

Domain Wall Fermions with 4-d EO preconditioning

Version 2.0.1

Andrew Pochinsky

January 2, 2004

Abstract

This version is optimized for slow network and uses vector extension of gcc 3.3.

1 DEFINITIONS

Here is the definition of the DWF Dirac operator we are using:

$$\begin{aligned} \chi_{s,x} = D\psi &= M_0\psi_{s,x} + \sum_{\mu} \left((1 + \gamma_{\mu})U_{x,\mu}\psi_{s,x+\hat{\mu}} + (1 - \gamma_{\mu})U_{x-\hat{\mu},\mu}^{\dagger}\psi_{s,x-\hat{\mu}} \right) \\ &+ (1 + \gamma_5)M_s^{(+)}\psi_{s+1,x} + (1 - \gamma_5)M_s^{(-)}\psi_{s-1,x} \end{aligned}$$

where

$$M_s^{(+)} = \begin{cases} 1, & \text{if } s < N_s - 1 \\ -m_f, & \text{if } s = N_s - 1 \end{cases}$$

and

$$M_s^{(-)} = \begin{cases} 1, & \text{if } s > 0 \\ -m_f, & \text{if } s = 0 \end{cases}$$

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

As Kostas has shown, one can do 4-d even/odd preconditioning to DWF in this form. This allows us to concentrate on computing the operator

$$\phi = Q_{ee}^{-1}Q_{eo}\psi$$

Up to a scale factor, one has

$$Q_{ee} = \frac{1 + \gamma_5}{2} \begin{pmatrix} 1 & a & 0 & \cdots & 0 \\ 0 & 1 & a & & 0 \\ \vdots & & & & \vdots \\ b & 0 & 0 & \cdots & 1 \end{pmatrix} + \frac{1 - \gamma_5}{2} \begin{pmatrix} 1 & 0 & \cdots & 0 & b \\ a & 1 & 0 & & 0 \\ \vdots & & & & \vdots \\ 0 & \cdots & a & 1 \end{pmatrix}$$

Now, $a = 2/M_0$, and $b = -2m_f/M_0$ are numbers and color structure of Q_{ee} is trivial.

The trickiest part to do on a vector architecture is computing Q_{ee}^{-1} with high efficiency. To achieve good performance, we employ a few tricks. First, one s -slice fits comfortably into L1 cache. This allows us to (a) reuse the U field, thus reducing memory traffic, and (b) apply Q_{ee}^{-1} to the result on a s -slice by slice basis while the result of $Q_{eo}\psi$ is still in cache.

2 γ MATRICES

Choice of γ -matrices made in `sse.nw` is still convenient for two reasons:

- γ_5 is diagonal. It allows one to simplify Q_{ee}^{-1} computation.
- Signs in γ_2 make the final step in Q_{eo} a tiny bit simpler. It might be not hugely advantageous, but it cuts a bit a number of floating point operations needed, which is good.

The separated projection part reads `psi` sequentially and stores the results into eight buffers `proj[d][xx.Fproj[d]]` rearranging sites to facilitate communication. It is non-obvious, that one loop with one reader and eight writers gives higher performance than eight loops with one reader and eight writers each. A small efficiency study left for the sake of getting something running yesterday.

$$\gamma_0 = 1 \otimes \sigma_2 = \begin{pmatrix} 0 & -i & 0 & 0 \\ i & 0 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{pmatrix}$$

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

$\langle \text{Build } (1 + \gamma_0) \text{ projection of } *f \text{ in } *g \rangle \equiv$

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

$\langle \text{Unproject and accumulate } (1 + \gamma_0) \text{ link} \rangle \equiv$

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

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

$\langle \text{Build } (1 - \gamma_0) \text{ projection of } *f \text{ in } *g \rangle \equiv$

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

$\langle \text{Unproject and accumulate } (1 - \gamma_0) \text{ link} \rangle \equiv$

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

$$\gamma_1 = \sigma_1 \otimes \sigma_1 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

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

$\langle \text{Build } (1 + \gamma_1) \text{ projection of } *f \text{ in } *g \rangle \equiv$

```
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;
```

$\langle \text{Unproject } (1 + \gamma_1) \text{ link} \rangle \equiv$

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

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

$\langle \text{Build } (1 - \gamma_1) \text{ projection of } *f \text{ in } *g \rangle \equiv$

```
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;
```

$\langle \text{Unproject and accumulate } (1 - \gamma_1) \text{ link} \rangle \equiv$

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

$$\gamma_2 = \sigma_2 \otimes \sigma_1 = \begin{pmatrix} 0 & 0 & 0 & -i \\ 0 & 0 & -i & 0 \\ 0 & i & 0 & 0 \\ i & 0 & 0 & 0 \end{pmatrix}$$

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

$\langle \text{Build } (1 + \gamma_2) \text{ projection of } *f \text{ in } *g \rangle \equiv$

```
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;
```

$\langle \text{Unproject and accumulate } (1 + \gamma_2) \text{ link} \rangle \equiv$

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

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

$\langle \text{Build } (1 - \gamma_2) \text{ projection of } *f \text{ in } *g \rangle \equiv$

```
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;
```

$\langle \text{Unproject and accumulate } (1 - \gamma_2) \text{ link} \rangle \equiv$

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

$$\gamma_3 = \sigma_3 \otimes \sigma_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

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

$\langle \text{Build } (1 + \gamma_3) \text{ projection of } *f \text{ in } *g \rangle \equiv$

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

$\langle \text{Unproject and accumulate } (1 + \gamma_3) \text{ link} \rangle \equiv$

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

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

$\langle \text{Build } (1 - \gamma_3) \text{ projection of } *f \text{ in } *g \rangle \equiv$

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

$\langle \text{Unproject and accumulate } (1 - \gamma_3) \text{ link} \rangle \equiv$

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

These γ -matrices were chosen to make γ_5 diagonal:

$$\gamma_5 = 1 \otimes \sigma_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

This, in turn, makes computation of $Q_{ee}^{-1}\psi$ conceptually easy, because $(1 \pm \gamma_5)$ acts only on upper/lower Dirac components. Therefore, we need a method to compute inverses of the following two matrices:

$$A = \begin{pmatrix} 1 & a & 0 & \cdots & 0 \\ 0 & 1 & a & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & & \cdots & 1 & a \\ b & 0 & \cdots & 0 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 0 & \cdots & 0 & b \\ a & 1 & \cdots & & 0 \\ 0 & a & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & 1 & 0 \\ 0 & 0 & \cdots & a & 1 \end{pmatrix}$$

If we know how to compute A^{-1} and B^{-1} , computing Q_{ee}^{-1} is easy:

$$Q_{ee}^{-1} = \frac{1 + \gamma_5}{2} A^{-1} + \frac{1 - \gamma_5}{2} B^{-1}.$$

If $A = L_A R_A$, $B = L_B R_B$, where R_A and R_B are bidiagonal:

$$R_A = \begin{pmatrix} 1 & a & 0 & \cdots & 0 \\ 0 & 1 & a & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & & \cdots & 1 & a \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix} \quad R_B = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ a & 1 & \cdots & & 0 \\ 0 & a & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & 1 & 0 \\ 0 & 0 & \cdots & a & 1 \end{pmatrix},$$

one can easily find

$$L_A = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ \vdots & & \ddots & & & \vdots \\ b & -ab & a^2b & -a^3b & \cdots & 1 + (-a)^{n-1}b \end{pmatrix} \quad L_B = \begin{pmatrix} 1 + (-a)^{n-1}b & (-a)^{n-2}b & \cdots & a^2b & -ab & b \\ 0 & 1 & 0 & \cdots & & 0 \\ \vdots & & \ddots & & & \vdots \\ 0 & \cdots & & & 0 & 1 \end{pmatrix}$$

In these terms,

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

$\langle \text{Compute } A^{-1}\psi \text{ on upper two components} \rangle \equiv$

$\langle \text{Compute } L_A^{-1} \text{ part} \rangle$
 $\langle \text{Compute } R_A^{-1} \text{ part} \rangle$

$\langle \text{Compute } B^{-1}\psi \text{ on lower two components} \rangle \equiv$

$\langle \text{Compute } L_B^{-1} \text{ part} \rangle$
 $\langle \text{Compute } R_B^{-1} \text{ part} \rangle$

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

$$\begin{aligned} z_k^{(A)} &= \begin{cases} \sum_{j=0}^{n-2} \frac{(-a)^j b}{1+(-a)^{n-1}b} x_j + \frac{1}{1+(-a)^{n-1}b} x_{n-1}, & \text{if } k = n-1 \\ x_k, & \text{otherwise} \end{cases} \\ z_k^{(B)} &= \begin{cases} \frac{1}{1+(-a)^{n-1}b} x_0 + \sum_{j=1}^{n-1} \frac{(-a)^{n-1-j} b}{1+(-a)^{n-1}b} x_j, & \text{if } k = 0 \\ x_k, & \text{otherwise} \end{cases} \end{aligned}$$

We can compute z *in situ*. Care should be taken, however, to use SSE in the sums.

```

⟨Compute  $L_A^{-1}$  part⟩≡
    vhfzero(&zV);
    fx = ab_LA;
    for (s = 0; s < S_4_1; s++, fx = fx * va4) {
        VecFermion *rs = &rx[s];
        ⟨Compute  $zV \leftarrow zV + fx * rx_{[s]}^{up}$ ⟩
    }
    {
        VecFermion *rs = &rx[S_4_1];
        vput_3(&fx, c0);
        ⟨Compute  $zV \leftarrow zV + fx * rx_{[s]}^{up}$ ⟩
        for (c = 0; c < 3; c++) {
            ⟨Compute wall value in zX[c]⟩

            zn.re = rs->f[0][c].re;
            zn.im = rs->f[0][c].im;
            vput_3(&zn.re, zX[0][c].re);
            vput_3(&zn.im, zX[0][c].im);
            rs->f[0][c].re = zn.re;
            rs->f[0][c].im = zn.im;

            zn.re = rs->f[1][c].re;
            zn.im = rs->f[1][c].im;
            vput_3(&zn.re, zX[1][c].re);
            vput_3(&zn.im, zX[1][c].im);
            rs->f[1][c].re = zn.re;
            rs->f[1][c].im = zn.im;
        }
    }
}

```

To avoid strange things gcc does when SSE data is declared local to a block, we place all such variables on the function level:

```

⟨QQxx locals⟩≡
    vreal fx;
    VecHalfFermion zV;
    vcomplex zn, z1, z2, z3;
    complex zX[2][3];

```

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

```

⟨Compute  $zV \leftarrow zV + fx * rx_{[s]}^{up}$ ⟩≡
    for (c = 0; c < 3; c++) {
        zV.f[0][c].re += fx * rs->f[0][c].re;
        zV.f[0][c].im += fx * rs->f[0][c].im;
        zV.f[1][c].re += fx * rs->f[1][c].re;
        zV.f[1][c].im += fx * rs->f[1][c].im;
    }
}

```

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

```

⟨Compute wall value in zX[c]⟩≡
    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);

```

Computing L_B^{-1} differs fro L_A^{-1} only in the direction of the loop over the fifth dimension and change from the upper to the lower half of fermion indices.

```

⟨Compute  $L_B^{-1}$  part⟩≡
    vhfzero(&zV);
    fx = ab_LB;
    for (s = S_4; --s; fx = fx * va4) {
        VecFermion *rs = &rx[s];
        ⟨Compute  $zV \leftarrow zV + fx * rx_{[s]}^{down}$ ⟩
    }
    {
        VecFermion *rs = &rx[0];
        vput_0(&fx, c0);
        ⟨Compute  $zV \leftarrow zV + fx * rx_{[s]}^{down}$ ⟩
        for (c = 0; c < 3; c++) {
            ⟨Compute wall value in zX[c]⟩

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

            zn.re = rs->f[3][c].re;
            zn.im = rs->f[3][c].im;
            vput_0(&zn.re, zX[1][c].re);
            vput_0(&zn.im, zX[1][c].im);
            rs->f[3][c].re = zn.re;
            rs->f[3][c].im = zn.im;
        }
    }
}

```

This piece is used twice: once in the loop over L_s , and the second time after correcting s_0 :

```

⟨Compute  $zV \leftarrow zV + fx * rx_{[s]}^{down}$ ⟩≡
    for (c = 0; c < 3; c++) {
        zV.f[0][c].re += fx * rs->f[2][c].re;
        zV.f[0][c].im += fx * rs->f[2][c].im;
        zV.f[1][c].re += fx * rs->f[3][c].re;
        zV.f[1][c].im += fx * rs->f[3][c].im;
    }
}

```

One can compute $y^{(A)} = R_A^{-1}x$ and $y^{(B)} = R_B^{-1}x$ iteratively:

$$\begin{aligned} y_k^{(A)} &= \begin{cases} x_k, & \text{if } k = n-1 \\ x_k - ay_{k+1}^{(A)}, & \text{otherwise} \end{cases} \\ y_k^{(B)} &= \begin{cases} x_0, & \text{if } k = 0 \\ x_k - ay_{k-1}^{(B)}, & \text{otherwise} \end{cases} \end{aligned}$$

We need one last step to make computation of R^{-1} vector-friendly. Unrolling iterative definitions four times, one gets:

$$\begin{aligned} y_k^{(A)} &= \begin{cases} x_k, & \text{if } k = n-1 \\ x_k - ay_{k+1}^{(A)}, & \text{if } n-4 \leq k \leq n-2 \\ x_k - ax_{k+1} + a^2x_{k+2} - a^3x_{k+3} + a^4y_{k+4}^{(A)}, & \text{otherwise} \end{cases} \\ y_k^{(B)} &= \begin{cases} x_0, & \text{if } k = 0 \\ x_k - ay_{k-1}^{(B)}, & \text{if } 1 \leq k \leq 3 \\ x_k - ax_{k-1} + a^2x_{k-2} - a^3x_{k-3} + a^4y_{k-4}^{(B)}, & \text{otherwise} \end{cases} \end{aligned}$$

If we extend x by setting $x_k = 0$ iff $k < 0$ or $k \geq n$ and also set $y_k^{(A)} = 0$ for $k \geq n$ and $y_k^{(B)} = 0$ for $k < 0$, we do not need special cases on the boundaries.

Again, it is better to place `xOut` and `yOut` on the function level:

```
<QQxx locals>+=
  VecHalfFermion xOut;
  VecHalfFermion yOut;
```

With such an extended x and y we can use the following formulae:

$$\begin{aligned} y_k^{(A)} &= x_k - ax_{k+1} + a^2x_{k+2} - a^3x_{k+3} + a^4y_{k+4}^{(A)} \\ y_k^{(B)} &= x_k - ax_{k-1} + a^2x_{k-2} - a^3x_{k-3} + a^4y_{k-4}^{(B)} \end{aligned}$$

```
<Compute R_A^{-1} part>=
  <Init out of bound x and y>
  for (s = S_4; s--;) {
    VecFermion *rs = &rx[s];
    for (c = 0; c < 3; c++) {
      <Compute y_{k,[0]}^{(A)}>
      <Compute y_{k,[1]}^{(A)}>
    }
  }
```

```
<Compute y_{k,[0]}^{(A)}>=
  zn.re = rs->f[0][c].re;
  zn.im = rs->f[0][c].im;
  z1.re = shift_down1(zn.re, xOut.f[0][c].re);
  z1.im = shift_down1(zn.im, xOut.f[0][c].im);
  z2.re = shift_down2(zn.re, xOut.f[0][c].re);
  z2.im = shift_down2(zn.im, xOut.f[0][c].im);
  z3.re = shift_down3(zn.re, xOut.f[0][c].re);
  z3.im = shift_down3(zn.im, xOut.f[0][c].im);
  rs->f[0][c].re = zn.re - va1*z1.re + va2*z2.re - va3*z3.re + va4*yOut.f[0][c].re;
  rs->f[0][c].im = zn.im - va1*z1.im + va2*z2.im - va3*z3.im + va4*yOut.f[0][c].im;
  yOut.f[0][c].re = rs->f[0][c].re;
  yOut.f[0][c].im = rs->f[0][c].im;
  xOut.f[0][c].re = zn.re;
  xOut.f[0][c].im = zn.im;
```

```

⟨Compute  $y_{k,[1]}^{(A)}\rangle \equiv$ 
  zn.re = rs->f[1][c].re;
  zn.im = rs->f[1][c].im;
  z1.re = shift_down1(zn.re, xOut.f[1][c].re);
  z1.im = shift_down1(zn.im, xOut.f[1][c].im);
  z2.re = shift_down2(zn.re, xOut.f[1][c].re);
  z2.im = shift_down2(zn.im, xOut.f[1][c].im);
  z3.re = shift_down3(zn.re, xOut.f[1][c].re);
  z3.im = shift_down3(zn.im, xOut.f[1][c].im);
  rs->f[1][c].re = zn.re - va1*z1.re + va2*z2.re - va3*z3.re + va4*yOut.f[1][c].re;
  rs->f[1][c].im = zn.im - va1*z1.im + va2*z2.im - va3*z3.im + va4*yOut.f[1][c].im;
  xOut.f[1][c].re = zn.re;
  xOut.f[1][c].im = zn.im;
  yOut.f[1][c].re = rs->f[1][c].re;
  yOut.f[1][c].im = rs->f[1][c].im;

```

```

⟨Compute  $R_B^{-1}$  part⟩≡
  ⟨Init out of bound x and y⟩
  for (s = 0; s < S_4; s++) {
    VecFermion *rs = &rx[s];
    for (c = 0; c < 3; c++) {
      ⟨Compute  $y_{k,[2]}^{(B)}\rangle$ 
      ⟨Compute  $y_{k,[3]}^{(B)}\rangle$ 
    }
  }

```

```

⟨Compute  $y_{k,[2]}^{(B)}\rangle \equiv$ 
  zn.re = rs->f[2][c].re;
  zn.im = rs->f[2][c].im;
  z1.re = shift_up1(xOut.f[0][c].re, zn.re);
  z1.im = shift_up1(xOut.f[0][c].im, zn.im);
  z2.re = shift_up2(xOut.f[0][c].re, zn.re);
  z2.im = shift_up2(xOut.f[0][c].im, zn.im);
  z3.re = shift_up3(xOut.f[0][c].re, zn.re);
  z3.im = shift_up3(xOut.f[0][c].im, zn.im);
  rs->f[2][c].re = zn.re - va1*z1.re + va2*z2.re - va3*z3.re + va4*yOut.f[0][c].re;
  rs->f[2][c].im = zn.im - va1*z1.im + va2*z2.im - va3*z3.im + va4*yOut.f[0][c].im;
  yOut.f[0][c].re = rs->f[2][c].re;
  yOut.f[0][c].im = rs->f[2][c].im;
  xOut.f[0][c].re = zn.re;
  xOut.f[0][c].im = zn.im;

```

```

⟨Compute  $y_{k,[3]}^{(B)}\rangle \equiv$ 
  zn.re = rs->f[3][c].re;
  zn.im = rs->f[3][c].im;
  z1.re = shift_up1(xOut.f[1][c].re, zn.re);
  z1.im = shift_up1(xOut.f[1][c].im, zn.im);
  z2.re = shift_up2(xOut.f[1][c].re, zn.re);
  z2.im = shift_up2(xOut.f[1][c].im, zn.im);
  z3.re = shift_up3(xOut.f[1][c].re, zn.re);
  z3.im = shift_up3(xOut.f[1][c].im, zn.im);
  rs->f[3][c].re = zn.re - va1*z1.re + va2*z2.re - va3*z3.re + va4*yOut.f[1][c].re;
  rs->f[3][c].im = zn.im - va1*z1.im + va2*z2.im - va3*z3.im + va4*yOut.f[1][c].im;
  yOut.f[1][c].re = rs->f[3][c].re;
  yOut.f[1][c].im = rs->f[3][c].im;
  xOut.f[1][c].re = zn.re;
  xOut.f[1][c].im = zn.im;

```

```

⟨Init out of bound x and y⟩≡
  vhfzero(&xOut);
  vhfzero(&yOut);

```


The only part left is computing lattice-independent values for the fifth dimension.
This completes computation of $Q_{ee}^{-1}x$.

3 TEST CODE

Here we put together a simple test code to gauge performance of the SSE implementation.

```

⟨main.c⟩≡
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "qcd.h"
int
main(int argc, char *argv[])
{
    ⟨Main local variables⟩

    ⟨Check arguments⟩
    ⟨Initialize everything⟩
    ⟨Start timer⟩
    for (i = 0; i < count; i++)
        qdwcg_QooQoe(phi, U, psi, 0.1, -0.01);
    ⟨Stop timer⟩
    ⟨Cleanup everything⟩
    ⟨Show time⟩
    return 0;
}

```

The test code takes a lot of arguments because it tries to mimic various run-time configurations of lattice sizes and network topology.

```

⟨Check arguments⟩≡
if (argc != 11) {
    fprintf(stderr, "Usage: dw X Y Z T S a b c d N\n");
    exit(1);
}
for (i = 1; i < 6; i++) {
    size[i-1] = atoi(argv[i]);
    if (size[i-1] < 2) {
        fprintf(stderr, "Bad value for %c (%d)\n", "XYZTS"[i-1], size[i-1]);
        exit(1);
    }
}
for (i = 6; i < 10; i++) {
    torus[i-6] = atoi(argv[i]);
    if (torus[i-6] < 1) {
        fprintf(stderr, "Bad value for %c (%d)\n", "abcd"[i-6], torus[i-6]);
        exit(1);
    }
}
count = atoi(argv[10]);

⟨Main local variables⟩≡
int size[5];
int torus[4];
static int position[4] = {0, 0, 0, 0};
int count, i;

```

In this performance meter we only need to get a (random) gauge field, a random even half fermion and an uninitialized odd half fermion.

```

⟨Initialize everything⟩≡
    qdwcg_init(size, torus, position);
    psi = qdwcg_allocate_even_hf();
    phi = qdwcg_allocate_odd_hf();
    U = qdwcg_allocate_gauge();
    qdwcg_random_gauge(U);
    qdwcg_random_even_hf(psi);

```

```

⟨Main local variables⟩+≡
    EvenFermion *psi;
    OddFermion *phi;
    SU3 *U;

```

Cleanup is simple the inverse of initialization:

```

⟨Cleanup everything⟩≡
    qdwcg_free_gauge(U);
    qdwcg_free_odd_hf(phi);
    qdwcg_free_even_hf(psi);
    qdwcg_fini();

```

4 MEASURING TIME

There are many reasons not to use `clock()` for performance measurements. Instead, we use `getrusage()` here. It's in the standard and does not show funny behavior that `clock()` does under Linux.

```

⟨Main local variables⟩+≡
    struct rusage us0, us1;
    struct timeval t0, t1;

```

Starting the timer is simple:

```

⟨Start timer⟩≡
    getrusage(RUSAGE_SELF, &us0);

```

When the timer is stopped, we take our time to extract `struct timeval` from both `us0` and `us1`:

```

⟨Stop timer⟩≡
    getrusage(RUSAGE_SELF, &us1);
    t0 = us0.ru_utime;
    t1 = us1.ru_utime;

```

To show the results of the run, convert `struct timeval` to the interval, and do the arithmetics:

```

⟨Show time⟩≡
    dt = t1.tv_sec - t0.tv_sec + 10e-6*(t1.tv_usec - t0.tv_usec);
    if (dt > 0) {
        printf("Performance %g Mflops/sec. SubLattice %d %d %d %d %d ."
               " %d Iterations. %g total seconds\n",
               (double)count * qdwcg_QQOps() / dt / 1e6,
               size[0] / torus[3],
               size[1] / torus[3],
               size[2] / torus[3],
               size[3] / torus[3],
               size[4], count, dt);
    } else {
        printf("*** QeeQleo ran too fast. Try increasing count\n");
    }

```

```

⟨Main local variables⟩+≡
    double dt;

```

5 INTERFACE

With the test code composed, let us put together the interface into the Domain Wall Fermion Conjugate Gradient. An advantage of classical C is that it allows for clear separation between an interface and an implementation. Abstraction barriers are good.

```
<qcd.h>≡
    #ifndef qcd_h
    #define qcd_h
    <QCD data type declarations>
    <QCD interface functions>
    #endif
```

For our present purposes we need three types only. There is no need to reveal the implementation details at this stage:

```
<QCD data type declarations>≡
    typedef struct SU3 SU3;
    typedef struct EvenFermion EvenFermion;
    typedef struct OddFermion OddFermion;
```

The interface itself is a set of functions. In order to avoid name conflicts, all of them start with prefix `qdwcg_`.

As one can see, the initializer takes the total lattice size and the geometry of the machine as arguments.

Next comes a set of memory allocators and deallocators. Since we do not reveal implementations of lattice objects, the allocators below are the only way to construct the objects.

```
<QCD interface functions>≡
    SU3      *qdwcg_allocate_gauge(void);
    void      qdwcg_free_gauge(SU3 *);
    EvenFermion *qdwcg_allocate_even_hf(void);
    void      qdwcg_free_even_hf(EvenFermion *);
    OddFermion *qdwcg_allocate_odd_hf(void);
    void      qdwcg_free_odd_hf(OddFermion *);
```

For tests, it is convenient to be able to fill objects with random floating point numbers. In case of gauge field it is not a SU_3 parallel transport, but it is good enough for us now.

```
<QCD interface functions>+≡
    void qdwcg_random_gauge(SU3 *);
    void qdwcg_random_even_hf(EvenFermion *);
```

Finally, the function we are timing:

```
<QCD interface functions>+≡
    void qdwcg_QooQoe(OddFermion *,
                      const SU3 *, const EvenFermion *,
                      double a, double b);
    int  qdwcg_QQOps(void);
```

6 THE IMPLEMENTATION

```
<qcd.c>≡
    #include <string.h>
    #include <stdlib.h>
    #include <math.h>
    #include "qcd.h"
    <Type definitions>
    <Global data>
    <Function prototypes>
    <Functions>
```

Let us start with memory allocators. For the interface, we need to provide an allocator for each exported datatype. One problem is, however, that SSE imposes restrictions on the data alignment for efficient memory accesses. This leads us to the following solution:

```

<Functions>≡
EvenFermion *
qdwcg_allocate_even_hf(void)
{
    return allocate(even_odd.size * S_4 * sizeof (VecFermion));
}

void
qdwcg_free_even_hf(EvenFermion *f)
{
    deallocate(f);
}

OddFermion *
qdwcg_allocate_odd_hf(void)
{
    return allocate(odd_even.size * S_4 * sizeof (VecFermion));
}

void
qdwcg_free_odd_hf(OddFermion *f)
{
    deallocate(f);
}

```

<Type definitions>≡

Though the gauge field does not need to be aligned at 16 bytes, let us use the same mechanism for it as well:

```

<Functions>+≡
SU3 *
qdwcg_allocate_gauge(void)
{
    return allocate(gauge_XYZT * sizeof (SU3));
}

void
qdwcg_free_gauge(SU3 *f)
{
    deallocate(f);
}

```

<Type definitions>+≡

```

typedef float real;

typedef struct {
    real re, im;
} complex;

struct SU3 {
    complex v[3][3];
};

```

We build an aligned allocator on top of malloc() to avoid extra problems:

```

<Type definitions>+≡
struct memblock {
    struct memblock *next, *prev;
    unsigned int size;
    void *ptr;
};

```

```

⟨Global data⟩≡
    static struct memblock memblock;

```

```

⟨Function prototypes⟩≡
    static void *allocate(unsigned);

```

One should not call the allocators before QDWCG has been initialized:

```

⟨Functions⟩+≡
    static void *
    allocate(unsigned size)
    {
        static int inited = 0;
        int s = size + 15 + sizeof (struct memblock);
        void *ptr = malloc(s);
        struct memblock *bl;

        if (!inited) {
            memblock.prev = memblock.next = &memblock;
            memblock.size = 0;
            memblock.ptr = 0;
            inited = 1;
        }
        if (ptr == 0)
            return 0;
        ⟨Correct alignment⟩
    }

```

```

⟨Correct alignment⟩≡
    bl = ptr;
    bl->size = size;
    bl->ptr = (void *) (~15 & (15 + (unsigned long)(bl + 1)));
    bl->next = memblock.next;
    bl->prev = &memblock;
    memblock.next->prev = bl;
    memblock.next = bl;
    return bl->ptr;

```

When memory is deallocated, we run through the chain of blocks, looking for the right pointer:

```

⟨Function prototypes⟩+≡
    static void deallocate(void *);

⟨Functions⟩+≡
    static void
    deallocate(void *p)
    {
        struct memblock *b;

        for (b = memblock.next; b != &memblock; b++) {
            if (b->ptr == p) {
                b->next->prev = b->prev;
                b->prev->next = b->next;
                free(b);
                return;
            }
        }
        /* report error here */
    }

```

6.1 Random values

For performance testing we fill the sources with random values. It is not very important to keep elements of the gauge field in $SU(3)$ for our present purposes.

```
<Functions>+≡
void
qdwcg_random_gauge(SU3 *U)
{
    int x, c1, c2;

    for (x = gauge_XYZT; x--; ) {
        for (c1 = 0; c1 < 3; c1++) {
            for (c2 = 0; c2 < 3; c2++) {
                U[x].v[c1][c2].re = xrnd();
                U[x].v[c1][c2].im = xrnd();
            }
        }
    }
}
```

For `EvenFermion`, we need to use one of `vreal` constructors to fill in the data. This is not strictly necessary, but it helps one to think right.

```
<Functions>+≡
void
qdwcg_random_even_hf(EvenFermion *f)
{
    int x, d, c;
    for (x = even_odd.size * S_4; x--;) {
        for (d = 0; d < 4; d++) {
            for (c = 0; c < 3; c++) {
                f[x].f.f[d][c].re = vreal_mk4(xrnd(), xrnd(), xrnd(), xrnd());
                f[x].f.f[d][c].im = vreal_mk4(xrnd(), xrnd(), xrnd(), xrnd());
            }
        }
    }
}
```

Any random generator will do for now:

```
<Function prototypes>+≡
static real xrnd(void);

<Functions>+≡
real
xrnd(void)
{
    return 2.0 * rand() / (double)RAND_MAX - 1.0;
}
```

7 THE DIRAC OPERATOR

The top level of the Dirac operator is simple:

```
<Functions>+≡
void
qdwcg_QooQoe(OddFermion *phi,
              const SU3 *U,
              const EvenFermion *psi,
              double a, double b)
{
    build_boundaries(&psi->f, &odd_even);
    <Start communication>
    compute_insides(&phi->f, U, &psi->f, a, b, &odd_even);
    <Finish communication>
    compute_boundaries(&phi->f, U, &psi->f, a, b, &odd_even);
}
```

7.1 Building the boundaries

When the geometry had been analyzed, we computed all information needed to construct the send buffers. Now, there is little left to do but to compute the projections

```
<Function prototypes>+≡
static void build_boundaries(const VecFermion *psi, const struct neighbor *nb);

<Functions>+≡
static void
build_boundaries(const VecFermion *psi, const struct neighbor *nb)
{
    int i, s, c, *src;
    const VecFermion *f;
    VecHalfFermion *g;

    <Construct (1 + γ0) send buffer>
    <Construct (1 - γ0) send buffer>
    <Construct (1 + γ1) send buffer>
    <Construct (1 - γ1) send buffer>
    <Construct (1 + γ2) send buffer>
    <Construct (1 - γ2) send buffer>
    <Construct (1 + γ3) send buffer>
    <Construct (1 - γ3) send buffer>

}

<Construct (1 + γ0) send buffer>≡
for (i = nb->snd_size[0], g = nb->snd_buf[0], src = nb->snd[0]; i--; src++) {
    for (s = S_4, f = &psi[*src]; s--; g++, f++) {
        for (c = 0; c < 3; c++) {
            <Build (1 + γ0) projection of *f in *g>
        }
    }
}

<Construct (1 - γ0) send buffer>≡
for (i = nb->snd_size[1], g = nb->snd_buf[1], src = nb->snd[1]; i--; src++) {
    for (s = S_4, f = &psi[*src]; s--; g++, f++) {
        for (c = 0; c < 3; c++) {
            <Build (1 - γ0) projection of *f in *g>
        }
    }
}

}
```

```

 $\langle \text{Construct } (1 + \gamma_1) \text{ send buffer} \rangle \equiv$ 
  for (i = nb->snd_size[2], g = nb->snd_buf[2], src = nb->snd[2]; i--; src++) {
    for (s = S_4, f = &psi[*src]; s--; g++, f++) {
      for (c = 0; c < 3; c++) {
         $\langle \text{Build } (1 + \gamma_1) \text{ projection of } *f \text{ in } *g \rangle$ 
      }
    }
  }

 $\langle \text{Construct } (1 - \gamma_1) \text{ send buffer} \rangle \equiv$ 
  for (i = nb->snd_size[3], g = nb->snd_buf[3], src = nb->snd[3]; i--; src++) {
    for (s = S_4, f = &psi[*src]; s--; g++, f++) {
      for (c = 0; c < 3; c++) {
         $\langle \text{Build } (1 - \gamma_1) \text{ projection of } *f \text{ in } *g \rangle$ 
      }
    }
  }

 $\langle \text{Construct } (1 + \gamma_2) \text{ send buffer} \rangle \equiv$ 
  for (i = nb->snd_size[4], g = nb->snd_buf[4], src = nb->snd[4]; i--; src++) {
    for (s = S_4, f = &psi[*src]; s--; g++, f++) {
      for (c = 0; c < 3; c++) {
         $\langle \text{Build } (1 + \gamma_2) \text{ projection of } *f \text{ in } *g \rangle$ 
      }
    }
  }

 $\langle \text{Construct } (1 - \gamma_2) \text{ send buffer} \rangle \equiv$ 
  for (i = nb->snd_size[5], g = nb->snd_buf[5], src = nb->snd[5]; i--; src++) {
    for (s = S_4, f = &psi[*src]; s--; g++, f++) {
      for (c = 0; c < 3; c++) {
         $\langle \text{Build } (1 - \gamma_2) \text{ projection of } *f \text{ in } *g \rangle$ 
      }
    }
  }

 $\langle \text{Construct } (1 + \gamma_3) \text{ send buffer} \rangle \equiv$ 
  for (i = nb->snd_size[6], g = nb->snd_buf[6], src = nb->snd[6]; i--; src++) {
    for (s = S_4, f = &psi[*src]; s--; g++, f++) {
      for (c = 0; c < 3; c++) {
         $\langle \text{Build } (1 + \gamma_3) \text{ projection of } *f \text{ in } *g \rangle$ 
      }
    }
  }

 $\langle \text{Construct } (1 - \gamma_3) \text{ send buffer} \rangle \equiv$ 
  for (i = nb->snd_size[7], g = nb->snd_buf[7], src = nb->snd[7]; i--; src++) {
    for (s = S_4, f = &psi[*src]; s--; g++, f++) {
      for (c = 0; c < 3; c++) {
         $\langle \text{Build } (1 - \gamma_3) \text{ projection of } *f \text{ in } *g \rangle$ 
      }
    }
  }

```


7.2 Computing inside nodes

There is no communication for the inside nodes which means we can fill the time between send start and communication completion with it.

```

<Function prototypes>+=
static void
compute_insidess(VecFermion * __restrict__ phi,
                 const SU3 *U,
                 const VecFermion *psi,
                 double a, double b,
                 const struct neighbor *n);

```

We try to keep insides of the box sequential in memory, but `compute_insidess()` uses the translation address service provided by the neighbor table. Otherwise, `compute_insidess()` is a simple loop over inside sites.

```

<Functions>+=
static void
compute_insidess(VecFermion *phi,
                 const SU3 *U,
                 const VecFermion *psi,
                 double a, double b,
                 const struct neighbor *nb)
{
    <QQxx locals>

    <Compute constant values for  $Q_{ee}^{-1}$ >
    for (i = nb->inside_size; i--;) {
        xyzt = nb->inside[i];
        xyzt5 = xyzt * S_4;
        <Extract 1-d addresses>
        <Build SSE SU(3) objects>
        <Compute  $Q_{eo}$  part on the inside s-slice>
        <Compute  $Q_{ee}^{-1}$  part on the s-slice>
    }

}

```

Computing one the inner part of Q_{eo} is simple.

```

<Compute  $Q_{eo}$  part on the inside s-slice>=
for (s = 0; s < S_4; s++) {
    <Compute inside  $\gamma$ -projections>
    <Inside multiply by Vs>
    <Compute  $\gamma$ -unprojections and sum the results>
}

```

To reuse the γ -projection code, we make `f` point into the `VecFermion` source and `g` point into the `VecHalfFermion` destinations.

```

<Compute inside  $\gamma$ -projections>=
<Construct neighbor pointers>
for (c = 0; c < 3; c++) {
    f = &psi[ps[0]]; g = &gg[0]; <Build  $(1 + \gamma_0)$  projection of *f in *g>
    f = &psi[ps[1]]; g = &gg[1]; <Build  $(1 - \gamma_0)$  projection of *f in *g>
    f = &psi[ps[2]]; g = &gg[2]; <Build  $(1 + \gamma_1)$  projection of *f in *g>
    f = &psi[ps[3]]; g = &gg[3]; <Build  $(1 - \gamma_1)$  projection of *f in *g>
    f = &psi[ps[4]]; g = &gg[4]; <Build  $(1 + \gamma_2)$  projection of *f in *g>
    f = &psi[ps[5]]; g = &gg[5]; <Build  $(1 - \gamma_2)$  projection of *f in *g>
    f = &psi[ps[6]]; g = &gg[6]; <Build  $(1 + \gamma_3)$  projection of *f in *g>
    f = &psi[ps[7]]; g = &gg[7]; <Build  $(1 - \gamma_3)$  projection of *f in *g>
}

```

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

```

<Inside multiply by Vs>≡
for (d = 0; d < 8; d++) {
    VecHalfFermion *h = &hh[d];
    VecSU3 *u = &V[d];
    g = &gg[d];
    <Multiply *u by *g and store the result in *h>
}

```

7.3 Computing boundary nodes

On the sites having remote neighbors, the computation requires handling various cases of the boundary position. Fortunately, these cases nicely factors as follows.

```

<Function prototypes>+≡
static void compute_boundaries(VecFermion *phi,
                                const SU3 *U,
                                const VecFermion *psi,
                                double a, double b,
                                const struct neighbor *n);

```

We try to keep boundaries of the box sequential in memory, but `compute_boundaries()` uses the translation address service provided by the neighbor table.

```

<Functions>+≡
static void
compute_boundaries(VecFermion *phi,
                    const SU3 *U,
                    const VecFermion *psi,
                    double a, double b,
                    const struct neighbor *nb)
{
    <QQxx locals>

    <Compute constant values for  $Q_{ee}^{-1}$ >
    for (i = nb->boundary_size; i--;) {
        int m = nb->boundary[i].mask;
        xyzt = nb->boundary[i].index;
        xyzt5 = xyzt * S_4;
        <Extract 1-d addresses>
        <Build SSE SU(3) objects>
        <Compute  $Q_{eo}$  part on the boundary s-slice>
        <Compute  $Q_{ee}^{-1}$  part on the s-slice>
    }

}

```

Computing the boundary slice is similar to the inside slice on the top level:

```

<Compute  $Q_{eo}$  part on the boundary s-slice>≡
for (s = 0; s < S_4; s++) {
    <Compute boundary  $\gamma$ -projections>
    <Boundary multiply by Vs>
    <Compute  $\gamma$ -unprojections and sum the results>
}

```

This is the first place where we handle boundaries specially. Only local neighbors are projected into g here.

```

⟨Compute boundary  $\gamma$ -projections⟩≡
  ⟨Construct neighbor pointers⟩
  for (c = 0; c < 3; c++) {
    if ((m & 0x01) == 0) { f = &psi[ps[0]]; g = &gg[0]; ⟨Build  $(1 + \gamma_0)$  projection of *f in *g⟩ }
    if ((m & 0x02) == 0) { f = &psi[ps[1]]; g = &gg[1]; ⟨Build  $(1 - \gamma_0)$  projection of *f in *g⟩ }
    if ((m & 0x04) == 0) { f = &psi[ps[2]]; g = &gg[2]; ⟨Build  $(1 + \gamma_1)$  projection of *f in *g⟩ }
    if ((m & 0x08) == 0) { f = &psi[ps[3]]; g = &gg[3]; ⟨Build  $(1 - \gamma_1)$  projection of *f in *g⟩ }
    if ((m & 0x10) == 0) { f = &psi[ps[4]]; g = &gg[4]; ⟨Build  $(1 + \gamma_2)$  projection of *f in *g⟩ }
    if ((m & 0x20) == 0) { f = &psi[ps[5]]; g = &gg[5]; ⟨Build  $(1 - \gamma_2)$  projection of *f in *g⟩ }
    if ((m & 0x40) == 0) { f = &psi[ps[6]]; g = &gg[6]; ⟨Build  $(1 + \gamma_3)$  projection of *f in *g⟩ }
    if ((m & 0x80) == 0) { f = &psi[ps[7]]; g = &gg[7]; ⟨Build  $(1 - \gamma_3)$  projection of *f in *g⟩ }
  }

```

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

```

⟨Boundary multiply by  $Vs$ ⟩≡
  for (d = 0; d < 8; d++) {
    VecHalfFermion *h = &hh[d];
    VecSU3 *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⟩
  }

```

7.4 Common node code pieces

Some local we definitely need:

```

⟨QQxx locals⟩+≡
  int i, xyzt, xyzt5, d, c1, c2;
  const SU3 *Uup, *Udown;
  VecSU3 V[8];

```

For the inside sites, compute the s -chain address of the neighbor:

```

⟨Extract 1-d addresses⟩≡
  for (d = 0; d < 8; d++)
    p5[d] = nb->site[d].F[xyzt];

```

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

```

⟨Construct neighbor pointers⟩≡
  for (d = 0; d < 8; d++) {
    ps[d] = p5[d] + s;
  }

```

Since the gauge field does not depend on the s , we save a lot of memory bandwidth by loading the gauge field into the cache and running over the s -slice of the fermions. Strange, but true: gcc 3.3.2 likes the negation where it is in `V[d*2+1].v[c1][c2].im`.

```

⟨Build SSE SU(3) objects⟩≡
  Uup = &U[nb->site[xyzt].Uup];
  for (d = 0; d < 4; d++, Uup++) {
    Udown = &U[nb->site[xyzt].Udown[d]];
    for (c1 = 0; c1 < 3; c1++) {
      for (c2 = 0; c2 < 3; c2++) {
        V[d*2+0].v[c1][c2].re = vreal_mk1(Uup->v[c1][c2].re);
        V[d*2+0].v[c1][c2].im = vreal_mk1(Uup->v[c1][c2].im);
        /* conjugate down-link */
        V[d*2+1].v[c1][c2].re = vreal_mk1(Udown->v[c2][c1].re);
        V[d*2+1].v[c1][c2].im = vreal_mk1(-Udown->v[c2][c1].im);
      }
    }
  }

```

```

⟨QQxx locals⟩+=
  int s, c;
  VecHalfFermion gg[8], hh[8];
  VecHalfFermion *g;
  const VecFermion *f;
  int ps[8], p5[8];

⟨Multiply *u by *g and store the result in *h⟩≡
  for (c = 0; c < 3; c++) {
    h->f[c].re=u->v[c][0].re*g->f[0][0].re-u->v[c][0].im*g->f[0][0].im
      +u->v[c][1].re*g->f[0][1].re-u->v[c][1].im*g->f[0][1].im
      +u->v[c][2].re*g->f[0][2].re-u->v[c][2].im*g->f[0][2].im;
    h->f[0][c].im=u->v[c][0].im*g->f[0][0].re+u->v[c][0].re*g->f[0][0].im
      +u->v[c][1].im*g->f[0][1].re+u->v[c][1].re*g->f[0][1].im
      +u->v[c][2].im*g->f[0][2].re+u->v[c][2].re*g->f[0][2].im;
    h->f[1][c].re=u->v[c][0].re*g->f[1][0].re-u->v[c][0].im*g->f[1][0].im
      +u->v[c][1].re*g->f[1][1].re-u->v[c][1].im*g->f[1][1].im
      +u->v[c][2].re*g->f[1][2].re-u->v[c][2].im*g->f[1][2].im;
    h->f[1][c].im=u->v[c][0].im*g->f[1][0].re+u->v[c][0].re*g->f[1][0].im
      +u->v[c][1].im*g->f[1][1].re+u->v[c][1].re*g->f[1][1].im
      +u->v[c][2].im*g->f[1][2].re+u->v[c][2].re*g->f[1][2].im;
  }

```

The final part of `compute_insidess()` is unprojecting the products and summing up contributions from all eight directions. Here we use the fact that $(1 + \gamma_1)$ has a special form which allows us to save eight additions.

```

⟨Compute  $\gamma$ -unprojections and sum the results⟩≡
  rs = &rx[s];
  for (c = 0; c < 3; c++) {
    ⟨Unproject  $(1 + \gamma_1)$  link⟩
    ⟨Unproject and accumulate  $(1 - \gamma_1)$  link⟩
    ⟨Unproject and accumulate  $(1 + \gamma_0)$  link⟩
    ⟨Unproject and accumulate  $(1 - \gamma_0)$  link⟩
    ⟨Unproject and accumulate  $(1 + \gamma_2)$  link⟩
    ⟨Unproject and accumulate  $(1 - \gamma_2)$  link⟩
    ⟨Unproject and accumulate  $(1 + \gamma_3)$  link⟩
    ⟨Unproject and accumulate  $(1 - \gamma_3)$  link⟩
  }

```

We run through the results by `xyzt` indices:

```

⟨Extract 1-d addresses⟩+=
  rx = &phi[xyzt5];

⟨QQxx locals⟩+=
  VecFermion * __restrict__ rx, * __restrict__ rs;

```

7.5 Computing Q_{ee}^{-1}

Once we get an s -chain of $Q_{eo}\psi$, we can compute a part of $Q_{ee}^{-1}Q_{eo}\psi$. The good news is that all data needed is still in the cache and we get good performance on both A^{-1} and B^{-1} below.

```

⟨Compute  $Q_{ee}^{-1}$  part on the  $s$ -slice⟩≡
  ⟨Compute  $A^{-1}\psi$  on upper two components⟩
  ⟨Compute  $B^{-1}\psi$  on lower two components⟩

```

7.6 Constants needed for Q_{ee}^{-1}

(Compute constant values for Q_{ee}^{-1})≡

```
real c0 = 1./(1+b*pow(a, S_4*4-1));
vreal va1 = vreal_mk1(a);
vreal va2 = va1 * va1;
vreal va3 = va1 * va2;
vreal va4 = va2 * va2;
vreal ab_LA = vreal_mk4(c0*b, -a*c0*b, a*a*c0*b, -a*a*a*c0*b);
vreal ab_LB = vreal_mk4(-a*a*a*c0*b, a*a*c0*b, -a*c0*b, c0*b);
```

7.7 Operation count

Finally, let us count the floating point operations.

γ -projections	$2(\text{complex}) \times 3(\text{colors}) \times 2(\text{half fermions}) \times 8(\text{directions}) \times 1(\text{add})$	=	96
γ -unprojections	$2(\text{complex}) \times 3(\text{colors}) \times 4(\text{full fermions}) \times 7(\text{directions}) \times 1(\text{add})$	=	168
Gauge multiplications	$2(\text{complex}) \times 3(\text{colors}) \times 2(\text{half fermions}) \times 8(\text{directions}) \times (6(\text{multiply}) + 5(\text{add}))$	=	1056
Applying L_A^{-1} and L_B^{-1}	$2(L_A^{-1} \text{ and } L_B^{-1}) \times 2(\text{complex}) \times 3(\text{colors}) \times 2(\text{half fermions}) \times (1(\text{multiply}) + 1(\text{add}))$	=	48
Applying R_A^{-1} and R_B^{-1}	$2(R_A^{-1} \text{ and } R_B^{-1}) \times 2(\text{complex}) \times 3(\text{colors}) \times 2(\text{half fermions}) \times (4(\text{multiply}) + 4(\text{add}))$	=	192
Total per site		=	1560

thus,

(Functions)+≡

```
int
qdwcg_QQOps(void)
{
    return even_odd.size * S_4 * 4 * 1560;
}
```

8 SSE

Here all vectors operations and related data types are collected.

All functions defined here are `inline` so that gcc optimizes corresponding function calls. Unfortunately, sometimes the optimization is not very good, hence, one must watch the generated code for results.

Let us start with the basic 4-vector of reals:

(Type definitions)+≡

```
typedef real vreal __attribute__((mode(V4SF),aligned(16)));
```

We use two constructors for packing floating point numbers into `vreal`:

(Function prototypes)+≡

```
static inline vreal vreal_mk4(real a0, real a1, real a2, real a3) {
    vreal v;
    real *r = (real *)&v;
    r[0] = a0;
    r[1] = a1;
    r[2] = a2;
    r[3] = a3;
    return v;
}
```

```
static inline vreal vreal_mk1(real a)
{
    vreal v = __builtin_ia32_loadss((float *)&a);
    asm("shufps\t$0,%0,%0" : "+x" (v));
    return v;
}
```

Next, not quite constructors, but useful operations modifying `vreal`s:

$\langle \text{Function prototypes} \rangle + \equiv$

```
static inline void vput_3(vreal *v, real a3)
{
    ((real *)v)[3] = a3;
}

static inline void vput_0(vreal *v, real a0)
{
    ((real *)v)[0] = a0;
}
```

It's convenient to have a cleaner for `VecHalfFermion`:

$\langle \text{Function prototypes} \rangle + \equiv$

```
static inline void vhfzero(VecHalfFermion *v)
{
    vreal z = vreal_mk1(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;
}
```

Computing the sum of the components of `vreal`:

$$s \leftarrow v_0 + v_1 + v_2 + v_3$$

$\langle \text{Function prototypes} \rangle + \equiv$

```
static inline real vsum(vreal v)
{
    real *vv = (real *)&v;
    return vv[0] + vv[1] + vv[2] + vv[3];
}
```

Given

$$\begin{aligned} a &= (a_0, a_1, a_2, a_3), \\ b &= (b_0, b_1, b_2, b_3), \end{aligned}$$

compute various shifts as follows:

$$\text{shift_up1} \leftarrow (a_1, a_2, a_3, b_0)$$

$\langle \text{Function prototypes} \rangle + \equiv$

```
static inline vreal shift_up1(vreal a, vreal b)
{
    vreal x = a;
    vreal y = b;
    asm("shufps\t$0x30,%0,%1\n\t"
        "shufps\t$0x29,%1,%0"
        : "+x" (x), "+x" (y));
    return x;
}
```

$$\text{shift_up2} \leftarrow (a_2, a_3, b_0, b_1)$$

```

<Function prototypes>+≡
static inline vreal shift_up2(vreal a, vreal b)
{
    vreal x = a;
    asm("shufps\t$0x4e,%1,%0"
        : "+x" (x): "x" (b));
    return x;
}

```

$$\text{shift_up3} \leftarrow (a_3, b_0, b_1, b_2)$$

```

<Function prototypes>+≡
static inline vreal shift_up3(vreal a, vreal b)
{
    vreal x = a;
    asm("shufps\t$0x03,%1,%0\n\t"
        "shufps\t$0x9c,%1,%0"
        : "+x" (x): "x" (b));
    return x;
}

```

$$\text{shift_down1} \leftarrow (a_3, b_0, b_1, b_2)$$

```

<Function prototypes>+≡
static inline vreal shift_down1(vreal a, vreal b)
{
    return shift_up3(a, b);
}

```

$$\text{shift_down2} \leftarrow (a_2, a_3, b_0, b_1)$$

```

<Function prototypes>+≡
static inline vreal shift_down2(vreal a, vreal b)
{
    return shift_up2(a, b);
}

```

$$\text{shift_down3} \leftarrow (a_1, a_2, a_3, b_0)$$

```

<Function prototypes>+≡
static inline vreal shift_down3(vreal a, vreal b)
{
    return shift_up1(a, b);
}

```

9 QCD DATATYPES

Complex vectors are simple:

```

<Type definitions>+≡
typedef struct {
    vreal re, im;
} vcomplex;

```

We also need a vector version of SU3:

```

<Type definitions>+≡
typedef struct {
    vcomplex v[3][3];
} VecSU3;

```

Full vector fermions:

```
<Type definitions>+≡
typedef struct {
    vcomplex f[4][3];
} VecFermion;
```

Projected vector fermions are next:

```
<Type definitions>+≡
typedef struct {
    vcomplex f[2][3];
} VecHalfFermion;
```

Type system helps us to keep track of sublattices:

```
<Type definitions>+≡
struct EvenFermion {
    VecFermion f;
};

struct OddFermion {
    VecFermion f;
};
```

10 DWFCG INITIALIZATION AND CLEANUP

All that remains is constructing index tables and arranging communication buffers as needed. For completeness, we also include a corresponding cleanup routine so that `dwcg` could be intergrated into a library.

Prototypes for the constructor and destructor come first:

```
<QCD interface functions>+≡
int qdwcg_init(int size[], int torus[], int pos[]);
void qdwcg_fini(void);
```

Here is the definition of `struct neighbor`

```
<Type definitions>+≡
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[8];    /* size of send buffers in 8 dirs */
    int          rcv_size[8];    /* size of receive buffers */
    int          *snd[8];        /* 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 */
    VecHalfFermion *snd_buf[8];  /* Send buffers */
    VecHalfFermion *rcv_buf[8];  /* Receive buffers */
};
```

For better spatial locality, we pack boundary index translation and outside neighbor mask together into `struct boundary`. It also simplifies allocation of the table.

```
<Type definitions>+≡
struct boundary {
    int index;    /* i->x translation */
    int mask;     /* only lower 8 bits are used */
};
```


For each site on sublattice, we compute the table of neighbor information. We need to know where to get eight gauge fields and eight neighbor fermions on the other sublattice. For the fermion, the index of the $s = 0$ neighbor element is stored for inside sites and for boundary sites with `mask & (1<<d) == 0` (e.g., inside neighbors of the boundary elements.) Otherwise, the index of $s = 0$ element in the corresponding receive buffer is stored. As a result, all stored fermion indices are multiples of `S_4`. For gauge fields, we can avoid storing three indices by the following observation. All forward links for a given site are always needed for computation, so we can store only the index of the first of them, `Uup`, and arrange the gauge fields in memory for that other three are `Uup+1`, `Uup+2`, and `Uup+3` respectively. For the backward links, some of them will come from truncated sites (where the originating site belongs to another node on the network.) In this case, we only need one gauge matrix on the node, which we store *after* all local gauge fields. Such an arrangement works for inside sites as well, because we index only one of four gauge links on the originating site.

<Type definitions>+≡

```
struct site {
    int Uup;          /* up-links are [[Uup]], [[Uup+1]], [[Uup+2]], [[Uup+3]] */
    int Udown[4];     /* four down-links */
    int F[8];         /* eight neighboring fermions on the other sublattice */
};
```

The DWF initializer creates all things necessary for communication and computation. Since we do not know the problem size until the runtime, memory is allocated here for index arrays.

<Functions>+≡

```
int
qdwcg_init(int lattice[], int network[], int coord[])
{
    struct neighbor tmp;
    struct bounds bounds;
    int i, v;

    init_neighbor(&bounds, &neighbor, lattice, network, coord);
    <Compute init sizes>
    tmp = neighbor;
    build_neighbor(&even_odd, &bounds, 0, &tmp, lattice, network, coord);
    build_neighbor(&odd_even, &bounds, 1, &tmp, lattice, network, coord);
    <Allocate and setup all send and receive buffers>

    return 0;
}
```

First, we set global data:

<Global data>+≡

```
/*
static int even_XYZTS_4;
static int odd_XYZTS_4;
*/
static int gauge_XYZT;
static int S_4, S_4_1;
```

<Compute init sizes>≡

```
S_4 = lattice[4] / 4;
S_4_1 = S_4 - 1;
for (v = 1, i = 0; i < 4; i++) {
    v *= bounds.hi[i] - bounds.lo[i];
}
gauge_XYZT = 4 * v;
for (i = 0; i < 4; i++) {
    if (network[i] < 2)
        continue;
    gauge_XYZT += v / (bounds.hi[i] - bounds.lo[i]);
}
```

The finilizer deallocates the allocated memory and sets pointers and sizes to zero for safety sake:

```

<Functions>+≡
void
qdwcg_fini(void)
{
/*
    int i;
*/

    <Deallocate send and receive buffers>

/*
    fini_neighbor(&neighbor);
*/
}

```

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

```

<Type definitions>+≡
struct bounds {
    int lo[4];
    int hi[4];
};

```

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

```

<Global data>+≡
static struct neighbor neighbor;
static struct neighbor odd_even;
static struct neighbor even_odd;

```

Let us start with computing the boundary of the sublattice

```

<Function prototypes>+≡
static 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 void
mk_sublattice(struct bounds *bounds,
              int lattice[],
              int network[],
              int coord[])
{
    int i;

    for (i = 0; i < 4; i++) {
        bounds->lo[i] = lattice_start(lattice[i], network[i], coord[i]);
        bounds->hi[i] = lattice_start(lattice[i], network[i], coord[i] + 1);
    }
}

```

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

```

<Function prototypes>+=
static void
init_neighbor(struct bounds *bounds,
              struct neighbor *neighbor,
              int lattice[],
              int network[],
              int coord[])
{
    int i;

    mk_sublattice(bounds, lattice, network, coord);
    <Compute inside_size and boundary_size>
    <Allocate inside table>
    <Allocate boundary table>
    <Compute send sizes and allocate index tables>
}

<Compute inside_size and boundary_size>=
for (neighbor->size = 1,
     neighbor->inside_size = 1,
     i = 0; i < 4; 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 = calloc(neighbor->size, sizeof (struct site));

<Allocate inside table>=
if (neighbor->inside_size)
    neighbor->inside = calloc(neighbor->inside_size, sizeof (int));
else
    neighbor->inside = 0;

<Allocate boundary table>=
if (neighbor->boundary_size)
    neighbor->boundary = calloc(neighbor->boundary_size, sizeof (struct boundary));
else
    neighbor->boundary = 0;

<Compute send sizes and allocate index tables>=
for (i = 0; i < 8; i++) {
    int d = i / 2;
    if (network[d] > 1) {
        neighbor->snd_size[i] = neighbor->size / (bounds->hi[d] - bounds->lo[d]);
        neighbor->snd[i] = calloc(neighbor->snd_size[i], sizeof (int));
    } else {
        neighbor->snd_size[i] = 0;
        neighbor->snd[i] = 0;
    }
}

```

The trickiest part is constructing the neighbor tables to take care of all possibilities.

Before going into it, however, 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.

(Function prototypes)+≡

```
static int
to_HFlinear(int p[],
            struct bounds *b,
            int lattice[],
            int q,
            int z)
{
    int x, d;
    for (x = 0, d = 4; d--;) {
        int v = p[d] + ((d == q)?z:0);
        int s = b->hi[d] - b->lo[d];
        int y = b->lo[d];
        if (v < 0)
            v += lattice[d];
        if (v >= lattice[d])
            v -= lattice[d];
        x = x * s + v - y;
    }
    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.

(Function prototypes)+≡

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

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

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

(Find index of a regular gauge link)≡

```
for (x = 0, d = 4; d--;) {
    int s = b->hi[d] - b->lo[d];
    int v = p[d] - ((q == d)?1:0);
    if (v < 0)
        v += lattice[d];
    x = x * s + v - b->lo[d];
}
return 4 * x + ((q < 0)?0:q);
```

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

⟨Find index of a borrowed gauge link⟩≡

```
int s0, v0;
for (d = 0, v0 = 1; d < 4; d++)
    v0 *= b->hi[d] - b->lo[d];
for (d = 0, s0 = 4 * v0; d < q; d++)
    s0 += v0 / (b->hi[d] - b->lo[d]);
for (d = 4, 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;
```

In addition to the above translations, we need a function that will walk through a boundary of a neighbor building the outside part of the `site[]`.F indices.

⟨Function prototypes⟩+≡

```
static void
construct_rec(struct neighbor *out,
              int parity,
              struct bounds *bounds,
              int lattice[],
              int network[],
              int coord[],
              int dir,
              int step)
{
    struct bounds xb;
    int xc[4], p[4];
    int s, d, x, k;
    int dz = dir * 2 + ((step>0)?1:0);

    ⟨Construct the neighbor's network coordinates xc and bounds xb⟩
    ⟨Construct the initial point of the hypersurface⟩
    ⟨Walk through the hypersurface⟩
}
```

Constructing the neighbor's network position is straightforward:

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

```
for (d = 0; d < 4; d++) {
    int v;

    if (d == dir) {
        v = coord[d] + step;
        if (v < 0)
            v += network[d];
        if (v >= network[d])
            v -= network[d];
    } else {
        v = coord[d];
    }
    xc[d] = v;
}
mk_sublattice(&xb, lattice, network, xc);
```

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

⟨Construct the initial point of the hypersurface⟩≡

```
for (d = 0; d < 4; d++) {
    if (d == dir) {
        if (step > 0)
            p[d] = xb.lo[d];
        else
            p[d] = xb.hi[d] - 1;
    } else {
        p[d] = xb.lo[d];
    }
}
```

Walking through the hypersurface is very much like walking through the sublattice below. There are only two differences: (a) we are walking opposite parity sublattice surface here and, (b) while advancing the point, we should stay on the surface selected above.

⟨Walk through the hypersurface⟩≡

```
k = 0;
do {
    for (d = 0, s = parity; d < 4; d++)
        s += p[d];
    if (!(s & 1))
        goto next;
```

⟨Translate p to target x⟩

⟨Insert k into site[x].F[dx]⟩

next:

```
for (d = 0; d < 4; d++) {
    if (d == dir)
        continue;
    if (++p[d] == xb.hi[d])
        p[d] = xb.lo[d];
    else
        break;
}
```

```
} while (d != 4);
```

```
out->rcv_size[dz^1] = k; /* XXX is it true? */
```

⟨Translate p to target x⟩≡

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

⟨Insert k into site[x].F[dx]⟩≡

```
out->site[x].F[dz] = S_4 * k++;
```

Now we can define `build_neighbor()`:

⟨Function prototypes⟩ +=

```
static void
build_neighbor(struct neighbor *out,
               struct bounds  *bounds,
               int             parity,
               struct neighbor *in,
               int             lattice[],
               int             network[],
               int             coord[])
{
    int d, s, x, m;
    int p[4];

    ⟨Initialize out and p⟩
    ⟨Walk through sublattice⟩
    ⟨Build outside indices⟩
}
```

First part is easy: we start with copying `in` to `out`, resetting fields which will be computed shortly and setting `p` to `bounds->lo`:

⟨Initialize out and p⟩ =

```
*out = *in;
out->size = 0;
out->inside_size = 0;
out->boundary_size = 0;
for (d = 0; d < 4; 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;
    p[d] = bounds->lo[d];
}
```

Walking through the sublattice is done without nested loops. See Knuth if it looks suspicious to you.

⟨Walk through sublattice⟩ =

```
do {
    for (d = 0, s = parity; d < 4; d++)
        s += p[d];
    if ((s & 1) != 0)
        goto next;
    ⟨Compute x and m⟩
    ⟨Setup boundary or inside⟩
    ⟨Build local neighbors⟩
    out->size++;
    in->size++;
next:
    for (d = 0; d < 4; d++) {
        if (++p[d] == bounds->hi[d])
            p[d] = bounds->lo[d];
        else
            break;
    }
} while (d != 4);
```

For x we use a function to compute it from p . 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*.

```

⟨Compute x and m⟩≡
  x = to_HFlinear(p, bounds, lattice, -1, 0);
  for (m = 0, d = 0; d < 4; d++) {
    if (network[d] > 1) {
      if (p[d] == bounds->lo[d])
        m |= 1 << (2 * d);
      if (p[d] + 1 == bounds->hi[d])
        m |= 1 << (2 * d + 1);
    }
  }

```

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

```

⟨Setup boundary or inside⟩≡
  if (m) {
    ⟨Setup boundary⟩
  } else {
    ⟨Setup inside⟩
  }

```

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

```

⟨Setup inside⟩≡
  *in->inside++ = x;
  out->inside_size++;

```

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

```

⟨Setup boundary⟩≡
  in->boundary->index = x;
  in->boundary->mask = m;
  in->boundary++;
  out->boundary_size++;
  for (d = 0; d < 8; d++) {
    if ((m & (1 << d)) == 0)
      continue;
    *in->snd[d]++ = x;
    out->snd_size[d]++;
  }

```

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.

```

⟨Build local neighbors⟩≡
  in->site->Up = to_Ulinear(p, bounds, lattice, network, -1);
  for (d = 0; d < 4; d++) {
    in->site->Udown[d] = to_Ulinear(p, bounds, lattice, network, d);
    if ((m & (1 << (2 * d))) == 0)
      in->site->F[2*d] = S_4 * to_HFlinear(p, bounds, lattice, d, -1);
    if ((m & (1 << (2 * d + 1))) == 0)
      in->site->F[2*d + 1] = S_4 * to_HFlinear(p, bounds, lattice, d, +1);
  }

```

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:

```

⟨Build outside indices⟩≡
  for (d = 0; d < 4; d++) {
    if (network[d] < 2)
      continue;
    construct_rec(out, parity, bounds, lattice, network, coord, d, +1);
    construct_rec(out, parity, bounds, lattice, network, coord, d, -1);
  }

```


10.1 Fake Send and Receive Buffers

In this version we only pretend to communicate, so each send buffer is the same as a corresponding receive.

\langle *Allocate and setup all send and receive buffers* $\rangle \equiv$

```
for (i = 0; i < 8; i++) {
    if (network[i/2] < 2)
        continue;
    even_odd.snd_buf[i] = even_odd.rcv_buf[i^1] = allocate(even_odd.snd_size[i] * S_4 * sizeof (VecHalfFermion))
    odd_even.snd_buf[i] = odd_even.rcv_buf[i^1] = allocate(odd_even.snd_size[i] * S_4 * sizeof (VecHalfFermion))
}
/* XXX Allocate and setup send and receive buffers */
```

11 THINGS TO DO

Things which will remain empty until real communication is implemented:

\langle *Start communication* $\rangle \equiv$

```
/* Start communication here */
```

\langle *Finish communication* $\rangle \equiv$

```
/* Wait for communication completion here */
```

\langle *Deallocate send and receive buffers* $\rangle \equiv$

```
/* XXX Deallocate send and receive buffers */
```