

DWF SSE Interface

Version 1.2.0

Andrew Pochinsky

January 20, 2005

Abstract

This is a definition of the interface to the SSE DWF CG solver to a Chroma-like environment. The CG engine requires gcc version 3.3.x or higher and must be compiled as C code to achieve good performance. The interface targets both C and C++ external environments with calling conventions compatible with the gcc compiler on x86.

This version extends the interface by exposing M and M^\dagger needed for HMC.

1 NOTATION

For the following it is convenient to introduce some notation and specify restriction that the inverter imposes on its input parameters and the environment.

We assume that the lattice is a 5-d torus with periodic boundary conditions in 4-directions and a domain wall in the fifth direction. Other boundary conditions in 4-directions may be implemented by appropriate modifications of the gauge field. Lattice sizes are $L_0 \times L_1 \times L_2 \times L_3 \times L_4$. The CG uses red-black preconditioning and, therefore, requires that $L_0 \dots L_3$ be even. Because of the way SSE instructions are used by the CG code, L_4 must be a multiple of 4.

We assume also that the cluster has logical geometry $N_0 \times N_1 \times N_2 \times N_3$ (some of N_i may be 1). The cluster network is a torus in all non-trivial extends, and we require $N_i \leq L_i$ for $i = 0, \dots, 3$. Otherwise there is no restrictions on N_i . (However, communications will be overlapped with computations only if $L_i/N_i \geq 3$ for all i . Nevertheless, the code will work correctly, albeit slowly, for smaller values of L_i/N_i .)

Before embarking upon memory layout details, let us introduce

$$\begin{aligned} a_{ij} &= \left\lfloor \frac{jL_i}{N_i} \right\rfloor, \\ b_{ij} &= \left\lfloor \frac{(j+1)L_i}{N_i} \right\rfloor = a_{ij+1}. \end{aligned}$$

Then a node with logical coordinates (n_0, n_1, n_2, n_3) hosts a sublattice with coordinates $(x_0, x_1, x_2, x_3, x_4)$, where $a_{in_i} \leq x_i < b_{in_i}$ for $i = 0, \dots, 3$ and $0 \leq x_4 < L_4$. It is required that at the time the interface functions are called all gauge and fermion fields needed by the CG are *local* on each node and the QMP layer has no outstanding communications.

2 QMP STATE ETC

It is required that before any of the solver interface functions is called, the message passing subsystem is set into a known state. In particular:

- `QMP_init_message_passing()` has successfully returned or its equivalent action had been performed.
- `QMP_declare_logical_topology()` was called with parameters corresponding to the outer layer lattice layout.
- All QMP messages had arrived and were handled.
- Memory subsystem is in such a state that it is possible to call functions of `malloc()` family on each node if `alloactor` is passed as `NULL` to `sse_dwf_init()`. Otherwise, it should be possible to allocate 128-bit aligned memory by calling `allocator(byte_count)` between the entry to `sse_dwf_init()` and return from `sse_dwf_fini()` in the program control flow.

In addition, the outer environment must provide a mechanism to call a given function on each node of the cluster without waiting for other nodes. In fact, all interface functions are expected to be called by such a mechanism.

3 INTERFACE

The CG interface consists of functions, opaque datatypes and call-back function types. To avoid requiring to reveal too much information about outer layer data types, the CG uses `void *` and `const void *` types for the outer layer lattice objects. In addition to the call-back functions passed as arguments, the CG uses QMP interface for internode communication.

The interface handles both single and double precision solvers. The accessor functions below are oblivious to the floating point length of the outer layer objects. The solver's behavior is controlled by `fp_size` argument of function `SSE_DWF_init()`. This allows one to switch precision with minimal changes to the upper level environment.

2a `<sse-dwf-cg.h 2a>`≡
 `#ifndef SSE_DWF_CG`
 `#define SSE_DWF_CG`
 <Start C binding region 2b>
 <Data types 3a>
 <Interface function prototypes 5a>
 <End C binding region 2c>
 `#endif`

Since the interface header file may be included from C++ source, we need to tell the compiler that external symbol have C bindings:

2b `<Start C binding region 2b>`≡
 `#if defined (__cplusplus)`
 `extern "C" {`
 `#endif`

2c `<End C binding region 2c>`≡
 `#if defined (__cplusplus)`
 `}`
 `#endif`

3.1 Opaque Types

Here are opaque datatypes used by the interface:

3a $\langle \text{Data types 3a} \rangle \equiv$
typedef struct SSE_DWF_Fermion SSE_DWF_Fermion;
typedef struct SSE_DWF_Odd_Fermion SSE_DWF_Odd_Fermion;
typedef struct SSE_DWF_Gauge SSE_DWF_Gauge;

Access to outer layer fields is done via accessor functions. Each of them takes a field to access (as `void *` for writers and `const void *` for readers), *global* lattice coordinates, component indices, and real/imaginary part selector. In addition, there is a `void *env` parameter that may be used to pass extra information to the accessor. This parameter is passed by the outer layer to export/import interface functions and is used by the CG only to give it to the call-back functions. Otherwise the CG completely ignores this argument—it does not try to read or write memory pointed to, the pointers are never stored in the internal structures etc..

3b $\langle \text{Data types 3a} \rangle + \equiv$
typedef double (*SSE_DWF_gauge_reader)(const void *OuterGauge,
void *env,
const int lattice_addr[4],
int dim,
int a, int b,
int re_im);

This is the type of access functions used by the CG to read gauge field components. The CG calls

`gauge_reader(U, env, x, dim, a, b, 0);`

to read $\Re U_{ab}(x)$. To access the imaginary part, `re_im` is set to 1. Arguments `a` and `b` vary from 0 to 2 inclusive. It is guaranteed that the CG will only pass the local sublattice coordinates in `lattice_addr[]`. Since this call-back is used only to setup the gauge field, the upper level environment is encouraged to do out-of-range checks on `lattice_addr` because it adds only small overhead while helping to catch data layout mismatch.

4a $\langle \text{Data types 3a} \rangle + \equiv$

```
typedef double (*SSE_DWF_fermion_reader)(const void *OuterFermion,
                                         void *env,
                                         const int lattice_addr[5],
                                         int color, int dirac,
                                         int re_im);
```

This is the type of access functions used to the CG to read input fermion field components. Argument `color` varies from 0 to 2 inclusive, argument `dirac` varies from 0 to 3. Argument `re_im` is 0 for the real part and 1 for the imaginary part. Notice that `lattice_addr` has five components.

4b $\langle \text{Data types 3a} \rangle + \equiv$

```
typedef void (*SSE_DWF_fermion_writer)(void *OuterFemrion,
                                       void *env,
                                       const int lattice_addr[5],
                                       int color, int dirac,
                                       int re_im,
                                       double value);
```

This is the type of writer functions used to convert back from the CGland to outer layer data format.

4c $\langle \text{Data types 3a} \rangle + \equiv$

```
typedef double (*SSE_DWF_odd_fermion_reader)(const void *OuterFermion,
                                             void *env,
                                             const int lattice_addr[5],
                                             int color, int dirac,
                                             int re_im);
```

This is the type of access functions used to the CG to read input odd half fermion field components. Argument `color` varies from 0 to 2 inclusive, argument `dirac` varies from 0 to 3. Argument `re_im` is 0 for the real part and 1 for the imaginary part. Notice that `lattice_addr` has five components and will only traverse the odd sublattice.

4d $\langle \text{Data types 3a} \rangle + \equiv$

```
typedef void (*SSE_DWF_odd_fermion_writer)(void *OuterFemrion,
                                           void *env,
                                           const int lattice_addr[5],
                                           int color, int dirac,
                                           int re_im,
                                           double value);
```

This is the type of writer functions used to convert back from the CGland to outer layer data format.

3.2 CG Initialization

The first function of the CG interface called by the upper level environment must be

5a \langle Interface function prototypes 5a $\rangle \equiv$

```
int
SSE_DWF_init(const int lattice[5],
             SSE_DWF_FP_SIZE fp_size,
             void *(*allocator)(size_t size),
             void (deallocater)(void *));
```

Here, `lattice` is size of the lattice (*not the local sublattice*), `allocator` is a pointer to the function the CG should use to allocate dynamic memory (if it is `NULL`, standard library's `malloc()` will be used.) Likewise, `deallocater` is a pointer to the function to free dynamic memory (if it is `NULL`, standard library's `free()` will be used.) These function pointers will be stored by `SSE_DWF_init()` in internal structures and may be called *after* after it returns.

This function does all initialization needed for the CG to run. Among other things, it allocates and initializes communication channels and constructs index tables needed for computing the Dirac operator. Argument `fp_size` is `SSE_DWF_FLOAT` for a single precision solver and `SSE_DWF_DOUBLE` if double precision should be used. `SSE_DWF_init()` will return 0 in success, otherwise a non-zero value is returned.

5b \langle Data types 3a $\rangle + \equiv$

```
typedef enum {
    SSE_DWF_FLOAT,
    SSE_DWF_DOUBLE
} SSE_DWF_FP_SIZE;
```

The upper level environment should complete all QMP communications before calling `SSE_DWF_init()`. This includes not only data arrays involved in the inverter, but all communications in the cluster. In addition, it is expected that QMP had been initialized as outlined above.

3.3 CG Cleanup

The very last CG function to be called by the upper level environment is

5c \langle Interface function prototypes 5a $\rangle + \equiv$

```
void
SSE_DWF_fini(void);
```

It deallocates all memory owned by the CG and returns QMP to a known state. Upon return from `SSE_DWF_fini()` all CG communication operations are finished and there is no QMP channels owned by the CG.

The upper level environment should wait until `SSE_DWF_fini()` returns on *all nodes of the cluster* before calling any QMP function.

3.4 Exporting Gauge Fields

The following function is used to convert outer layer gauge field into a format suitable for the CG. For simplification of the non-critical parts of the CG we require two gauge field parameters: assuming that $U[\mu]$ is the gauge field in the canonical form (link in the μ direction at each lattice site,) let $V[\mu]$ be its cyclic shift, namely $V[i] = \text{cshift}(U[i], i, \text{UP})$. In these conventions, the prototypes of the gauge field loaders are

```
6a <Interface function prototypes 5a>+≡
    SSE_DWF_Gauge *
    SSE_DWF_load_gauge(const void *OuterGauge_U,
                       const void *OuterGauge_V,
                       void *env,
                       SSE_DWF_gauge_reader reader);
```

While in the loader, `reader` will be called to access the outer layer data. On return, `NULL` indicates that the load operation failed. Otherwise, the returned value is suitable for `SSE_DWF_solve()`. Gauge fields loaded into the CG should be freed by calling the following function:

```
6b <Interface function prototypes 5a>+≡
    void
    SSE_DWF_delete_gauge(SSE_DWF_Gauge *);
```

3.5 Exporting Fermion Fields

For domain wall fermions, let us start with a function used to load the right hand side and the initial guess of the Dirac equation. One does the conversion by the following function:

```
6c <Interface function prototypes 5a>+≡
    SSE_DWF_Fermion *
    SSE_DWF_load_fermion(const void *OuterFermion,
                        void *env,
                        SSE_DWF_fermion_reader reader);
```

This function allocates and initializes 5-d fermion fields that are suitable as arguments for the solver proper. To allocate an uninitialized fermion field for the CG, one can use the following function:

```
6d <Interface function prototypes 5a>+≡
    SSE_DWF_Fermion *SSE_DWF_allocate_fermion(void);
```

Either allocated or loaded, CG's fermion fields should be freed after use to reclaim memory by calling

```
6e <Interface function prototypes 5a>+≡
    void
    SSE_DWF_delete_fermion(SSE_DWF_Fermion *);
```

3.6 Importing the Result

We also need a way to convert solutions of the domain wall Dirac equation to the upper level format. Here are functions to do that:

```
6f <Interface function prototypes 5a>+≡
    void
    SSE_DWF_save_fermion(void *OuterFermion,
                        void *env,
                        SSE_DWF_fermion_writer writer,
                        SSE_DWF_Fermion *CGfermion);
```

It will iterate through the local subvolume on each node and call `writer()` with appropriate arguments to convert data into the outer layer format.

3.7 Exporting Odd Half Fermion Fields

Very much like full fermions, here is the importer for odd half fermion fields:

```
7a <Interface function prototypes 5a>+≡
    SSE_DWF_Odd_Fermion *
    SSE_DWF_load_odd_fermion(const void *OuterFermion,
                             void *env,
                             SSE_DWF_odd_fermion_reader reader);
```

Sometimes we need to create an SSE odd half fermion without initialization.

```
7b <Interface function prototypes 5a>+≡
    SSE_DWF_Odd_Fermion *
    SSE_DWF_allocate_odd_fermion(void);
```

Either allocated or loaded, odd half fermion fields should be freed after use to reclaim memory by calling

```
7c <Interface function prototypes 5a>+≡
    void
    SSE_DWF_delete_odd_fermion(SSE_DWF_Odd_Fermion *);
```

3.8 Importing the Odd Half Fermion

```
7d <Interface function prototypes 5a>+≡
    void
    SSE_DWF_save_odd_fermion(void *OuterFermion,
                             void *env,
                             SSE_DWF_odd_fermion_writer writer,
                             SSE_DWF_Odd_Fermion *CGfermion);
```

It will iterate through the local subvolume on each node and call `writer()` with appropriate arguments to convert data into the outer layer format.

3.9 Solver Engine

The solver proper takes fields converted into CG's format and a few extra parameters:

8a \langle Interface function prototypes 5a $\rangle + \equiv$

```

int
SSE_DWF_cg_solver(SSE_DWF_Fermion *result,
                  double *out_eps, int *out_iter,
                  const SSE_DWF_Gauge *gauge,
                  double M_0, double m_f,
                  const SSE_DWF_Fermion *guess,
                  const SSE_DWF_Fermion *rhs,
                  double eps, int max_iter);

```

It returns 0 if it believes that a reasonable approximation to the solution was found and a non-zero value otherwise. Number of conjugate gradient iterations used is returned in `out_iter`, an estimate of the residue after the last iteration is returned in `out_eps`. It uses the operator and preconditioner described in `dwf.pdf`.

3.10 $M^\dagger M$ Solver Engine

It is also convenient to have exposed a solver for

$$M^\dagger M \psi = \eta$$

8b \langle Interface function prototypes 5a $\rangle + \equiv$

```

int
SSE_DWF_MxM_solver(SSE_DWF_Odd_Fermion *result,
                   double *out_eps, int *out_iter,
                   const SSE_DWF_Gauge *gauge,
                   double M_0, double m_f,
                   const SSE_DWF_Odd_Fermion *guess,
                   const SSE_DWF_Odd_Fermion *rhs,
                   double eps, int max_iter);

```

It returns 0 if it believes that a reasonable approximation to the solution was found and a non-zero value otherwise. Number of conjugate gradient iterations used is returned in `out_iter`, an estimate of the residue after the last iteration is returned in `out_eps`. It uses the operator and preconditioner described in `dwf.pdf`.

3.11 Dirac Operator

It is convenient to have the Dirac operator as a standalone function. Here we compute

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

8c \langle Interface function prototypes 5a $\rangle + \equiv$

```

void
SSE_DWF_Dirac_Operator(SSE_DWF_Fermion *chi,
                       const SSE_DWF_Gauge *gauge,
                       double M_0, double m_f,
                       const SSE_DWF_Fermion *psi);

```

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

8d \langle Interface function prototypes 5a $\rangle + \equiv$

```

void
SSE_DWF_Dirac_Operator_conjugate(SSE_DWF_Fermion *chi,
                                  const SSE_DWF_Gauge *gauge,
                                  double M_0, double m_f,
                                  const SSE_DWF_Fermion *psi);

```


3.12 Preconditioned Operator

HMC needs access to the preconditioned operator.

$$\chi \leftarrow M\psi.$$

9a \langle Interface function prototypes 5a $\rangle + \equiv$
void
SSE_DWF_M_Operator(SSE_DWF_Odd_Fermion *chi,
const SSE_DWF_Gauge *gauge,
double M_0, double m_f,
const SSE_DWF_Odd_Fermion *psi);

$$\chi \leftarrow M^\dagger \psi.$$

9b \langle Interface function prototypes 5a $\rangle + \equiv$
void
SSE_DWF_M_Operator_conjugate(SSE_DWF_Odd_Fermion *chi,
const SSE_DWF_Gauge *gauge,
double M_0, double m_f,
const SSE_DWF_Odd_Fermion *psi);

3.13 Little Helpers

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

9c \langle Interface function prototypes 5a $\rangle + \equiv$
void
SSE_DWF_Add_Fermion(SSE_DWF_Fermion *psi,
const SSE_DWF_Fermion *phi,
double a,
const SSE_DWF_Fermion *eta);

$$v \leftarrow \langle \bar{\psi} | \varphi \rangle$$

9d \langle Interface function prototypes 5a $\rangle + \equiv$
void
SSE_DWF_Fermion_Dot_Product(double *v_re,
double *v_im,
const SSE_DWF_Fermion *psi,
const SSE_DWF_Fermion *phi);

4 SAMPLE USAGE PSEUDOCODE

Here is a pseudo-code showing a possible use of the CG by the upper level environment. It is possible to use the CG interface in different ways, e.g., to solve many equations with the same gauge field without going through the full initialization dance. The changes needed to accomplish that should be obvious to the reader by now.

```
OuterSolver(U, eta, guess)
{
    OuterGauge V;
    OuterFermion solution;

    for (int i = 0; i < 4; i++)
        V[i] = cshift(U[i], i, UP);

    // Finalize all outer layer QMP operations
    SSE_DWF_init(lattice, SSE_DWF_FLOAT, NULL, NULL); // single precision
    SSE_DWF_Gauge *g = SSE_DWF_load_gauge(U, V,
                                           NULL,
                                           gauge_reader);
    SSE_DWF_Fermion *rhs = SSE_DWF_load_fermion(eta,
                                                NULL,
                                                fermion_reader);
    SSE_DWF_Fermion *x0 = SSE_DWF_load_fermion(guess,
                                                NULL,
                                                fermion_reader);
    SSE_DWF_Fermion *x = SSE_DWF_allocate_fermion();
    double out_epsilon;
    int out_iterations;

    SSE_DWF_gc_solver(x, &out_epsilon, &out_iterations,
                     g, M_0, m_f,
                     x0, rhs,
                     1e-14, 5000);
    SSE_DWF_save_fermion(solution,
                        NULL,
                        fermion_writer);
    SSE_DWF_delete_gauge(g);
    SSE_DWF_delete_fermion(rhs);
    SSE_DWF_delete_fermion(x0);
    SSE_DWF_delete_fermion(x);

    SSE_DWF_fini();

    return solution;
}
```

5 CHUNKS

⟨Data types [3a](#)⟩
⟨End C binding region [2c](#)⟩
⟨Interface function prototypes [5a](#)⟩
⟨Start C binding region [2b](#)⟩
⟨sse-dwf-cg.h [2a](#)⟩