Physics Environment

QCDSP Group

July 4, 2002

Contents

1 Introduction

2 Global Variables

Objects with the following names are declared external in the corresponding header files. These objects are defined outside main.

GJP Global Job Parameter is a class that contains all globally needed parameters such as lattice size, verbose level etc.
Header File: util/include/global_job_parameters.h
Source Directory: util/global_job_parameter

VRB VeRBose is a class that controls all verbose output.
Header File: util/include/verbose.h
Source Directory: util/verbose

ERR Error is a class that controls all error output. It prints a message and exits with a given exit value.
Header File: util/include/error.h
Source Directory: util/error

3 Coding Style

- There should be no explicit use of the `float` or `double` native types in the code. The `Float` typedef should be used instead, thus allowing compile-time precision control.
- One may use slightly more modern c++ techniques (template, standard template library), as long as this code is distinct from the Columbia code, and does not affect up the compilation on the qcdsp.

4 Internal Data Formats

4.1 Lattice: The Gauge Configuration

Each instance of the Lattice class contains a pointer to the guage configuration, `gauge_field`, which is a pointer to an array of Matrix objects. The value of this pointer can be accessed using the Lattice::GaugeField() method.

While the internal format of the guage configuration can be changed, the ”canonical” format will be described here. The array consists of 4*GPJ.VolNodesSites() Matrix objects, i.e. there four SU(3) matricies (0, 1, 2, 3 for Ux, Uy, Uz, Ut) for each site on that processor node. This is the “fastest-moving index”. The next is the count in the x-direction, then the y-direction, then z and finally t. This detail is hidden from the user via a simple method: Lattice::GsiteOffset(int* x). Given an integer array indicating the *local* position of a site (such that x[i] is the ith coordinate where i = 0,1,2,3 = x,y,z,t), this method returns a pointer to the first of the four matrix objects associated with that site.

Each Matrix object is a single SU(3) matrix, build of 2×3×3 Floats (i.e. two floats to one complex number, and three by three colors). [It is not clear what the internal structure of the SU(3) matrix is, but presumably the rows and colums of this matrix must map onto the rows and colums of the matricies read from other codes if binary reproducibility is to be ensured. - SHOULD CHECK THIS - ANJ]. [UPDATE - ADDING A TRANSPOSE FLAG]

5 Introduction

The aim of this document is to put forward a plan for the way in which currently available lattice QCD codes can be adapted to run on the forthcoming QCD-OC hardware. This will focus on the C++ QCD-SP code, and in particular on how this code may be made a portable as possible, so that a minimum of new code will be required for it to run on any future hardware platform.

The Core Aim:

To develop the codes required to produce good science using the QCD-OC machine.

The quality of the science that can be determined using the code is predominately determined by the functionality supplied by it. This is primary issue is outlined in §6 below. Having defined the functionality we require, we can then consider the best route by which to produce such a code. There is a large volume of trusted code already in existence, and so this secondary issue breaks down as follows. We evaluate the available codes, estimating the pro's and con's of each (see §7), and then put forward plans for ways in which these code may be developed to meet our aims (§**??**).

Overall Constraints

- The most important constraint is the project time-scale. The hardware may be available in as little as 12 months, and the software development plan should fit *within* this ETA.

6 Functionality Specification for the QCD-OC code

The follow specification is a breakdown of the *ideal* functionality that the new code could provide. A version of the code which contains all of the assorted bells & whistles listed below is probably unattainable on the project time-scale. However, all HIGH priority functionality should be included, and the code-developers will keep in mind the possible implementation of the lower-priority aims at a later date.

6.1 The Physical System

Description	Priority
Wilson gauge field	HIGH
Wilson-clover fermions	HIGH
Ginsburg-Wilson fermion action	Medium (long-term HIGH)

6.2 Evolution Mechanisms

Description	Priority
Hybrid MC.	HIGH
Tweakability of algorithm parameters.	HIGH
Output options?	?

6.3 Measurement Capabilities

Description	Priority
$\psi\psi$?	?

6.4 Constraints & Overall Priorities

Description	Priority
Usable within $\sim$12 months.	HIGH
Capable of using 5x5x5x5 sublattices, i.e. the odd local lattice size issue.	HIGH
Run-time machine partitioning in software	Medium
Portability and future proofing against future hardware.	Medium/low

7 Available Codes

7.1 Colombia/QCD-SP

Written in pre-ANSI C++. What do to about this? (what are Colombia going to do?) See §**??**. But review of this code in §8.

7.2 SZIN

This is a macro-based application code, and as such its treatment is split into two parts. Firstly, the definition of the language itself, and secondly the application code that has been built up using that language.

7.2.1 SZIN: The Language

[m4 macros? links?]

- In Brief: http://www.jlab.org/~edwards/szin_manual.html
- Manual: http://www.jlab.org/~edwards/macros_v2.ps

7.2.2 SZIN: The Application Code

7.3 UKQCD

[What fortran codes are availible, and are there any ways in which this code may be of use to use despite the (probable???) lack of a compiler.]

8 Overview of the Colombia/QCD-SP code

The core concept behing this implementation of a lattice QCD library is the unification of the way in which different gauge and fermion actions are treated, via `lattice` abstract class (§8.2). The algorithms to be applied to the chosen system are also abstracted in order to unify the interface by which algorithms are 'run' on the system (§8.4). These classes, along with simple mechanisms for specifing the simulation parameters, allow the user to construct LQCD applications using relatively little source code. A simple source file can be constructed as follows:

- Specify the global parameters, such as the size of the processor array (§8.1).
- Define the system by choosing the gauge and fermion types (§8.2).
- Choose an algorithm from the suite available (§8.4).
- Specify any parameters specific to the algorithm (§8.4.1).
- Run the algorithm for a given number of steps (§8.4).

The central aim of the following code breakdown is twofold:

1. To clarify the library interface, so that any shortcomings in the way in which the application interacts with the library are made clear.
2. To indicate any gaps between the desired code functionality and that supplied by this code, particularly where the gaps are due to QCD-SP specific parts of the code.

[???, What to say about sub-lattice size limitations???] [???, What to say about software partiioning issue???]

8.1 Global Parameters

These either specify some property of the simulation itself (§8.1.1 & §8.1.2), or control the textual output of the code (§8.1.3 & §8.1.4).

8.1.1 The GlobalJobParameter class and the DoArg & CommonArg structs

To control the most general parameters of the simulation, a specific instance of **GlobalJobParameter** class (called `GJP` and with global scope) *must* be created. The parameters contained in the `GJP` can be read via a set of methods, but these parameters should be set by using a single call and a **DoArg** struct.

The parameters held in an instance of **DoArg** are listed in full in table **??**. In brief, this allows the user to set:

- The parameters of the regular domain decomposition, i.e. the size of the processor grid and the size of the lattice on each processor.
- The boundary conditions (periodic or antiperiodic in each direction).
- The kind of initial configuration.
- The initial value of the random number generator seed.
- The number of colors.
- Various parameters for controlling the different gauge/fermion actions, such as the value of the gauge β.

The parameters held within a **DoArg** structure are then used to initialise the global job parameters held in `GJP`.

8.1.2 The CommonArg struct

This structure is used to control the output of the code, and an instance of it is used to initialise each of the algorithms (§8.4). It contains a unique simulation identifier and the name of the file to be used for simulation output. [???, The unique ID is not used in the test codes, it seems. Does this have something to do with the task list and job manager stuff???]

8.1.3 Verbosity: The Verbose class

As in the case of the **GlobalJobParameter** class, a specific instance of the **Verbose** class (called `VRB`) is used to control the diagnostic textual output of the code. This information includes subroutine call traces, memory allocation and deallocation and so on. It also allows the program to control the LED status. The level of diagnostic output is initially set via the **GlobalJobParameter** initialisation, but can be explicitly set using `VRB.Level`.

8.1.4 Error Reporting: The Error class

Again, a specific global instance of the **Error** class (called `ERR`) is used to control the error reporting mechanism. This is very similar to the **Verbose** class, but for serious problems (such as attempts to use `NULL` pointers, file I/O failures and so on) which require the code to cease execution.

8.2 The Lattice Class

This class forms the core of the QCDSP code. It holds the gauge-field configuration, and defines the interface for the gauge-field and fermion-field operations in such a way as to ensure that the interface remains the same no matter what specific action is used. It also defines the interface to the random number generator. [???, anything else??? Converting???] Note that the fermion field is actually stored elsewhere (usually in the **Alg** class, §8.4). A large number of classes are derived from this base class, for each fermion and gauge action. The actual class that is used for a simulation will be one of these derived classes, chosen by the dessired combination of actions.

8.2.1 Gauge Field Types

There are four possible gauge-field types (see table 8.2.1), each of which implements the gauge-operation interface specified by the **Lattice** class (see §8.2.2).

8.2.2 Gauge Operations Interface

[What functionality for acting upon the Lattices is supplied within Lattice.]

- **GclassType Gclass** (void)

  It returns the type of gauge class.

- void **GactionGradient** (**Matrix** &grad, int $*$x, int mu)

  Calculates the partial derivative of the gauge action w.r.t. the link U_mu(x). Typical implementation has this func called with **Matrix** *&grad = $*$mp0, so avoid using it.*

- void **GforceSite** (**Matrix** &force, int $*$x, int mu)

  It calculates the gauge force at site x and direction mu.

- void **EvolveMomGforce** (**Matrix** $*$mom, **Float** step_size)

  It evolves the canonical momentum mom by step_size using the pure gauge force.

- **Float GhamiltonNode** (void)

  Returns the value of the pure gauge Hamiltonian of the node sublattice.

Class	Description	Dependencies
Gnone	Acts as if there is no gauge action (i.e. $\beta = 0$)	???
Gwilson	Uses the standard Wilson single plaquette action.	???,The Wilson Library?
GimprRect	Uses the standard Wilson plaquette operator plus the second order rectangle operator. ???	
GpowerPlaq	This action is the same as the standard Wilson action with the irrelevant power plaquette term added to it. The full action is: $\sum_p [\beta * -Tr[U_p]/3 + (1 - Tr[U_p]/3/c)^k]$ with `c = GJP.PowerPlaqCutoff()` and `k = GJP.PowerPlaqExponent()`. This action supresses plaquettes with $1 - ReTr[U_p]/3 > c$ and threfore reduces lattice dislocations.	???
GpowerRect	Uses the standard Wilson plaquette operator plus the second order rectangle operator plus a power plaquette term plus a power rectangle term. The full action is: $(\sum_p [c_0 * \beta * -Tr[U_p]/3 + (1 - Tr[U_p]/3/c)^k] + \sum_r [c_1 * \beta * -Tr[U_r]/3 + (1 - Tr[U_r]/3/c)^k])$ with `c = GJP.PowerPlaqCutoff()`, `k = GJP.PowerPlaqExponent() c_0 = 1 - 8 * c_1`, `c_1 = GJP.C1()`. This action supresses plaquettes with $1 - ReTr[U_p]/3 > c$ and rectangles with $1 - ReTr[U_r]/3 > c$ and therefore reduces lattice dislocations.	???

Table 1: Gauge fields supported by the QCD-SP code.

[Plus some action specific functions.]

[Details of dependencies.vector_util, Matrix, Float, cbuf (this last one turns into nothing on a workstation, I think).]

8.2.3 Fermion Action Types

8.2.4 Fermion Operations Interface

[What functionality for acting upon the Lattices is supplied within Lattice.] [NB It contains an interface for a Ritz eigenvec/val solver... Which is implemented in eigen_wilson.C. This is relevant to the GW stuff]

- **FclassType Fclass** (void)

  It returns the type of fermion class.

- int **FsiteOffsetChkb** (const int $*$x) const

Class	**Description**	**Dependencies**
Fnone	Its functions do nothing and return values as if there is no fermion action or fermion fields. The number of spin components is zero	???
FstagTypes:	Staggered fermions:	
Fstag	Defines the **Lattice** functions for staggered fermions.	???
FwilsonTypes:	Wilson fermions:	
Fclover	Wilson-clover fermions.	???,The Wilson Library?
Fdwf	Wilson-domain-wall fermions.	???
Fwilson	Wilson fermions.	???

Table 2: Fermion actions supported by the QCD-SP code.

Sets the offsets for the fermion fields on a checkerboard. The fermion field storage order is not the canonical one but it is particular to the fermion type. This function is not relevant to fermion types that do not use even/odd checkerboarding. x[i] is the ith coordinate where i = {0,1,2,3} = {x,y,z,t}.

- int **FsiteOffset** (const int ∗x) const

  Sets the offsets for the fermion fields on a checkerboard. The fermion field storage order is the canonical one. X[I] is the ith coordinate where i = {0,1,2,3} = {x,y,z,t}.

- int **ExactFlavors** (void)

  Returns the number of exact flavors of the matrix that is inverted during a molecular dynamics evolution.

- int **SpinComponents** (void)

  Returns the number of spin components.

- int **FsiteSize** (void)

  Returns the number of fermion field components (including real/imaginary) on a site of the 4-D lattice.

- int **FchkbEvl** (void)

  0 -> If no checkerboard is used for the evolution or the CG that inverts the evolution matrix. 1 -> If the fermion fields in the evolution or the CG that inverts the evolution matrix are defined on a single checkerboard (half the lattice).

- int **FmatEvlInv** (**Vector** ∗f_out, **Vector** ∗f_in, **CgArg** ∗cg_arg, **Float** ∗true_res, **CnvFrmType** cnv_frm=CNV_FRM_YES)

It calculates f_out where A ∗ f_out = f_in and A is the preconditioned (if relevant) fermion matrix that appears in the HMC evolution (typically some preconditioned form of [Dirac^dag Dirac]). The inversion is done with the conjugate gradient. cg_arg is the structure that contains all the control parameters, f_in is the fermion field source vector, f_out should be set to be the initial guess and on return is the solution. f_in and f_out are defined on a checkerboard. If true_res !=0 the value of the true residual is returned in true_res. true_res = |src - MatPcDagMatPc ∗ sol| / |src| The function returns the total number of CG iterations.

- int **FmatEvlInv** (**Vector** ∗f_out, **Vector** ∗f_in, **CgArg** ∗cg_arg, **CnvFrmType** cnv_frm=CNV_-FRM_YES)

  Same as original but with true_res=0;.

- int **FmatInv** (**Vector** ∗f_out, **Vector** ∗f_in, **CgArg** ∗cg_arg, **Float** ∗true_res, **CnvFrmType** cnv_frm=CNV_FRM_YES, **PreserveType** prs_f_in=PRESERVE_YES)

  It calculates f_out where A ∗ f_out = f_in and A is the fermion matrix (Dirac operator). The inversion is done with the conjugate gradient. cg_arg is the structure that contains all the control parameters, f_in is the fermion field source vector, f_out should be set to be the initial guess and on return is the solution. f_in and f_out are defined on the whole lattice. If true_res !=0 the value of the true residual is returned in true_res. true_res = |src - MatPcDagMatPc ∗ sol| / |src| cnv_frm is used to specify if f_in should be converted from canonical to fermion order and f_out from fermion to canonical. prs_f_in is used to specify if the source f_in should be preserved or not. If not the memory usage is less by the size of one fermion vector or by the size of one checkerboard fermion vector (half a fermion vector). For staggered fermions f_in is preserved regardles of the value of prs_f_in. The function returns the total number of CG iterations.

- int **FmatInv** (**Vector** ∗f_out, **Vector** ∗f_in, **CgArg** ∗cg_arg, **CnvFrmType** cnv_frm=CNV_-FRM_YES, **PreserveType** prs_f_in=PRESERVE_YES)

  Same as original but with true_res=0;.

- int **FeigSolv** (**Vector** ∗∗f_eigenv, **Float** ∗lambda, **Float** chirality[ ], int valid_eig[ ], **Float** ∗∗hsum, **EigArg** ∗eig_arg, **CnvFrmType** cnv_frm=CNV_FRM_YES)

  It finds the eigenvectors and eigenvalues of A where A is the fermion matrix (Dirac operator). The solution uses Ritz minimization. eig_arg is the structure that contains all the control parameters, f_eigenv are the fermion field source vectors which should be defined initially, lambda are the eigenvalues returned on solution. f_eigenv is defined on the whole lattice. hsum are projected eigenvectors. The function returns the total number of Ritz iterations.

- void **SetPhi** (**Vector** ∗phi, **Vector** ∗frm1, **Vector** ∗frm2, **Float** mass)

  It sets the pseudofermion field phi from frm1, frm2.

- void **EvolveMomFforce** (**Matrix** ∗mom, **Vector** ∗frm, **Float** mass, **Float** step_size)

It evolves the canonical momentum mom by step_size using the fermion force.

- **Float FhamiltonNode** (**Vector** ∗phi, **Vector** ∗chi)

  The fermion Hamiltonian of the node sublattice. chi must be the solution of Cg with source phi.

- void **Fconvert** (**Vector** ∗f_field, **StrOrdType** to, **StrOrdType** from)

  Convert fermion field f_field from -> to.

- **Float BhamiltonNode** (**Vector** ∗boson, **Float** mass)

  The boson Hamiltonian of the node sublattice.

[Plus action specific functions.]

[Details of dependencies.]

8.2.5 Putting the fermion & gauge fields together

[The system is chosen from a list of all possible F and G combinations: GtypeFtype]

8.3 Dirac Operators

8.4 The Algorithms

8.4.1 Algorithm Parameters