# The MILC Code

version —4.0—

**Claude Bernard** (*Washington U.*) <cb@lump.wustl.edu>
**Tom Blum** (*Brookhaven Nat'l Lab*) <tblum@wind.phy.bnl.gov>
**Tom DeGrand** (*U. of Colorado*) <degrand@aurinko.colorado.edu>
**Carleton DeTar** (*U. of Utah*) <detar@mail.physics.utah.edu>
**Steve Gottlieb** (*Indiana U.*) <sg@denali.physics.indiana.edu>
**Urs Heller** (*SCRI*) <heller@scri.fsu.edu,>
**James Hetrick** (*U. of Arizona*) <hetrick@physics.arizona.edu>
**Craig McNeile** (*U. of Utah*) <mcneile@mail.physics.utah.edu>
**Kari Rummukainen** (*Indiana U.*) <kari@trek.physics.indiana.edu>
**Bob Sugar** (*U.C. Santa Barbara*) <sugar@sarek.physics.ucsb.edu>
**Doug Toussaint** (*U. of Arizona*) <doug@klingon.physics.arizona.edu>
**Matt Wingate** (*U. of Colorado*) <wingate@haggis.colorado.edu>

The MILC Code is a body of high performance research software for doing SU(3) and SU(2) lattice gauge theory on several different (MIMD) parallel computers in current use. In scalar mode, it runs on a variety of workstations making it extremely versatile for both production and exploratory applications. Currently supported code runs on:

- Most scalar machines ("vanilla" version)
- Intel Paragon
- Thinking Machines CM5
- Cray T3D and T3E
- PVM (version 3.2)
- MPI

This is a TEXinfo document; an HTML version is accessible at:

http://www.physics.arizona.edu/~hetrick/milc.html
http://physics.indiana.edu/~sg/milc.html

Last change: [hetrick:19960725.1630CET]]

# 1 Obtaining the MILC Code

The most up-to-date information and access to the MILC Code can be found

- via WWW at:

  `http://physics.indiana.edu/~sg/milc.html`

  `http://www.physics.arizona.edu/~hetrick/milc`
- via email request to the authors at:

  `doug@klingon.physics.arizona.edu`

## 1.1 Usage conditions

The MILC Code is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation.

**Publications of research work done using this code or derivatives of this code should acknowledge its use.** The MILC project is supported in part by grants for the US Department of Energy and National Science Foundation and we ask that you use (at least) the following string in publications which derive results using this material:

*This work was in part based on the MILC collaboration's public lattice gauge theory code. See* **http://physics.indiana.edu/~sg/milc.html**

This software is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. See the GNU General Public License for more details, a copy of which License can be obtained from

```
Free Software Foundation, Inc.,
675 Mass Ave, Cambridge, MA 02139, USA.
```

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

# 2 General description

**The MILC Code** is a set of codes developed by the MIMD Lattice Computation (MILC) collaboration for doing simulations of four dimensional SU(3) lattice gauge theory on MIMD parallel machines. The latest version of this code includes libraries and routines for SU(2) gauge theory as well, but is not yet fully supported. Note that the MILC Code is publicly available for research purposes. Publications of work done using this code or derivatives of this code should acknowledge this use. Section 1.1 [Usage conditions], page 1

MORE...

## 2.1 Portability

One of our aims in writing this code was to make it very portable across machine architectures and configurations. While the code must be compiled with the architecture specific low level communication files (see Section 5.2 [Building the MILC Code], page 34), the application codes contain a minimum of **#ifdef**'s which mostly are for machine dependent performance optimizations in the conjugate gradient routines, etc.

Similarly, with regard to random numbers, care has been taken to ensure convenience and reproducablility. With **SITERAND** set (see Section 4.10 [Random numbers], page 28), the random number generator will produce the same sequence for a given seed, across architectures and varying the number of nodes.

## 2.2 Supported architectures

This manual covers **version 4.0** which is currently supposed to run on:

- Most scalar machines (the "vanilla" version for workstations)
- The Intel Paragon
- Thinking Machines' CM5
- The Cray T3D and T3E
- PVM (version 3.2)
- MPI

In addition it has run in the past, and may still work, on

- The Intel iPSC-860
- The Ncube 2

If you have a machine which is not listed above, or would like specific information about a particular workstation, write to `doug@klingon.physics.arizona.edu`. It is quite possible that code for your machine is in developement, or that certain routines for your workstation have been optimized.

Since this is our working code, it is continually in a state of development. We informally support the code as best we can by answering questions and fixing bugs. We will be very grateful for reports of problems and suggestions for improvements, which may be sent to

`doug@klingon.physics.arizona.edu`

## 2.3 Directory layout

Each **application**, or major project of a research program, will have its own directory. Applications are things like **ks_dynamical** (simulations with dynamical Kogut-susskind quarks), **wilson_invert** (Wilson Dirac operator inversion and spectroscopy), etc. All applications share the **libsu3** and **libsu2** directories containing low-level stuff, and the **generic_su3** and **generic_su2** directory containing high level stuff that is more or less independent of the physics. Examples of **generic_suN** code are the random number routines, the layout routines, routines to evaluate the plaquette or Polyakov loop, etc. The **su3** or **su2** codes are independent so that one can run SU(3) simulations without building the **libsu2** and **generic_su2** directories. At the moment both **libsu3** and **libsu2** contain a copy of the files for the library '`complex.a`' which really only needs to be compiled in one of the directories, so long as the include path to it is correctly specified in '`Makefiles`'.

The MILC Code should unpack into at least the following directories. (see Section 5.2 [Building the MILC Code], page 34)

*SUPPORT ROUTINES*

**libsuN:** Low level routines and include files (N=2,3).

- complex.h

  Header file of definitions and macros for complex numbers.

- complex.a

  Library of routines for complex numbers.

- suN.h

  Header file of definitions and macros for SU(N) matrix and fermion operations.

- suN.a

  Library of routines for SU(N) operations.

**generic_suN:**

High level code for generic SU(3) simulation. The other directories, which are for real applications, should use the routines in this directory where possible, otherwise copy routines from this directory and modify them or write new routines.

*APPLICATIONS*

**ks_dynamical:**

Simulations with dynamical Kogut-Susskind fermions. Variants include the "R", "phi" and hybrid Monte Carlo updating algorithms. Measurements of the plaquette, Polyakov loop, $< \psi\bar{\psi} >$ , and fermions energy and pressure are included. Optional measurements include the hadron spectrum, screening spectrum, and some wave functions (FFT routines are included in wave function codes)

**wilson_dynamical:**

Simulations with dynamical Wilson fermions. Variants include the "R", "phi" and hybrid Monte Carlo updating algorithms. Measurements of the plaquette, Polyakov loop, $< \psi\bar{\psi} >$ ,and fermion energy and pressure. Optional measurements include hadron spectrum, screening spectrum, axial current quark mass, and Landau gauge quark propagators.

**pure_gauge:**

Simulation of the pure gauge theory

**wilson_invert:**

Inversion of Wilson fermion matrix (conjugate gradient, MR, and BiCG algorithms) and measurements with Wilson quarks. Lattices are supposed to be generated by someone else. Not well maintained.

**ks_invert:** Inversion of Kogut-Susskind fermion matrix and measurements with Kogut-Susskind quarks. Lattices are supposed to be generated by someone else. Not well maintained.

**heavy_quarks:**

Heavy—light quarkonium, Weak matrix element physics. Under development (email: `cb@lump.wustl.edu`)

**wilson_hybrids:**

Spectrum of exotic mesons. (email: `doug@klingon.physics.arizona.edu`)

**perfect_actions**

Lattice nirvana. (email: `degrand@aurinko.colorado.edu`)

Each directory contains a '`README`' file with specific information on how to `make` that directory (see Section 5.2 [Building the MILC Code], page 34).

Generally, to make the code for a particular machine, use the makefile '`Make_MACHINE`'. The biggest machine dependency is that each machine uses the file '`com_MACHINE.c`' in the **generic_suN** directory. Hopefully, all of the communication calls specific to your machine are in this file. '`MACHINE`' is something like *paragon*, *cm5*, *vanilla*, etc.

# 3 Overview of applications

NON-EXISTENT.

## 3.1 Setup and initialization

# 4 Programming with MILC Code

These notes document some of the features of the MILC QCD code. They are intended to help people understand the basic philosophy and structure of the code, and to outline how one would modify existing applications for a new project.

## 4.1 Header files

Various header files define structures, macros, and global variables. They are, at the moment:

'`complex.h`'
: code for complex numbers (see Section 4.8.4 [Library routines], page 26).

'`su3.h`'
: routines for SU(3) operations, eg. matrix multiply (see Section 4.8.4 [Library routines], page 26).

'`su2.h`'
: routines for SU(2) operations, eg. matrix multiply (see Section 4.8.4 [Library routines], page 26).

'`globaldefs.h`'
: in SU(2) code, things that are really common to SU(2) and SU(3)

'`comdefs.h`'
: macros and defines for communication routines

'`lattice.h`'
: defines lattice fields and global variables

This last file is rather special being where the physical structure of the lattice is defined. While other header files are generally stored in the **libsuN** directories and not changed, a '`lattice.h`' file is stored in the application directory (see Section 2.3 [Directory layout], page 3), since it is modified to contain the fields specific to a particular application.

Some of these include files depend on previous ones, so the order of inclusion matters. Each header file should have a wrapper

```
#ifndef _HEADERFILE_H
#define _HEADERFILE_H
 ...body
#endif
```

around the body of the file so that one just inserts the following `#include` lines:

```
#include <stdio.h>
#include <math.h>
#include "complex.h"
#include "su3.h"
#include "lattice.h"
#include "comdefs.h"
```

in every file. The wrapper above short circuits re-reading a header which is already known to the compiler.

The external variables are defined using a macro `EXTERN` which is usually just "extern", but when `CONTROL` is defined, is null. The effect is to reserve storage in whichever file `CONTROL` is defined, so that exactly one file, typically the *main()* program, should contain a `#define CONTROL` before all the other includes (C++ would fix this nonsense).

## 4.2 Global variables

There are a number of global variables available. Most of these are defined in '`lattice.h`'. Unless specified, these variables are initialized in the function `initial_set()`.

`int this_node`
: number of this node

`int number_of_nodes`
: number of nodes in use

`int sites_on_node`
: number of sites on this node. [*This variable is set in:* `setup_layout()`]

`int even_sites_on_node`
: number of evensites on this node. [*This variable is set in:* `setup_layout()`]

`int odd_sites_on_node`
: number of odd sites on this node. [*This variable is set in:* `setup_layout()`]

`int nx,ny,nz,nt`
: lattice dimensions

`int volume`
: volume = nx * ny * nz * nt

`int iseed` random number seed

There are other variables which are not fundamental to the layout of the lattice but vary from application to application. These dynamical variables are part of a `params` struct which is passed between nodes by `initial_set()` in '`setup.c`' (see Section 3.1 [Setup and initialization], page 6). For example, a pure gauge simulation might have a `params` struct like this:

```
/* structure for passing simulation parameters to each node */
typedef struct {
        int nx,ny,nz,nt;  /* lattice dimensions */
        int iseed;      /* for random numbers */
        int warms;      /* the number of warmup trajectories */
        int trajecs;    /* the number of real trajectories */
        int steps;      /* number of steps for updating */
        int stepsQ;     /* number of steps for quasi-heatbath */
        int propinterval;  /* number of trajectories between measurements */
        int startflag;  /* what to do for beginning lattice */
        int fixflag;    /* whether to gauge fix */
        int saveflag;   /* what to do with lattice at end */
        float beta;     /* gauge coupling */
        float epsilon;  /* time step */
        char startfile[80],savefile[80];
} params;
```

These run-time variables are usually loaded by a loop over `readin()` defined in '`setup.c`'.

## 4.3 Lattice storage

The fields on the lattice are in structures of type `site`. This structure is defined in '`lattice.h`' (see Section 4.1 [Header files], page 7). Each *node* of the parallel machine has an array of such structures called `lattice`, with as many elements as there are sites on the *node*. In scalar mode there is only one *node*. The `site` structure looks like this:

```
struct site {
    /* The first part is standard to all programs */
        /* coordinates of this site */
        short x,y,z,t;
        /* is it even or odd? */
        char parity;
        /* my index in the lattice array */
        int index;

    /* Now come the physical fields, application dependent. We will
       add or delete whatever we need. This is just an example. */
        /* gauge field */
        su3_matrix xlink, ylink, zlink, tlink;
```

```
          /* antihermitian momentum matrices in each direction */
           anti_hermitmat xmom, ymom, zmom, tmom;

           su3_vector phi; /* Gaussian random source vector */
};
typedef struct site site;
```

Thus, to refer to the `phi` field on a particular lattice site, site "`i`" on this node, you say

```
lattice[i].phi,
```

and for the real part of color 0

```
lattice[i].phi.c[0].real,
```

etc. See (see Section 4.9 [Distributing sites among nodes], page 26) for how to figure out the index `i`. (Actually you usually won't need it.)

In general, there is a pointer to a site around, and then you would refer to the field as:

```
site *s; ...
/* s gets set to &(lattice[i]) */
s->phi
```

The coordinate, parity and index fields are used by the gather routines and other utility routines, so it is probably a bad idea to mess with them unless you want to change a lot of things. Other things can be added or deleted with abandon.

The routine `make_lattice()` is called from `setup()` to allocate the lattice on each node. This routine currently is in the file '`setup.c`' (see Section 3.1 [Setup and initialization], page 6).

In addition to the fields in the `site` structure, there are two sets of vectors whose elements correspond to lattice sites. These are the eight vectors of integers:

```
int *neighbor[MAX_GATHERS]
```

`neighbor[XDOWN][i]` is the index of the site in the **XDOWN** direction (see Section 4.4 [Moving around the lattice], page 11) from the i'th site on the node, if that site is on the same node. If

the neighboring site is on another node, this pointer will be **NOT_HERE (= -1)**. These vectors are mostly used by the gather routines, so application code usually doesn't have to worry about them.

There are a number of important vectors of pointers used for accessing fields at other (usually neighboring) neighboring sites,

```
char **gen_pt[MAX_GATHERS]
```

These vectors of pointers are declared in '`lattice.h`' and allocated in `make_lattice()`. They are filled by the gather routines, `start_gather()` and `start_general_gather()`, with pointers to the gathered field. See Section 4.5 [Accessing fields at other sites], page 14. You use one of these pointer vectors for each simultaneous gather you have going.

*Comments:*

- This storage scheme seems to allow the easiest coding, and likely the fastest performance. It certainly makes gathers about as easy as possible. However, it is somewhat wasteful of memory, since all fields are statically allocated. (You can use unions if two fields are needed in mutually exclusive parts of the program.) Also, there is no mechanism for defining a field on only even or odd sites.
- The "*site major*" ordering of variables in memory probably means that variables will fairly often be in cache. The contrasting "*field major*" ordering ( `su3_matrix`, `xlink[volume]`, `ylink[volume]`...) is more suitable for vectorizing in the traditional sense.
- The **EVENFIRST** option causes all the even sites to be stored first in the array, followed by all the odd sites. This makes looping over sites of a given parity more efficient. At some point, this will quit being an option and become a requirement for all layout schemes. At the moment it is useful for debugging to be able to turn this on and off.

## 4.4 Moving around the lattice

Various definitions, macros and routines exist for dealing with the lattice fields. So far, the only macros which are really necessary are **F_OFFSET**(`field_offset`) and **F_PT**(`field_pointer`). The definitions and macros (to be defined in '`globaldefs.h`' are:

```
/* Directions, and a macro to give the opposite direction */
/* These must go from 0 to 7 because they will be used to index an
   array. */
/* Also define NDIRS = number of directions */
```

```
#define XUP 0
#define YUP 1
#define ZUP 2
#define TUP 3
#define TDOWN 4
#define ZDOWN 5
#define YDOWN 6
#define XDOWN 7
#define OPP_DIR(dir) (7-(dir))  /* Opposite direction */
                                /* for example, OPP_DIR(XUP) is XDOWN */
/* number of directions */
#define NDIRS 8
```

The parity of a site is **EVEN** or **ODD**, where **EVEN** means (x+y+z+t)%2=0. Lots of routines take **EVEN**, **ODD** or **EVENANDODD** as an argument. Specifically:

```
#define EVEN 0x02
#define ODD 0x01
#define EVENANDODD 0x03
```

Often we want to use the name of a field as an argument to a routine, as in `dslash(chi,phi)`. There is a macro to convert the name of a field into an integer, and another one to convert this integer back into an address at a given site. A type `field_offset`, which is secretly an integer, is defined to help make the programs clearer.

**F_OFFSET**(`fieldname`) gives the offset in the site structure of the named field. **F_PT**(`*site`, `field_offset`) gives the address of the field whose offset is `field_offset` at the site `*site`. An example is certainly in order:

```
main() {
  copyfield( F_OFFSET(phi), F_OFFSET(chi) );
  /* "phi" and "chi" are names of su3_vector's in site. */
}

/* Copy an su3_vector field over the whole lattice */
copyfield(field_offset off1, field_offset off2) {
  register int i;
  register site *s;
  register su3_vector *v1,*v2;

  for(i=0;i<nsites_on_node;i++) { /* loop over sites on node */
     s = &(lattice[i]); /* pointer to current site */
     v1 = (su3_vector *)F_PT( s, off1); /* address of first vector */
     v2 = (su3_vector *)F_PT( s, off2);
     *v2 = *v1; /* copy the vector at this site */
  }
```

```
}
```

*Comments:*

- It will generally be good form to typecast the result of the **F_PT** macro to the appropriate pointer type. It naturally produces a character pointer. The code for copyfield could be much shorter at the expense of clarity. Here we use a macro to be defined below.

```
copyfield(field_offset off1, field_offset off2) {
  register int i;
  register site *s;
  FORALLSITES(i,s) {
    *(su3_vector *)F_PT(s,off1) = *(su3_vector *)F_PT(s,off2);
  }
}
```

- The following macros are not necessary, but are very useful. You may use them or ignore them as you see fit. Loops over sites are so common that we have defined macros for them (*These are soon to be in* '`globaldefs.h`'). These macros use an integer and a site pointer, which are available inside the loop. The site pointer is especially useful for accessing fields at the site.

```
/* macros to loop over sites of a given parity, or all sites on a node.
   Usage:
        int i;
        site *s;
        FOREVENSITES(i,s) {
            Commands go here, where s is a pointer to the current
            site and i is the index of the site on the node.
            For example, the phi vector at this site is "s->phi".
        } */
#define FOREVENSITES(i,s) \
    for(i=0,s=lattice;i<nsites_on_node;i++,s++)if(s->parity==EVEN)
#define FORODDSITES(i,s) \
    for(i=0,s=lattice;i<nsites_on_node;i++,s++)if(s->parity==ODD)
#define FORALLSITES(i,s) \
    for(i=0,s=lattice;i<nsites_on_node;i++,s++)
#define FORSOMEPARITY(i,s,parity) \
    for(i=0,s=lattice;i<nsites_on_node;i++,s++) \
       if(s->parity & (parity) != 0)
```

The first three of these macros loop over even, odd or all sites on the node, setting a pointer to the site and the index in the array. The index and pointer are available for use by the commands inside the braces. The last macro takes an additional argument which should be one of **EVEN**, **ODD** or **EVENANDODD**, and loops over sites of the selected parity. The actual definitions are not quite those written above if the **EVENFIRST** option is turned on, but they are logically equivalent.

## 4.5 Accessing fields at other sites

At present, each node only writes fields on its own sites. To read fields at other sites, there are gather routines. These are portable in the sense that they will look the same on all the machines on which this code runs, although what is inside them is quite different. All of these routines return pointers to fields. If the fields are on the same node, these are just pointers into the lattice, and if the fields are on sites on another node some message passing takes place. Because the communcation routines may have to allocate buffers for data, it is necessary to free the buffers by calling the appropriate cleanup routine when you are finished with the data. These routines are in '`com_XXXXX.c`', where `XXXXX` identifies the machine.

The gather routines provide a hopefully optimized way to gather a field from the neighboring sites. There is another type defined, `msg_tag`, which remembers the information needed from one gather routine to the next. The destination of the gather is one of the vectors of pointers, `gen_pt[0]`, etc. (see Section 4.3 [Lattice storage], page 9). On each site, or on each site of the selected parity, this pointer will be set to the address of the desired field on the neighboring site, or a copy thereof if the site lives on a different node.

These routines use asynchronous sends and receives when possible, so it is possible to start one or more gathers going, and do something else while awaiting the data. If you are doing more than one gather at a time, just use different `*msg_tags` for each one to keep them straight.

To set up the data structures required by the gather routines, `make_nn_gathers()` is called in the setup part of the program. This must be done *after* the call to `make_lattice()`.

```
/* "start_gather()" starts asynchronous sends and receives required
    to gather neighbors. */

msg_tag * start_gather(field,size,direction,parity,dest);
  /* arguments */
  field_offset field;  /* which field? Some member of structure "site" */
  int size;            /* size in bytes of the field
                          (eg. sizeof(su3_vector))*/
  int direction;       /* direction to gather from. eg XUP */
  int parity;          /* parity of sites whose neighbors we gather.
                          one of EVEN, ODD or EVENANDODD. */
  char * dest;         /* one of the vectors of pointers */


/* "wait_gather()" waits for receives to finish, insuring that the
   data has actually arrived.  The argument is the (msg_tag *) returned
   by start_gather. */
```

```
void wait_gather(msg_tag *mbuf);


/* "cleanup_gather()" frees all the buffers that were allocated, WHICH
    MEANS THAT THE GATHERED DATA MAY SOON DISAPPEAR. */

void cleanup_gather(msg_tag *mbuf);
```

Nearest neighbor gathers are done as follows. In the first example gathers `phi` at all even sites from the neighbors in the **XUP** direction. (*Gathering at* **EVEN** *sites means that* `phi` *at odd sites will be made available for computations "at* **EVEN** *sites.*)

```
msg_tag *tag;
site *s;
int i;

tag = start_gather( F_OFFSET(phi), sizeof(su3_vector), XUP,
                    EVEN, gen_pt[0] );

/* do other stuff */

wait_gather(tag);
/* *(su3_vector *)gen_pt[0][i] now contains the address of the
    phi vector (or a copy therof) on the neighbor of site i in the
    XUP direction for all even sites i.  (The type cast
    "(su3_vector *)" is usually not necessary.) */

FOREVENSITES(i,s) {
/* do whatever you want with it here.
   (su3_vector *)(gen_pt[0][i]) is a pointer to phi on
   the neighbor site. */
}

cleanup_gather(tag);
/* subsequent calls will overwrite the gathered fields. but if you
don't clean up, you will eventually run out of space */
```

This second example gathers `phi` from two directions at once:

```
msg_tag *tag0,*tag1;
tag0 = start_gather( F_OFFSET(phi), sizeof(su3_vector), XUP,
                     EVENANDODD, gen_pt[0] );
tag1 = start_gather( F_OFFSET(phi), sizeof(su3_vector), YUP,
                     EVENANDODD, gen_pt[1] );

/** do other stuff **/
```

```
wait_gather(tag0);
/* you may now use *(su3_vector *)gen_pt[0][i], the
   neighbors in the XUP direction. */

wait_gather(tag1);
/* you may now use *(su3_vector *)gen_pt[1][i], the
   neighbors in the YUP direction. */

cleanup_gather(tag0);
cleanup_gather(tag1);
```

Of course, you can also simultaneously gather different fields, or gather one field to even sites and another field to odd sites. Just be sure to keep your `msg_tag` pointers straight. The internal workings of these routines are far too horrible to discuss here. Consult the source code and comments in '`com_XXXXX.c`' if you must.

There is another type of gather for getting a field from an arbitrary displacement. It works like the gather described above except that instead of specifying the direction you specify a four component array of integers which is the relative displacement of the field to be fetched. This mechanism is much slower than gathering from neighbors, but far faster than `field_pointers()` (see *Comments* below). Thus if you plan to do a particular gather frequently, use `make_gather()` to define it.

Also, there can only be one `general_gather` at a time working. Thus if you need to do two `general_gathers` you must wait for the first gather before starting the second. (It is not necessary to cleanup the first gather before starting the second.)

Chaos will ensue if you use `wait_gather()` with a message tag returned by `start_general_gather()`, or vice-versa. `start_general_gather()` has the following format:

```
/* "start_general_gather()" starts asynchronous sends and receives
    required to gather neighbors. */
msg_tag * start_general_gather(field,size,displacement,parity,dest)
  /* arguments */
  field_offset field; /* which field? Some member of structure site */
  int size;           /* size in bytes of the field
                         (eg. sizeof(su3_vector))*/
  int *displacement;  /* displacement to gather from,
                         a four component array */
  int parity;         /* parity of sites whose neighbors we gather.
                         one of EVEN, ODD or EVENANDODD. */
  char ** dest;       /* one of the vectors of pointers */
```

```
/* "wait_general_gather()" waits for receives to finish, insuring that
    the data has actually arrived.  The argument is the (msg_tag *)
    returned by start_general_gather. */
void wait_general_gather(msg_tag *mbuf);

/* "cleanup_general_gather()" frees all the buffers that were
    allocated, WHICH MEANS THAT THE GATHERED DATA MAY SOON
   DISAPPEAR.  */
void cleanup_general_gather(msg_tag *mbuf);
```

This example gathers `phi` from a site displaced by +1 in the x direction and -1 in the y direction.

```
msg_tag *tag;
site *s;
int i, disp[4];

disp[XUP] = +1; disp[YUP] = -1; disp[ZUP] = disp[TUP] = 0;

tag = start_general_gather( F_OFFSET(phi), sizeof(su3_vector), disp,
                            EVEN, gen_pt[0] ); /* do other stuff */

wait_general_gather(tag);
/* gen_pt[0][i] now contains the address of the phi
   vector (or a copy therof) on the site displaced from site i
   by the vector "disp" for all even sites i. */

FOREVENSITES(i,s) {
/* do whatever you want with it here.
   (su3_vector *)(gen_pt[0][i]) is a pointer to phi on
   the other site. */
}

cleanup_general_gather(tag);
```

*Comments:*

- The code was originally designed with functions to provide a general way of accessing fields at any site. However, they require the ability for one node to interupt another. So far as we know, this works reliably on the iPSC-860, unreliably on the Intel Paragon, and not at all on most other parallel machines. So we actually don't use the following routines. However, for historical purposes, and in anticipation of better hardware in the future, we include the following notes on arbitrary data access across the nodes.

  To set up the interrupt handlers required by the `field_pointer` routines, call `start_handlers()` in the setup part of the program.

  ```
  /* "field_pointer_at_coordinates()" returns a pointer to a field in
      the lattice given its coordinates. */
  ```

```
char * field_pointer_at_coordinates( field, size, x,y,z,t );
  /* arguments */
  field_offset field; /* offset of one of the fields in lattice[] */
  int size;           /* size of the field in bytes */
  int x,y,z,t;        /* coordinates of point to get field from */

/* "field_pointer_at_direction()" returns a pointer to a field in the
    lattice at a direction from a given site. */ char *
field_pointer_at_direction( field,size, s, direction );
  /* arguments */
  int field;         /* offset of one of the fields in lattice[] */
  int size;          /* size of the field in bytes */
  site *s;           /* pointer to a site on this node */
  int direction;     /* direction of site's neighbor to get data from. */

/* "cleanup_field_pointer()" frees the buffers that field_pointer...
    allocated. */
void cleanup_field_pointer(char *buf);
```

An example of the usage of these routines is:

```
su3_matrix *pt; int x,y,z,t;
/* set x,y,z,t to the coordinates of the desited site here, then: */

  pt = (su3_matrix *)field_pointer_at_coordinates( F_OFFSET(xlink),
                                                   sizeof(su3_matrix),
                                                   x,y,z,t );

/* now "pt" points to the xlink at the site whose coordinates are
   x,y,z,t. It may point either to the original data or a copy.
   Use it for whatever you want, and when you are done with it: */

  cleanup_field_pointer( (char *)pt );

/* subsequent calls to malloc may overwrite *pt, so don't use it any
   more */
```

## 4.6 Details of gathers and creating new ones

A *gather* is basically defined by a mapping, where each site receives data from some other site. To speed up gathers, there are routines which prepare tables on each node containing information about what sites must be sent to other nodes or received from other nodes. Before calling such a gather, a routine must be called to set up the tables. The call to this routine is:

```
#include <comdefs.h>
```

```
int make_gather( function, arg_pointer, inverse, want_even_odd,
                 parity_conserve )
  int (*function)();
  int *arg_pointer;
  int inverse;
  int parity_conserve;
```

The "function" argument is a pointer to a function which defines the mapping. This function must have the following form:

```
int function( x, y, z, t, arg_pointer, forw_back, xpt, ypt, zpt, tpt)
  int x,y,z,t;
  int *arg_pointer;
  int forw_back;
  int *xpt,*ypt,*zpt,*tpt;
```

Here x,y,z,t are the coordinates of the site *RECEIVING* the data. arg_pointer is a pointer to a list of integers, which is passed through to the function from the call to make_gather(). This provides a mechanism to use the same function for different gathers. For example, in setting up nearest neighbor gathers we would want to specify the direction. See the examples below.

forw_back is either **FORWARDS** or **BACKWARDS**. If it is **FORWARDS**, the function should compute the coordinates of the site that sends data to x,y,z,t. If it is **BACKWARDS**, the function should compute the coordinates of the site which gets data from x,y,z,t. It is necessary for the function to handle **BACKWARDS** correctly even if you don't want to set up the inverse gather (see below). At the moment, only one-to-one (invertible) mappings are supported.

The inverse argument to make_gather() is one of **OWN_INVERSE**, **WANT_INVERSE**, or **NO_INVERSE**. If it is **OWN_INVERSE**, the mapping is its own inverse. In other words, if site x1,y1,z1,t1 gets data from x2,y2,z2,t2 then site x2,y2,z2,t2 gets data from x1,y1,z1,t1. Examples of mappings which are there own inverse are the butterflys in FFT's. If inverse is **WANT_INVERSE**, then make_gather() will make two sets of lists, one for the gather and one for the gather using the inverse mapping. If inverse is **NO_INVERSE**, then only one set of tables is made.

The want_even_odd argument is one of **ALLOW_EVEN_ODD** or **NO_EVEN_ODD**. If it is **ALLOW_EVEN_ODD** separate tables are made for even and odd sites, so that start gather can be called with parity **EVEN**, **ODD** or **EVENANDODD**. If it is **NO_EVEN_ODD**, only one set of tables is made and you can only call gathers with parity **EVENANDODD**.

The `parity_conserve` argument to `make_gather()` is one of **SAME_PARITY**, **SWITCH_PARITY**, or **SCRAMBLE_PARITY**. Use **SAME_PARITY** if the gather connects even sites to even sites and odd sites to odd sites. Use **SWITCH_PARITY** if the gather connects even sites to odd sites and vice versa. Use **SCRAMBLE_PARITY** if the gather connects some even sites to even sites and some even sites to odd sites. If you have specified **NO_EVEN_ODD** for `want_even_odd`, then the `parity_conserve` argument does nothing. Otherwise, it is used by `make_gather()` to help avoid making redundant lists.

`make_gather()` returns an integer, which can then be used as the `direction` argument to `start_gather()`. If an inverse gather is also requested, its `direction` will be one more than the value returned by `make_gather()`. In other words, if `make_gather()` returns 10, then to gather using the inverse mapping you would use 11 as the direction argument in start_gather.

Notice that the nearest neighbor gathers do not have their inverse directions numbered this convention. Instead, they are sorted so that **OPP_DIR**(`direction`) gives the gather using the inverse mapping.

Now for some examples which hopefully clarify all this.

First, suppose we wished to set up nearest neighbor gathers. (Of course, `make_comlinks()` already does this for you, but it is a good example. The function which defines the mapping is basically `neighbor_coords()`, with a wrapper which fixes up the arguments. `arg` should be set to the address of the direction — **XUP**, etc.

```
/* The function which defines the mapping */
neighbor_temp(x,y,z,t, arg, forw_back, xpt, ypt, zpt, tpt)
  int x,y,z,t;
  int *arg;
  int forw_back;
  int *xpt,*ypt,*zpt,*tpt;
{
  register int dir; /* local variable */
  dir = *arg;
  if(forw_back==BACKWARDS)dir=OPP_DIR(dir);
  neighbor_coords(x,y,z,t,dir,xpt,ypt,zpt,tpt);
}

/* Code fragment to set up the gathers */
/* Do this once, in the setup part of the program. */
int xup_dir, xdown_dir;
int temp;
temp = XUP; /* we need the address of XUP */
```

```
xup_dir = make_gather( neighbor_temp, &temp, WANT_INVERSE,
                       ALLOW_EVEN_ODD, SWITCH_PARITY);
xdown_dir = xup_dir+1;

/* Now you can start gathers */
start_gather( F_OFFSET(phi), sizeof(su3_vector), xup_dir, EVEN,
              gen_pt[0] );
/* and use wait_gather, cleanup_gather as always. */
```

Again, once it is set up it works just as before. Essentially, you are just defining new `directions`. Again, `make_comlinks()` does the same thing, except that it arranges the directions so that you can just use **XUP**, **XDOWN**, etc. as the `direction` argument to `start_gather()`.

A second example is for a gather from a general displacement. You might, for example, set up a bunch of these to take care of the link gathered from the second mearest neighbor in evaluating the plaquette in the pure gauge code. In this example, the mapping function needs a list of four arguments — the displacement in each of four directions. Notice that for this displacement even sites connect to even sites, etc.

```
/* The function which defines the mapping */
/* arg is a four element array, with the four displacements */
general_displacement(x,y,z,t, arg, forw_back, xpt, ypt, zpt, tpt)
  int x,y,z,t;
  int *arg;
  int forw_back;
  int *xpt,*ypt,*zpt,*tpt;
{
    if( forw_back==FORWARDS ) { /* add displacement */
       *xpt = (x+nx+arg[0])%nx;
       *ypt = (y+ny+arg[1])%ny;
       *zpt = (z+nz+arg[2])%nz;
       *tpt = (t+nt+arg[3])%nt;
    }
    else { /* subtract displacement */
       *xpt = (x+nx-arg[0])%nx;
       *ypt = (y+ny-arg[1])%ny;
       *zpt = (z+nz-arg[2])%nz;
       *tpt = (t+nt-arg[3])%nt;
    }
}

/* Code fragment to set up the gathers */
/* Do this once, in the setup part of the program. */
/* In this example, I set up to gather from displacement -1 in
   the x direction and +1 in the y direction */
int plus_x_minus_y;
int disp[4];
```

```
disp[0] = -1;
disp[1] = +1;
disp[2] = 0;
disp[3] = 0;
plus_x_minus_y = make_gather( general_displacement, disp,
                              NO_INVERSE, ALLOW_EVEN_ODD, SAME_PARITY);

/* Now you can start gathers */
start_gather( F_OFFSET(link[YUP]), sizeof(su3_matrix), plus_x_minus_y,
              EVEN, gen_pt[0] );
/* and use wait_gather, cleanup_gather as always. */
```

Finally, an FFT butterfly we would want to set up roughly as follows. Here the function wants two arguments: the direction of the butterfly and the level.

```
/* The function which defines the mapping */
/* arg is a two element array, with the direction and level */
butterfly_map(x,y,z,t, arg, forw_back, xpt, ypt, zpt, tpt)
  int x,y,z,t;
  int *arg;
  int forw_back;
  int *xpt,*ypt,*zpt,*tpt;
{
int direction,level;
    direction = arg[0];
    level = arg[1];
    /* Rest of code goes here */
}

/* Code fragment to set up the gathers */
/* Do this once, in the setup part of the program. */
int butterfly_dir[5]; /* for nx=16 */
int args[2];
args[0]=XUP;
for( level=1; level<=4; level++ ) {
    args[1]=level;
    butterfly_dir[level] = make_gather( butterfly_map, args,
       OWN_INVERSE, NO_EVEN_ODD, SCRAMBLE_PARITY);
}
/* similarly for y,z,t directions */
```

## 4.7 Data types

Various data structures have been defined for QCD computations. More will be defined as we progress, notably Wilson fermion spinors. Again, you are free to use these or not, and to define any other types.

In names of members of structure, I will use the following conventions:

`c` means color, and has an index which takes three values (0,1,2).

`d` means Dirac spin, and its index takes four values (0-4).

`e` means element of a matrix, and has two indices which take three values - row and column.

Complex numbers: (in '`complex.h`')

```
typedef struct { /* standard complex number declaration for single- */
   float real;    /* precision complex numbers */
   float imag;
} complex;

typedef struct { /* standard complex number declaration for double- */
   double real;   /* precision complex numbers */
   double imag;
} double_complex;
```

Three component complex vectors, 3x3 complex matrices, and 3x3 antihermitian matrices stored in triangular (compressed) format. (in '`su3.h`')

```
typedef struct { complex e[3][3]; } su3_matrix;
typedef struct { complex c[3]; } su3_vector;
typedef struct {
  float m00im,m11im,m22im;
  complex m01,m02,m12;
} anti_hermitmat;
```

Wilson vectors:

```
typedef struct { su3_vector d[4]; } wilson_vector;
```

Projections of Wilson vectors, using $1 \pm \gamma_\mu$

```
typedef struct { su3_vector h[2]; } half_wilson_vector;
```

A definition to be used in the next definition:

```
typedef struct { wilson_vector c[3]; } color_wilson_vector;
```

A four index object — source spin and color by sink spin and color:

```
typedef struct { color_wilson_vector d[4]; } wilson_matrix
```

Examples:

```
su3_vector phi; /* declares a vector */
su3_matrix m1,m2,m3;          /* declares 3x3 complex matrices */
wilson_vector wvec;           /* declares a Wilson quark vector */

phi.c[0].real = 1.0;          /* sets real part of color 0 to 1.0 */
phi.c[1] = cmplx(0.0,0.0);    /* sets color 1 to zero (requires
                                 including "complex.h" */
m1.e[0][0] = cmplx(0,0);      /* refers to 0,0 element */
mult_su3_nn( &m1, &m2, &m3); /* subroutine arguments are usually
                                 addresses of structures */
wvec.d[2].c[0].imag = 1.0;    /* How to refer to imaginary part of
                                 spin two, color zero. */
```

## 4.8 Library routines

### 4.8.1 Complex numbers

'`complex.h`' and '`complex.a`' contain macros and subroutines for complex numbers. For example:

```
complex a,b,c;
CMUL(a,b,c); /* macro: c <- a*b */
```

Note that all the subroutines (`cmul()`, etc.) take addresses as arguments, but the macros generally take the structures themselves. These functions have separate versions for single and double precision complex numbers. The macros work with either single or double precison (or mixed). '`complex.a`' contains:

```
complex cmplx(float r, float i);      /* (r,i) */
complex cadd(complex *a, complex *b); /* a + b */
complex cmul(complex *a, complex *b); /* a * b */
complex csub(complex *a, complex *b); /* a - b */
complex cdiv(complex *a, complex *b); /* a / b */
complex conjg(complex *a);            /* conjugate of a */
complex cexp(complex *a);             /* exp(a) */
```

```
complex clog(complex *a);             /* ln(a) */
complex csqrt(complex *a);            /* sqrt(a) */
complex ce_itheta(float theta);       /* exp( i*theta) */

double_complex dcmplx(double r, double i);                  /* (r,i) */
double_complex dcadd(double_complex *a, double_complex *b); /* a + b */
double_complex dcmul(double_complex *a, double_complex *b); /* a * b */
double_complex dcsub(double_complex *a, double_complex *b); /* a - b */
double_complex dcdiv(double_complex *a, double_complex *b); /* a / b */
double_complex dconjg(double_complex *a);          /* conjugate of a */
double_complex dcexp(double_complex *a);                   /* exp(a) */
double_complex dclog(double_complex *a);                   /* ln(a) */
double_complex dcsqrt(double_complex *a);                 /* sqrt(a) */
double_complex dce_itheta(double theta);           /* exp( i*theta) */
```

and macros:

```
CONJG(a,b)        b = conjg(a)
CADD(a,b,c)       c = a + b
CSUM(a,b)         a += b
CSUB(a,b,c)       c = a - b
CMUL(a,b,c)       c = a * b
CDIV(a,b,c)       c = a / b
CMUL_J(a,b,c)     c = a * conjg(b)
CMULJ_(a,b,c)     c = conjg(a) * b
CMULJJ(a,b,c)     c = conjg(a*b)
CNEGATE(a,b)      b = -a
CMUL_I(a,b)       b = ia
CMUL_MINUS_I(a,b) b = -ia
CMULREAL(a,b,c)   c = ba with b real and a complex
CDIVREAL(a,b,c)   c = a/b with a complex and b real
```

### 4.8.2 SU(3) operations

'su3.h' and 'su3.a' contain functions for SU(3) operations. For example:

```
void mult_su3_nn(su3_matrix *a, su3_matrix *b, su3_matrix *c);
/* matrix multiply, no adjoints
   *c <- *a * *b (arguments are pointers) */

void mult_su3_mat_vec_sum(su3_matrix *a, su3_vector *b, su3_vector *c);
/* su3_matrix times su3_vector multiply and add to another su3_vector
   *c <- *A * *b + *c */
```

There have come to be a great many of these routines, too many to keep a duplicate list of here. Consult the include file 'su3.h' for a list of prototypes and description of functions. An html version of 'su3.h' is available at http://www.physics.arizona.edu/~hetrick/su3.html.

### 4.8.3 SU(2) functions

Most of the SU(3) utility routines also exist for SU(2). See the directory **libsu2**, which contains the include file 'su2.h' and the file 'globaldefs.h'. 'globaldefs.h' includes things that are really common to both SU(2) and SU(3), and in some later code version these routines should be removed from 'su3.h'.

### 4.8.4 Miscellaneous functions

These will probably be collected somewhere as the code evolves.

```
/* utility function for finding coordinates of neighbor */
/* x,y,z,t are the coordinates of some site, and x2p... are
   pointers.  *x2p... will be set to the coordinates of the
   neighbor site at direction "dir".

neighbor_coords( x,y,z,t,dir, x2p,y2p,z2p,t2p)
  int x,y,z,t,dir; /* coordinates of site, and direction (eg XUP) */
  int *x2p,*y2p,*z2p,*t2p;
```

## 4.9 Distributing sites among nodes

Four functions are used to determine the distribution of sites among the parallel *nodes*. These functions may be changed, but chaos will ensue if they are not consistent. For example, it is a gross error for the node_index function to return a value larger than or equal to the value returned by num_sites of the appropriate node. In fact, node_index must provide a one-to-one mapping of the coordinates of the sites on one node to the integers from 0 to num_sites(node)-1.

setup_layout() is called once on each node at initialization time, to do any calculation and set up any static variables that the other functions may need. At the time setup_layout() is called the global variables ns,ny,nz and nt (the lattice dimensions) will have been set.

setup_layout() must initialize the global variables:

```
sites_on_node,
even_sites_on_node,
odd_sites_on_node.
```

The following functions are available for node/site reference:

`num_sites(node)`
: returns the number of sites on a node

`node_number(x,y,z,t)`
: returns the node number on which a site lives.

`node_index(x,y,z,t)`
: returns the index of the site on the node.

Thus, the site at `x,y,z,t` is `lattice[node_index(x,y,z,t)]`.

A good choice of site distribution on nodes will minimize the amount of communication. These routines are in '`layout_XXX.c`'. There are currently several layout strategies to choose from; select one in your '`Makefile`' (see Section 5.2 [Building the MILC Code], page 34).

'`layout_gen.c`'
: "generic" stupid layout, mostly for testing.

'`layout_planes.c`'
: distributes 2-d planes enely among nodes.

'`layout_squares.c`'
: divides longest two directions of the lattice by factors of two. Fails if there aren't enough powers of two in the dimensions.

'`layout_yztcubes.c`'
: useful on the CM5

'`layout_hyper.c`'
: divides the lattice up into hypercubes by dividing dimensions by factors of two. Fails if there aren't enough powers of two in the dimensions.

Below is a completely simple example, which just deals out the sites among nodes like cards in a deck. It works, but you would really want to do much better.

```
int Num_of_nodes; /* static storage used by these routines */
```

```
void setup_layout() {
  Num_of_nodes = numnodes();
  sites_on_node = nx*ny*nz*nt/Num_of_nodes;
  even_sites_on_node = odd_sites_on_node = sites_on_node/2;
}

int node_number(x,y,z,t)
int x,y,z,t;
{
  register int i;
  i = x+nx*(y+ny*(z+nz*t));
  return( i%Num_of_nodes );
}

int node_index(x,y,z,t)
int x,y,z,t;
{
  register int i;
  i = x+nx*(y+ny*(z+nz*t));
  return( i/Num_of_nodes );
}

int num_sites(node)
int node;
{
  register int i;
  i = nx*ny*nz*nt;
  if( node< i%Num_of_nodes ) return( i/Num_of_nodes+1 );
  else return( i/Num_of_nodes );
}
```

Some of the layout files have options, set in '`lattice.h`'. The **EVENFIRST** option causes all the even sites to be stored first in the array, followed by all the odd sites. This makes looping over sites of a given parity more efficient. **GRAYCODE** and **ACCORDION** are obsolete options for hypercube architectures.

*Note*: At some future time, **EVENFIRST** will become required.

## 4.10 Random numbers

The random number generator is the exclusive-OR of a 127 bit feedback shift register and a 32 bit integer congruence generator. It is supposed to be machine independent. Each node or site uses a different multipler in the congruence generator and different initial state in the shift register, so

all are generating different sequences of numbers. If **SITERAND** is defined, each lattice site has its own random number generator state. This takes extra storage, but means that the results of the program will be independent of the number of nodes or the distribution of the sites among the nodes.

## 4.11 Files

The files which make up the program are listed here. This list depends very much on which application of the program (Kogut-Susskind/Wilson? Thermodynamics/Spectrum?) is being built. The listing here is for a bare bones Kogut-Susskind application.

'`Make_XXX:`'
: Contains instructions for compiling and linking. There are three things you can make, "su3_rmd", "su3_phi" and "su3_hmc", which are programs for the R algorithm, the phi algorithm, or the hybrid Monte Carlo algorithm. Definitions of **COMPILER** and **CFLAGS** in this file need to be changed to move from machine to machine.

**HIGH LEVEL ROUTINES:**

'`control.c`'
: main procedure - directs traffic it must call initialize_machine() first thing.

'`setup.c`'
: most of the initialization stuff - called from control.c

'`update.c`'
: update the gauge fields by refreshing the momenta and integrating for one trajectory. Knows three different algorithms through preprocessor switches.

'`update_h.c`'
: Update the gauge momenta

'`update_u.c`'
: Update the gauge fields

'`grsource.c`'
: Heat bath update of the phi field - "Gaussian random source"

'`d_congrad5.c`'
: A Kogut-Susskind inverter

'`reunitarize.c`'
: Reunitarize the gauge fields

'action.c'

Measure the action. Used only in the hybrid Monte Carlo algorithm.

'ranmom.c'

Produce Gaussian random momenta for the gauge fields.

'plaquette.c'

'plaquette2.c'

Measure the plaquette. plaquette2 is the good one, the first one is obsolete except for testing field_pointer routines.

'ploop.c' Measure Polyakov loop

'f_measure.c'

Measure fermion stuff - psi-bar-psi, fermion energy and pressure.

'spectrum.c'

Measure hadron propagators

'spectrum_s.c'

Measure hadron screening propagators

'gaugefix.c'

Fix to lattice Landau or Coulomb (in any direction) gauge

### LOWER LEVEL STUFF

'ranstuff.c'

Routines for random numbers on multiple nodes: initialize_prn() and myrand()

'layout_XXX.c'

currently one of layout_gen.c, layout_planes.c, layout_squares.c or layout_hyper.c. Routines which tell which node a site lives on, and where on the node it lives.

'com_XXX.c'

Communication routines - gather, field_pointer, setup routines for communications. This one is very machine dependent. Choose the appropriate one for your target machine.

'io_lat.c,'

Input and output routines - read and write lattices. Some optimized versions for special machines exist such as 'io_t3d.c' amd 'io_paragon.c'.

### LIBRARIES

'`complex.a`'
: complex number operations. See section on "Library routines".

'`su3.a`'
: 3x3 matrix and 3 component vector operations. See "utility subroutines".

## 4.12 Bugs and features

The i860 assembler language subroutines use doubleword loads to get complex numbers, and if they are not aligned on doubleword boundaries the program slows down at best and crashes at worst. I do not yet know an elegant way to insure this - putting a double in the site structure ahead of all the complex stuff doesn't work because the compiler doesn't align doubles to doubleword boundaries! The current workaround is to insert integers in the site structure as needed. Currently, only one such kludge is needed; the random number generator state is 11 words so it is padded to 12 words with an integer to keep things aligned on double words (See '`ks_dynamical/lattice.h`' for example).

Ideally the code will terminate smoothly at the end of the input file. Some applications like **ks_dynamical** loop over blocks of input. On some machines, we have found that the mechanism for node 0 to kill all other nodes does not work (the T3D using PVM for instance). Therefore most of our applications contain code in '`setup.c`' which stops the program if `beta < 0`.

# 5 Building the MILC Code

At the moment, building the MILC Code is a bit kludgey. There is a '`makefile`' for each type of parallel architecture:

```
Make_vanilla (for scalar mode on workstations and PC's)
Make_cm5
Make_intel
Make_paragon
Make_t3d
Make_t3d_mpi
Make_mpi
```

Sometime soon we hope to use the *Cygnus'* `configure` package to dynamically test for architectural type and compiling environment, and to then produce appropriate makefiles. This will mean one types simply: `configure` and `make`.

For now however, the procedure goes as follows, where we `make` each directory individually. For the most part the procedure is identical in each of the application directories. Since we will need the **libsu3** (or *su2*) libraries before any applications or generic code, let's first go through their compilation as an example.

## 5.1 Making the Libraries

There are two libraries needed for SU(3) operations:

- **complex.a** contains struct definitions and routines for operations on complex numbers. See '`complex.h`' for a summary.
- **su3.a** contains struct definitions and routines for operations on SU3 matrices, 3 element complex vectors, and Wilson vectors (12 element complex vectors). See '`su3.h`' for a summary.

For SU(2) code you will need to additionally make

- **su2.a** which contains routines for operations on SU2 matrices, 2 element complex vectors, and SU(2) Wilson vectors (*Note that these are not yet implimented!*). See '`su3.h`' for a summary.

First `cd` to the **libsu3** directory, and choose the makefile for your machine. You will have to edit this file according to your environment and directory structure by uncommenting the correct **COMPILER**, **CFLAGS**, etc. from the list as shown below for `gcc` compatible machines.

```
# Makefile for Libraries for QCD programs # # Vanilla version, for
workstations

#CFLAGS = -O -fsingle -DFAST #Sun 3 #CFLAGS = -O4 -fsingle -DFAST #Sun 4
#CFLAGS = -O -DFAST -DPROTO -float #Dec alpha compiler #CFLAGS = -O -f
-DFAST -DPROTO #Mips #CFLAGS = -O -DFAST -DPROTO #IBM RS6000 #CFLAGS =
-O3 -qarch=pwr2 -DFAST -DPROTO #IBM POWER2 CFLAGS = -O -DFAST -DPROTO
#gnu c compiler

#COMPILER = cc #most #COMPILER = xlc #IBM RS6000 ANSI c COMPILER = gcc
#gnu c compiler
```

At this point you can `make` the libraries for your particular architecture.

There are a few files which are specific to particular machines. **i860** assembler code for some routines is in the files with the '`.m4`' suffix. In each case a '`.c`' file of the same name contains a C routine which should work identically. Similarly, **T3D** assembler code is in files with the '`.t3d`' suffix. The existence of assembler code for the DEC Alpha workstation is a byproduct of producing T3D assembler—the nodes use the same chip, so we just need to change the assembler directives, mneumonics, and stack usage conventions.

The scalar workstation code on a SUN4 and the CM5 code can be compiled either with the Sun `cc` compiler or with the gnu C compiler `gcc`. Note that if the library code is compiled with `gcc` the application directory code must also be compiled with `gcc`, and vice versa. This is because `gcc` understands prototypes and the sun4 `cc` compiler doesn't, and they therefore pass float arguments differently. We generally recommend `gcc` on workstations and the CM5, with the exception of the DEC Alpha, where `cc` produces better optimized code. Then, to make the libraries say, on

*a scalar workstation (not a DEC Alpha),*
*for PVM,*
*or the CM5:*

Edit '`Make_vanilla`' to get the directories, compiler, flags, etc. correct; then do

```
make -f Make_vanilla all >& make.log &
```

*the DEC Alpha*

Edit '`Make_alpha`' to get the directories, compiler, flags, etc. correct.

```
make -f Make_alpha all >& make.log &
```

*the Intel Paragon*

Edit '`Make_paragon`' to get the directories, compiler, flags, etc. correct.

```
make -f Make_paragon all >& make.log &
```

*the Cray T3D*

Edit '`Make_t3d`' to get the directories, compiler, flags, etc. correct.

```
make -f Make_t3d all >& make.log &
```

The same procedure should be followed in the **libsu2** directory to make '`su2.a`', using the command:

```
make -f Make_MACHINE su2.a >& make.log &
```

Similarly, on can make the individual libraries by name with the command `make -f Make_MACHINE complex.a su3.a >& make.log &`

## 5.2 Making the Applications

Most application directories have more or less the same structure. There is code to do various high level things specific to the application, a '`control.c`' file with the `main()` program, and various '`Make_MACHINE`' files.

# Concept Index

## R



## S



## T



## U



## W

# Variable Index

# Table of Contents