



Matrix distributed processing: a set of C++ tools for implementing generic lattice computations on parallel systems ☆,☆☆

Massimo Di Piero

Fermi National Accelerator Laboratory, P.O. Box 500, Batavia, IL 60510, USA

Received 26 April 2001; received in revised form 26 May 2001

Abstract

We present a set of programming tools (classes and functions written in C++ and based on Message Passing Interface) for fast development of generic parallel (and non-parallel) lattice simulations. They are collectively called MDP 1.2.

These programming tools include classes and algorithms for matrices, random number generators, distributed lattices (with arbitrary topology), fields and parallel iterations. No previous knowledge of MPI is required in order to use them.

Some applications in electromagnetism, electronics, condensed matter and lattice QCD are presented. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Lattice; Parallel computing; Numerical software

PROGRAM SUMMARY

Title of the library: MDP, Matrix Distributed Processing (version 1.2)

Catalogue identifier: ADPA

Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland

Program Summary URL: <http://cpc.cs.qub.ac.uk/summaries/ADPA>

Computer for which the library is designed: Any computer or parallel computer, including clusters of workstations

Software required in order to run MDP: MPI (Message Passing Interface) is required to run MDP in parallel. MPI is not required to run MDP in single process mode

Computer(s) on which the library has been tested: SUN SparcSTATION with Solaris, Linux PCs, cluster of PCs (connected by Ethernet and Myrinet). The code has also been tested partially on a Cray T3E

Programming language used: ANSI C++, to be compiled with `g++ -ansi -pedantic [input program] -o [output program]`

☆ The realization of this paper and of the software FermiQCD was performed at Fermilab, a U.S. Department of Energy Lab (operated by the University Research Association, Inc.), under contract DE-AC02-76CHO3000.

☆☆ This program can be downloaded from the CPC Program Library under catalogue identifier: <http://cpc.cs.qub.ac.uk/summaries/ADPA>
E-mail address: mdp@fnal.gov (M. Di Piero).

No. of bytes in distributed software, including test data, etc.:
2 500 222

Distribution format: tar gzip file

Nature of the physical problem

Any problem that can be described in terms of interacting fields discretized on a lattice of some arbitrary shape and topology

Parallel applications provided together with the library as examples

A program that solves electrostatic problems (`application1.C`).

A program that computes total impedance in a net of resistors (`application2.C`). An Ising model simulation (`application3.C`). A parallel implementations of the Vegas multidimensional integration algorithm (`MDP_PVegas.h`, not described here). A complete Lattice QCD package (`FermiQCD`, not described here)

Web site: <http://www.phoenixcollective.org/mdp/mdp.html>

LONG WRITE-UP

1. Introduction

One of the biggest discoveries of modern physics is that long distance forces are mediated by fields and these fields are subject to local interactions. Local differential equations, in some field variables, seem to describe any aspect of fundamental physics (from particle physics to gravitation, from thermodynamics to condensed matter).

Often the equations describing complex interacting systems cannot be solved exactly and require a numerical approach, i.e. discretize on a lattice the space on which the equations are defined. The derivative terms of a local equation, once discretized, give origin to quasi-local interaction terms on the lattice.

The good thing of having to deal with quasi-local equations is that one can partition the problem by partitioning the lattice (the space) on which the problem is defined, and only a minimal set of communications is required to pass information about the field variables at the boundary of these partitions (because first and second derivatives of the differential equations in the continuum correspond to nearest neighbor interaction terms on the lattice).

Parallel computers and clusters are the natural hardware for implementing the numerical algorithms that can solve this kind of problems.

MDP is a set of C++ tools for fast development of any kind of parallel numerical computations that involve the lattice discretization of a finite portion of space (and/or time) over a number of parallel processors (supporting Message Passing Interface). It considerably reduce that development costs since one can concentrate on the algorithm instead of focusing on communications.

A number of tricks and optimizations are implemented that make of MDP 1.2 an efficient package for developing, among others, fast and robust Lattice QCD applications.¹

1.1. Content

The main characteristics of MDP 1.2 are:

- It includes `Matrix`, a class to manipulate matrices of complex (`Complex`) numbers. The assignment operator and the copy constructor have been overloaded to optimize the speed and the memory usage. In fact a `Matrix` object contains a dynamically allocated pointer to the location of memory where the matrix elements are stored. The matrix elements are automatically copied only when it is inevitable, otherwise their memory address is copied. In particular, the temporary `Matrix` object returned by a function stores its elements in the same location of memory used by the function that will take that object as argument. This class is used to implement an interface to the fields.

¹ For example, MDP 1.2 has been used here at Fermilab to develop, in a relatively short time, a parallel software for Lattice QCD simulations (see Section 5.5).

- All the calls to MPI are done through a wrapper object, `mpi`, which takes care of generating communication tags and computing the length of the objects to be sent/received by the parallel processes.
- The most important classes in MDP 1.2 are

```
generic_lattice
site
generic_field
```

They enable the user to define lattices of any size, shape and topology and associate any kind of structure to the sites. Other properties of these classes are: automatic communications between the parallel processes (when necessary), parallel I/O for the field variables and functions to move on the lattice.

- For each user defined lattice, one random generator per site is automatically created and is a member variable of `generic_lattice`. This guarantees that any simulation will give the same results whether it runs on a single PC or in parallel, independently of the way the lattice is partitioned. The random generator itself can be seen as a field and it can be saved and restored at any point.
- Each lattice (and each field defined on that lattice) can be partitioned arbitrarily among the different processes (or automatically if the user does not bother to do it) and the constructor of `generic_lattice`, for each process, takes care of allocating the memory for storing the neighbor sites that have to be passed by a neighbor process. The code has been optimized in order to minimize the need for communications and the low level optimization tricks are completely hidden to the high level programmer.
- The same code can run on single process on one PC or multiprocess on a cluster of PCs (or on a supercomputer supporting MPI), without any modification. The results are independent from the number of processes.²

As an example of what one can do with MDP 1.2 we present a parallel program that creates a field of 5×5 matrices on a $4 \times 4 \times 4$ lattice (distributed on 4 processors) and sets each matrix (for each site) equal to the inverse of the exponential of a random SU(5) matrix (just to make things a little complicated!). Then it averages each matrix with its 6 nearest neighbors.

```
00: // Program: example01.C
01: #define PARALLEL
02: #define Nproc 4
03: #include "MDP_Lib2.h"
04: #include "MDP_MPI.h"
05:
06: int main(int argc, char **argv) {
07:     mpi.open_wormholes(argc,argv);
08:     int ndim=3;
09:     int mybox[]={4,4,4};
10:     generic_lattice mylattice(ndim, mybox);
11:     Matrix_field F(mylattice,5,5);
12:     site x(mylattice);
13:     forallsites(x)
14:         F(x)=inv(exp(mylattice.random(x).SU(5)));
15:     F.update();
16:     forallsites(x) {
17:         F(x)=1.0/7.0*(F(x)+F(x+0)+F(x-0)+F(x+1)+F(x-1)+F(x+2)+F(x-2));
18:         printf("F(x) for x=(%i,%i)\n", x(0), x(1));
19:         print(F(x));
```

² Apart for rounding errors that usually are compiler and hardware dependent.

```

20:    };
21:    mpi.close_wormholes();
22:    return 0;
23: };

```

Note that each line has a meaning and none of them is superfluous:

- Line 1 tells that the job will run in parallel.
- Line 2 tells that it will run on 4 processes (this statement is optional since the number of processes can be determined at runtime).
- Line 3 and 4 includes the libraries of MDP 1.2.
- Line 7 establishes the communications among the processes.
- Line 8 sets a variable containing the dimensions of the lattice.
- Line 9 fills an array containing the size of the lattice.
- Line 10 defines the lattice, `mylattice`.
- Line 11 defines a field of 5×5 matrices F on `mylattice` (`Matrix_field` is built-in field based on `generic_field`).
- Line 12 defines a site variable x on `mylattice`.
- Lines 13, 14 initialize each field variable $F(x)$ with the inverse of the exponential of a random $SU(5)$ (generated using the local random generator of site x).
- Line 15 performs all the communication to update boundary sites, i.e. sites that are shared by different processes. They are determined automatically and optimally when the lattice is declared.
- Lines 16, 17 average $F(x)$ with its 6 next neighbor sites.
- Lines 18, 19 print out the results.
- Line 21 closes all the communications.

The source code of MDP 1.2 is contained in the files

- `MDP_Lib2.h`,
- `MDP_MPI.h`,
- `MDP_ANSI.h`,
- `MDP_Measure.h`,
- `MDP_Fit.h`,
- `MDP_PVegas.h`,
- `MDP_Prompt.h`,
- `MDP_utils.h`,

plus some examples and applications. Only the classes defined in the first two files will be discussed here.

A reader interested only in parallelization issues can jump directly to Section 4, and go back to Sections 2 and 3 later.

2. Nonparallel tools

Four basic classes are declared in the file `MDP_Lib2.h`:³

- `myreal`. It is equivalent to `float` unless the user defines the global constant `USE_DOUBLE_PRECISION`. In this case `myreal` stands for `double`.
- `Complex`. Declared as `complex<myreal>`. The imaginary unit is implemented as a global constant `I=Complex(0,1)`.

³ The classes defined in the file `MDP_Lib2.C` have extensively been used in some analysis programs written by the Southampton Theory Group for Lattice QCD applications.

- `DynamicArray<object, n>`. It is a container for n -dimensional arrays of objects (for examples arrays of `Matrix`). `DynamicArrays` can be resized any time, moreover they can be passed to (and returned by) functions.
- `Matrix`. An object belonging to this class is effectively a matrix of `Complex` numbers and it may have arbitrary dimensions.
- `Random_generator`. This class contains the random number generator and member functions to generate random `float` numbers with uniform distribution, Gaussian distribution or any user defined distribution. It can also generate random $SU(n)$ matrices. Different `Random_generator` objects can be declared and initialized with different seeds.
- `JackBoot`. It is a class to store sets of data to be used for computing the average of any user defined function acting on the data. It includes member functions for computing Jackknife and Bootstrap errors on this average. Moreover the following constants are declared

```
#define and          &&
#define or           ||
#define Pi           3.14159265359
#define I            Complex(0,1)
#define PRECISION    1e-16
#define CHECK_ALL
#define TRUE 1
#define FALSE 0
```

If the definition of `CHECK_ALL` is removed the code will be faster but some checks will be skipped resulting a lesser safe code.

2.1. On `DynamicArray` and `Matrix`

The main idea on which these two classes have been built is the solution of the problem of returning objects containing pointers to dynamically allocated memory [1].

Consider the following code involving matrices:

```
Matrix A,B(10,10);
A=B;
A=B*A;
```

In the first assignment one wants each element of `B` to be copied in the corresponding element of `A`, while the second assignment is faster if `B` and `A` are passed to the local variables of the `operator*` by reference (i.e. without copying all the elements). Moreover one wants the local variable created by the `operator*` to occupy the same memory location as the variable that is returned (this avoids copying and wasting space). To implement this idea each `Matrix` object contains a `FLAG` and a pointer to dynamically allocated memory (where the numbers are stored). The copy constructor and the `operator=` have been overloaded in such a way to take care of the status of the `FLAG` and eventually to copy the pointer to the memory, instead of copying the memory containing the real matrix.

A physical location of memory may be pointed by different `Matrix` objects, but this never generates confusion if a few safety rules, stated later, are followed. An automatic system of garbage collecting deallocates the unused memory when there are no objects alive pointing to it.

In this way, in the first assignment of the example, $11 + 800$ bytes⁴ are copied, while in the second assignment only 11 bytes are copied three times (when passing `A` and `B` to the `operator*()` and when returning the result)

⁴ The number 11 is the size in bytes of a `Matrix` object. It is independent from the size of the memory occupied by the “real” matrix.

to be compared with the 800 bytes of the matrix. The pointer of A is swapped only when the multiplication is terminated without generating confusion between the input and the output of the function. The FLAG takes care of everything and the procedure works also in any recursive expression. For big matrices this is, in principle, faster than it would be possible in FORTRAN or C. (In FORTRAN, for example, it would be necessary to create a temporary array where to store the result of the multiplication and copy it into A.) In practice this is not true because the class `Matrix` uses dynamic allocation and modern compilers are not able to optimize its algorithms as well as they can optimize algorithms using normal arrays.

We do not suggest to use the classes `DynamicArray` and `Matrix` to implement those parts of the user's algorithms that are critical for speed. Despite this, they can be used to implement an efficient and practical interface to field variables (as shown in Section 4.3.4).

2.2. Safety rules

The safety rules for `DynamicArray` and `Matrix` objects are:

- Always, before returning an *object*, call `prepare(object)`;
- Never explicitly call the copy constructor.
- Do not call the assignment operator of the argument of a function within the function itself.

2.3. Using `DynamicArray`

`DynamicArrays` are declared using templates. For example the command

```
DynamicArray<float,4> myarray(2,3,6,5);
```

declares a $2 \times 3 \times 6 \times 5$ array of `float` called `myarray`. The first argument of the template specifies the type of object (i.e. `float`) and the second argument is the number of dimensions of the array (i.e. 4). An array of this kind can be resized at any time. For example,

```
myarray.dimension(4,3,2,2);
```

transforms `myarray` in a new $4 \times 3 \times 2 \times 2$ array of `float`. A `DynamicArray` can have up to 10 dimensions. The array elements can be accessed by reference using `operator()` in the natural way

```
myarray(i,j,k,l)
```

where `i, j, k, l` are integers. The total number of dimensions (in this example 4) is contained in the member variable

```
myarray.ndim
```

Two useful member functions are

```
myarray.size()
```

that returns the total number of elements of the array (as `long`) and

```
myarray.size(n)
```

that returns the size of the of the dimension `n` of the array (as `long`). The argument `n` must be an integer in the range $[0, \text{ndim} - 1]$.

`DynamicArrays` can be passed to function and can be returned by functions but, before a `DynamicArray` is returned the function, `prepare` must be called. Here is an example of a program that returns a `DynamicArray`.

```
// Program: example02.C
#include "MDP_Lib2.h"

DynamicArray<float,2> f(float x) {
    DynamicArray<float,2> a(2,2);
    a(0,0)=x;
    prepare(a); // IMPORTANT !!
    return a;
};

int main() {
    float x=f(Pi)(0,0);
    printf("%f\n", x);
    return 0;
};
```

It prints 3.141593. Note that `f()` returns a 2×2 Matrix. The following is a program that contains a function that takes a `DynamicArray` as argument

```
// Program: example03.C
#include "MDP_Lib2.h"

DynamicArray<float,2> f(float x) {
    DynamicArray<float,2> a(2,2);
    a(0,0)=x;
    prepare(a);
    return a;
};

void g(DynamicArray<float,2> a) {
    printf("%f\n", a(0,0));
};

int main() {
    DynamicArray<float,2> a;
    a=f(Pi);
    g(a);
    return 0;
};
```

The output of this program is again 3.1415. Note that when `f()` is assigned to `a`, the latter is automatically resized.

Bare in mind that even if the function `g()` does not take its arguments by reference the object `DynamicArray` contains a pointer to the memory where the real data are stored, therefore it is *as if* it was passed by reference. This does not generate confusion providing one follows the three safety rules.

2.4. Using Matrix

A `Matrix` object, say `M`, is very much like `DynamicArray<Complex,2>`. A `Matrix` can be declared either by specifying its size or not

```
Matrix M(r,c);    // r rows times c columns
Matrix M;         // a general matrix
```

Even if the size of a matrix has been declared it can be changed anywhere with the command

```
M.dimension(r,c); // r rows times c columns
```

Any `Matrix` is automatically resized, if necessary, when a value is assigned to it. The following lines

```
Matrix M(5,7), A(8,8);
M=A;
printf("%i,%i\n", M.rowmax(),M.colmax());
```

prints 8,8. The member functions `rowmax()` and `colmax()` return, respectively, the number of rows and columns of a `Matrix`.

The element (i, j) of a matrix `M` can be accessed with the natural syntax

```
M(i,j)
```

where i, j are integers. Moreover the class contains functions to perform standard operations among matrices:

```
+, -, *, /, +=, -=, *=, /=, inv, det, exp, sin, cos, log,
transpose, hermitiam, minor, identity,....
```

As an example a program to compute

$$\left[\exp \begin{pmatrix} 2 & 3 \\ 4 & 5i \end{pmatrix} \right]^{-1} \quad (1)$$

looks like this:

```
// Program: example04.C
#include "MDP_Lib2.h"
int main() {
    Matrix a(2,2);
    a(0,0)=2; a(0,1)=3;
    a(1,0)=4; a(1,1)=5*I;
    print(inv(exp(a)));
    return 0;
};
```

It is straightforward to add new functions copying the following prototype

```
Matrix f(const Matrix a, const Matrix b, ...) {
    Matrix M;
    // body,
    prepare(M);
    return M;
};
```

We repeat again the three safety rules one should always remember: one should call `prepare()` before returning a `Matrix`; one should not return a `Matrix` by reference; one should not initialize a `Matrix` using the copy constructor nor call the assignment operator of a `Matrix` argument.

Here is one more example of how to use the class `Matrix`:

```
// Program: example05.C
#include "MDP_Lib2.h"
Matrix cube(Matrix X) {
    Matrix Y;
    Y=X*X*X;
    prepare(Y);
    return Y;
};
int main() {
    Matrix A,B;
    A=Random.SU(3);
    B=cube(A)*exp(A)+inv(A);
    print(A);
    print(B);
    return 0;
};
```

This code prints on the screen a random $SU(3)$ matrix A and $B = A^3 e^A + A$. Some example statements are listed in Table 1. Note that the command

```
A=mul_left(B,C);
```

is equivalent but faster than

```
A=C*transpose(B);
```

because it does not involve allocating memory for the transposed of B .

As one more example of a powerful application of `DynamicArray` here is a program that defines a multidimensional `DynamicArray` of `Matrix` objects and pass it to a function.

Table 1

Examples of typical instructions acting on `Matrix` objects. A, B, C, D are assumed to be declared as `Matrix`; r, c as `int`; a, b may be any kind of number

| Example | C++ with MDP_Lib2.h |
|---------------------------------------|--------------------------------------|
| $A \in M_{r \times c}(\mathbf{C})$ | <code>A.dimension(r,c)</code> |
| A_{ij} | <code>A(i,j)</code> |
| $A = B + C - D$ | <code>A=B+C-D</code> |
| $A^{(ij)} = B^{(ik)} C^{(kj)}$ | <code>A=B*C</code> |
| $A^{(ij)} = B^{(jk)} C^{(ik)}$ | <code>A=mul_left(B,C)</code> |
| $A = aB + C$ | <code>A=a*B+C</code> |
| $A = a\mathbf{1} + B - b\mathbf{1}$ | <code>A=a+B-b</code> |
| $A = B^T C^{-1}$ | <code>A=transpose(B)*inv(C)</code> |
| $A = B^\dagger \exp(iC)$ | <code>A=hermitian(B)*exp(I*C)</code> |
| $A = \cos(B) + i \sin(B) * C$ | <code>A=cos(B)+I*sin(B)*C</code> |
| $a = \text{real}(\text{tr}(B^{-1}C))$ | <code>a=real(trace(inv(B)*C))</code> |
| $a = \det(B) \det(B^{-1})$ | <code>a=det(B)*det(inv(B))</code> |

```
// Program: example06.C
#include "MDP_Lib2.h"

DynamicArray<Matrix,3> initialize() {
    DynamicArray<Matrix,3> d(20,20,20);
    int i,j,k;
    for(i=0; i<20; i++)
        for(j=0; j<20; j++)
            for(k=0; k<20; k++) {
                d(i,j,k).dimension(2,2);
                d(i,j,k)(0,0)=k;
                d(i,j,k)(0,1)=i;
                d(i,j,k)(1,0)=j;
                d(i,j,k)(1,1)=k;
            };
    prepare(d);
    return(d);
}

DynamicArray<Matrix,3> f(DynamicArray<Matrix,3> c) {
    DynamicArray<Matrix,3> d(c.size(0),c.size(1),c.size(2));
    int i,j,k;
    for(i=0; i<c.size(0); i++)
        for(j=0; j<c.size(1); j++)
            for(k=0; k<c.size(2); k++)
                d(i,j,k)=sin(c(i,j,k));
    prepare(d);
    return(d);
};

int main() {
    DynamicArray<Matrix,3> a, b;
    a=initialize();
    b=f(a);

    int i=1, j=2, k=3;
    print(a(i,j,k));
    print(b(i,j,k));
    return 0;
};
```

a, b, c, d are 3D $20 \times 20 \times 20$ arrays of 2×2 matrices. The program prints

```
[ [ 3.000+0.000i  1.000+0.000i ]
  [ 2.000+0.000i  3.000+0.000i ] ]
[ [ 0.022+0.000i  -0.691+0.000i ]
  [ -1.383+0.000i  0.022+0.000i ] ]
```

Note that the instruction

```
d(i,j,k)=sin(c(i,j,k));
```

computes the $\sin()$ of the Matrix $c(i,j,k)$.

2.5. classRandom_generator

The random generator implemented in `Random_generator` is the Marsaglia random number generator described in Refs. [4,5] (the same generator is also used by the UKQCD collaboration in many large scale numerical simulations). It presents the nice feature that it can be initialized with a single long random number and its correlation time is relatively large compared with the standard random generator `rand()` of C++. To define a random generator, say `myrand`, and use the integer `seed` as seed one should simply pass `seed` to the constructor:

```
Random_generator myrand(seed);
```

For simple non-parallel applications one only needs one random generator. For this reason one `Random_generator` object called `Random` is automatically created and initialized with seed 0.

`Random_generator` contains some member variables for the seeds and four member functions:

- `plain()`; It returns a random float number in the interval $[0, 1)$ with uniform distribution.
- `gaussian()`; It returns a random float number x generated with a Gaussian probability $P(x) = \exp(-x^2/2)$.
- `distribution(float (*P)(float, void*), void* a)`; It returns a random number x in the interval $[0, 1)$ generated with a Gaussian probability $P(x, a)$, where a is any set of parameters pointed by void * a (the second argument passed to `distribution`). The distribution function P should be normalized so that $0 \leq \min P(x) \leq 1$ and $\max P(x) = 1$.
- `Random_generator::SU(int n)`; it returns a random Matrix in the group $SU(n)$ or $U(1)$ if the argument is $n = 1$.

Here is a simple program that creates one random number generator, generates a set of 1000 random Gaussian numbers and counts how many of them are in the range $[n/2, (n+1)/2)$ for $n = 0, \dots, 9$

```
// Program: example07.C
#include "MDP_Lib2.h"
int main() {
    Random_generator random;
    int i,n,bin[10];
    float x;
    for(n=0; n<10; n++) bin[n]=0;
    for(i=0; i<1000; i++) {
        x=random.gaussian();
        for(n=0; n<10; n++)
            if((x>=0.5*n) && (x<0.5*(n+1))) bin[n]++;
    };
    for(n=0; n<10; n++)
        printf("bin[%i] = %i\n", n, bin[n]);
    return 0;
};
```

Here is a program that computes the average of the sum of two sets of 1000 numbers, a and b , where a is generated with probability $P(a) = \exp(-(a - \bar{a})^2/(2\sigma))$ and b with probability $Q(b) = \sin(\pi b)$.

```
// Program: example08.C
#include "MDP_Lib2.h"

float Q(float x, void *a) {
    return sin(Pi*x);
};

int main() {
    Random_generator random;
    int i,n,bin[10],N=100;
    float a,b,average=0, sigma=0.3, a_bar=1;
    for(i=0; i<N; i++) {
        a=(sigma*random.gaussian()+a_bar);
        b=random.distribution(Q);
        average+=a+b;
        printf("average=%f\n", average/(i+1));
    };
    return 0;
};
```

For large N the output asymptotically approaches $a_bar+0.5=1.5$.

The algorithm for $SU(2)$ is based on map between $O(3)$ and $SU(2)$, realized by

$$\{\hat{a}, \alpha\} \rightarrow \exp(i\alpha\hat{a} \cdot \sigma) = \cos(\alpha) + i\hat{a} \cdot \sigma \sin(\alpha), \quad (2)$$

where $(\sigma^1, \sigma^2, \sigma^3)$ is a vector of Pauli matrices, $\hat{a} \in S^2$ is a uniform vector on the sphere and $\alpha \in [0, \pi)$ is a uniform rotation angle around that direction.

A random $SU(n)$ is generated using the well-known Cabibbo–Marinari iteration [2] of the algorithm for $SU(2)$.

2.6. classJackBoot

Suppose one has n sets of m different measured float quantities $x[i][j]$ (where $i = 0, \dots, n$ and $j = 0, \dots, m$) and one wants to compute the average over i , of a function $F(x[i])$. For example,

```
float x[n][m]={...}
float F(float *x) {
    return x[1]/x[0];
};
float result=0;
for(i=0; i<n; i++)
    result+=F(x[i])/n;
};
```

Then one may ask: what is the error on the result? In general there are two algorithms to estimate this error: Jackknife and Bootstrap [3]. They are both implemented as member functions of the class JackBoot. They work for any arbitrary function F .

A JackBoot object, let's call it `jbb`, is a sort of container for the data. After it has been filled it can be asked to return the mean of $F()$ and its errors. Here is an example of how it works:

```
JackBoot jbb(n,m);
```

```

jb.f=F;
for(i=0; i<n; i++)
    for(j=0; j<m; j++)
        jb(i,j)=x[i][j];
printf("Result          = %f\n", jb.mean());
printf("Jackknife error = %f\n", jb.j_err());
printf("Bootstrap error = %f\n", jb.b_err(100));

```

Note that

- The constructor of the class JackBoot takes two arguments: the first is the number of configuration; the second is the number of the quantities measured on each configuration.
- `jb.f` is the pointer to the function used in the analysis.
- `jb.mean()` returns the mean.
- `jb.j_err()` returns the Jackknife error.
- `jb.b_err()` returns the Bootstrap error. It takes as argument the number of Bootstrap samples. The default value is 200.

It is possible to declare arrays of JackBoot objects, but it is rarely necessary. It is simpler to declare different functions and repeat the analysis using the same JackBoot object assigning the pointer `JackBoot::f` to each of the functions at the time.

As another example consider the following program. It generates an array of 100 SU(6) matrices. For each matrix it computes trace and determinant, and returns the average of the ratio between the real part of the trace and the real part of the determinant (with its Jackknife and Bootstrap errors), and the average of the product of the real part of the trace and the real part of the determinant (with its Jackknife and Bootstrap errors)

```

// Program: example09.C
#include "MDP_Lib2.h"
#define n 100
float f1(float *x, float *a) { return x[0]/x[1]; };
float f2(float *x, float *a) { return x[0]*x[1]; };
int main() {
    Matrix A;
    JackBoot jb(n,2);
    int i;
    for(i=0; i<n; i++) {
        A=Random.SU(6);
        jb(i,0)=real(det(inv(A)));
        jb(i,1)=real(det(A));
    };
    jb.f=f1;
    printf("Result x[0]/x[1] = %f\n", jb.mean());
    printf("Jackknife error = %f\n", jb.j_err());
    printf("Bootstrap error = %f\n", jb.b_err(100));
    jb.f=f2;
    printf("Result x[0]*x[1] = %f\n", jb.mean());
    printf("Jackknife error = %f\n", jb.j_err());
    printf("Bootstrap error = %f\n", jb.b_err(100));
    return 0;
};

```

Note that any user defined function used by JackBoot (`f1` and `f2` in the example) has to take two arguments: an array of `float` (containing a set of measurements done on a single configuration) and a pointer to `void`. This is useful to pass some extra data to the function. The extra data must be passed to JackBoot by assigning the member variable

```
void* JackBoot::handle;
```

to it.

JackBoot has one more member function `plain(int i)`, where the call

```
/* define JackBoot jb(...) */
/* assign a value to the integer i */
jb.plain(i)
```

is completely equivalent to

```
/* define JackBoot jb(...) */
/* assign a value to the integer i */
float f(float *x, void *a) {
    return x[i];
};
jb.f=f
```

Using `plain` saves one from defining trivial functions for JackBoot.

3. Parallel tools

With a parallel program we mean a job constituted by many programs running in parallel (eventually on different processors) to achieve a global goal. Each of the running programs, part of the same job, is called a process. Different processes can communicate to each other by exchanging messages (in MPI) or by accessing each-other memory (in computers with shared memory). The latter case is hardware dependent and will not be taken in consideration. With the term “partitioning” we will refer to the way different variables (in particular the field variables defined on a lattice) are distributed among the different processes.

The most common and portable parallel protocol is Message Passing Interface (MPI). It is implemented on a number of different platforms, for example Unix, Linux, Solaris, Windows NT and Cray T3D/E.

When writing a program using MPI, one essentially writes one program for each processor (Multiple Instructions Multiple Data) and MPI provides the functions for sending/receiving information among them.

We present here an example of how MPI works, even if a detailed knowledge of MPI is not required to understand the rest of this paper.

Suppose one wants to compute $(5 * 7) + (4 * 8)$ in parallel using two processes. A typical MPI program to do it is the following:

```
// Program: example10.C
#include "stdio.h"
#include "mpi.h"
int main(int argc, char **argv) {
    int ME, Nproc;
    MPI_Status status;
    MPI_Init(&argc, &argv);
```

```

MPI_Comm_rank(MPI_COMM_WORLD, &ME);
MPI_Comm_size(MPI_COMM_WORLD, &Nproc);
if(ME==1) {
    int b;
    b=4*8;
    MPI_Send(&b,1,MPI_INT,0,45,MPI_COMM_WORLD);
};
if(ME==0) {
    int a,b;
    a=5*7;
    MPI_Recv(&b,1,MPI_INT,1,45,MPI_COMM_WORLD,&status);
    printf("%i\n", a+b);
};
MPI_Finalize();
};

```

ME is a variable that contains the unique identification number of each running process and Nproc is the total number of processes running the same code. Since different processes have different values of ME they execute different parts of the program (if (ME==n) ...).

In the example $(5 * 7)$ is computed by process 0 while, at the same time $(4 * 8)$ is computed by process 1. Then process 1 sends (MPI_Send) its partial result to process 0, which receives it (MPI_Recv) and prints out the sum.⁵

MPI instructions can be classified in

- initialization functions (like MPI_Init, MPI_Comm_rank, MPI_Comm_size),
- one to one communications (like MPI_Send, MPI_Recv),
- collective communication (where for example one variable can be broadcasted by one process to all the others).

It is not a purpose of this paper to explain how MPI works therefore we refer to [6].

3.1. mpi on top of MPI

MPI is not Object Oriented and it is so general that MPI function calls usually require many arguments. This makes it very easy to do mistakes when programming with MPI.

For this reason the class `mpi_wormhole_class`, a wrapper to MPI, has been created. It has only one global representative object called `mpi`. We believe that `mpi` is easier to use than MPI in the context of lattice simulation.

`mpi` is not intended as a substitute or an extension to MPI, in fact only a subset of the features of MPI are implemented in `mpi` and occasionally one may need explicit MPI calls. Its main purpose is to build an intermediate programming level between MPI and `generic_lattice/generic_field`, which constitute the very essence of MDP 1.2. There are some advantages of having this intermediate level, for example:

- It allows one to compile the code without linking with MPI. In this case `mpi` substitutes some dummy functions to the MPI calls. This allows one to compile and run any parallel program on a single processor computer even if MPI is not installed, and it is also important for testing and debugging the code.
- In case one wants to run MDP 1.2 on a parallel machine that does not support MPI but does support a different protocol, one does not have to rewrite everything but only minor modifications to the class `mpi_wormhole_class` will be necessary.

⁵ Needless to say that in such a simple program the parallelization is completely useless because the message passing takes more time than the computation of the whole expression!

3.2. mpi and wormholes

From now on we will use here the word “wormhole”, in its intuitive sense, to represent the virtual connections between the different processes. One can think of each process to have one wormhole connecting it with each of the other processes. Each wormhole has a number, the number of the process connected to the other end. Process X can send something to process Y just putting something into its wormhole n.Y. Process Y has to get that something from its wormhole n.X. `mpi` is the global object that allows the user to access these wormholes (put/get stuff to/from them). A schematic representation of a 3 processes job is shown in Fig. 1.

Using `mpi`, instead of `MPI`, program `example10.C` becomes:

```
// Program: example11.C
#define PARALLEL
#define Nproc 2
#include "MDP_Lib2.h"
#include "MDP_MPI.h"
int main(int argc, char **argv) {
    mpi.open_wormholes(argc,argv);
    if(ME==1) {
        int b;
        b=4*8;
        mpi.put(b,0);
    }
```

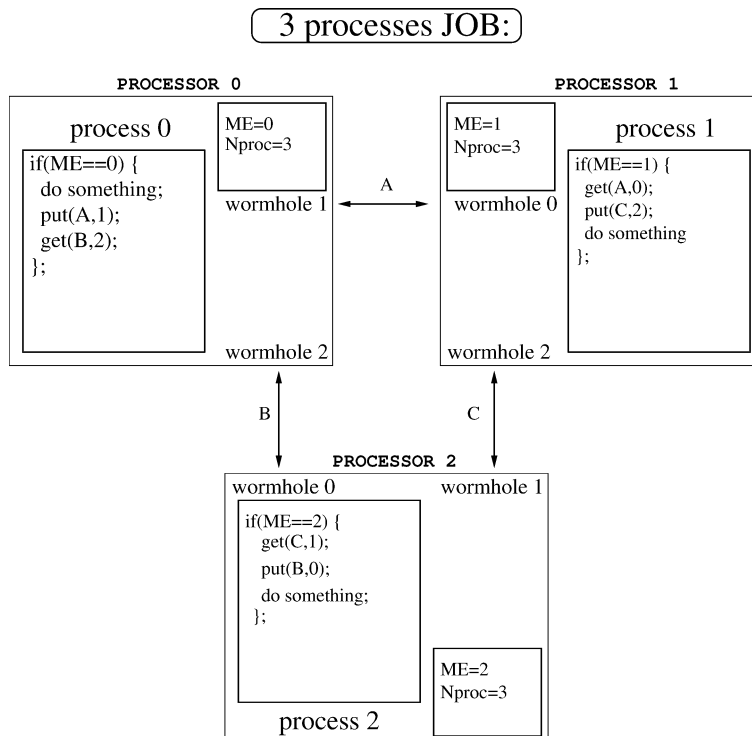


Fig. 1. Example of a 3 processes jobs. ME and Nproc are the global variables that characterize the different processes running (eventually) on different processors. The term wormhole refers to the virtual connection between each couple of processes.


```

};
if(ME==0) {
    int a,b;
    a=5*7;
    mpi.get(b,1);
    printf("%i\n" , a+b);
};
mpi.close_wormholes();
};

```

The most important things to notice are:

- (1) A constant (without any value assigned), `PARALLEL`, is defined on top. It tells the compiler to use the MPI functions. When it is omitted some dummy functions are substituted to the MPI calls and communications are skipped. This is useful for testing the behavior of a particular process in a multiprocess job without actually going parallel.
- (2) `Nproc` is the total number of processes. If it is not defined by the user, `Nproc` it is defined automatically (as a variable) when `MDP_MPI.h` is included and its value is determined at runtime.
- (3) Two libraries `MDP_Lib2.h` and `MDP_MPI.h` are included. The standard headers (`stdio.h`, `math.h`, `complex.h`, `iostream.h`) are included by `MDP_Lib2.h`.
- (4) `ME` is a global macro meaning `mpi.my_id`, i.e. the identification number of the running process. Its value is assigned by the call to `mpi.open_wormholes` when running with `PARALLEL`. Otherwise it has to be specified by the user after that call.
- (5) All of the initialization functions have been replaced by a single one:
`void mpi_wormhole_class::open_wormholes(int&,char**);`
 It is compulsory even if the program does not run in parallel. It takes as input the same arguments of `main()`.⁶
- (6) `MPI_Send` and `MPI_Recv` have been replaced by `mpi.put` and `mpi.get` respectively. They check automatically the size and type of their arguments.
- (7) At the end of the program one has to `close_wormholes`. Actually if one forgets about this the destructor of `mpi` does take care of it.

The programmer is forced to use the names `Nproc` and `ME` for the total number of processes and the process identification number, respectively. They are keywords for `MDP 1.2`.

3.2.1. `get()` and `put()`

The functions `put/get` have the following syntax

```

template <class T> mpi_wormhole_class::put(T&,int);
template <class T> mpi_wormhole_class::put(T&,int,mpi_request&);
template <class T> mpi_wormhole_class::get(T&,int)

```

The first argument (passed by reference) is the object to be sent (`put`) or where to store the received one (`get`). The second argument is the destination (`put`) or the source (`get`). Forget about `mpi_request` for the moment, it is optional.

All the variables related with message passing are members of `mpi`. In this way variables that are not explicitly necessary remain hidden inside it (for example the communicator and the communication tag).

⁶ This is due to the fact that when running in parallel (using the `runmpi` command) a script submits the same program to the different processors with different command parameters, which are passed as arguments to `main()`. This is how a process knows its own identification number: `ME`.

The functions `put` and `get` that replaced `MPI_Send` and `MPI_Recv` are Object Oriented in the sense that they can be used to send any kind of object.⁷ The size and the tag of the communication are automatically computed and do not have to be specified by the programmer. For example with the same syntax a Complex number can be put/get instead of an integer. The only price paid for this is that a process, X, cannot send more than one object to another process, Y, until the first object has been received. This is not such a price because in the kind of computations we are interested in, we want to minimize the number of put/get and we want to pack as much data as possible in each single communication, instead of making many small ones.

To avoid confusion, if the third argument of `put()` is not specified this function will not return until the send process has been completed (synchronous send).⁸ Sometimes one wants the `put()` command to return even if the send is not completed, so that the process is free to do more computations, and eventually check at a later stage if the previous `put()` has completed its task (asynchronous send). One typical case is when the object to be send was dynamically allocated and it cannot be deallocated until the communication is concluded. The right way to implement this send is writing a code like the following:

```
/* mystruct is any built in variable
   or user defined structure */
mystruct *myobj=new mystruct;
mpi_request request;
put(*myobj,destination,request);
/* do something else */
mpi.wait(request);
delete myobj;
```

The `wait()` member function of `mpi` stops the process until the `put()` command has terminated its task. Its only argument is the object belonging to the class `mpi_request` that was passed to the corresponding `put()`. The function `wait()` can also take a pointer to an array of requests and wait for all of them. The general definition of this function is

```
void mpi_wormhole_class::wait(mpi_request&);
void mpi_wormhole_class::wait(mpi_request*, int);
```

the second argument is the length of the array.

It is also possible to put/get arrays of objects by using the overloaded functions

```
template <class T> mpi_wormhole_class::put(*T,long,int)
template <class T> mpi_wormhole_class::put(*T,long,int,mpi_request&)
template <class T> mpi_wormhole_class::get(*T,long,int)
```

where the first argument is a pointer to the first element of the array, the second is the length, the third is the destination/source.

3.2.2. Put/get matrices

We have said that with the put/get commands it is not possible to pass objects that contain a pointer to dynamically allocated memory and this is the case for a `Matrix` object. This does not mean that one cannot put/get a `Matrix`. The correct way to send a `Matrix` is

⁷ Assuming it does not contain a pointer to dynamically allocated memory.

⁸ Actually in `mpi`, `put` an synchronous send is internally implemented using `MPI_Isend + MPI_Wait`. This is because `MPI_Send` is ambiguous in some cases: the same program can have different behavior depending on the size of the objects that are passed. Our implementation avoids any ambiguity.

```
Matrix A;
/* do something with A */
mpi.put(A.address(),A.size(),destination);
```

The correct way to receive a Matrix is

```
Matrix B;
/* dimension B as A */
mpi.get(B.address(),B.size(),destination);
```

It is extremely important to stress that when one puts/gets a matrix one only sends/receives its elements. Therefore (in the last example) B must have the same dimensions of A at the moment get is called, otherwise one gets wrong results.

3.2.3. Global communications

As an example of global communication we will consider the following code

```
// Program: example12.C
#define PARALLEL
#define Nproc 2
#include "MDP_Lib2.h"
#include "MDP_MPI.h"
int main(int argc, char **argv) {
    mpi.open_wormholes(argc,argv);
    int a;
    if(ME==1) {
        a=4*8;
        mpi.add(a);
    };
    if(ME==0) {
        a=5*7;
        mpi.add(a);
        printf("%i\n",a);
    };
    mpi.close_wormholes();
};
```

The member function:

```
void mpi_wormhole_class::add(float&);
```

sums the first argument passed by all the processes. In this way each process knows the result of the sum operation.

To sum an array of float the corresponding sum for arrays of float is implemented as:

```
void mpi_wormhole_class::add(float*, long);
```

(the second argument is the length of the array). The same function add also works for double, long, int, Complex and Matrix and arrays of these types. To add Complex numbers:

```
Complex x;
mpi.add(x);
```

To add Matrix objects

```
Matrix A;
mpi.add(A);
```

Some more member functions of `mpi` for collective communications are:

- `barrier()` which sets a barrier and all processes stop at that point until all the processes reach the same point.
- `abort()` which forces all the processes to abort.
- `template<class T> broadcast(T a, int p)` which broadcast the object `a` (belonging to an arbitrary class `T`) of process `p` to all other processes.
- `template<class T> broadcast(T *a, long n, int p)` which broadcast the array `a` of objects `T` of process `p` to all other processes. `n` here is the number of elements of the array.

Here is an example of how to use these collective communications

```
// Program: example13.C
#define PARALLEL
#define Nproc 2
#include "MDP_Lib2.h"
#include "MDP_MPI.h"
int main(int argc, char **argv) {
    mpi.open_wormholes(argc,argv);
    int i,j;
    for(i=0; i<5; i++) {
        if(ME==0) j=i;
        else      j=0;
        if(i%2==0) mpi.barrier();
        mpi.broadcast(j,0);
        printf("I am process %i, i=%i, j=%i\n", ME, i,j);
    };
    mpi.close_wormholes();
};
```

This programs prints

```
I am process 0, i=0, j=0
I am process 0, i=1, j=1
I am process 0, i=2, j=2
I am process 0, i=3, j=3
I am process 0, i=4, j=4
I am process 1, i=0, j=0
I am process 1, i=1, j=1
I am process 1, i=2, j=2
I am process 1, i=3, j=3
I am process 1, i=4, j=4
```

4. Introducing lattices and fields

4.1. class generic_lattice

By a lattice we mean a generic set of points and a set of functions to move from one point to another. In MDP 1.2 a lattice is implemented as any subset of a regular n -dimensional grid with a given (arbitrary) topology.

A lattice of such a kind is implemented as an object belonging to the class `generic_lattice` using the command:

```
generic_lattice mylattice(ndim,mybox,mypartitioning,
                          mytopology,seed,next_next);
```

or the command

```
generic_lattice mylattice(ndim,ndir,mybox,mypartitioning,
                          mytopology,seed,next_next);
```

where:

- `mylattice` is a user defined variable that will contain the actual lattice structure.
- `ndim` is the dimension of the basic regular grid.
- `ndir` is the number of directions. It can be omitted and, by default, it is assumed to be equal to `ndim`.
- `mybox` is a user defined `ndim`-ensional array of `int` that contains the size of the basic regular grid.
- `mypartitioning` is a user defined function that for each site of the basic grid returns the number of the process that stores it (or it returns `NOWHERE` if the point has to be excluded from the lattice). If `mypartitioning` is omitted all sites of the original grid (specified by `mybox`) will be part of the final lattice, and they will be equally distributed between the processes according with the value of coordinate 0 of each point.⁹
- `mytopology` is a user defined function that for each site of the final (remaining) grid, and for each dimension in the `mydim`-ensional space, returns the coordinates of the neighbor sites in the up and down direction. If `mytopology` is omitted the lattice will, by default, have the topology of an `ndim`-ensional torus, i.e. the same of the basic grid plus periodic boundary conditions.
- `seed` is an integer that is used to construct a site dependent seed to initialize the random generators associated to each lattice sites. If omitted it is assumed to be 0.
- `next_next` is an integer that may have only three values (1, 2 or 3). It essentially fixes the thickness of the boundary between different processes. If omitted the default value is 1. It's use will not be discussed further.

For the moment we will restrict ourselves to the study of lattices with the basic (torus) topology, postponing the rules concerning the specifications for `mypartitioning` and `mytopology`.

Here is a short program to create a 2D 8^2 lattice (but no field yet associated to it) running in parallel on 4 processes.

```
// Program: example14.C
#define PARALLEL
#define Nproc 4
#include "MDP_Lib2.h"
#include "MDP_MPI.h"
int main(int argc, char **argv) {
    mpi.open_wormholes(argc,argv);
```

⁹ For example, consider a lattice of size 8 in the 0 direction distributed on 4 processes. Process 0 will contain sites coordinate 0 equal to 0 and 1. Process 1 will contain sites coordinate 0 equal to 2 and 3. An so on.

```

int mybox[]={8,8};
generic_lattice mylattice(2,mybox);
mpi.close_wormholes();
};

```

When it runs (since `Nproc` is set to 4) the lattice will be automatically subdivided into 4 sublattices of size $\{2, 8\}$. `mylattice` contains information about the sites and how they are distributed among the different processes. Moreover the different processes automatically communicate to each other this information so that each of them knows which local sites will have to be passed to the neighbor processes. All the communications to establish the topology are done only once and the information are inherited by all the fields defined on `mylattice`. The program produces the following output

```

PROCESS 0 STARTING
PROCESS 2 STARTING
PROCESS 1 STARTING
PROCESS 3 STARTING
=====
Starting [ Matrix Distributed Processing ] ...
This program is using the packages: MDP_Lib2 and MDP_MPI
Created by Massimo Di Pierro (mdp@FNAL.GOV) version 17-11-1999
=====
Going parallel ... YES

Initializing a generic_lattice...
Communicating...
Initializing random per site...
Done. Let's begin to work!
PROCESS 1 ENDING AFTER 0.010 sec.
PROCESS 2 ENDING AFTER 0.008 sec.
PROCESS 3 ENDING AFTER 0.011 sec.
=====
Fractional time spent in communications by processor 0 is 0.03
Fractional time spent in communications by processor 1 is 0.04
Fractional time spent in communications by processor 2 is 0.02
Fractional time spent in communications by processor 3 is 0.03
Ending this program.
Any runtime error below this point means mess with mpi and/or
deallocation of memory.
=====
PROCESS 0 ENDING AFTER 0.012 sec.

```

Note that the output automatically includes the execution time (using the wall clock) and the fractional time spent in communication by each processor (0.03 means 3%). The latter is computed by measuring the total time spent inside the communication function `update()`.

Two other member functions of `generic_lattice` are

```

long generic_lattice::global_volume();
long generic_lattice::local_volume();

```

The first return the size of the total lattice volume (i.e. the total number of sites), and the second returns the size of the portion of volume stored by the calling process.

The member function

```
long generic_lattice::size();
```

returns the total volume of the box containing the lattice and

```
long generic_lattice::size(int mu);
```

returns the size, in direction μ , of the box containing the lattice.

4.2. class site

To access a site one can define a `site` variable using the syntax

```
site x(mylattice);
```

This tells the compiler that `x` is a variable that will contain the coordinate of a site and a reference to the lattice `mylattice` on which the point is defined.

Let's look at a modification of the preceding code that create a lattice and, for each site `x`, prints out the identification number of the process that stores it:

```
// Program: example15.C
#define PARALLEL
#define Nproc 5
#include "MDP_Lib2.h"
#include "MDP_MPI.h"
int main(int argc, char **argv) {
    mpi.open_wormholes(argc,argv);
    int mybox[]={10,10};
    generic_lattice mylattice(2,mybox);
    site x(mylattice);
    forallsites(x)
        printf("Site=(%i,%i) is stored by %i\n", x(0), x(1), ME);
    mpi.close_wormholes();
};
```

Note that

```
x(mu)
```

returns the coordinate μ of `x`.

This simple program allows one to print a map of the lattice. The output of this code depends on the number of processors on which it is running.¹⁰ The loop command

```
forallsites(x) /* do something with x */
```

spans first all the even sites and then all the odd sites.

¹⁰ The buffers of the different processors are copied to the standard output at a random time partially scrambling the output of the different processes. For this reason it is in general a good rule to print only from process `ME==0`.

It is possible to set the value of a `site` variable `x` to a particular site of given coordinates $((3, 7)$, for example) using the command

```
x.set(3,7);
```

but one must be careful and execute this statement only on the process which contains the site $(3, 7)$ or, eventually, in a process that contains a neighbor of this site. In this case, in fact, the process will also contain a copy of the original site therefore it will be able to access the location. By default each process will store its own sites plus a copy of next-neighbor sites in each direction and a copy of next-next neighbor sites (moving in two different directions).¹¹ They will be referred to as boundary sites. Boundary sites are different for different processes.

It is possible to check if a site is a local site in the following way:

```
x.set(3,7)
if(x.is_in()) /* then do something */
```

The member function `is_in()` returns TRUE if the site is local. A better way to code it would be

```
if(on_which_process(3,7)==ME) {
    x.set(3,7);
    /* do something */
}
```

Other member functions of `site` are

- `is_in_boundary()` returns TRUE if a site is in the boundary of that process.
- `is_here()` returns TRUE is the site `is_in()` or `is_in_boundary()`.
- `is_equal(int,int,int...)` returns TRUE if the site is equal to the site specified by the coordinates that are arguments of `is_equal()` (`x0,x1,x2,...`). **Accessing a site that `!is_here()` crashes the program.**

To move from one site to another is quite simple:

```
x=x+mu; // moves x up in direction mu (integer) of one step
x=x-mu; // moves x down in direction mu (integer) of one step
```

Note that `mu` is an integer and in general

$$x + \mu \neq x - \mu \quad (3)$$

even when `mu` is zero.

Here is a test program to explore the topology:

```
// Program: example16.C
#define PARALLEL
#define Nproc 2
#include "MDP_Lib2.h"
#include "MDP_MPI.h"
int main(int argc, char **argv) {
    mpi.open_wormholes(argc,argv);
    int mybox[]={10,10};
```

¹¹ This is required, for example, by the clover term in QCD.


```

generic_lattice mylattice(2,mybox);
site x(mylattice);
int mu=0;
if(ME==0) {
    x.set(0,0);
    do {
        printf("x=(%i,%i)\n", x(0), x(1));
        if(x.is_in_boundary()) error("I found the boundary");
        x=x+mu;
    } while(TRUE);
};
mpi.close_wormholes();
};

```

This program stops after 5 iterations if `Nproc==2`, after 4 if `Nproc==3`, after 3 if `Nproc==4`, after 2 if `Nproc==5` and so on. It will never stop on a single process job (`Nproc==1`) because of periodic boundary conditions. Since all sites with the same `x(0)`, by default, are stored in the same process, if one moves in direction `mu=1` the job will never stop despite the number of processes.

4.3. class generic_field

A `generic_field` is the simplest field one can define on a given `generic_lattice`.

Suppose, for example, one wants to associate a given structure, `mystruct`, to each site of `mylattice` (in the example a structure that contains just a float) and initialize the field variable to zero:

```

// Program: example17.C
#define PARALLEL
#define Nproc 2
#include "MDP_Lib2.h"
#include "MDP_MPI.h"
struct mystruct {
    float value; /* or any other structure */
};
int main(int argc, char **argv) {
    mpi.open_wormholes(argc,argv);
    int mybox[]={10,10};
    generic_lattice      mylattice(2,mybox);
    generic_field<mystruct> myfield(mylattice);
    site x(mylattice);
    forallsites(x)
        myfield(x).value=0;
    myfield.update();
    mpi.close_wormholes();
};

```

The command

```
generic_field<mystruct> myfield(mylattice);
```

defines `myfield`, a field of `mystruct` on the sites of `mylattice`. The command `myfield(x)` returns by reference the structure associated to the site `x`.

The function

```
void generic_lattice::update(int,int,int)
```

is the most important of all. For the moment we will restrict to that case when it is called with no arguments. Its task is to perform all the communications necessary in order for each process to get an updated copy of the boundary sites stored by a neighbor process.

After the loop all sites that are local (`forallsites`) are initialized, but sites that are in boundary (since they are copies of sites initialized on a different process) still contain “random” numbers. One way to initialize also the sites in the boundary and avoid some time consuming communications is replacing the lines

```
forallsites(x)
    myfield(x).value=0;
mylattice.update();
```

with

```
forallsitesandcopies(x) myfield(x).value=0;
```

(Note that the command `forallsitesandcopies` only works for a local expression that does not call the local random generator.)

It is also possible to loop on all the sites of a given parity, for example, EVEN (or ODD), in a slow way

```
int parity=EVEN;
forallsites(x)
    if(x.parity()==parity)
        /* then do something */
```

or in a fast way

```
int parity=EVEN;
forallsitesofparity(x,parity)
    /* do something */
```

The second expression is faster because sites with the same parity are stored contiguously in memory.

4.3.1. Local random generator

Once a `generic_lattice` is declared one random generator per site is automatically created and initialized with a function of the site coordinates. The random generator (and its member functions) of site `x` of `mylattice` can be accessed (by reference) with the command

```
Random_generator& generic_lattice::random(site)
```

Program `example17.C` can be modified so that `myfield` is initialized, for example, with a gaussian random number:

```
// Program: example18.C
#define PARALLEL
#define Nproc 2
#include "MDP_Lib2.h"
```

```

#include "MDP_MPI.h"
struct mystruct {
    float value; /* or any other structure */
};
int main(int argc, char **argv) {
    mpi.open_wormholes(argc,argv);
    int mybox[]={10,10};
    generic_lattice      mylattice(2,mybox);
    generic_field<mystruct> myfield(mylattice);
    site x(mylattice);
    forallsites(x)
        myfield(x).value=mylattice.random(x).gaussian();
    myfield.update();
    mpi.close_wormholes();
};

```

4.3.2. Accessing a lattice from a field

It is always possible to access a `generic_lattice` from its field using `lattice()`, a member function of `generic_field` that returns by reference the `generic_lattice` on which the field is defined.

For example, in the preceding program one could substitute

```
myfield(x).value=mylattice.random(x).gaussian();
```

with

```
myfield(x).value=myfield.lattice().random(x).gaussian();
```

and get the same result. This is useful because one can avoid passing a `generic_lattice` object to a function of a `generic_field`.

4.3.3. More on update()

In some cases it may be useful to have a field that contains `n` `mystruct` variables at each site.

A field of this type can be defined with the command

```
generic_field<mystruct> myfield(mylattice,n);
```

Consider, for example, the simple case of a tensor field $T_{\mu}^{ij}(x)$, where $i, j \in \{0, \dots, 5\}$, $\mu \in \{0, \dots, 3\}$. It can be defined with

```
generic_field<Complex[4][5][5]> T(mylattice);
```

and its elements can be accessed with the natural expression:

```
T(x)[mu][i][j]
```

where `mu`, `i`, `j` are integers.

Alternatively the tensor T_{μ}^{ij} can be defined with the command

```
generic_field<Complex[5][5]> T(mylattice,4);
```

and its elements can be accessed with the expression:

```
T(x,mu)[i][j]
```

Sometimes the second choice is better for the following reason: In many algorithms one often needs to update all the boundary site variables having a given a parity and a given μ . One wants to do the update in one single communications. The `update()` function allows one to do exactly this:

```
int parity=EVEN, int mu=3;
T.update(parity,mu);
```

4.3.4. operator()

Some care is required when declaring a `generic_field`. The structure associated to the sites cannot contain a pointer to dynamically allocated memory, therefore it cannot contain a `Matrix` (or a `DynamicArray`) object. This is not a limitation since the structure associated to the sites can be a multidimensional array and one can always map it into a `Matrix` by creating a new field, inheriting `generic_field` and redesigning `operator()`.

In this subsection we explain how to use `Matrix` to build and interface to the field variables.

Let's go back to the example of a tensor field $T_{\mu}^{ij}(x)$. One can define it as

```
class mytensor: public generic_lattice<Complex[5][5]> {};
mytensor T(my lattice,4);
```

Instead of accessing its elements as `Complex` numbers one may want to access the two-dimensional array as a `Matrix` object. This is done by overloading the `operator()` of the class `generic_field`:

```
Matrix myfield::operator() (site x, int mu) {
    return Matrix(address(x,mu),5,5);
};
```

where `address()` is a member function of `generic_lattice` that returns the location of memory where the field variables at (x, μ) are stored. After this definitions one can simply access the field using the expression:

```
T(x,mu)
```

that now returns a `Matrix`.

Now `mytensor T` looks like a field of 5×5 matrices and its elements can be used as discussed in Section 2. For example, one can initialize the tensor using random `SU(5)` matrices generated independently by the local random generators associated to each site

```
forall sites(x)
    for(mu=0; mu<4; mu++)
        T(x,mu)=T.lattice().random(x).SU(5);
```

or print a particular element

```
x.set(3,5);
if(x.is_in()) print(T(x,2));
```

4.3.5. A few derived fields

Following the directives of the last subsections four more basic fields are implemented in MDP 1.2.

- `Matrix_field`: a field of `Matrix`;
- `NMatrix_field`: a field of n `Matrix`;
- `Vector_field`: a field of vectors (a vector is seen as a `Matrix` with one single column);

- `NVector_field`: a field of n vectors;

To explain their usage we present here a few lines of code that define a field of 3×4 matrices, `A`, a field of 4-vectors, `u`, and a field of 3-vectors, `v`; then for each site compute

$$v(x) = A(x)u(x). \quad (4)$$

This can be implemented as

```
Matrix_field A(mylattice,3,4);
Vector_field u(mylattice,4);
Vector_field v(mylattice,3);
/* assign values to the fields */
forallsites(x) v(x)=A(x)*u(x);
```

and its generalization to

$$v_i(x) = A_i(x)u_i(x) \quad (5)$$

for i in the range $[0, N - 1]$

```
int N=10; // user defined variable
NMatrix_field A(mylattice,N,3,4);
NVector_field u(mylattice,N,4);
NVector_field v(mylattice,N,3);
/* assign values to the fields */
forallsites(x) for(i=0; i<N; i++) v(x,i)=A(x,i)*u(x,i);
```

4.4. Input/output

The class `generic_field` contains two I/O member functions:

```
void generic_lattice::save(char[],int,int);
void generic_lattice::load(char[],int,int);
```

They take three arguments

- The `filename` of the file to/from which to save/load the data
- The identification number of the master process that is will physically perform to the I/O
- The size (in sites) of the buffer used by the master process to communicate with the other processes.

The last two arguments are optional and by default the master process is process 0 and the buffer size is 1024 sites. The identification number of the master process has to correspond to the process running on the processor that is physically connected with the I/O device where the file `filename` is. The size of the buffer will not affect the result of the I/O operation but may affect the speed (which should increase with the size of the buffer). If the size is too big there may be an “out of memory” problem (that usually results in obscure error messages when compiling with MPI). Note that the I/O functions `save/load` have to be called by all processes (not just the master one). In fact all processes which contain sites of the lattice are involved in the I/O by exchanging informations with the master.

As an example of I/O we consider the case of a `save()` operation. Each of the processes arranges the local field variables to be saved into packets and sends the packets to the process that performs the I/O (we assume it is process 0). Process i arranges its site into packets ordered according with the global parameterization and sends one packet at the time to process 0. In this way process 0 only receives one packet at the time for each of the processes and already finds the sites stored in the correct order so that it does not need to perform a seek. Process

0 saves the sites according to the global parameterization (which is independent from the lattice partitioning over the parallel processes). Its only task is to sweep all the sites and, for each of them, pick the corresponding field variables from the packet sent by the process that stored it.

We believe this is the most efficient way to implement a parallel I/O for field variables. As a particular case, assuming process 0 has enough memory, one could set the size of the buffer equal to the maximum number of sites stored by a process. In this case the parallel I/O would be done performing only one communication per process.

As an example of efficiency: a field as large as 100 MB can be read and distributed over 8 500 MHz PentiumIII PCs in parallel (connected with Ethernet) in a few of seconds.

The save/load function are inherited by every field based on `generic_field`. For example:

```
/* define mylattice */
class myfield: public generic_field<Complex[7][3]> {};
myfield F(mylattice);
/* do something */
F.save("test.dat");
F.load("test.dat");
```

If a field contains member variables other than the structure at the sites, these member variables are ignored by save/load. The only variables that are saved together with the field are:

- the number of dimensions,
- the size of the box containing the lattice,
- the total number of sites belonging to the lattice,
- the size in bytes of the structure at each lattice site,
- a code to detect the endianness of the computer that saved the data,
- the time and date when the file was created.

We also provide a small program, `inspect.C`, that can open a file and extract this information.

It is always possible to add any other information to the bottom of the file (in binary or ASCII format) without compromising its integrity.

4.5. The internal representation of a lattice

A lattice is a collection of sites and, for sake of simplicity, from now on we will refer to the lattice defined by

```
int ndim=2;
int mybox[]={6,6};
int mypartitioning(int x[], int ndim, int nx[]) {
    if(x[0]<3) return 0;
    if(x[1]<4) return 1;
    return 2;
};
generic_lattice mylattice(ndim,mybox,mypartitioning);
```

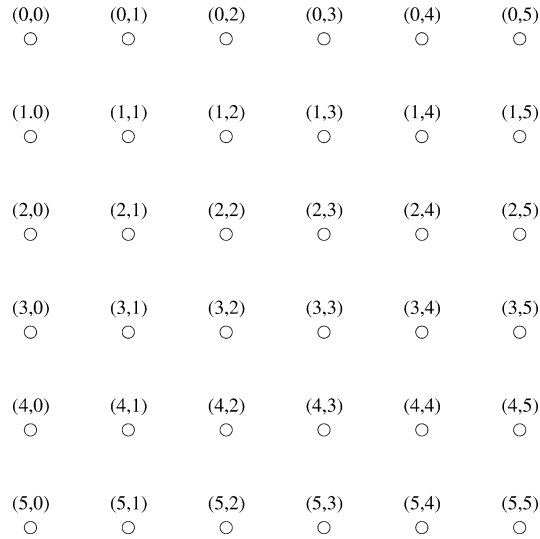
This lattice is represented in Fig. 2 and partitioned as in Fig. 5.

To speed up the communication process MDP 1.2 uses different parametrizations for labelling the sites:

- the global coordinates (as shown in Fig. 2),
- a global index (as shown in Fig. 3),
- a local index (as shown in Fig. 8 from the point of view of process 0),

and it provides functions to convert one parameterization into another.

The global parameterization is quite natural. For example: the point $x[] = \{3, 2\}$ has a global index given by $x[1]*mybox[0]+x[0]$; The two global parametrizations have a one to one correspondence. The local

Fig. 2. Example of a 6×6 grid. The points are labelled by their coordinates.Fig. 3. Example of a 6×6 grid. The points are labelled by their global index.

parameterization is more complicated and it represents the order in which each process stores the sites in the memory.

When the lattice is defined the constructor of the class `generic_lattice` spans all the sites using the global coordinates (Fig. 2) and it assigns them the global index (Fig. 3) and a parity (Fig. 4). Then each process checks which sites are local sites (Fig. 5). After the topology is assigned to the lattice (Fig. 6), each process identifies which sites are neighbors of the local sites (Fig. 7 for process 0). At this point each process has all the information needed to build a local parameterization for the sites. All even sites belonging to process 0 are labeled with a progressive number; then all the odd sites from process 0 are labelled in progression. The same procedure is repeated by each

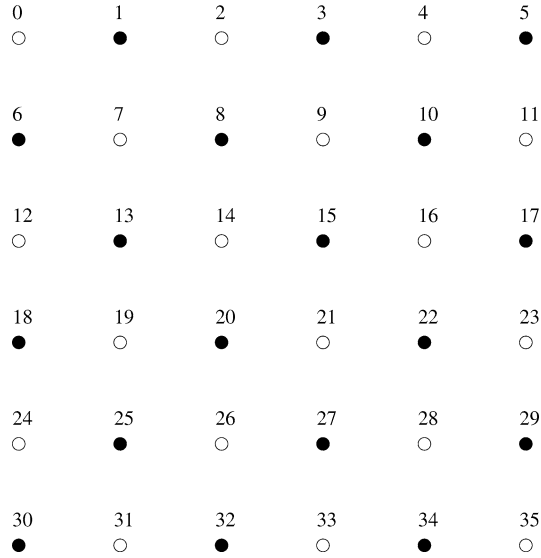


Fig. 4. Example of a 6×6 grid. The points are labelled by their global index. Points with even (white) and odd (black) parity are distinguished.

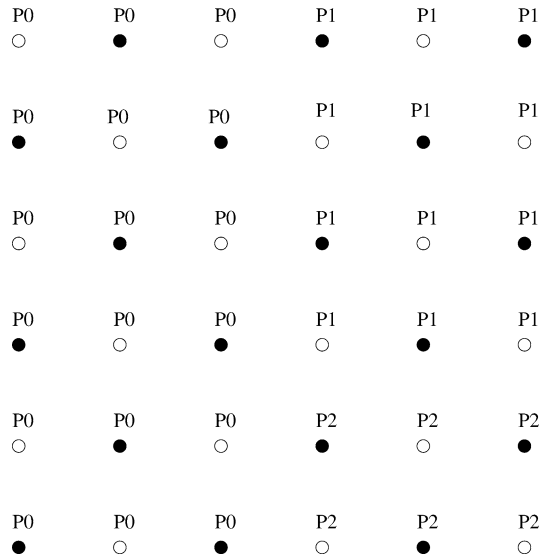


Fig. 5. Example of a 6×6 grid distributed on 3 processors (according with some user defined partitioning). Each point is labelled by the identification number of the process that stores it.

process for each process (including itself). The result (for the case in the example) is shown in Fig. 8 (from the point of view of process 0).

When a field is associated to the lattice the field variables are stored by each process according with the local parameterization. This guarantees that:

- Boundary sites that have to be copied from the same process are stored continuously in the memory. This minimizes the need for communications.

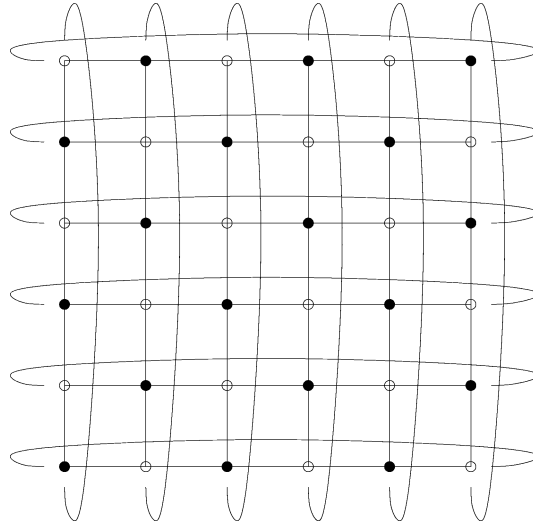


Fig. 6. Example of a 6×6 grid endowed with the default (torus) topology. When the topology is assigned to the grid, it is promoted to the rank of lattice.

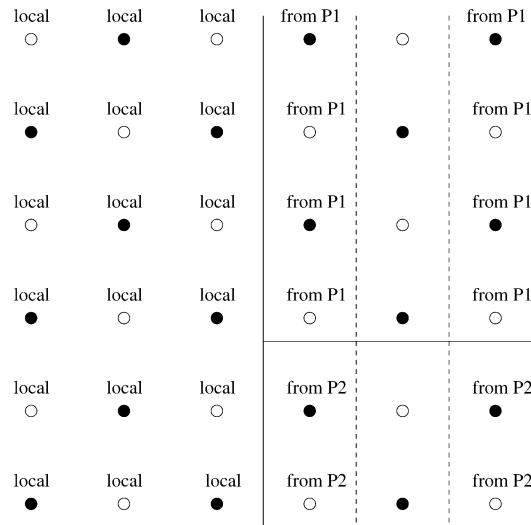


Fig. 7. Example of a 6×6 lattice from the point of view of process 0. Process 0 knows which sites are local, which non-local sites are neighbors of the local sites and from which process to copy them.

- Boundary sites copied from the same process (and the local sites as well) of given parity are also stored continuously in memory. This speeds up loops on sites of given parity.
- When a process X copies a neighbor site from process Y it does not need to allocate a buffer because the data can be received directly in the memory location where it is normally stored.

Given a site variable `x` on `mylattice` one can ask for the `mu` coordinate using the syntax

`x(mu)`

or the global index

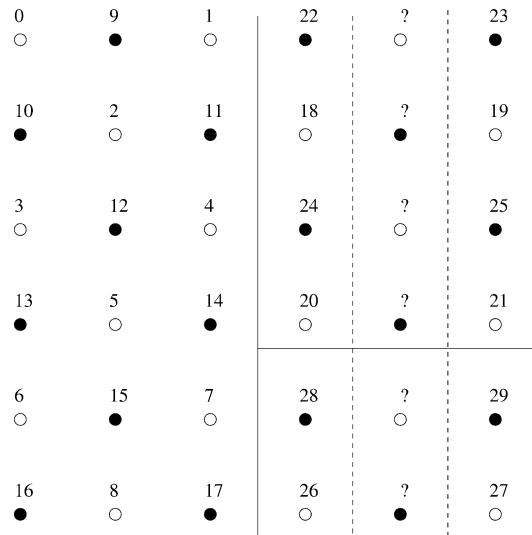


Fig. 8. Example of a 6×6 lattice form the point of view of process 0. Process zero assigns a local parameterization to the local sites and to the neighbor sites of the local points. It cannot access the rest of the sites.

```
x.global_index()
```

or the local index

```
x.local_index()
```

Moreover one can initialize a site variable specifying the coordinates

```
x.set(3,2);
```

the global index

```
x.set_global(26);
```

or the local index

```
x.set_local(7); /* assuming ME is 0 */
```

If `x` and `y` are two site variables in the same code defined on the same lattice, but using different partitioning. One can assign `y` to `x` using the assignment operator

```
x=y;
```

This is equivalent to

```
x.set_global(y.global_index());
```

4.6. Partitioning and topology

One of the most important characteristics of MDP 1.2 is that one is not limited to a box-like lattice (even if it has to be a subset of the points in a box) and the sites of the lattice can be associated to an arbitrary topology. Before stating with weird stuff here is the definition of the default function for the lattice partitioning:

```

template<int dim>
int default_partitioning(int *x, int ndim, int *nx) {
    float tpp1=nx[dim]/NPROC;
    int    tpp2=(int) tpp1;
    if(tpp1-tpp2>0) tpp2+=1;
    return x[dim]/tpp2;
};

```

Any partitioning function takes three arguments:

- An array `x[ndim]` containing the `ndim` coordinates of a site.
- The dimensions, `ndim`, of the box containing the lattice sites.
- An array, `nx[ndim]`, containing the size of box in each dimension.

It returns the identification number of the process that stores the site corresponding to the coordinates in `x[]`.¹²

One can define a different partitioning just by defining another function that takes the same arguments as the default one and pass it as third argument to the constructor of `generic_lattice`. If the new partitioning function, for a particular input site, returns a number outside the range 0 to `NPROC-1`, that site is excluded from the lattice. To this scope the macro `NOWHERE` is defined. If a site has to be excluded from the lattice the partition function should return `NOWHERE`. If one defines a partitioning function that excludes some points from the lattice one is forced to create a user defined topology, otherwise one ends up with sites which are neighbors of sites that are `NOWHERE` and the program fails.

The default torus topology is defined as

```

void torus_topology(int mu, int *x_dw, int *x, int *x_up,
                   int ndim, int *nx) {
    for(int nu=0; nu<ndim; nu++) if(nu==mu) {
        x_dw[mu]=(x[mu]-1+nx[mu]) % nx[mu];
        x_up[mu]=(x[mu]+1) % nx[mu];
    } else x_up[nu]=x_dw[nu]=x[nu];
};

```

Any topology function takes six arguments, they are:

- A direction `mu` in the range 0 to `ndim-1`.
- A first array of coordinates, `x_dw[]`.
- A second array of coordinates, `x[]`.
- A third array of coordinates, `x_up[]`.
- The dimensions, `ndim`, of the box containing the lattice sites.
- An array, `nx[ndim]`, containing the size of box in each dimension.

The coordinates in `x[]` are taken as input and the topology function fills the the arrays `x_dw[]` and `x_up[]` with the coordinates of the neighbor points when moving from `x[]` one step in direction `mu` down and up, respectively.

The topology function will only be called for those sites `x[]` that belong to the lattice. It has to fill `x_dw[]` and `x_up[]` with the coordinates of points that have not been excluded from the lattice.

¹² Note that at this level `site` variables are not used. In fact the partitioning function (as well the topology function) is called before the lattice is defined, and it is used to define it. A site variable can be defined only after a lattice is defined.

4.6.1. Nonperiodic lattice

As an example one can create a lattice with a true physical boundary by saying, for example, that moving up (or down) in a particular direction from a particular subset of sites one remains where one is.¹³ In the simple case of a finite box one could define the topology:

```
void box_topology(int mu, int *x_dw, int *x, int *x_up,
                 int ndim, int *nx) {
    torus_topology(mu, x_dw, x, x_up, ndim, nx);
    if(x[mu]==0) x_dw[mu]=x[mu];
    if(x[mu]==nx[mu]-1) x_up[mu]=x[mu];
};
```

4.6.2. Hexagonal lattice

Another example of a lattice with a weird topology is the hexagonal grid shown in Fig. 9. The trick to do it is having a 3D lattice flat in the third direction, put NOWHERE the centers of the hexagons and define the proper topology for the remaining points. It can be done by the following code

```
int exagonal_partitioning(int x[], int ndim, int nx[]) {
    if((x[0]+x[1])%3==0) return NOWHERE;
    else return default_partitioning<0>(x, ndim, nx);
}
```

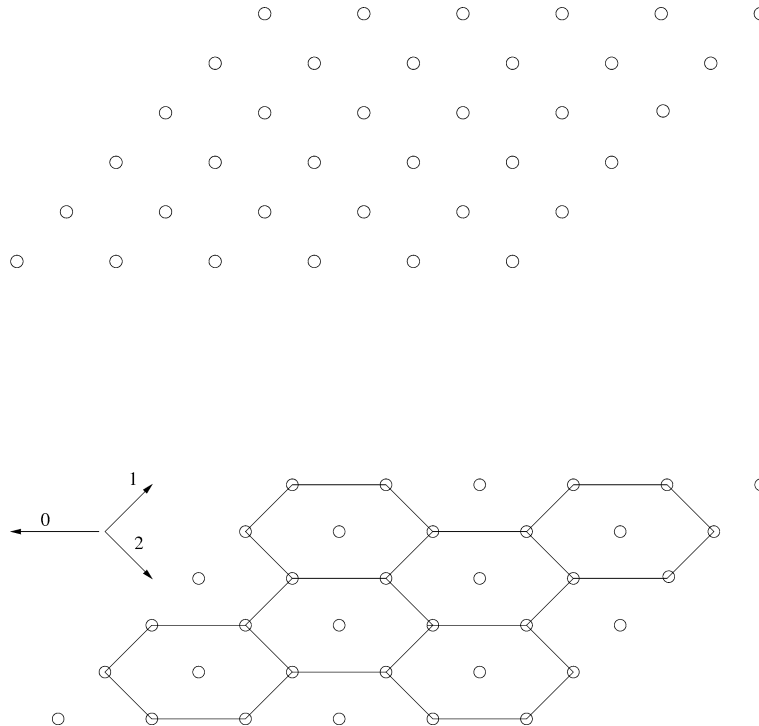


Fig. 9. Example of a 6×6 grid to be used to build an hexagonal lattice (top) and the metric associated to this grid (bottom). The points at center of the hexagons are excluded from the lattice by the partitioning function.

¹³ This kind of lattice can be used, for example, to study a 3D solid under stress and study its deformations.

```

};

void exagonal_topology(int mu, int *x_dw, int *x, int *x_up,
                      int ndim, int *nx) {
    if((x[0]+x[1])%3==1) {
        switch(mu) {
            case 0: x_up[0]=(x[0]+1)%nx[0]; x_up[1]=x[1]; break;
            case 1: x_up[1]=(x[1]+1)%nx[1]; x_up[0]=x[0]; break;
            case 3: x_up[0]=(x[0]-1+nx[0])%nx[0];
                   x_up[1]=(x[1]-1+nx[1])%nx[1]; break;
        };
        x_up[2]=x[2]; x_dw[0]=x[0];
        x_dw[1]=x[1]; x_dw[2]=x[2];
    };
    if((x[0]+x[1])%3==2) {
        switch(mu) {
            case 0: x_dw[0]=(x[0]-1+nx[0])%nx[0]; x_dw[1]=x[1]; break;
            case 1: x_dw[1]=(x[1]-1+nx[1])%nx[1]; x_dw[0]=x[0]; break;
            case 3: x_dw[0]=(x[0]+1)%nx[0]; x_dw[1]=(x[1]+1)%nx[1]; break;
        };
        x_dw[2]=x[2]; x_up[0]=x[0];
        x_up[1]=x[1]; x_up[2]=x[2];
    };
};

int n=2;
int mybox[]={3*n, 3*n, 1};

```

```

generic_lattice exagonal(3,mybox,exagonal_partitioning,
                        exagonal_topology);

```

One can observe that we need a flat third dimension to allow for three possible direction. For this reason MDP 1.2 allows to define a lattice with a number of directions different from the number of dimensions. To avoid entering in a number of technical details we will avoid discussing this possibility here. What one can do with such a lattice is another matter and will not be discussed. From now on we will concentrate on more regular lattices which tend to have a more intuitive interpretation.

4.6.3. Antiperiodic boundary conditions

There are two possible meaning for antiperiodic boundary conditions: (1) antiperiodic boundary conditions for the lattice (i.e. a Moebious topology). (2) antiperiodic boundary conditions for a field defined on the lattice.

The first case can be handled in the same way as any other topology. The second case has nothing to do with the lattice topology, but is a property of the field and one has to take care of it in the definition of `operator()` for the field in question.

4.7. Memory optimization

Actually the class `generic_field` has more properties of those described. For example each field can be deallocated at any time if memory is needed:

```

struct mystruct { /* any structure */ };
generic_lattice mylattice1(mydim1,mybox1);
generic_field<mystruct> myfield(mylattice1);
/* use myfield */
myfield.deallocate_memory();
generic_lattice mylattice2(mydim2,mybox2);
myfield.allocate_field(mylattice2);

```

Note that one cannot reallocate a field using a different structure at the site. This would be very confusing.

5. Examples of parallel applications

We will present here a few example of parallel application that use MDP 1.2. To keep things simple we will not make use of fields of matrices.

5.1. Electrostatic potential for a given distribution of charges

We consider here the vacuum space inside a cubic metal box connected to ground and containing a given (static) distribution of charge. We want to determine the electrostatic potential in the box.

The vacuum is implemented as a finite 20^3 lattice. Two fields, a charge density $q(x)$ and a potential $u(x)$, are defined on it. In the continuum the potential is obtained by solving the Gauss law equation

$$\nabla^2 u(x) = q(x). \quad (6)$$

Its discretized form reads

$$\sum_{i=0}^2 [u(x + \hat{i}) - 2u(x) + u(x - \hat{i})] = q(x) \quad (7)$$

which can be solved in $u(x)$

$$u(x) = \frac{1}{6} \left[q(x) + \sum_{i=0}^2 u(x + \hat{i}) + u(x - \hat{i}) \right]. \quad (8)$$

Therefore the static potential solution is obtained by iterating Eq. (8) on the vacuum until convergence.

As initial condition we assume that only two charges are present in the box:

$$q(x) = 3\delta(x - A) - 5\delta(x - B), \quad (9)$$

where $A = (3, 3, 3)$ and $B = (17, 17, 17)$. Moreover since the box has a finite extension the `box_topology` will be used.

Here is the parallel program based on MDP 1.2.

```

// Program: application1.C
#define PARALLEL
#define Nproc 4
#include "MDP_Lib2.h"
#include "MDP_MPI.h"
int main(int argc, char **argv) {
    mpi.open_wormholes(argc,argv);
    int mybox[]={20,20,20};

```

```

generic_lattice vacuum(3,mybox,
                      default_partitioning<0>,
                      box_topology);
generic_field<float> u(vacuum);
generic_field<float> q(vacuum);
site x(vacuum);
site A(vacuum);
site B(vacuum);
A.set(3,3,3);
B.set(17,17,17);
float precision, old_u;
forallsitesandcopies(x) {
    u(x)=0;
    if(x==A)      q(x)=3;
    else if(x==B) q(x)=-5;
    else          q(x)=0;
};
do {
    precision=0;
    forallsites(x) if((x(0)>0) && (x(0)<mybox[0]-1) &&
                      (x(1)>0) && (x(1)<mybox[1]-1) &&
                      (x(2)>0) && (x(2)<mybox[2]-1)) {
        old_u=u(x);
        u(x)=(q(x)+u(x+0)+u(x-0)+u(x+1)+u(x-1)+u(x+2)+u(x-2))/6;
        precision+=pow(u(x)-old_u,2);
    };
    u.update();
    mpi.add(precision);
} while (sqrt(precision)>0.0001);
u.save("potential.dat");
mpi.close_wormholes();
};

```

The saved values can be used to produce a density plot representing the potential in the volume.

5.2. Total impedance in net of resistors

Another problem we want to solve is that of determining the total resistance between two arbitrary points (A and B) on a semi-conducting finite cylindrical surface. The cylinder is obtained starting from a torus topology and cutting the torus in one direction (say 0). The total resistance is probed by connecting the terminals of a current generator (J) to the points A and B and evaluating the potential at points A and B using Ohm's law

$$R_{AB} = \frac{u(A) - u(B)}{J}. \quad (10)$$

The surface of the semiconductor is implemented as a periodic net of resistances so that each site has four resistances connected to it, Fig. 10. For simplicity we assume that all the resistances are the same (i.e. the material is homogeneous). A different choice would be equally possible.

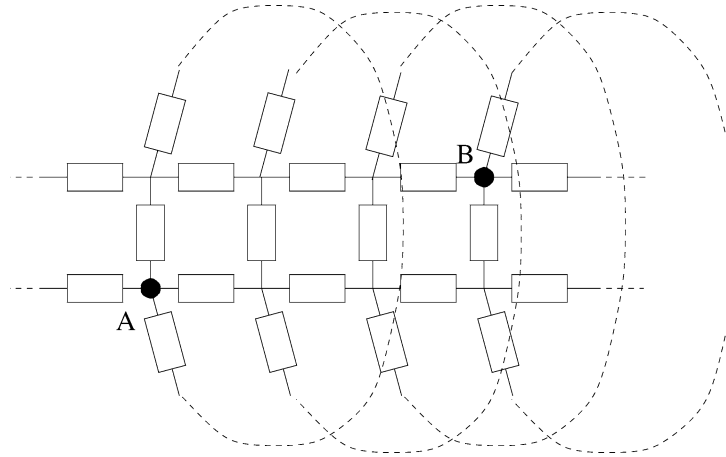


Fig. 10. Example of a cylindrical net of resistors.

The potential $u(x)$ is computed using the Kirchoff law at each vertex

$$\sum_{i=0,1} R_i(x - \hat{i})[u(x - \hat{i}) - u(x)] + R_i(x)[u(x + \hat{i}) - u(x)] + J = 0 \quad (11)$$

that, solved in $u(x)$, leads to

$$u(x) = \frac{\sum_{i=0,1} R_i(x - \hat{i})u(x - \hat{i}) + R_i(x)u(x + \hat{i}) + J}{\sum_{i=0,1} R_i(x - \hat{i}) + R_i(x)}. \quad (12)$$

This equation is iterated on each site until convergence.

Here is the parallel program based on MDP 1.2.

```
// Program: application2.C
#define PARALLEL
#define Nproc 4
#include "MDP_Lib2.h"
#include "MDP_MPI.h"
void open_cylinder(int mu, int *x_dw, int *x, int *x_up,
                  int ndim, int *nx) {
    torus_topology(mu, x_dw, x, x_up, ndim, nx);
    if((mu==0) && (x[0]==0)) x_dw[0]=x[0];
    if((mu==0) && (x[0]==nx[0]-1)) x_up[0]=x[0];
};
float resistance(int x0, int x1, int mu) {
    return (Pi/100*x0);
};
int main(int argc, char **argv) {
    mpi.open_wormholes(argc, argv);
    int mybox[]={100,20};
    generic_lattice cylinder(2, mybox, default_partitioning<0>,
                           open_cylinder);
    generic_field<float> u(cylinder);
```



```

generic_field<float> r(cylinder,2);
site x(cylinder);
site A(cylinder);
site B(cylinder);
float precision, old_u;
float c, J=1, deltaJ, deltaR, Rtot, local_Rtot;
A.set(15,7);
B.set(62,3);
forallsitesandcopies(x) {
    u(x)=0;
    r(x,0)=resistance(x(0),x(1),0);
    r(x,1)=resistance(x(0),x(1),1);
};
do {
    precision=0;
    forallsites(x) {
        old_u=u(x);
        if(x==A) { c+=J; printf("%i\n", ME); };
        if(x==B) c-=J;
        deltaJ=c; deltaR=0;
        /* the next two lines take care of the finite cylinder */
        if(x+0!=x) { deltaJ+=u(x+0)*r(x,0); deltaR+=r(x,0); };
        if(x-0!=x) { deltaJ+=u(x-0)*r(x-0,0); deltaR+=r(x-0,0); };
        deltaJ+=u(x+1)*r(x,1); deltaR+=r(x,1);
        deltaJ+=u(x-1)*r(x-1,1); deltaR+=r(x-1,1);
        u(x)=deltaJ/deltaR;
        precision+=pow(u(x)-old_u,2);
    };
    u.update();
    mpi.add(precision);
} while (sqrt(precision)>0.00001);
Rtot=0;
if(A.is_in()) Rtot+=u(A)/J;
if(B.is_in()) Rtot-=u(B)/J;
mpi.add(Rtot);
if(ME==0) printf("R_A-R_B=%f\n", Rtot);
mpi.close_wormholes();
};

```

5.3. Ising model

One more full application we wish to consider is the study of the total magnetization of a spin system under a magnetic field (this is also known as the Ising model) as function of the temperature. As usual we define a lattice that represents the geometry of the spin system, and associate to the sites a field $s(x) \in \{-1, +1\}$. This is schematically represented in Fig. 11. The Hamiltonian of this system is

$$H = -\beta \sum_{x,y} s(x) \Delta(x, y) s(y) - \kappa \sum_x s(x) h(x), \quad (13)$$

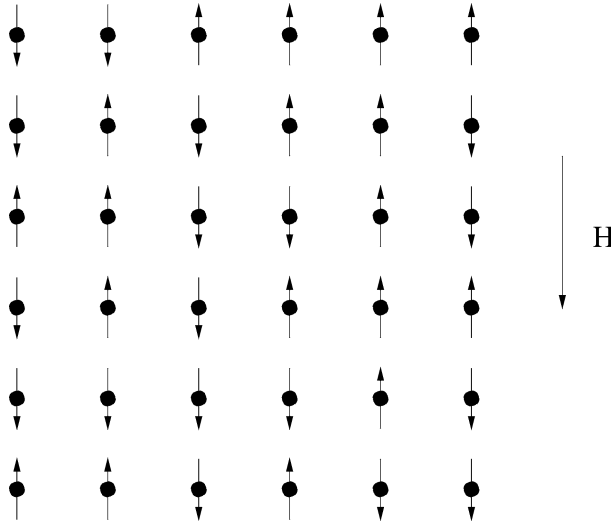


Fig. 11. Example of a 2D spin system.

where $h(x)$ is an external magnetic field, $\beta = 1/T$ is the inverse temperature and κ is a constant typical of the material (in principle one could promote κ to be a field for non-homogeneous materials. $\Delta(x, y)$ is 1 if $|x - y| = 1$, 0 otherwise. In the example we consider a two dimensional spin system but the extension to arbitrary dimensions is trivial: just change mybox. Moreover we implement a simple montecarlo_multihit algorithm [7] to minimize the action in presence of thermal fluctuations.

Here is the parallel program based on MDP 1.2.

```
// Program: application3.C
#define PARALLEL
#include "MDP_Lib2.h"
#include "MDP_MPI.h"

// //////////////////////////////////////
class scalar_field: public generic_field<float> {
public:
    int ndim,nc;
    scalar_field(generic_lattice &a) {
        allocate_field(a, 1);
        ndim=a.ndim;
    };
    float &operator() (site x) {
        return *(address(x));
    };
};

// //////////////////////////////////////
class ising_field: public generic_field<int> {
public:
    int ndim,nc;
```

```

float beta, kappa;
ising_field(generic_lattice &a) {
    allocate_field(a, 1);
    ndim=a.ndim;
};
int &operator() (site x) {
    return *(address(x));
};
friend void set_cold(ising_field &S) {
    site x(S.lattice());
    forallsites(x) S(x)=1;
};

friend void montecarlo_multihit(ising_field &S, scalar_field &H,
int n_iter=10, int n_hits=3) {
    int iter, parity, hit, new_spin, mu;
    float delta_action;
    site x(S.lattice());
    for(iter=0; iter<n_iter; iter++) {
        for(parity=0; parity<=1; parity++)
            forallsitesofparity(x,parity) {
                for(hit=0; hit<n_hits; hit++) {
                    delta_action=S.kappa*H(x);
                    for(mu=0; mu<S.ndim; mu++)
                        delta_action-=S(x+mu)+S(x-mu);
                    new_spin=(S.lattice().random(x).plain()>0.5)?1:-1;
                    delta_action*=S.beta*(new_spin-S(x));
                    if(exp(-delta_action)>S.lattice().random(x).plain())
                        S(x)=new_spin;
                }
            };
        S.update(parity);
    };
};
friend float average_spin(ising_field &S) {
    float res=0;
    site x(S.lattice());
    forallsites(x) res+=S(x);
    mpi.add(res);
    return res/(S.lattice().nvol_gl);
};
};

int main(int argc, char **argv) {
    mpi.open_wormholes(argc, argv);

    int conf,Ntherm=100,Nconfig=100;
    int mybox[]={64,64};

```

```

generic_lattice mylattice(2, mybox);
ising_field      S(mylattice); /* ths spin field +1 or -1      */
scalar_field     H(mylattice); /* the external magnetic field */
site             x(mylattice);
JackBoot         jb(Nconfig,1);
jb.plain(0);

S.beta=0.5;      /* inverse square temperature                */
S.kappa=0.1;    /* coupling with the total extarnal field */

/* setting initial conditions */
forallsites(x) {
    S(x)=1; /* initial spins set to +1 */
    H(x)=0; /* external magnetic field set to zero */
};
S.update();
H.update();

if(ME==0)
    printf("Beta\tMagnetization\terror\n");

/* looping on values of beta */
for(S.beta=0.20; S.beta<0.60; S.beta+=0.01) {
    /* termalizing configuration */
    for(conf=0; conf<Ntherm; conf++)
        montecarlo_multihit(S,H);

    /* generating Nconfig configurations for each value of beta */
    for(conf=0; conf<Nconfig; conf++) {
        montecarlo_multihit(S,H);
        jb(conf,0)=average_spin(S);
    };
    if(ME==0)
        printf("%f\t%f\t%f\n",
            S.beta, abs(jb.mean()), jb.j_err());
};
mpi.close_wormholes();
return 0;
};

```

The output of this program (for $h(x) = 0$) has been plotted and reported in Fig. 12. It clearly shows that when the temperature is high ($\beta = 1/T$ is low) the average magnetization fluctuates around zero while, when the temperature is low (β is high), all the spins tend to become parallel giving origin to a total magnetization different from zero. The critical temperature where the phase transition (the jump) occurs is, for the 2D model

$$T_c = \frac{1}{\beta_c} = \frac{2}{\log(1 + \sqrt{2})} \simeq 2.269. \quad (14)$$

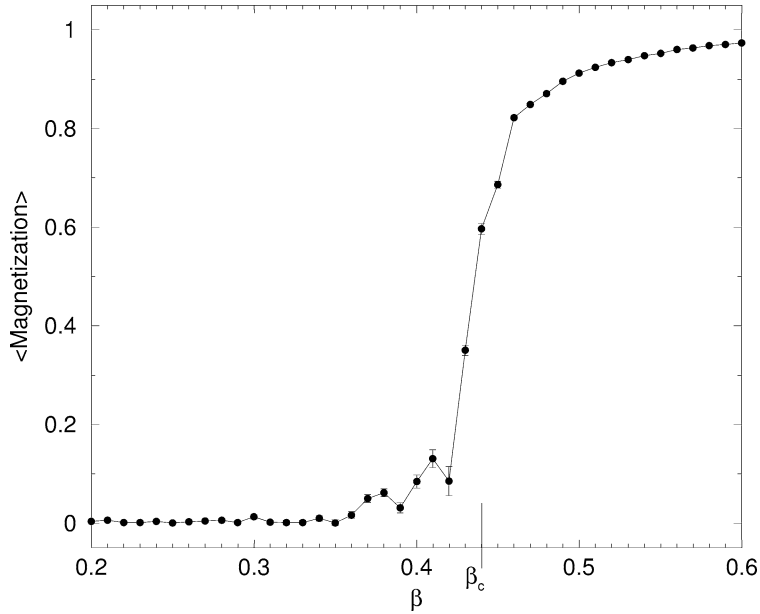


Fig. 12. Total average magnetization in a two-dimensional spin system as function of the inverse temperature. The jump corresponds to a phase transition.

Note that the only reference to the number of dimensions is in the function `main()` in the lines

```
int mybox[]={64, 64};
generic_lattice mylattice(2, mybox);
```

These are the only lines to change to generalize the program to more than 2D. For bigger volumes one may also want to change the total number of computed configurations `Nconfig` and the steps to thermalization `Ntherm`.

5.4. Solid under stress

Many other parallel applications could be developed relatively easy using MDP 1.2. For example, one may be interested in studying the deformation of a solid under stress. The solid can be approximated with a finite lattice of given arbitrary shape and each site would be connected to its next-neighbor and next-next-neighbor sites with springs. One needs at least 4 fields on this lattice: M , K , G , Q and P , where M is the mass density of the solid; K the spring constant; G a parameter determining the internal friction of the solid; Q the physical (spatial) coordinates of each site, P the momentum associated to each site. The problem of determining the deformations of this solid can be solved by iterating a discretized form of the Hamilton equations

$$P_{t+\Delta t}(x) - P_t(x) = \frac{\partial H(P, Q, K, G)}{\partial Q(x)} \Delta t = F(x, Q, P, K, G) \Delta t, \quad (15)$$

$$Q_{t+\Delta t}(x) - Q_t(x) = \frac{\partial H(P, Q, K, G)}{\partial P(x)} \Delta t = P(x)/M(x) \Delta t. \quad (16)$$

These equations translates into

```
float delta_t=0.000001; /* a small nonzero positive number */
forallsites(x)
```

```

for(mu=0; mu<3; mu+) {
  F[mu]=/* some function of K,G,Q,P */
  P(x,mu)=P(x,mu)+F(mu)*delta_t;
  Q(x,mu)=Q(x,mu)+P(x,mu)/m*delta_t;
};
P.update();
Q.update();

```

The expression of the function F can be quite complicated depending on how one models the solid.

5.5. Lattice QCD

MDP 1.2 has been used to develop a parallel package for large scale Lattice QCD simulations called FermiQCD [8,9]. Here we only list some of its main features.

The typical problem in QCD (Quantum Chromodynamics) is that of determining the correlation functions of the theory as a function of the parameters. From the knowledge of these correlation functions one can extract particle masses and matrix elements to compare with experimental results from particle accelerators.

On the lattice, each correlation function is computed numerically as the average of the corresponding operator applied to elements of a Markov chain of gauge field configurations. Both the processes of building the Markov chain and of measuring operators involve quasi-local algorithms. Some of the main features implemented in FermiQCD are:

- works on a single process PC,
- works in parallel with MPI,
- arbitrary lattice partitioning,
- parallel I/O (partitioning independent),
- arbitrary space-time dimension,
- arbitrary gauge group $SU(n)$,
- anisotropic Wilson gauge action,
- anisotropic Wilson fermionic action,
- anisotropic Clover improved action,
- D234 improved action,
- Kogut–Susskind improved action,
- ordinary and stochastic propagators,
- minimal residue inversion,
- stabilized biconjugate gradient inversion,
- twisted boundary conditions for large β numerical perturbation theory (all fields),
- reads existing CANOPY data,
- reads existing UKQCD data,
- reads existing MILC data.

In Table 2 we show a few examples of FermiQCD Object Oriented capabilities (compared with examples in the standard textbook notation for Lattice QCD [9]).

6. Timing and efficiency issues

The issue of efficiency of a parallel program is not a simple one, in fact there are many factors to take into account. In particular the efficiency scales with the number of parallel processes, with the size of the data that need to be communicated between processes, and with the total volume of the lattice.

Table 2
Example of FermiQCD statements

| QCD | FermiQCD |
|--|--|
| Algebra of Euclidean gamma matrices | |
| $A = \gamma^\mu \gamma^5 e^{3i\gamma^2}$ | Matrix A; A=Gamma[mu]*Gamma5*exp(3*I*Gamma[2]); |
| Multiplication of a fermionic field for a spin structure | |
| $\forall x: \chi(x) = (\gamma^3 + m)\psi(x + \hat{\mu})$ | /* assuming the following definitions generic_lattice space_time(...); fermi_field chi(space_time,Nc); fermi_field psi(space_time,Nc); site x(space_time); */ forallsites(x) chi(x)=(Gamma[3]+m)*psi(x+mu); |
| Translation of a fermionic field | |
| $\forall x, a: \chi_a(x) = U(x, \mu)\psi_a(x + \hat{\mu})$ | forallsites(x) for(a=0; a<psi.Nspin; a++) chi(x,a)=U(x,mu)*psi(x+mu,a); |

The dependence of the efficiency on these variables cannot be in general factorized because of non-linearities introduced by modern computer architectures (for example, caches effects, finite memory, memory speed/CPU speed, latencies, etc.).

To give an idea of how a parallel computer program based on MDP scales with the number of processes, we consider the following program.

```
// program: laplace.C
#define PARALLEL
#include "MDP_Lib2.h"
#include "MDP_MPI.h"

int main(int argc, char **argv) {
    mpi.open_wormholes(argc, argv);
    int box[4]={8,12,12,12};
    generic_lattice space(4,box);
    Matrix_field U(space,3,3);
    Matrix_field V(space,3,3);
    site x(space);
    forallsites(x) {
        V(x)=space.random(x).SU(3);
        U(x)=0;
    };
    U.update();
}
```

```

V.update();
for(int i=0; i<100000; i++) {
  forallsites(x)
    U(x)=0.125*(cos(U(x)+V(x))+
               U(x+0)+U(x-0)+
               U(x+1)+U(x-1)+
               U(x+2)+U(x-2)+
               U(x+3)+U(x-3));
  /* uncomment to print convergence
  if(ME==0) {
    x.set(0,0,0,0);
    printf("%f\n", real(U(x,0,0)));
  };
  */
  U.update();
};
V.save("V_field.dat");
U.save("U_field.dat");
mpi.close_wormholes();
return 0;
};

```

It solves a 4-dimensional Laplace equation for a field of 3×3 matrices, and performs some parallel IO as well. The reader can work out the details of the program as an exercise.

We ran the same program for different lattice volumes $V = T \times L^3$ and on different sets of processors (N = number of processors). Each processor (node) is a 700 MHz PentiumIII PC. The nodes are connected using Myrinet cards/switches.

For each run we measured the total time $t(T, L, N)$ and we then computed the efficiency according with two possible definitions:¹⁴

$$e_1(T, L) = 100 \frac{t(T, L, 1)}{N t(T, L, N)}, \quad (17)$$

$$e_2(N, L, x) = 100 \frac{t(x, L, 1)}{t(Nx, L, N)}. \quad (18)$$

The first definition is applicable to computations in which the problem size is fixed and one varies the numbers of processors. The second definition is applicable to computations for which the size of the problem increases with the number of processors available. In both cases the efficiency is normalized to 100%. In an ideal world (without latencies and with instantaneous communications) both these efficiencies should be constant and equal to 100%.

In Figs. 13, 14 we plot e_1 and e_2 for our runs and the maximum time spent in communications by the nodes (as computed by MDP) that we call r . The normalizations are given by

$$t(60, 4, 1) = 788 \text{ s},$$

$$t(60, 8, 1) = 6417 \text{ s},$$

$$t(60, 12, 1) = 6417 \text{ s},$$

$$t(2, 8, 1) = 201 \text{ s},$$

¹⁴ In the context of Lattice QCD simulation, up to an overall normalization, the first definition is the one used at Fermilab to benchmark parallel software, the second is the definition used by the MILC Collaboration.

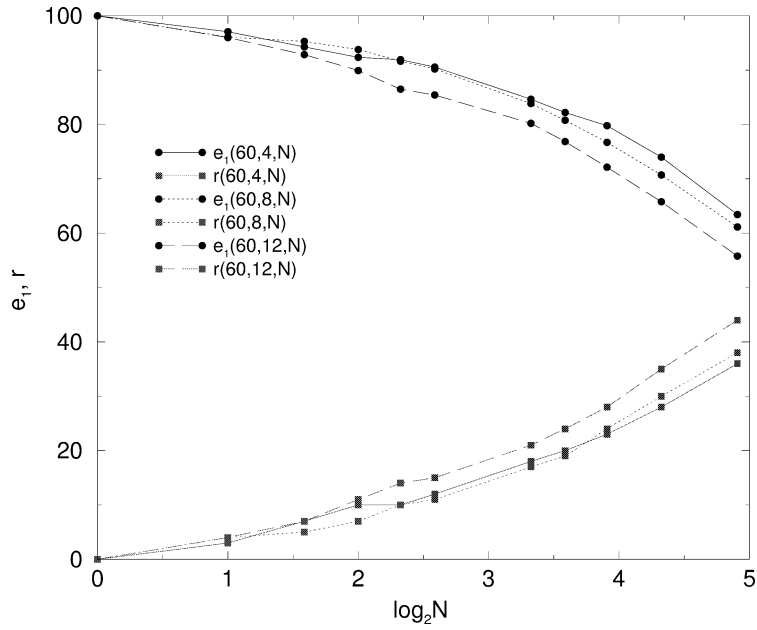


Fig. 13. Efficiency e_1 as function of N and L , for $T = 60$ (the number 60 has been chosen because it has the biggest number of integer factors within the number of available processors).

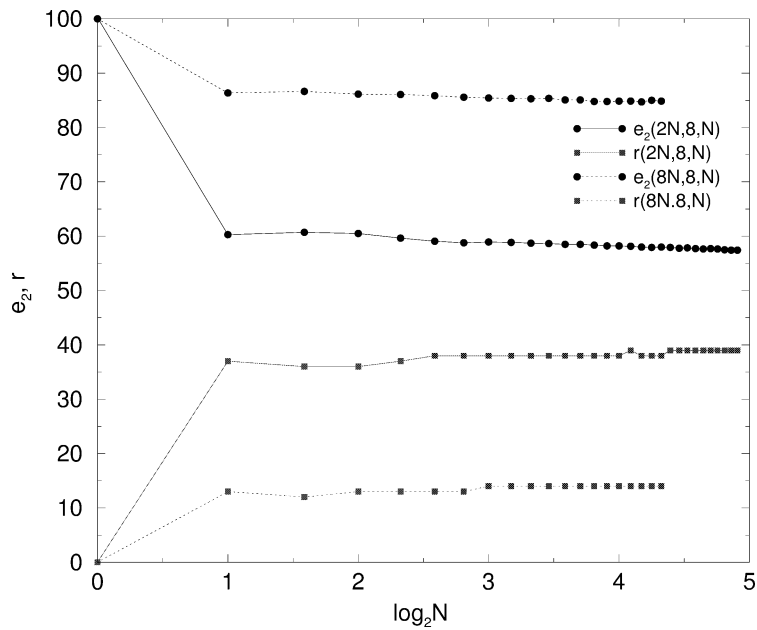


Fig. 14. Efficiency e_2 as function of N , for $T = 2N$ and $L = 8$.

$$t(8, 8, 1) = 834 \text{ s},$$

These numbers can be further reduced by writing a user defined function that optimizes the critical computation done in the line

$$U(\mathbf{x}) = 0.125 * (\cos(U(\mathbf{x}) + V(\mathbf{x})) + \dots).$$

From the plots we observe that with good approximation

$$e_i \simeq 100 - r \quad (19)$$

for both the definitions of efficiency.

Eventually, for N bigger than some N_0 , the total time becomes totally dominated by communication (the time spent in communications goes to 100%) and e_1 starts to scale as $1/N$.

Note that the answer to the question “how many processors should I use?” is not addressed in this paper because, in financial terms, this depends on the subjective utility function of each individual user. The only general conclusion is that N should be less than N_0 .

A different algorithm may scale in a different way even if the qualitative behavior of the efficiency should be the same. Some algorithms may require more communications than others and may result in a bigger loss of efficiency when running in parallel.

Acknowledgements

I thank Chris Sachrajda, Jonathan Flynn and Luigi Del Debbio of the University of Southampton (UK) where the MDP project started.

The development of the code MDP 1.2 in its final (but not yet definitive) form and its main application, FermiQCD, have greatly benefited from discussions and suggestions with members of the Fermilab Theory Group. In particular I want to thank Paul Mackenzie, Jim Simone, Jimmy Juge, Andreas Kronfeld and Estia Eichten for comments and suggestions. Moreover I freely borrowed many ideas from existing Lattice QCD codes (including the UKQCD code [10], the MILC code [11], QCDF90 [12] and CANOPY [13]). I here thank their authors for making their software available to me.

References

- [1] G. Satir, D. Brown, C++ The Core Language, O'Reilly and Associates, 1995.
- [2] N. Cabibbo, E. Marinari, A new method for updating $SU(N)$ matrices in computer simulations of gauge theories, Phys. Lett. 119B (1982) 387.
- [3] J. Shao, D. Tu, The Jackknife and Bootstrap, Springer Verlag, 1995.
- [4] P.D. Coddington, Random Number Generators for Parallel Computers, 1997 (unpublished).
- [5] G.A. Marsaglia, A current view on random number generators, in: L. Balliard (Ed.), Computational Science and Statistics: The Interface, Elsevier, Amsterdam, 1985.
- [6] P.S. Pacheco, Parallel Programming with MPI, San Francisco, CA, Morgan Kaufmann, 1997.
- [7] G. Banhot, The Metropolis algorithm, Rep. Prog. Phys. 51 (1988) 429.
- [8] M. Di Pierro, Matrix distributed processing and FermiQCD, hep-lat/0011083.
- [9] M. Di Pierro, From Monte Carlo integration to lattice quantum chromodynamics: An introduction, hep-lat/0009001.
- [10] The UKQCD collaboration web page: <http://www.ph.ed.ac.uk/ukqcd/>.
- [11] The MILC collaboration web page: <http://physics.indiana.edu/~sg/milc.html>.
- [12] I. Dasgupta, A.R. Levi, V. Lubicz, C. Rebbi, Comput. Phys. Commun. 98 (1996) 365–397.
- [13] M. Fischler, J. Hockney, P. Mackenzie, M. Uchima, Canopy 7.0 manual, Fermilab preprint TM-1881. PDF text available at web site <http://fnalpubs.fnal.gov/archive/tm/TM-1881.pdf>.

LICENSE for MDP 1.2

(including examples and applications suchas FermiQCD)

- MDP 1.2 has been created by Massimo Di Pierro. MDP 1.2 is a property of the author. (The application FermiQCD, also covered by this License is a joined property of the author and of Fermilab).
- MDP 1.2 is free of charge for educational and research institutions worldwide.
- You may copy and distribute exact replicas of MDP 1.2 as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice including the author's name and the disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipient of MDP 1.2 a copy of this License along with the software.
- You may modify your copy or copies of MDP 1.2 or any portion of it and distribute such modifications or work under the terms of Section 1 and 2 above, provided that the modified content carries prominent notices stating that it has been changed, the exact nature and content of the changes, and the date of any change.
- BECAUSE MDP 1.2 IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR IT. EXCEPT WHEN OTHERWISE STATED IN WRITING. THE AUTHOR PROVIDES MDP 1.2 "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED. THE ENTIRE RISK OF USE OF THIS SOFTWARE IS WITH THE RECIPIENT. SHOULD MDP 1.2 OR ONE OF ITS APPLICATIONS PROVE FAULTY, INACCURATE, OR OTHERWISE UNACCEPTABLE YOU ASSUMES THE COST OF ALL NECESSARY REPAIR OR CORRECTION.
- To request MDP 1.2 (including the examples and applications described in this paper) please sign this license and fax or mail this page to the author:

Massimo Di Pierro
Email: mdp@fnal.gov
Fax: 001-630-840-5435
Address: MS 106, Fermilab, Batavia, IL 60510-0500 (USA)

Name:
Date:
Email:
Signature: