Parallel Inversion of the Dirac Matrix

Johannes Gerer

Supervised by
Prof. Dr. Andreas Schfer,
Institute of Theoretical Physics, University of Regensburg

Contents

0 Notations

Throughout the work summation conventions are used for spatial and Minkowsky indices and indices of the adjoint SU(3) representation.
Color and Dirac indices will be omitted if possible.

The following notation will be used:

Natural units	$c = \hbar = 1$	(1)
Contravariant vector	$x^\mu = \left(x^0, x^1, x^2, x^3\right) = (t, x, y, z) = (t, \vec{x})$	(2)
Metric tensor	$g^{\mu\nu} = g_{\mu\nu} = \text{diag}\,(1, -1, -1, -1)$	(3)
Covariant vector	$x_\mu = g_{\mu\nu} x^\nu = (t, -\vec{x})$	(4)
Scalar product	$x \cdot y = x^\mu y_\mu = x^0 y^0 - \vec{x} \cdot \vec{y}$	(5)
Four-momentum	$p^\mu = (E, -\vec{p}) = i\partial^\mu = i\frac{\partial}{\partial x_\mu} = i\left(\frac{\partial}{\partial t}, -\vec{\nabla}\right)$	(6)
Dirac matrices	$\gamma^\mu = (\gamma^0, \vec{\gamma})$, with $\{\gamma^\mu, \gamma^\nu\} = 2\, g^{\mu\nu}$	(7)
	$\gamma^5 = i\,\gamma^0\gamma^1\gamma^2\gamma^3$	(8)
Feynman dagger	$\not{a} = a_\mu \gamma^\mu$	(9)
Quark spinor	$\psi_f^{\alpha,c}(x)$,with$x \in \mathbb{R}^4$ color index$c = 1, 2, 3 (= N_{\text{color}})$, Dirac spinor index $\alpha = 1, 2, 3, 4 (= N_{\text{Dirac}})$ and flavor$f$	(10)
Dirac adjungation	$\bar{\psi} = \psi^\dagger \gamma_0$	(11)
SU(3) generators	T^a, with $a = 1, \ldots, 8$	(12)
Four potential	$\mathrm{A}_\mu\,(x) = A^a{}_\mu\,(x)\,\mathrm{T}^a$	(13)
Euclidean vector	$x_\mu^{\mathrm{E}} = (x_1^{\mathrm{E}}, x_2^{\mathrm{E}}, x_3^{\mathrm{E}}, x_4^{\mathrm{E}}) = (\vec{x}, it)$	(14)
Euclidean Dirac matrices	$\gamma_\mu^{\mathrm{E}} = (-i\boldsymbol{\gamma}, \gamma^0)$, with $\left\{\gamma_\mu^{\mathrm{E}}, \gamma_\nu^{\mathrm{E}}\right\} = 2\,\delta^{\mu\nu}$	(15)
Lattice constant	a	(16)
Lattice point	$x_\mu = a n_\mu$ with $n_\mu \in \mathbb{N}$	(17)

Lattice extend	$L_\mu = aN_\mu$	(18)
Number of sites	$N_{\text{sites}} = N_0 N_1 N_2 N_3$	(19)

1 Introduction

The most promising way to probe the non-perturbative regime of QCD is Lattice QCD. Lattice QCD is a discreet theory that is formulated such that its limit of vanishing lattice spacing and infinite lattice size equals the continuum formulation of QCD. Also the limit process should be preferably smooth.
In this theory it is possible to calculate the path integral expressions numerically using statistical sampling and thus within statistical uncertainties that are to be minimized.
In order to derive anything meaningful, the lattice volume and the sample of configurations have to be large. These requirements drive the computation time far beyond what standard personal computers can achieve in years.
One part of this calculation is the inversion if the Dirac operator. As this operator in Lattice QCD is a square-matrix with $N_{\text{sites}} N_{\text{Dirac}} N_{\text{color}}$ rows, its inversion can be quite computational intense.

To overcome these obstacles, particle physicists have turned to supercomputers. Modern supercomputing means parallel computing. So in this work I will give a small overview over parallel computing, show where the inverse of the Dirac operator is needed in QCD, and explain the algorithm of the parallel inversion. It will be accompanied by examples of my previous work, that has been done within the Bern-Graz-Regensburg collaboration: I calculated the tensor- and the axial-charge (g_A) of the lowest negative parity states of the nucleon on a $32 \times 16 \times 16 \times 16$ lattice using two mass-degenerate flavours of chirally improved fermions and 100 configurations generated using the improved Lüscher-Weisz action.

2 Basics of Parallel Computing and MPI

2.1 Motivation

If a single computer takes years to complete a certain calculation, one is forced to speed up the computation. Assuming that the algorithm is already as efficient and optimized as it can be, there are in general two ways to speed up a calculation.

The first is to use a faster cpu. This possibility however has become less feasible, as higher speeds which require smaller scales of electric circuits, meet physical boundaries. Instead, manufactures started to produce multi-core cpus to meet the demand for more performance. This inevitably leads to the seconds possibility:
Split up the task in parts that can be computed in parallel by separate processing units. Depending on the specific application and requirements, there are numerous realizations of parallelized environments.
They can differ in aspects of memory - distributed or shared. It is also important to know whether the different parallel processes need to be synchronized or each can run separately and if each process follows the same instructions or if there is a different program for every different processor.

Recent supercomputers follow a hybrid approach in most of the above aspects: They consist of numerous identical nodes connected through some networking interface. Every node is equipped with a number of cpus - possibly multi-core - and memory that all the processors can access.
And although it is generally possible to execute different programs on different nodes, the most convenient method is to use only one executable that contains all the different instructions for the different processors and additional code that controls which parts are executed on what processor.

In the particular example of the Dirac operator inversion in lattice QCD, parallelization at first sight seams quite easy, because the number of independent inversions with separate data that have to be done can be very large. In our case, we needed to perform one inversion for each out of 100 configurations, 10 different quark masses, two different parity projections and sink times, the four Dirac and three color indices. This multiplies to a total of 24,000 independent inversions. So unless you have more than 24,000 processor cores, the most efficient method would be to use sequential code, that can perform one inversion using exactly one core and run it 24,000 times using all the processors. This way you achieve linear speed-up, as the parallelization produced no overhead and if t_{seq} is the time one core needs for one inversion, the total time needed per inversion is

$$t_{\text{par}} = \frac{t_{\text{seq}}}{n_{\text{cores}}}.$$

The problem however was caused by the required amount of memory. In the case above, about 24 GiB were needed for one inversion, and as the RAM per core was not higher than 4 GiB on any node, the inversion algorithm had in fact to be parallized to run on up to 16 cores from two nodes in parallel.

If several processes on different cores, processors and nodes actually have to work on the same inversion at a time using an iterative algorithm, they need to be synchronized and be able to communicate and exchange data. There are of course several ways this can be realized, but as basic functionalities coincide for many areas of application, some parallel programming standards emerged over the last two decades. They can be split up roughly in two complementary programming approaches: Threaded shared memory programming and parallel message passing systems.

2.2 MPI

The message passing interface (MPI) has been used in our case. It is a very popular specification for an API that implements a parallel message passing system. MPI provides source-code portability across a variety of architectures through its C and Fortran language bindings. There exist several implementations in form of libraries containing a set of routines that provide MPI functionalities to a variety of programming languages.

In order for a MPI program to work, it has to fulfill some requirements. The MPI header file has to be included (typically `mpi.h` in C and `mpif.h` in Fortran), the function `MPI_INIT` has to be called at the beginning of the program and the function `MPI_FINALIZE` at the end, by all processes. Also an MPI program typically is started by an MPI agent. This agent starts the desired number of processes, which are separate instances running a copy of the actual MPI executable. Processes created in this way share the same *communicator*, which handles communication among processes and enables them to send and receive messages by MPI function calls. This "root" communicator is created when `MPI_INIT` is called, as well as a globally defined variable `MPI_COMM_WORLD` that will be used to references to this communicator. Additionally, user-defined communicators can be created from arbitrary subsets of processes.
Within a communicator each process is assigned a unique *rank* that is used to address a process in point-to-point communication. Furthermore the rank can be used to distinguish the roles of the processes and one typically identifies the process with rank zero as the master-process, that will perform unique tasks like reading input files, writing results or screen output. The rank of a process within a communicator can be determined via

```
MPI_COMM_RANK(communicator,rank)
```

as well as the total number of processes within a communicator:

```
MPI_COMM_SIZE(communicator,size)
```

2.3 Communication

As the name suggest, communication within MPI is done by sending and receiving messages. A message is an array of one particular datatype. Every MPI routine possesses the three arguments `buf`, `count`, `type` characterizing the message. `buf` contains the memory address of the data, `count` specifies the number of elements of the type `type` that the message should contain.
The communication functions provided by MPI can be classified in two types:

2.3.1 Point-to-Point communication

One process sends a message that is received by another process. Both processes have to share a communicator and each other's rank within the communicator has to be known to them. There are different versions of the send and receive commands that differ in their communication mode. The communication mode determines mainly whether the program halts until the send or receive is completed or continues regardless of the completion. This difference is also named blocking or non-blocking communication.

```
MPI_SEND(buf,count,type,dest,tag,comm)
```

is the standard blocking send. This call will return only after the data has been moved to the receiver. `comm` contains the communicator, `dest` the rank of the destination process and `tag` is an integer value that can be used by the programmer to mark different types of messages.

```
MPI_RECV(buf,count,type,source,tag,comm,status)
```

This is the standard blocking receive, which will return when a message has ben received. `buf` specifies the address where the received data should be stored, `source` contains the rank of the source process and `status` will contain some information about the message after it has been received.

2.3.2 Collective communication

Collective communication always involves every process in the specified communicator. Consequently, every process in the communicator has to call the routine. As the inversion algorithm will exclusively use collective communication, I will explain the important functions in more detail:

Broadcast: `MPI_BCAST(buf,count,type,root,comm)`

All processes must specify the same `root` and `comm`. This function acts like a send on the root process and a receive on all other processes. After completion every process will hold the same data in `buf`.

Scatter:
```
MPI_SCATTER(sendbuf,sendcount,sendtype,
recvbuf,recvcount,recvtype,root,comm)
```

The root process scatters the data in portions specified by `sendcount` and `sendtype` to each process in the given communicator, including itself. This routine explicitly gives the possibility, as opposed to `MPI_BCAST`, to specify different send and receive types. These only have to match if primitive datatypes are used. When using derived datatypes, this restriction is loosened, but one still has to ensure via the variables `sendcount` and `recvcount`, that the amount of data is identical.

Gather: `MPI_GATHER(sendbuf,sendcount,sendtype,`
`recvbuf,recvcount,recvtype,root,comm)`

The root process gathers data specified by `sendcount` and `sendtype` by each process in the given communicator, including itself and stores it in portions specified by `recvcount` and `recvtype` sorted by process rank in the `recvbuf`.

```
MPI_ALLGATHER(sendbuf,sendcount,sendtype,
recvbuf,recvcount,recvtype,comm)
```

While the `MPI_GATHER` command stored the gathered data only in the root process' memory, this command will provide every process in the communicator with a copy of the gathered data.

Reduce: `MPI_REDUCE(sendbuf,recvbuf,count,type,op,root,comm)`

This routine will take the i-th element of the `sendbuf` of every process within the communicator, reduce it using some reduction operation `op` and store it in the i-th element of the root process in `recvbuf`. This is done for $i = 1$...`count`.
For primitive datatypes there exists a number of predefined operators: Maximum, minimum, sum, product, bitwise/logical and/or/xor, rank of the process holding the maximum or minimum value. It is also possible to create user-defined operants.

```
MPI_ALLREDUCE(sendbuf,recvbuf,count,type,op,comm)
```

While the `MPI_REDUCE` command stored the reduced data only in the root process' memory, this command will provide every process in the communicator with a copy of the reduced data.

3 Physical Motivation

In the path integral formalism the vacuum expectation value of an operator $\mathcal{O}\left[\bar{\psi}, \psi, A\right]$ can be calculated through the path integral (using Euclidean time)

$$\left\langle \mathcal{O}\left[\bar{\psi}, \psi, A\right] \right\rangle = \frac{\int \mathcal{D}\psi \mathcal{D}\bar{\psi} \mathcal{D}A \; \mathcal{O}\left[\bar{\psi}, \psi, A\right] \; e^{-\mathcal{S}[\bar{\psi}, \psi, A]}}{\int \mathcal{D}\psi \mathcal{D}\bar{\psi} \mathcal{D}A \; e^{-\mathcal{S}[\bar{\psi}, \psi, A]}} \tag{20}$$

The integration measures (e.g. $\mathcal{D}\psi$) are defined as the multi-dimensional integral over all degrees of freedom of the fields (cp. eqs. (10) and (13)). $\mathcal{S}$ denotes the action defined by the integral over the Lagrangian density of lattice QCD.

$$\mathcal{S} = \sum_{f,x,y} \bar{\psi}_f(x) K[A](x,y) \psi_f(y) + \mathcal{S}_G \tag{21}$$

$K[A](x,y)$ represents the lattice Dirac operator, which depends on the gauge configuration, and $\mathcal{S}_G$ the action of the gauge fields.
If the spacetime coordinates are also understood as vector indices, all indices can symbolically be merged into one.
In order to account for the anti-commuting character of the fermion field operator, the 'classical' fermion fields need to be Grassmann degrees of freedom, hence anti-commuting numbers. By following the rules for integration over Grassmann variables, the fermionic part of eq. (20) can be performed analytically:

$$\left\langle \psi^{\alpha_1} \cdots \psi^{\alpha_j} \bar{\psi}^{\beta_1} \cdots \bar{\psi}^{\beta_j} \ \mathcal{O}_G[A] \right\rangle \tag{22}$$

$$= \frac{\int \mathcal{D}A \ \mathcal{O}_G[A] \ e^{-\mathcal{S}_G[A]} \left\langle \psi^{\alpha_1} \cdots \psi^{\alpha_j} \bar{\psi}^{\beta_1} \cdots \bar{\psi}^{\beta_j} \right\rangle_A}{\int \mathcal{D}A \ e^{-\mathcal{S}_G[A]}} \tag{23}$$

Occurring determinants were set constant and canceled out (this is the quenched approximation). The factor $\left\langle \psi^{\alpha_1} \cdots \psi^{\alpha_j} \bar{\psi}^{\beta_1} \cdots \bar{\psi}^{\beta_j} \right\rangle_A$ is the n-point function of the non-interacting theory on a static gauge background, which can be calculated from products of the inverse of the Dirac operator:

$$\begin{aligned} \left\langle \psi^{\alpha_1} \cdots \psi^{\alpha_j} \bar{\psi}^{\beta_1} \cdots \bar{\psi}^{\beta_j} \right\rangle_A &= \frac{\int \mathcal{D}\psi \mathcal{D}\bar{\psi} \ \psi^{\alpha_1} \cdots \psi^{\alpha_j} \bar{\psi}^{\beta_1} \cdots \bar{\psi}^{\beta_j} e^{-\mathcal{S}_F[\bar{\psi},\psi,A]}}{\int \mathcal{D}\psi \mathcal{D}\bar{\psi} \ e^{-\mathcal{S}_F[\bar{\psi},\psi,A]}} \\ &= \xi_j \sum_P (-1)^{\sigma_P} \mathrm{K}[A]^{-1}_{\alpha_1 \beta_{P_1}} \cdots \mathrm{K}[A]^{-1}_{\alpha_j \beta_{P_j}} \end{aligned} \tag{24}$$

$\xi_j = (-1)^{j(j-1)/2}$, P stands for all permutations of the indices β_i and σ_P is the signum of the permutation P.

This results shows where the inverse of the Dirac matrix is needed in lattice QCD.

The inverse of the Dirac matrix $K^{-1}(x,y)$ is also called propagator, because it is the amplitude for a non-interacting particle to propagate from y to x. We introduce a new symbol for the propagator:

$$G(x,y) \equiv K^{-1}(x,y) \tag{25}$$

One spacetime column of this matrix, i.e. $G(x,y)$ for fixed y, is the solution vector to the following system of equations:

$$\sum_{x'} K(x,x') G(x',y) = \delta_{x,y} \mathbb{1}_{\text{Dirac,color}} \ \text{, for all } x \tag{26}$$

At this point, one should mention that a similar equation has to be solved elsewhere in lattice QCD. The so called sequential propagators (Σ) are the solution to this equation:

$$\sum_{x'} K(x,x') \gamma_5 \Sigma(x')^\dagger = \gamma_5 S(x)^\dagger \ \text{, for all } x \tag{27}$$

$S(x)$ is the sequential source.

In order to be able to use the same program for both problems, the algorithm should be able to invert the Dirac matrix on any given source vector.

4 The Inversion Algorithm

In order to make programming and the notation a little easier, we number all lattice sites using the following mapping:

$$x_\mu = an_\mu \longmapsto i = n_0 + n_1 N_0 + n_2 N_0 N_1 + n_3 N_0 N_1 N_2 \ , \qquad \text{with } 0 \le i < N_{\text{sites}} \tag{28}$$

This way we can define all quantities using only one index for the spacetime coordinate and the equation that has to be solved is:

$$\sum_j^{N_{\text{sites}}} D(i,j)G(j) = S(i) \tag{29}$$

G is one row of the propagator, S the source vector, and D the Dirac operator. Dirac and color indices have been dropped as they play no role in the algorithm or the parallelization.

4.1 Parallelization

The crucial part of the parallelization is to let every process handle just a subset of lattice sites. Requiring that the number of processes involved in the calculation is a divisor of N_{sites}, the workload can be equally distributed, if every process handles N_{sub} sites, with

$$N_{\text{sub}} = N_{\text{sites}}/n_{\text{cores}}. \tag{30}$$

In this manner we define 'local' quantities, that will satisfy the following relations:

$$S^r_{\text{loc}}(i) = S(r \cdot N_{\text{sub}} + i) \tag{31}$$

$$G^r_{\text{loc}}(i) = G(r \cdot N_{\text{sub}} + i) \tag{32}$$

$$D^r_{\text{loc}}(i,j) = D(r \cdot N_{\text{sub}} + i, j) \tag{33}$$

$0 \le i < N_{\text{sub}}$ and $0 \le j < N_{\text{sites}}$. r is the rank of the process with $0 \le r < n_{\text{cores}}$. Every process 'knowns' only his part of the complete vector or matrix. This knowledge is sufficient for vector addition, as the i-th component of the resulting vector only involves the i-th components of the operands. But operations that involve the complete set of sites, need to be emulated using MPI. Operations of this kind, that will be used in this algorithm, are calculation of a scalar product of two vectors and matrix on vector multiplication.

4.1.1 Vector on vector

The dot product of two vectors A, B with N_{sites} entries that are distributed over all processes is calculated in the following way:

$$c = A \cdot B = \sum_{r}^{n_{\text{cores}}} c^r_{\text{loc}} = \sum_{r}^{n_{\text{cores}}} \left(\sum_{i}^{N_{\text{sub}}} A^r_{\text{loc}}(i) B^r_{\text{loc}}(i) \right) \tag{34}$$

In FORTRAN-code:

```
c_loc=dot_product(A_loc,B_loc)

call MPI_ALLREDUCE(c_loc,c,1,MPI_DOUBLE_PRECISION,&
MPI_SUM,MPI_COMM_WORLD,ierr)
```

The sum over i in eq. (34) is performed in the routine `dot_product`, the sum over r is performed by the MPI reduction routine, that provides every process with the result of the full dot product in variable `c`.

4.1.2 Matrix on vector

In this case, a square matrix D with N_{sites} rows that are distributed over all processes, has to be multiplied on a column vector A with N_{sites} entries that are also distributed over all processes. The local parts of the resulting vector B are calculated:

$$B^r_{\text{loc}}(i) = \sum_{j}^{N_{\text{sites}}} D^r_{\text{loc}}(i,j) A(j) \text{ , for all } 0 < i < N_{\text{sub}} \tag{35}$$

To perform this calculation very process has to have the complete vector A in its memory. So the different parts of the vector A have to be gathered, which is done by the corresponding MPI routine:

```
call MPI_ALLGATHER(                                 &
      A_loc,N_dirac*N_color*N_sub,MPI_DOUBLE_COMPLEX, &
      A,    N_dirac*N_color*N_sub,MPI_DOUBLE_COMPLEX, &
    MPI_COMM_WORLD,ierr)

do i=0,n_sub-1
  B_loc(i)=dot_product(D_loc(i,:),A)
enddo
```

As a result of this operation, every process will have only his local part of the result vector B.

4.2 The sparse Dirac operator

The ultra-local nature of the Dirac operator is reflected in the fact that the Dirac matrix in configuration space is a sparse matrix . In our case of the chirally improved Dirac operator, only one in 1016 entries was non-zero. These entries only connect points with a physical distance of less than $2.3a$. This means that every point is only connected to 128 neighbors.
Of course internally the Dirac matrix is not handled as a sparse two-dimensional array, but instead only non-zero entries are stored. If N_{neigh} is the number of neighboring sites that are connected via the Dirac matrix to every single site (this might also include the site itself), the matrix is stored in a $N_{\text{sites}} \times N_{\text{neigh}}$ two-dimensional array with one spacetime and a neighbor index:

$$D^r_{\text{loc,sparse}}(i,j) \text{ , with } 0 \leq i < N_{\text{sub}}, 0 \leq j < N_{\text{neigh}} \tag{36}$$

In order to be able to reconstruct the full matrix, another two-dimensional array is needed. It contains the addresses of every neighbor of all local lattice sites:

$$a^r_{\text{loc,neigh}}(i,j) \in \{0,\dots,N_{\text{sites}}-1\} \text{ , with } 0 \leq i < N_{\text{sub}} \text{ and } 0 \leq j < N_{\text{neigh}} \tag{37}$$

It returns the global site index of the j-th neighbor of the i-th site handled by the process with rank r.
With this new definition of D the matrix on vector multiplication of eq. (35) can be performed in the following way:

$$B^r_{\text{loc}}(i) = \sum_j^{N_{\text{neigh}}} D^r_{\text{loc,sparse}}(i, a^r_{\text{loc,neigh}}(i,j)) A(a^r_{\text{loc,neigh}}(i,j)) \tag{38}$$

4.3 BiCG-stab

Using the methods described in this section, the parallelization of an inversion algorithm of the Dirac matrix is straight forward. First an algorithm is chosen that is suited best for the kind of problem: Sparse matrix, non-symmetric, etc. We chose the Bi-conjugate gradient stabilized (BiCG-stab) algorithm, which is an iterative method that takes an arbitrary starting vector and transforms it iteratively into the solution vector after a finite number of steps. Usually the algorithm terminates before the exact solution is reached, e.g. when the estimated distance of the intermediate vector from the solution vector is smaller than the specified tolerance.

The next step is the parallelization of the inversion routine. This is done by replacing the vector operations of the sequential routine by the corresponding parallel versions for local quantities as described above. For the BiCG-stab this includes only vector on vector multiplication and Dirac matrix on vector multiplication, which also proves the implementation of the sparse version of the Dirac matrix, as described above, to be quite easy.

If, as in the case with the free quark propagators, one has to do a whole set of inversion, where the sources are identical and the Dirac matrices only differ by a constant (i.e. the different quark masses)

$$\sum_{j}^{N_{\text{sites}}} \left(D_{\text{massless}}(i,j) + m_{\text{quark}}\right) G(j) = \delta(i,k), \tag{39}$$

a modified version of the algorithm is used. The so called BiCGstab-M [1] can perform these different inversions using only as many matrix-vector operations as the solution of the most difficult single system requires.

This is however not possible for the calculation of sequential propagators, because the sources are also mass dependent. (cp. eq. 27)

5 Inversion on the HPC-Cluster Regensburg

The inversions of the Dirac operator on sequential sources using configurations from the Bern-Graz-Regensburg collaboration were performed in the test-run of the 2009 new high-performance computing cluster (HPCC) at the University of Regensburg. This cluster is made up of 187 nodes that are connected via *InfiniBand*-interfaces. Every node possesses two *AMD Barcelona B3 2,2 GHz* processors with four processors-cores each. This gives a total of 1496 cores that can execute instructions in parallel. The main-memory of the nodes is 16Gib or 32 Gib. They all have 250 to 500Gib local hard-disc storage and are connected to NFS-servers via Gigabit-Lan. The servers run *SUSE Enterprise Linux (Novell)* and have numerous compiler suites and MPI libraries installed. The queuing system *Torque* handles the distribution and execution of jobs on the nodes.
As explained above, 24,000 independent inversion had to be performed. As each inversion takes about 24GiB of memory, at least two nodes had to be used if one of them was a 16GiB machine. Which implies that, in order to achieve maximum performance, one inversion has to run on 16 cores in parallel (8 from each node).
The program is written in Fortran.
The total task took one week to complete.

References

[1] B. Jegerlehner. Multiple mass solvers. *Nuclear Physics B - Proceedings Supplements*, 63:958, 1998. http://arxiv.org/abs/hep-lat/9708029v1. 4.3