Hybrid Monte Carlo for less colourful lattice QCD

Tadeusz Janowski

August 18, 2011

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2011
Abstract

The aim of this project was to improve the performance and maintainability of the 2 colour QCD code developed by physicists at Swansea University. The code was modified to follow FORTRAN90 standards, which greatly improved its readability. Initial profiling has helped to identify the most computationally expensive subroutines, which turned out to be hdslash, hds slash and MPI_WAIT. Changes that improve cache reuse and reduce the number of branches from within loops were introduced in hdslash and hds slash subroutines. They resulted in an increase of serial performance by about 6%. MPI call structure was changed to reduce waiting time and unnecessary halo swaps were removed. This resulted in dramatic improvement in scaling behaviour of the code. The random number generator was found to be broken and was replaced with a different one. Also, the number generator was replaced with a parallel version to give consistent results on different processor grids. Initially the code was restricted to run on processor grids which are cubic in spatial dimensions. These restrictions were lifted, which resulted in a higher flexibility of the code.
# Contents

1. **Introduction**  
   - 1

2. **Preliminary work**  
   - 2
      - 2.1 Introduction ............................................. 2
      - 2.2 Restructuring the code ................................. 2
      - 2.3 Profiling ............................................... 4
      - 2.4 Correctness testing .................................... 4
      - 2.5 Scripting and job submission ......................... 5

3. **Serial improvements**  
   - 6
      - 3.1 Introduction ............................................. 6
      - 3.2 Merging loops to improve data locality ............... 6
      - 3.3 Removing loop variable dependent conditional expressions from the body of the loop 7
      - 3.4 Results ................................................ 7

4. **Parallel improvements**  
   - 8
      - 4.1 Mixed OpenMP/MPI code ................................. 8
         - 4.1.1 Why use the hybrid code? ......................... 8
         - 4.1.2 OpenMP in FORTRAN77 ............................. 10
         - 4.1.3 Parallelizing loops using OpenMP ................. 10
      - 4.2 Improving MPI structure ............................... 10
         - 4.2.1 Redundant halo swaps ............................ 12

5. **Functional improvements**  
   - 13
      - 5.1 Replacing the random number generator .............. 13
         - 5.1.1 Motivation ........................................... 13
         - 5.1.2 Overview of pRNGs ................................. 13
         - 5.1.3 Linear Congruential ................................ 14
         - 5.1.4 Shift Register ...................................... 14
         - 5.1.5 Lagged Fibonacci .................................. 15
         - 5.1.6 Mersenne Twister .................................. 15
      - 5.2 pRNGs in FORTRAN ................................. 16
      - 5.3 Parallel implementation of RNGs .................... 17
      - 5.4 Testing RNGs ........................................... 18
      - 5.5 Removing restrictions on the dimensions of processor grid 19
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Initial profiling results on 16 processors of HECToR XE6</td>
<td>4</td>
</tr>
<tr>
<td>4.1</td>
<td>Schematic initial MPI structure of the code for 1D array of processes</td>
<td>11</td>
</tr>
<tr>
<td>4.2</td>
<td>Schematic improved MPI structure of the code for 1D array of processes</td>
<td>12</td>
</tr>
<tr>
<td>4.3</td>
<td>Schematic improved MPI structure of the code for 1D array of processes. Neighbouring processes not connected</td>
<td>12</td>
</tr>
<tr>
<td>5.1</td>
<td>Parking lot test for linear congruential pRNG with $a=100$, $c=0$, $m=2^{32}$</td>
<td>18</td>
</tr>
<tr>
<td>5.2</td>
<td>Parking lot test for linear congruential pRNG with $a=1664525$, $c=1013904223$, $m=2^{32}$</td>
<td>19</td>
</tr>
<tr>
<td>6.1</td>
<td>$S_{1024}$ against number of processors for the original code</td>
<td>23</td>
</tr>
<tr>
<td>6.2</td>
<td>$S_{1024}$ against number of processors for the new code</td>
<td>23</td>
</tr>
<tr>
<td>6.3</td>
<td>Number of threads vs time at 16 processor cores</td>
<td>25</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to thank all the people who helped me with the project, especially my supervisor, Dr Chris Johnson for his invaluable advice in all areas of the project. I would also like to thank Prof. Simon Hands and Dr Jonivar Skullrend from Swansea University, who gave me an opportunity to work on this project.
Chapter 1

Introduction

Quantum Chromodynamics (QCD) is the study of strong interactions that bind quarks together to create hadrons, in particular protons and neutrons that are constituents of atomic nuclei. [1] Even though the theory is well known to work in vacuum, not much is known about the physics at finite particle densities. This knowledge could be useful, for example, in studying occurring inside neutron stars. [2] When attempting to model QCD problems arise when the Euclidean action becomes complex. For that reason, instead of studying the full SU(3) (i.e. 3 colour) theory we study the simpler SU(2) (2 colour) theory, where the action is guaranteed to be real.

Lattice simulations are one of the main ways of researching QCD. Unlike the alternative perturbation theory, they do not depend on any particular values of input parameters, making them a flexible and powerful tool.

The aim of this project is to improve the performance of the 2 colour QCD code, originally developed by physicists at Swansea University. The main algorithms used by the program are the conjugate gradient algorithm [3] for matrix inversion which allows computation of fermionic fields [4] and Hybrid Monte Carlo [5] which is used to generate gauge fields with desired probability. The initial parallelization of the code consists of a 4 dimensional (3 spatial + time) domain decomposition with halo swaps. The following report outlines the changes made to the code and explains how they affect its performance and maintainability.

These changes were divided into three categories:

- Serial - ones that improve the serial performance of the code
- Parallel - ones that improve parallel scaling of the code
- Functional - ones that fix existing bugs or add new features to the code

Each of these categories is described in detail in a corresponding chapter of this report.

Furthermore, the performance of the code was measured and the results are described in the performance section.
Chapter 2
Preliminary work

2.1 Introduction

The following section describes the work that was done before any performance change have been made. The first section describes the changes that make the code more readable and maintainable. The second section shows the initial profiling results, which helped to identify the main sources of overhead, both in serial and in parallel.

2.2 Restructuring the code

The original code was written in FORTRAN77. Unfortunately, this means that the syntax was cumbersome and outdated in many places. FORTRAN90, on the other hand, provides an excellent alternative syntax. In order to make the code more readable and hence easier to work with some changes to the structure of the code had to be made. We focused on the following aspects:

- indentation
- removing line markers (i.e. numbers at the beginning of the lines)
- using array syntax wherever possible

These are briefly outlined below.

Converting do loops to FORTRAN 90 standard: Before

```
do n i=a,b
   CODE
   n continue
```

After

```
do i=a,b
```
Removing goto statements: The if-goto statements. Before:

```plaintext
if(condition) goto n
CODE
n continue
```

After:

```plaintext
if(.not.condition) then
    CODE
end if
```

Combined with exiting subroutines. Before:

```plaintext
subroutine [name]
CODE
if(condition) goto n
MORE CODE
n continue
    return
end
```

After:

```plaintext
subroutine [name]
    CODE
    if(condition) return
    MORE CODE
    return
end
```

Mixed if-goto and do loop. Before

```plaintext
do n i=a,b
    if(condition) goto n
    CODE
    n continue
```

After

```plaintext
do i=a,b
    if(condition) continue
    CODE
end do
```

These changes have been made extensively throughout the entire code.
2.3 Profiling

In order to identify fragments of the code that take the most out of the runtime and therefore require the most attention in the performance improvement process, profiling was needed. The profiling was done on HECToR XE6 using vendor-specific tool called CrayPat.

The summary is presented in Figure 2.1

Clearly, the hdslash and hdslashd subroutines require the most attention as far as serial code is concerned. The largest parallel overhead is introduced by MPI_WAIT calls.

2.4 Correctness testing

When modifying a code it is always important to have a way to check whether it works correctly. It is also important to be able to revert to the latest working version of the code should a change ruin it. To safely keep track of the changes, a CVS repository was set up.

Initially, we could simply compare the output of the original code with the modified one. As long as the output values were within the rounding error the program was assumed to be working.

After replacing the random number generator this approach was no longer feasible. To check further updates for correctness, it was necessary to temporarily set the random number generator to always return a fixed value. This value had to be chosen by trial and error in such a way to avoid generating ‘Not A Number’ (NaN) floating point values. Of course the random number generator can not be tested this way, so instead it was tested separately using e.g. ‘parking lot’ test as presented in Figure 5.2.

Some odd behaviours was observed with the imaginary part of qbqb variable. As the number of processors changes, all other variables, including the real part of qbqb stay the same (up to rounding errors), but the imaginary part of qbqb varies greatly. This behaviour requires further investigation.
2.5 Scripting and job submission

Performance testing can become a tedious task when a large number of jobs must be submitted to the queue. Scripting allows to execute this process automatically. For the sake of performance testing many runs on different numbers of processors are required. Unfortunately, the program requires recompiling every time the number of processors is changed. For this reason many executable copies of the program are required, each compiled to run on different number of processors.

Another limitation is the number of jobs that can be submitted to the batch queue. One way to get around this problem is using job chaining, which means having a batch script submit another script upon completion. In practice this means adding the following line to the end of the script file:

```
qsub newjob.pbs
```

In summary, the job submission process would proceed as follows:

1. Compile the code into several executables each corresponding to different processor number.
2. Create several batch scripts, each to run a different executable and chain them together.
3. Submit the first script in the chain.
Chapter 3

Serial improvements

3.1 Introduction

One of the ways of improving the performance of the code is to reduce the time taken in time consuming subroutines. As the profiling has shown, these are hslash and hslashd. The following chapter will therefore focus on changes that will reduce the time spent in these subroutines and results are presented.

3.2 Merging loops to improve data locality

One way of improving the performance of dslash and dslashd is merging loops. For example there are two consecutive loops with the following structure:

```fortran
  do i=1,kvol
    do mu=1,3
      do igorkov=1,ngorkov
        Phi(1,igorkov,i)=Phi(1,igorkov,i)+some terms
        Phi(2,igorkov,i)=Phi(2,igorkov,i)+some terms
      enddo
    enddo
  enddo
```

Having two such loops in succession can cause unnecessary cache misses, because each segment of Phi array has to be loaded into cache twice - once per every loop. Merging these loops would require only one load which is certainly an improvement.
3.3 Removing loop variable dependent conditional expressions from the body of the loop

In the dslash and dslashd subroutines there are some conditional statements inside the loop. For example:

```fortran
  do igorkov=1,ngorkov
    if(kvol.le.4)then
      condition 1
    else
      condition 2
    end if
  enddo
```

This is inefficient, because it introduces unnecessary branches, which will lose time in case the processor takes the wrong branch. A better way of writing this code (which admittedly works correctly only for $ngorkov \leq 4$) is:

```fortran
  do igorkov=1,4
    condition 1
  end do
  do igorkov=5,ngorkov
    condition 2
  end do
```

3.4 Results

This section describes the results of two runs: one before and one after the above changes have been made. Both were run on the backend of Ness. In both cases the lattice size used was $16^3 \times 32$, and all other parameters of sizes.h file were fixed. Total runtime has been measured in both cases and the comparison is presented in a table below.

<table>
<thead>
<tr>
<th>Run time before the changes</th>
<th>Run time after the changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>57.5 s</td>
<td>54.2 s</td>
</tr>
</tbody>
</table>

This corresponds to 6% improvement in the final run time, which is a small but significant improvement.
Chapter 4

Parallel improvements

4.1 Mixed OpenMP/MPI code

4.1.1 Why use the hybrid code?

One of the main goals of parallel programming is to achieve the best possible performance. How does creating a hybrid code compare with pure OpenMP or pure MPI code? Let us begin with some theoretical considerations. First, let us assume that we have a hybrid code that consists of four disjoint parts:

- A part that scales with the number of MPI processes and OpenMP threads
- A part that scales with the number of MPI processes but not with the number of OpenMP threads
- A part that scales with the number of OpenMP threads but not with the number of MPI processes
- A serial part

Let us label the serial time taken by each of these parts as $T_{mix}$, $T_{MPI}$, $T_{OMP}$ and $T_s$ respectively. Call the number of MPI processes used $m$ and the number of OpenMP threads used $t$. Also assume that we are trying to find the optimal OpenMP/MPI distribution on $p = mt$ processors. Then the total time is given by:

$$T(m, t) = \frac{T_{mix}}{p} + \frac{T_{MPI}}{m} + \frac{T_{OMP}}{t} + T_s$$  \hspace{1cm} (4.1)

Since $m = p/t$:

$$T(p, t) = \frac{T_{mix}}{p} + \frac{T_{MPI}t}{p} + \frac{T_{OMP}}{t} + T_s$$  \hspace{1cm} (4.2)

We want to find the value of $t$, for fixed $p$, that will minimise this function. To find it, use the derivative:

$$\frac{\partial T(p, t)}{\partial t} = \frac{T_{MPI}}{p} - \frac{1}{t^2} T_{OMP}$$  \hspace{1cm} (4.3)
$T$ has an extremum at $\partial T/\partial t = 0$, i.e. at:

$$t = \sqrt{\frac{T_{OMP}}{T_{MPI}}}$$  \hspace{1cm} (4.4)

We can check that it is indeed a minimum by performing a second derivative test:

$$\frac{\partial^2 T}{\partial t^2} = \frac{2}{t^3} T_{OMP} \geq 0$$  \hspace{1cm} (4.5)

where the inequality follows from the fact that both $T_{OMP}$ and $t$ are non-negative. Thus, the solution is a local minimum.

As a check, we can use $p = mt$ to find the optimal value of $m$:

$$m = \sqrt{\frac{T_{MPI}p}{T_{OMP}}}$$  \hspace{1cm} (4.6)

which is the expected result, given that equation 4.1 is symmetric under simultaneous $T_{OMP} \leftrightarrow T_{MPI}$ and $m \leftrightarrow t$ exchange, and after making these exchanges we arrive at equation 4.4.

Let $t_{max}$ be the node size on the given architecture. Then, as long as:

$$1 < \sqrt{\frac{T_{OMP}}{T_{MPI}}} < t_{max} \leq p$$  \hspace{1cm} (4.7)

the hybrid code is expected to perform better than pure MPI ($t = 1$) or pure OpenMP ($t = p$) code.

The above reasoning is simplified. It does not take into account communication and memory access overheads, which depend on the quality of the interconnect and are therefore architecture dependent. It also does not take into account the fact that, for most systems, it is impossible to use only some processors in the node and all of them have to be used instead. For example, if we have an architecture that consists of 24 processor nodes and we find that the optimal number of threads is 20, then we may have to either use 12 or 20 threads instead to avoid having idle processors.

Performance is not the only aspect of hybrid codes that may be advantageous. Another aspect is portability. When porting the code that has been optimised for one architecture to a different architecture some performance loss will often be observed. Hybrid codes, thanks to their flexibility, can be quickly modified to accommodate for the new architecture. As an example, when porting a code from a hybrid architecture with 24-processor nodes to the one with 16 processor nodes, changing the number of threads in use (e.g. from 24 to 16) can result in a performance boost.

It may seem that hybrid codes are architecture-dependent and that is indeed the case, however hybrid architecture is becoming increasingly popular, which in itself is a very good motivation for using hybrid codes.
4.1.2 OpenMP in FORTRAN77

OpenMP implementation in FORTRAN77 is more restrictive than the one in FORTRAN90. Each OpenMP instruction must be headed by c$omp (and not !$omp or *$omp, which are treated as comments). Furthermore, the 72-characters per line restriction also applies to F77 (but not F90) OpenMP instructions. Lines that are too long must therefore be split, with continuation lines starting with c$omp&.

The above remarks are relevant due to the way that the FORTRAN PGI compiler interprets file extensions. The ‘.f’ files are interpreted as FORTRAN77 files as far as OpenMP is concerned (above remarks apply) and ‘.f90’ files are compiled as FORTRAN90 files. Since the source code files is called ‘su2hmc.f’, the above restrictions apply.

4.1.3 Parallelizing loops using OpenMP

The greatest benefit is expected by parallelizing the loops that take the most of the program’s run time, i.e. ones in hdslash and hdslashd subroutines. These are of the form:

\[
\begin{align*}
doi=1,kvol \\
doi ddirac=1,ndirac \\
domi=1,3 \\
\text{Phi}(1,ddirac,i)=\text{Phi}(1,ddirac,i)+... \\
\text{Phi}(2,ddirac,i)=\text{Phi}(1,ddirac,i)+... \\
\text{igork1}=... \\
[\text{more terms of the same form}] \\
end do \\
end do \\
end do
\end{align*}
\]

The only private variable, other than inner loop variables ddirac and mu, is igork1. All other variables can be declared as shared. Note that since the outermost loop will be decomposed (this makes sense as kvol, the local lattice volume, is large), the Phi array can be declared as shared, because all instances of Phi are indexed with i and therefore no access conflicts can occur.

4.2 Improving MPI structure

The message passing structure of the code is created from several ‘building block’ subroutines: zuphaloswapdir and zdnhaloswapdir, cuphaloswapdir and cdnhaloswapdir. Each of these is responsible for sending off and receiving into upper and lower halo respectively in the specified direction. Both subroutines have a similar MPI structure:
one non-blocking send (MPI_ISEND) followed by a receive (MPI_RECV) and finally wait for the send to finish (MPI_WAIT). Using pseudocode:

```fortran
subroutine zuphaloswapdir(arguments)
    MPI_ISEND(...) !send halo
    MPI_RECV(...) !receive halo from neighbouring process
    MPI_WAIT(...) !wait for the first non-blocking send to finish
end subroutine zuphaloswapdir
```

Because of this structure, sending data in more than one direction becomes inefficient, because each process has to wait twice for each direction (once for up halo and once for down halo). This is illustrated in Figure 4.1. One obvious improvement is to have only one wait instruction just before the data that is being swapped becomes necessary as shown in Figure 4.2. One way to implement it is to replace receives with non-blocking receives (MPI_IRECV) and removing MPI_WAIT call from these routines. This requires changing the argument list of these calls by adding two ‘request’ integers - one for non-blocking send and one for non-blocking receive. The synchronisation can be performed using MPI_WAITALL subroutine. Example pseudocode:

```fortran
subroutine zuphaloswapdir(arguments, request1, request2)
    !Added 2 requests to the argument list: one for the non-blocking send and the other for non-blocking receive
    MPI_ISEND(...) !send halo, return request1
    MPI_IRECV(...) !receive halo from neighbouring process, return request2
end subroutine zuphaloswapdir

subroutine wait(requestarray)
    MPI_WAITALL(requestarray,...)
end subroutine wait
```
4.2.1 Redundant halo swaps

There are several arrays that halo swaps are used on in the main code. These are: \( u11 \), \( u12 \), \( dk4m \), \( dk4p \). They have a structure such that one of their dimensions is of size base+halo, where base is the base size (no halo) of the array, and halo is the halo size. The elements 1 to base are unaffected by halo swaps. Loops over these arrays typically have an index going from 1 to base. In some instances, the array index is just a loop variable, i.e. there is no need for a halo swap.

An example would be \( dk4p \) array inside \( hdslash \) subroutine. The loop only uses \( dk4p(i) \), (not \( dk4p(ind(i,mu)) \), which can include halos), so this halo swap is redundant and can be removed, because \( i \) can only point the non-halo elements. This is illustrated in Figure 4.3.
Chapter 5

Functional improvements

5.1 Replacing the random number generator

5.1.1 Motivation

During the tests it was discovered that the random number generator occasionally generates numbers from outside of the desired [0,1] interval. There was a ‘quick fix’ in the code that replaced negative numbers with a small positive random number and numbers greater than 1 with 1, while printing a warning to the standard output when any of these replacements occur. Unfortunately, for certain choices of parameters this would result in over 300 000 warnings (13.7 MB file of warnings). This undesirable behaviour results in a non-uniformity of random numbers generated and effectively giving the wrong results. For this reason the random number generator had to be replaced completely.

Another improvement that was made was making the random number generator work in parallel. This allows for the output of the program to be the same for any number of processors which in turn helps in correctness testing and debugging.

5.1.2 Overview of pRNGs

In this section a selected few pseudo-random number generators (pRNGs) are described. Other pRNGs exist but they are not as good for our purposes.

In all pRNG methods, a certain bit pattern, called the seed is used to generate random numbers by considering the seed as an integer and dividing by a certain factor. A new seed is then created by performing certain operations on the existing seed. [6] [7]

The first pair of generators has a small seed, which is an advantage in a parallel implementation, where multiple seeds are necessary. The second pair have larger seeds but produce better quality random numbers. In the context of random numbers, the term...
‘quality’ refers to period length (length of non-repetitive sequence of random numbers) and also correlations between generated numbers.

5.1.3 Linear Congruential

Linear congruential pRNG generates the new seed $X_{n+1}$ from the old seed $X_n$ by using the following formula:

$$X_{n+1} = aX_n + c \mod m$$  (5.1)

where $a, c$ and $m$ are integer constants. A commonly used value for $m$ is $2^{32}$, which means that by choosing $X_k$ to be a 4-byte (32 bit) unsigned integer, the modulo operation can be omitted, because it will overflow naturally.

Advantages:
- Easy to implement
- Fast
- Used a relatively small seed (32 bits)

Disadvantages:
- Lower quality random numbers compared to other methods

5.1.4 Shift Register

This family of algorithms selects individual bits from the bit pattern of the seed to generate the value of the new bit. It is typically done using the exclusive or (EOR) operation. The new bit is added to the end of the beginning sequence and the last bit is removed.

Example: 3-bit RNG, EOR on 2nd and 3rd bits:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

This corresponds to the sequence: 1,4,2,5,6,7,3,1. This RNG therefore has a period of 7.

Advantages:
- Fast
- Only 1 seed

Disadvantages:
- Lower quality random numbers than alternative generators

### 5.1.5 Lagged Fibonacci

This family of pRNGs uses an array of seeds rather than an individual seed. Only the last value in the array is used to generate random numbers. New seeds are generated from the set of previous ones by using a certain operation, typically EOR. The idea is very similar to shift register algorithms, but instead of working on individual bits it works on seeds.

Advantages:
- Good quality of random numbers
- R(250,103) is widely used

Disadvantages:
- Requires to store seeds in a large array, which uses a lot of memory
- Some problems have been reported with the Ising model [8]

### 5.1.6 Mersenne Twister

The Mersenne Twister [9] algorithm uses a complicated set of bitwise operations (mainly shifts, bitwise EOR and bitwise AND) on an array of seeds to generate new seeds.

Advantages:
- Very long period; the shortest repetitive sequence of random numbers has length of $2^{19937}$.
- Good quality of random numbers.

Disadvantages:
- Uses an array of 624 seed - large memory requirement.
- Slow
- Relatively complicated to program
5.2 pRNGs in FORTRAN

The FORTRAN standard supports bitwise operations. These are implemented as sub-
routines. The most useful in this context are:

<table>
<thead>
<tr>
<th>Name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAND</td>
<td>bitwise and</td>
</tr>
<tr>
<td>IEOR</td>
<td>bitwise exclusive or</td>
</tr>
<tr>
<td>IOR</td>
<td>bitwise or</td>
</tr>
<tr>
<td>ILS</td>
<td>left shift</td>
</tr>
<tr>
<td>IRS</td>
<td>right shift</td>
</tr>
</tbody>
</table>

FORTRAN does not support unsigned integers. This creates problems during type con-
version from integer (seed) to real (random number). Fortunately, signed integers can
be successfully used to generate numbers in $[-0.5, 0.5]$ range instead of the usual $[0, 1]$ range. To convert them to $[0, 1]$ range, we can simply add 0.5. Using pseudocode:

```
unsigned integer seed

RNG=(real)seed/(2^number_of_bits)
```

works as well as:

```
integer seed

RNG=(real)seed/(2^number_of_bits) + 0.5
```

Strictly speaking these are not the same, because a given bit pattern will generate dif-
ferent random numbers. To see this, consider a pattern which consists of 0s only. In
both the signed and unsigned cases this corresponds to the integer 0, so in the first case,
the generated number will be 0.0 and in second case it will be 0.5. However, since the
seeds are generated randomly, any bijective mapping onto a set of uniformly distributed
real numbers will give the desired effect.

If we wanted to get exactly the same random number as in the unsigned case, we could
use:

```
integer seed

RNG=(real)seed/(2^number_of_bits)
if (RNG<0) RNG=RNG+1
```

This solution is less efficient, because of the additional branching point.
5.3 Parallel implementation of RNGs

In this section, for generality, the term ‘seed’ will refer to array of seeds in case of R(250,103) and Mersenne Twister. Even though they have not been implemented, they still remain a point of interest.

The main challenge of random number generation is obtaining the same results regardless of the number of processors given initial seed. In this particular case, this can be obtained by assigning different seeds for different elements of the local lattice.

Random numbers are either generated on the master process and broadcasted (par_granf subroutine) or generated within loops over the local volume, kvol. The latter is the source of the problem. To fix it, an array of seeds of size kvol was created. Each element of this array was indexed by $global\_index = kvol \times nprocs + local\_index$, where $nprocs$ is the number of processors. This ensures that the same set of indices is present in the simulation regardless of the number of processors.

Seeds can then be initialised by setting them equal to the global index value of the corresponding element (or any function of it).

The implementation requires drastic changes to the interface of the ranf function. It has to take an argument that is either an index (probably neater) or an actual seed. It is also better to use a separate module for RNG that has the seed array as a private variable. The template for such module would be as follows:

module rngmod
  implicit none
  include ‘sizes.h’ !Just to get the value of kvol
  private
  public :: ranset
  public :: ranget
  public :: ranf
  integer(kind=something) seed(kvol)

contains

  subroutine ranset(newseed)
    integer(kind=something),intent(in) :: newseed(kvol)
  ...

  subroutine ranget(getseed)
    integer(kind=something),intent(out) :: newseed(kvol)
  ...

  real function ranf(int index)
    integer, intent(in) :: index
  ...
As a result, each call to ranf() will be replaced by the call to ranf(i) (i denotes the index). The parallel function, par_granf, will have to be changed to use the seed value with global index 1 (which is seed(1) at process 0).

### 5.4 Testing RNGs

One very simple test that can be performed is usually called the ‘parking lot’ test. In this test, pairs of random numbers are plotted on the 2D plane. A good random number generator should show no distinguishable patterns.

Figures [5.1] and [5.2] are two examples, one of a bad pRNG and one of a good one. Overall, this test is not very good; it took a lot of effort to obtain a regular pattern.

In the actual code, we decided to implement the linear congruential RNG whose test is shown in Figure [5.2]. Although the quality of random number may be insufficient for our purposes, this RNG was the simplest to implement and has a small seed, which is an advantage when parallel implementation is considered.
5.5 Removing restrictions on the dimensions of processor grid

Initially, the program was set up to handle only processor grids that are cubic in spatial dimensions, i.e. of the form $a^3 \times b$. This makes the main code much simpler, because the numbers of processors on the edges of the hypercube corresponding to spatial directions, called $k_{x}$, $k_{y}$ and $k_{z}$, could be replaced with a single variable called $k_{size}$. This simplifies the main code (where $k_{size}$ could be used rather than $k_{x}$, $k_{y}$ and $k_{z}$) at the expense of some functionality. This functionality, however, is very desirable for performance testing.

In order to make the program work without the aforementioned restriction, it is necessary to make appropriate replacements of $k_{size}$ with $k_{x}$, $k_{y}$ and $k_{z}$. This process is very intuitive. For example, $k_{size} \times k_{size} \times k_{size}$ term has to be replaced with $k_{x} \times k_{y} \times k_{z}$, because it clearly represents a volume of the grid. When describing how to replace each $k_{size}$, symmetry arguments are especially convincing: since the spatial directions are symmetric, there is no ‘special’ direction that has to be treated differently from others. To see this simply rotate the system by 90 degrees. For that reason, for example, the term $k_{z} \times k_{x} \times k_{x}$ has to be wrong, because it treats the $x$ direction differently than $y$ and $z$ directions. Thus it is clear that $k_{x} \times k_{y} \times k_{z}$ is the only combination that preserves the symmetry.

The piece of code presented below, however, does not preserve the symmetry.

```plaintext
do  k=1,ksizet
  do  l=1,ksize
    do  j=1,ksize
```
do i=1,ksize
    ic=(((k-1)*ksize+(l-1))*ksize+(j-1))*ksize+i
    ...
end do
end do
end do
end do

The variable ic is a unique label for each element of the local lattice. The symmetry is broken for labelling purposes only. The code can be modified as follows:

do k=1,ksizet
    do l=1,ksizez
        do j=1,ksizey
            do i=1,ksizex
                ic=(((k-1)*ksizez+(l-1))*ksizey+(j-1))*ksizex+i
                ...
            end do
        end do
    end do
end do

Where we chose x to be fastest moving index followed by y, z and t.
Chapter 6

Performance Measurement

6.1 The effect of random number generation

Despite the fact that the profiling results show that random number generation takes very little of the program runtime, it can have a great effect on the final timings. This is mostly because of the classical time evolution loop:

\[
\text{do iter=1,itermax} \\
\quad \text{...} \\
\quad \text{if(condition) then} \\
\quad \quad \text{...} \\
\quad \quad \text{exit} \\
\quad \quad \text{end if} \\
\end{array}
\]

Because there is branching out of the loop, the number of iterations may vary depending on the branching condition, which is dependent on the values of the fields which in turn depend on the random number generator.

Loops within congradp and congradq have a very similar structure:

\[
\text{do niterx=1,niterc} \\
\quad \text{...} \\
\quad \text{if(condition) then} \\
\quad \quad \text{...} \\
\quad \quad \text{return} \\
\quad \quad \text{end if} \\
\end{array}
\]

Since replacing the random number generator is part of the project, there is a need to find a way to mitigate this problem. This can be done either by setting itermax=niterc=1 in sizes.h file or by (temporarily) replacing the random number generator to always return a fixed value. The latter option is more attractive, because it also helps in testing
6.2 Scaling

Measuring the scaling is non-trivial, considering the following observations:

- Low number of processors is unsuitable for running large problems, due to time and memory constraints (large arrays must be stored on a limited number of processors).
- Large number of processors is suitable for large problems only, because smaller problems would require less computation per processor and parallel overheads would dominate.

For these reasons it makes sense to measure performance for two separate cases: once for low number of processors and once for large number of processors.

When measuring performance, the main quantity of interest is the speedup, defined as:

\[ S(p) = \frac{T(1)}{T(p)} \]  

(6.1)

where \( T(n) \) denotes the time it takes for the program to complete at \( n \) processors.

When the number of processors is large, measuring speedup is no longer possible, simply because large problems are unfeasible for serial execution. Instead, we define a quantity \( S_n(p) \) and call it ‘Speedup relative to \( n \) processors’:

\[ S_n(p) = \frac{T(n)}{T(p)} \]  

(6.2)

with \( T(n) \) defined as above. Note that \( S_1 \) is just the usual speedup definition.

Here, a ‘large’ problem is a \( 64^3 \times 32 \) lattice on a \( 8^3 \times t \) processor grid, where \( t \in \{2, 4, 8, 16\} \)

Unfortunately, due to carelessness and budget restrictions, simulations for large number of processors were run on slightly different set of parameters. Even though this affects individual timings, speedups should remain the same.

Scaling of the original code can be seen in Figure 6.1. Even though there is a speedup from 1024 to 2048 processors, the program starts slowing down for numbers of processors greater than 2048. In contrast, speedups for the new version of the program are presented in Figure 6.2. It can be seen that speedups, although still far from ideal, are much better than they were in the case of the original code.

The main cause of this difference are unnecessary MPI_WAIT calls and also some redundant halo swaps which increase the time taken by the communication between
Figure 6.1: $S_{1024}$ against number of processors for the original code

Figure 6.2: $S_{1024}$ against number of processors for the new code
MPI processes. The amount of time gained due to parallelism is then outweighed by the added communication cost, which causes the program to run slower as the number of processors increase. Changes to the parallel structure of the code had therefore a huge and beneficial impact on the performance of the code at large number of processors.

### 6.3 Hybrid code testing

The purpose of this test is to compare the performance of mixed OpenMP/MPI code with pure MPI and pure OpenMP code. During the test, the total number of processors in use remains fixed, but it is distributed differently among MPI processes and OpenMP threads. The number of MPI processes, $m$ for a fixed number of processors $p$ will be given by

$$m \in \{ k \in \mathbb{Z} | p \mod k = 0 \}$$

(6.3)

In other words, $m$ is an integer divisor of $p$. For any $m$ in the set, the number of OpenMP threads, $t$, is given by:

$$t = \frac{p}{m}$$

(6.4)

which is an integer, because $p \mod m = 0$. We therefore have $tm = p$, as desired.

Note the special case of $t = 1, m = p$ corresponds to the pure MPI code. Also note that the case $t = p, m = 1$ corresponds to the pure OpenMP case. Thus, on the $m$ against time graph, the endpoints correspond to ‘pure’ cases, while points in between are ‘hybrid’ cases.

The results ran on Ness for $p = 16$ on a $16^4$ lattice with $13 \times p$ processor grid are presented in Figure 6.3.

It is clear that pure OpenMP code does not work well, despite Ness being a shared memory machine. This is most likely the result of many threads trying to access the shared memory and thus using up the bandwidth. Hybrid code with 2 threads seems to perform slightly better than pure MPI code, although this may due to measurement error. Hybrid code, however, gives more flexibility, allowing for simulations to run on small lattices compared to the size of processor grid. In pure MPI code doing so would cause each process to work on a small part of the lattice and the simulation time would be dominated by communication. Adding OpenMP allows for larger chunks of the lattice per MPI process and decompose loops using OpenMP threads.
Figure 6.3: Number of threads vs time at 16 processor cores
Chapter 7

Conclusions

In conclusion, several changes were introduced, which are beneficial to the correctness, performance and maintainability of the code.

The code was restructured to follow FORTRAN90 syntax, which is more readable than older FORTRAN syntax. Buggy random number generator has been replaced with a working one and it has been modified to work in parallel. Changes have been made to hdl and hdlhd to improve the serial performance. The order of MPI calls was changed to reduce the waiting time and unnecessary communications were removed. Thread parallelism was added using OpenMP to create a hybrid code.

The performance results showed a great improvement in parallel scaling of the code. Furthermore, since the serial version has also become faster, the overall performance is much higher than that of the original code, especially at large numbers of processors.

Further work may include:

- Replacing the random number generator with a one of higher quality that is more suitable for the given problem
- Measuring correlations between subsequent lattice configurations to determine the optimal choice of parameters for the simulation
- Measuring the performance on a different architecture
- Obtaining more performance results
- Investigating the behaviour of the imaginary part of qbqb variable.
Appendix A

List of changes

A.1 Extensive changes

This section outlines the changes that occur throughout the code. These include:

- replacing the do loops using the FORTRAN90 standard
- removing goto statements
- using the array syntax
- fixing the indentation

A.2 hdslash and hdslashd subroutines

The changes to hdslash and hdslashd include:

- Merging of loops
- Removing the conditional statement from the loop by splitting it into two loops
- Adding OpenMP calls
- Removing redundancies in halo swaps

The comparison of hdslash before and after changes is presented below. Before:

```fortran
subroutine hdslash(Phi,R,u11,u12)
  c
  c calculates Phi = M*R
  c
  implicit none
  include "precision.h"
  include "sizes.h"
```

After:
c include common block definition
#include "common_mat.h"
#include "common_para.h"
#include "common_neighb.h"
#include "common_dirac.h"

complex(kind=cmplxkind) u11(kvol+halo,ndim),u12(kvol+halo,ndim)
complex(kind=cmplxkind) Phi(2,4,kvol+halo),R(2,4,kvol+halo)

integer idirac,ic,i,mu,igork1

c write(6,*), 'hi from hdslash'
c

c Makes references to

c R(.,.,iu(i,mu)) and R(.,.,id(i,mu)) with ncpt = 8 ** NOT 16 **
c need a zhaloswapall

c also u(id(i,mu),mu)
c so need a zuphaloswapdir(mu) for u(mu)

Also dk4p(id(i,4)) and dk4m(id(i,4)), ncpt = 1

need to do a cuphaloswapdir(4)
c
Actually, this routine does not seem to access dk4p(id(),..) so one of those halo swaps could be omitted

call zhaloswapall(R, 8)

do mu = 1, ndim
  call zuphaloswapdir(u11(1,mu), 1, mu)
  call zuphaloswapdir(u12(1,mu), 1, mu)
end do

call cuphaloswapdir(dk4p,1,4)
call cuphaloswapdir(dk4m,1,4)
c
mass term
do i=1,kvol
doi=1,ndirac
doi=1,nc
Phi(ic,dirac,i)=R(ic,dirac,i)
enddo
enddo
enddo

cWilson term
do i=1,kvol
doi=1,ndirac
domu=1,3
Phi(1,dirac,i)=Phi(1,dirac,i)
&-akappa*(u11(i,mu)*R(1,dirac,iu(i,mu))
&+u12(i,mu)*R(2,dirac,iu(i,mu))
&+conjg(u11(id(i,mu),mu))*R(1,dirac,id(i,mu))
&-u12(id(i,mu),mu)*R(2,dirac,id(i,mu)))
Phi(2,dirac,i)=Phi(2,dirac,i)
&-akappa*(-conjg(u12(i,mu))*R(1,dirac,iu(i,mu))
&+conjg(u11(i,mu))*R(2,dirac,iu(i,mu))
&+conjg(u12(id(i,mu),mu))*R(1,dirac,id(i,mu))
&+u11(id(i,mu),mu)*R(2,dirac,id(i,mu)))
enddo
enddo
enddo

cDirac term
do i=1,kvol
doi=1,ndirac
domu=1,3
igork1=gamin(mu,dirac)
Phi(1,dirac,i)=Phi(1,dirac,i)
&+gamval(mu,dirac)*
& (u11(i,mu)*R(1,igork1,iu(i,mu))
&+u12(i,mu)*R(2,igork1,iu(i,mu))
&-conjg(u11(id(i,mu),mu))*R(1,igork1,id(i,mu))
&+u12(id(i,mu),mu)*R(2,igork1,id(i,mu)))
Phi(2,dirac,i)=Phi(2,dirac,i)
&+gamval(mu,dirac)*
& (-conjg(u12(i,mu))*R(1,igork1,iu(i,mu))
&+conjg(u11(i,mu))*R(2,igork1,iu(i,mu))
&-conjg(u12(id(i,mu),mu))*R(1,igork1,id(i,mu))
&-u11(id(i,mu),mu)*R(2,igork1,id(i,mu)))
enddo
c Timelike Wilson term
  do i=1,kvol
    do idirac=1,ndirac
      Phi(1,idirac,i)=Phi(1,idirac,i)
      & -dk4p(i)*(u11(i,4)*R(1,idirac,iu(i,4))
      & +u12(i,4)*R(2,idirac,iu(i,4)))
      &-dk4m(id(i,4))*(conjg(u11(id(i,4),4))*R(1,idirac,id(i,4))
      & -u12(id(i,4),4) *R(2,idirac,id(i,4)))
      Phi(2,idirac,i)=Phi(2,idirac,i)
      & -dk4p(i)*(-conjg(u12(i,4))*R(1,idirac,iu(i,4))
      & +conjg(u11(i,4))*R(2,idirac,iu(i,4)))
      &-dk4m(id(i,4))*(conjg(u12(id(i,4),4))*R(1,idirac,id(i,4))
      & +u11(id(i,4),4) *R(2,idirac,id(i,4)))
    enddo
  enddo

c Timelike Dirac term
  do i=1,kvol
    do idirac=1,ndirac
      igork1=gamin(4,idirac)
      Phi(1,idirac,i)=Phi(1,idirac,i)
      &+dk4p(i)*
      & (u11(i,4)*R(1,igork1,iu(i,4))
      & +u12(i,4)*R(2,igork1,iu(i,4)))
      &-dk4m(id(i,4))*
      &(conjg(u11(id(i,4),4))*R(1,igork1,id(i,4))
      & -u12(id(i,4),4) *R(2,igork1,id(i,4)))
      Phi(2,idirac,i)=Phi(2,idirac,i)
      &+dk4p(i)*
      & (-conjg(u12(i,4))*R(1,igork1,iu(i,4))
      & +conjg(u11(i,4))*R(2,igork1,iu(i,4)))
      &-dk4m(id(i,4))*
      & (conjg(u12(id(i,4),4))*R(1,igork1,id(i,4))
      & +u11(id(i,4),4) *R(2,igork1,id(i,4)))
    enddo
  enddo
c return
end
subroutine hdslash(Phi,R,u11,u12)
  c
  calculates Phi = M*R
  c
implicit none
include 'omp_lib.h'
include "precision.h"
include "sizes.h"
include common block definition
include "common_mat.h"
include "common_para.h"
include "common_neighb.h"
include "common_dirac.h"

complex(kind=cmplxkind) u11(kvol+halo,ndim),u12(kvol+halo,ndim)
complex(kind=cmplxkind) Phi(2,4,kvol+halo),R(2,4,kvol+halo)

integer idirac,ic,i,mu,igork1
  c
double precision t1, t2
  c
write(6,*) 'hi from hdslash'
  c

Makes references to
R(.,.,iu(i,mu)) and R(.,.,id(i,mu)) with ncpt = 8  ** NOT 16  **
need a zhaloswapall
also u(id(i,mu),mu)
so need a zuphaloswapdir(mu) for u(mu)
Also dk4p(id(i,4)) and dk4m(id(i,4)), ncpt = 1
need to do a cuphaloswapdir(4)
Actually, this routine does not seem to access dk4p(id(),...) so
one of those halo swaps could be omitted

  c

  call zhaloswapall(R, 8)
do mu = 1, ndim
    call zuphaloswapdir(u11(1,mu), 1, mu, reqarray(mu),
    &   reqarray(mu+ndim))
    call zuphaloswapdir(u12(1,mu), 1, mu, reqarray2(mu),
    &   reqarray2(mu+ndim))
end do

call cuphaloswapdir(dk4p,1,4,reqarray3(1),reqarray3(2))
call cuphaloswapdir(dk4m,1,4,reqarray3(3),reqarray3(4))

! Mass term
Phi(1:nc,1:ndirac,1:kvol)=R(1:nc,1:ndirac,1:kvol)

! Wilson term

! t1=omp_get_wtime()
c$omp parallel do default(none) shared(kvol,ndirac,phi,
c$omp& akappa,u11,u12,R,id,dk4p,iu,dk4m,gamval,gamin)
c$omp& private(mu,idirac,
c$omp& igork1)
do i=1,kvol
    do idirac=1,ndirac
        do mu=1,3
            Phi(1,idirac,i)=Phi(1,idirac,i)
            & -akappa*( u11(i,mu)*R(1,idirac,iu(i,mu))
            & +u12(i,mu)*R(2,idirac,iu(i,mu))
            & +conjg(u11(id(i,mu),mu))*R(1,idirac,id(i,mu))
            & -u12(id(i,mu),mu) *R(2,idirac,id(i,mu))
            &
            Phi(2,idirac,i)=Phi(2,idirac,i)
            & -akappa*(-conjg(u12(i,mu)))*R(1,idirac,iu(i,mu))
            & +conjg(u11(i,mu))*R(2,idirac,iu(i,mu))
            & +conjg(u12(id(i,mu),mu))*R(1,idirac,id(i,mu))
            & +u11(id(i,mu),mu) *R(2,idirac,id(i,mu))
            &
            igork1=gamin(mu,idirac)
            Phi(1,idirac,i)=Phi(1,idirac,i)
            & +gamval(mu,idirac)*
            & ( u11(i,mu)*R(1,igork1,iu(i,mu))
            & +u12(i,mu)*R(2,igork1,iu(i,mu))
            & -conjg(u11(id(i,mu),mu))*R(1,igork1,id(i,mu))
            & +u12(id(i,mu),mu) *R(2,igork1,id(i,mu)))
        end do
    end do
end do
Phi(2, idirac, i) = Phi(2, idirac, i) 
& + gamval(mu, idirac) * 
& (-conjg(u12(i, mu)) * R(1, igork1, iu(i, mu)) 
& + conjg(u11(i, mu)) * R(2, igork1, iu(i, mu)) 
& - conjg(u12(id(i, mu), mu)) * R(1, igork1, id(i, mu)) 
& - u11(id(i, mu), mu) * R(2, igork1, id(i, mu)))

Enddo

Phi(1, idirac, i) = Phi(1, idirac, i) 
& - dk4p(i) * (u11(i, 4) * R(1, idirac, iu(i, 4)) 
& + u12(i, 4) * R(2, idirac, iu(i, 4))) 
& - dk4m(id(i, 4)) * (conjg(u11(id(i, 4), 4)) * 
& R(1, idirac, id(i, 4)) 
& - u12(id(i, 4), 4) * R(2, idirac, id(i, 4)))

Phi(2, idirac, i) = Phi(2, idirac, i) 
& - dk4p(i) * (-conjg(u12(i, 4)) * R(1, idirac, iu(i, 4)) 
& + conjg(u11(i, 4)) * R(2, idirac, iu(i, 4))) 
& - dk4m(id(i, 4)) * (conjg(u12(id(i, 4), 4)) * 
& R(1, idirac, id(i, 4)) 
& + u11(id(i, 4), 4) * R(2, idirac, id(i, 4)))

igork1 = gamin(4, idirac)

Phi(1, idirac, i) = Phi(1, idirac, i) 
& + dk4p(i) * 
& (u11(i, 4) * R(1, igork1, iu(i, 4)) 
& + u12(i, 4) * R(2, igork1, iu(i, 4))) 
& - dk4m(id(i, 4)) * 
& (conjg(u11(id(i, 4), 4)) * R(1, igork1, id(i, 4)) 
& - u12(id(i, 4), 4) * R(2, igork1, id(i, 4)))

Phi(2, idirac, i) = Phi(2, idirac, i) 
& + dk4p(i) * 
& (-conjg(u12(i, 4)) * R(1, igork1, iu(i, 4)) 
& + conjg(u11(i, 4)) * R(2, igork1, iu(i, 4))) 
& - dk4m(id(i, 4)) * 
& (conjg(u12(id(i, 4), 4)) * R(1, igork1, id(i, 4)) 
& + u11(id(i, 4), 4) * R(2, igork1, id(i, 4)))

Enddo

Enddo

C$OMP end parallel do

return

End
A.3 The random number generator

The subroutines related to the random number generation were moved to newly created rngmod.f file. The ranf function is now a function of the seed to ensure maximum flexibility. The body of ranf function was replaced with a linear congruential random number generator.

A.4 Changes to par_mpi.f

In zuphaloswapdir, zdhaloswapdir cuphaloswapdir and cdhaloswapdir MPI_RECV calls were replaced with MPI_IRECV calls and MPI_WAIT calls were removed. The request for both MPI_ISEND and MPI_IRECV were added to the argument list.

A new subroutine wait was added.
Bibliography


