Testing the Columbia code.

A.N. Jackson & S. Booth

Revision: 1.3 Date: 2001/10/03 15:33:36

Contents

1 Proposed Testing Framework

1.1 Parallelism Issues

Some of the test cases should be done on big lattices, as there are subtle communication bugs that will only show with more than 32 nodes. Therefore, the code should ensure the same results can be calculated over a range of machine partition sizes.

1.2 Input Parameters

One of the issues is how to maintain the tests as the input parameters to the code gets changed. One way to proceed would be to write a library of perl scripts to write the input parameters to the codes. This would localise most of the input parameters to the code in one place. (This approach is recommended in Software Test Automation by Fewster and Graham).

1.3 Output Format

One of problems with testing is comparing the output against the ”correct” output. The issue is how to extract the important output (pion correlators, and residues) from the unimportant output (dates). It is suggested that we use XML to tag the important information. For example, the output could look ”something like”:

```
<notimportant>
STATUS:solver:make_source:0:Using point source on spin 0 colour 0
 STATUS:solver:make_clover:0:Making clover for p = 1
 STATUS:solver:make_source:0:Using point source on spin 0 colour 0
 STATUS:solver:solver_g5_driver:0:real residue is
</notimportant>
<residue> 0.97820832E-09 </residue>
<notimportant>
STATUS:solver:solver_data_save:0:Saving solver data...
 STATUS:solver:solver_mpp_save:0:Saving solver data...
</notimportant>
```

The plan would be to use ”attributes” to index things such as the residue as well.

An XML parser such as expat (http://www.jclark.com/xml/expat.html) could be used to do the comparison. Another option might be to use xmldiff (http://www.logilab.org/xmldiff/) We could of course try to use perl/grep/awk to extract the information we need, but this is somewhat opposed to the motivation for using XML.

2 Types of Test

2.1 Gauge Invarience

We need a gauge invariance test subroutine in the Columbia code. We can use the standalone UKQCD code to produce gauge rotated configurations.

http://www.ph.ed.ac.uk/ukqcd/collaboration/t3e_codes/transform_gauge/gaugetransform.html

These configurations could be read into the Columbia code.

3 Parallel Random Number Generation

Concerning the need to reproducability of results over a system of a given size, no matter how that system is decomposed over a set of processors. To achieve this, we must consider distributed and decomposition independent (Pseudo Random Number Generators) PRNGs.

Date: Tue, 11 Sep 2001 12:41:07 +0100 (BST)
From: ”S.Booth” ¡spb@epcc.ed.ac.uk¿
Reply-To: s.booth@epcc.ed.ac.uk
To: qcdoc-app@epcc.ed.ac.uk
Subject: Parallel random number generation

First of all some trivial notation

- S - the state of a generator
- X - the output of a generator
- U - update transform (maps one state to the next)
- F - output function (maps state to output)
- P - period of generator

i.e.

$$S_{n+1} = U.S_n \tag{1}$$

$$X_n = F.S_n \tag{2}$$

$$U^P \equiv I \tag{3}$$

Assume we are generating random numbers in large batches (size N) corresponding to a lattice worth of random numbers. (The problem of accept/reject algorithms that consume different amounts of random numbers on different sites is not too hard but I’ll neglect it here for clarity)

The easiest way of doing this in a decomposition independent way is to store an RNG state at each site and use U^N as the update transform.

$$S_0 S_1 S_2 ... S_{N+1} \tag{4}$$

goes to

$$S_N S_{N+1} S_{N+2} ... S_{2N-1} \tag{5}$$

giving the same result as if a single generator had been run on a sequential machine.

Vectorisable random number generators often used this trick, e.g for a multiplicative generator:

$$X_{n+1} = a.X_n \text{mod} M \tag{6}$$

implies

$$X_{n+64} = b.X_n \text{mod} M; \quad b = a^{64} \text{mod} M \tag{7}$$

allowing the sequence to be vectorised with a vector length of 64 storing 64 words of state instead of 1. (Linear congruential generators LCGs are very similar to this). Even for a large offset $Va^V \text{mod} M$ can be calculated in $\log(V)$ time.

The same trick can also be used with other PRNG algorithms e.g. lagged fibonacci. The difference here being that U^N may be more expensive to compute that U. A lagged fibonacci update might be of the form

$$X_n = X_{n-p} \pm X_{n-q} \text{mod} M \tag{8}$$

U^N will need q multiplications and additions (mod M)

$$X_{n+N} = \sum_{i=1}^{q} a_i.X_{n-i} \text{mod} M \tag{9}$$

(Note this cost is independent of the size of N but the cost of calculating the constants $\{a_1, a_2 ... a_q\}$ is proportional to $\log(N)$, however you only need to do this once)

This additional cost is an artifact of how the original generator was designed. The lagged fibonacci generators are just a special case of a more general class of generator the Multiply recursive generator (MRG)

$$X_n = \sum_{i=1}^{q} a_i.X_{n-i} \text{mod} M \tag{10}$$

It is just as easy (if not easier) to design just as good a generator where U^N is cheap to evaluate and U is the more expensive. There is no such thing as a "good" random number generator. Generators should really be anlysed in terms of their prospective use. However RNG designers usually use some fairly arbitrary criteria based on short distance correlation in the sequance that are thought to be valid for the majority of applications.

As far as a QCD simulation is concerned the important thing is the statistical properties of U_x U_y U_z U_t and U_s the operators that encode the transformation the RNG state from one lattice site to

the next and between algorithmic iterations. For a naive sequential implementation:

$$\begin{aligned} U_x &= U \\ U_y &= U^{Lx} \\ U_z &= U_y^{Lx} \\ U_t &= U_z^{Lz} \\ U_s &= U_t^{Lt} \end{aligned}$$

and we could investigate these properties with a varient of the spectral test.

In the parallel implementation we can take

$$U_s = U \tag{11}$$

and are free to choose $U_x\ U_y\ U_z\ U_t$ for their statistical properties rather than as a side effect of the order we loop through the lattice.

The only problem with this approach is the increase in storage needed for the generator state (q times the size of the lattice). This is only the "working" storage, as the states at each site are all related by $U_{x,y,z,t}$ it is only necessary to checkpoint a single state the others can be regenerated easily.

Its possible to trade off computation for storage in three ways.

1. We can get away with a smaller q if we use a prime modulus M rather than a power of 2. a 32 bit power of 2 generator has maximum period $2^31.(2^q - 1)$ if M is prime the maximum period is $(M^q - 1)$ Prime modulus at approx 31 bits won't be too expensive if we have 64-bit integer types available

2. Only store a single RNG state per processor. This is an alternative approach to getting decomposition independence where we arrange for each lattice of random numbers generated to be the same as if they had been produced by a single processor. Each processor only stores a single RNG state but applies a different transformation at the end of each X, Y, Z, loop across the lattice. These transforms are more expensive than the normal update transform (roughly q^2 additions and multiplies for lagged fibonacci above) This is fine for large local volumes but is probably a significant expense for our target local volumes. (On the T3E I did something like this but used a different decomposition for the random numbers as for the physics lattices. However that requires non local communications)

3. The option of having one generator per sub-cube of the lattice as sugessted by bob would also work fine, this uses less storage than the one generator / site approach but requires greater book-keepin. The problem being that it constrains the possible lattice decompositions to sub-cube boundaries. We could make the size of sub-cube a run-time parameter so the same code can also run with 1 generator/site though this increases the book-keeping overhead even futher.

For HMC the RNG is not a very significant part of the runtime so we have a lot of freedom about what to do. We could also design an interface that could support any of the above approaches making it easy to change later.

At the moment I would suggest a 1 generator/site approach using a prime modulus Multiply recursive generator (MRG) and 5 words of state per site. (that's 5K bytes for a 4^4 lattice) this generator would have a period of $(P^5 - 1)$ with $p = 2^31 - 1$ i.e. approx 2^155

In many ways this is overkill but the RNG is such a small part of the runtime a gold plated solution seems like a good idea to me.