Benchmarking, Profiling and Optimising DL_POLY_3 on HECToR

Valène PELLISSIER

August 24, 2011

Contents

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 About DL_POLY

DL_POLY_3 is a general purpose parallel molecular dynamics simulation package developed at Daresbury Laboratory by W. Smith and I.T. Todorov. The package was developed under the auspices of the Engineering and Physical Sciences Research Council (EPSRC) for the EPSRC's Collaborative Computational Project for the Computer Simulation of Condensed Phases (CCP5), the Computational Chemistry Group (CCG) at Daresbury Laboratory and the Natural Environment Research Council (NERC) for the NERC's eScience project Computational Chemistry in the Environment (eMinerals), directed by M.T. Dove.

It supports both the Velocity and Leap-Frog Verlet algorithms for a large number of ensembles, with a number of different baro- and thermo-stats. Further a very large wide variety of force fields are supported, making simulation of a many very different chemical systems possible. For more details see
`http://www.cse.scitech.ac.uk/ccg/software/DL$_$POLY/MANUALS/USRMAN4.01.pdf/`
which also covers the licensing and availability of the code.

1.2 Optimising

In this report I will present results from optimising a number of routines in DL_POLY_3. Throughout the report the GNU compiler has been used as this was found to be superior to both the PGI and Pathscale compilers for DL_POLY , and for the report a standard test case TEST8 has been used (though in practice other cases were examined as well). This is a simulation of a complex biomolecule, gramicidin A, in water which both exercises many routines in the code and is sufficently large to exhibit good scalability. Professor Richard Catlow has also observed that it is a representative case for the work performed by the Materials Chemistry Consortium. The various input files for this test case are available from :
`ftp://ftp.dl.ac.uk/ccp5/DL$_$POLY/DL$_$POLY$_$4.0/DATA/`
Extensive profiling of this (and related) cases pinpointed a number of potential bottlenecks in the code, and the rest of the report examines the improvements made in these routines, and the impact that the changes make on the run time of DL_POLY.

I shall also compare the performance on 2 of the Cray MPP architectures which have formed the main part of the HECToR service: The XT4 which consists of 4 core nodes connected by the SeaStar network,

and the XE6 (24 core nodes and Gemini). For more details of the architecture see :
`http://www.hector.ac.uk/service/hardware/`

Chapter 2

Optimisations

2.1 *link_cell_pairs*

2.1.1 Results

In table [2.1] a comparison of different runs of testcase TEST8 is shown for different numbers of cores. Bak XT4 and Bak XE6 corresponds to “vanilla” DL_POLY_3 compiled and run on XT4 and XE6 respectively, while Opt link is for DL_POLY with an optimised implementation link_cell_pairs algorithm and run on the XE6. Except for 16 and 512 cores it can be seen that the vanilla code is noticeabaly faster on the XE6 when compared to the XT4. It can also be seen that Opt link is faster than the original code, resulting in a 23% improvement in performance on 512 cores.

Nb. Procs	Bak XT4	Bak XE6	Opt link
16	199.154	218.722	211.924
32	106.790	98.113	95.955
64	63.129	57.494	51.436
128	42.036	39.360	34.150
256	27.471	29.492	23.760
512	22.137	25.951	19.961

Table 2.1: Timing comparison of different runs with Bak on XT4, XE6 and optimised link_cell_pairs on XE6

Table [2.2] shows the percentage improvement of the optimised code compared to the XT4 and XE6. The improvement due to Opt link increases with the number of cores, reaching 23.08 % at 512 processes on the XE6, as noted above. On average, the optimised version of DL_POLY with the optimised link_cell_pairs routine is 10.73 % faster than vanilla code on the XT4, and 11.93 % faster than on the XE6.

Nb. Procs	Opt link comp. to	
	Bak XT4	Bak XE6
16	6.41	-3.11
32	-10.15	-2.2
64	-18.52	-10.54
128	-18.76	-13.24
256	-13.51	-19.44
512	-9.83	-23.08
Average	-10.73	-11.93

Table 2.2: Variation rate of different runs of optimised link_cell_pairs on XE6 with Bak on XT4 and XE6

2.1.2 Modifications

The main modification of *link_cell_pairs* consists of removing the condition $ll > mxlist$ in the loop over *jj* (the inner loop over atoms). This is shown below in Algorithm 1. It should be noted that it is not possible to do the same for the **If** conditon dependent on *rsq* as *rsq* depends on *jj*.

The main point is to divide the loop over jj into two parts : one part when, for every jj in $[j\_start, lct\_start(jc+1)]$, $ll > mxlist$; the second part for all the other cases. For the former, the condition above can be written as $ll < mxlist$ and $ll + jj\_span < mxlist$. As $ll = list(0, i) + 1$ and $j\_span = lct\_start(jc+1) - j\_start$ testing whether $(list(0, i)+1 < mxlist).and.(list(0, i)+1+(lct\_start(jc+1)-1-j\_start) < mxlist)$ is true outside the loop allows us to evaluate whether the $ll > mxlist$ condition need be tested inside the loop. If this new condition is true the $ll > mxlist$ condition in the original code can **never** be true, and so in the the loop over jj it is sufficient just to evaluate the *If* condition over rsq. On the other hand if it is not true the original version of the loop over jj must be executed, with both the conditions over rsq and ll evaluated.

This optimisation should improve the performance of the code whenever $ll > mxlist$ for every *jj* in $[j\_start, lct\_start(jc+1)]$, which is the common case; if this condition is not true it actually indicates a run time error for the code. A comparison of the two algorithms can be seen below.

Algorithm 1 Bak Version of link_cell_pairs main loop

```
for jj=j_start, lct_start(jc+1) do
  !get domain local particle index
  j=at_list(jj)
  !distance in real space
  rsq=(xxx(j)-xxx(i))**2+(yyy(j)-yyy(i))**2+(zzz(j)-zzz(i))**2
  !check cutoff criterion
  if (rsq <= rcsq) then
    !check for overfloat and add an entry
    ll=list(0,i)+1
    if ll > mxlist then
      ibig=Max(ibig,ll)
      safe=.false.
    else
      ibig=Max(ibig,ll)
      list(0,i)=ll
      list(ll,i)=j
    end if
    !end of if-block for cutoff criterion
  end if
  !end of loop over secondary cell contents
end for
```

Algorithm 2 Optimised Version of link_cell_pairs main loop

```
!check if overfloat condition not verified
if ( list(0,i)+1 < mxlist) .and. (list(0,i)+1+( lct_start(jc+1)-1 - j_start) < mxlist) then
  for jj=j_start, lct_start(jc+1) do
    !get domain local particle index
    j=at_list(jj)
    !distance in real space
    rsq=(xxx(j)-xxx(i))**2+(yyy(j)-yyy(i))**2+(zzz(j)-zzz(i))**2
    !check cutoff criterion
    if (rsq <= rcsq) then
      ll=list(0,i)+1
      ibig=Max(ibig,ll)
      list(0,i)=ll
      list(ll,i)=j
    end if
  end for
else
  Same loop over jj than Bak Version
  !end of inner if-block on new condition on mxlist
end if
```

2.2 *ewald_spme_forces*

2.2.1 Results

Table [2.3] compares of the vanilla version of DL_POLY on the XT4 and XE6 with DL_POLY with an optimised version of the routine *ewald_spme_forces*. This evaluates the reciprocal space contribution to the Ewald energy and forces, and uses the SPME algorithm [Darden *et al.*, J. Chem. Phys., **103**, 19, (1995)]. A small decrease in timings can be observed with the Opt ewald version.

In the next table [2.4], as before with *link_cell_pairs*, I show the percentage speed up due to the optimised code. In this case no improvement is found for the 16 processes run, but in all other cases there is a small improvement a with a peak of 13% less time for the 256 processes run.

2.2.2 Modifications

For this function the modifications lie in unrolling the loops which evaluate the representation of the charge density on a grid. As discussed in the reference above this involves evaluating a fairly low order cardinal B-spline, values of 6-16 being common (for TEST8 the value is 8). Unfortunately the inner most loop in the implementation has an upper bound which is at most equal to the order of the spline ($mxspl$

Nb. Procs	Bak XT4	Bak XE6	Opt ewald
16	199.154	218.722	216.172
32	106.790	98.113	94.852
64	63.129	57.494	53.736
128	42.036	39.360	35.639
256	27.471	29.492	27.062
512	22.137	25.951	21.960

Table 2.3: Timing comparison of different runs with Bak on XT4, XE6 and optimised ewald_spme_forces on XE6

Nb. Procs	Opt ewald comp. to	
	Bak XT4	Bak XE6
16	8.55	-1.17
32	-11.18	-3.32
64	-14.88	-6.54
128	-15.22	-9.45
256	-1.49	-8.24
512	-0.80	-15.38
Average	-5.84	-7.35

Table 2.4: Variation rate of different runs of optimised ewald_spme_forces on XE6 with Bak on XT4 and XE6

in the code), while the outer loops are much longer. The optimised version consists, therefore, of using a "Select Case" statement in order to manually unroll the loop over j for the different values maximum values that it can take (up to 8 in this case). This is especially effective as the case construction can be moved into one of the outer loops, thus leaving long inner loops with no conditionals and stride 1 memory access, which are therefore good candidates for vectorisation:

Algorithm 3 Bak Version of ewald_spme_forces main loop

```
for j=1,mxspl do
    qsum=qqc_domain(jj,kk,ll)
    bdxj=qsum*bdxk*bsdx(j,i)
    bdyj=qsum*bdyk*bspx(j,i)
    bdzj=qsum*bdzk*bspx(j,i)
    fix=fix+bdxj
    fiy=fiy+bdyj
    fiz=fiz+bdzj
end for
```

Algorithm 4 Optimised Version of ewald_spme_forces main loop

```
Select Case (mxspl)
Case Default
for j=1,mxspl do
    Calculation
end for
Case ( 0 )
Case ( 1 )
    j=1
    Calculation for j=1
Case ( 2 )
    j=1
    Calculation for j=1
    j=2
    Calculation for j=2
Case ( 3 )
...
Case ( 8 )
    j=1
    Calculation for j=1
...
    j=8
    Calculation for j=8
End Select
```

2.3 *constraints_shake_vv*

For this function, two versions have been implemented. The no frozen Opt constraints relates to an optimised version of constraints_shake_vv which does not implement frozen atoms. Generic Opt constraints takes account of this case. The difference is purely a function of the testcase and can be determined once and for all just after the input files have been read and before the main time stepping loop starts. As the no frozen atoms case is the more common and offers greater potential for optimisation it is useful to differentiate the two cases.

2.3.1 Results

In the following table [2.5], timings obtained from no frozen Opt constraints and generic Opt constraints are presented. The generic version is faster for small number of processes and then the trend changes for greater number of processes. However the differences are small, and may simply be due to noise in the timings.

Nb. Procs	Bak XT4	Bak XE	Opt constraints	
			no frozen	generic
16	199.154	218.722	213.314	210.336
32	106.790	98.113	94.112	92.688
64	63.129	57.494	51.421	53.652
128	42.036	39.360	35.527	34.627
256	27.471	29.492	22.997	24.787
512	22.137	25.951	20.190	21.397

Table 2.5: Timing comparison of different runs with Bak on XT4, XE6 and optimised constraints_shake_vv on XE6

Nb. Procs	No frozen Opt constraints comp. to	
	Bak XT4	Bak XE6
16	7.11	-2.47
32	-11.87	-4.08
64	-18.55	-10.56
128	-15.48	-9.74
256	-16.29	-22.02
512	-8.8	-22.20
Average	-10.65	-11.85

Table 2.6: Variation rate of different runs of no frozen atoms optimised constraints_shake_vv on XE6 with Bak on XT4 and XE6

In tables [2.6] and [2.7] , the percentage improvements are again presented for the no frozen and generic cases respectively. In every case an improvement is observed. For the no frozen case the best improvement is 22%, which is for the 256 and 512 cores runs. The results for the generic version are slightly slower. For this case on average an improvement of 10.26 % si observed, with a peak at 17.55% when compared to the vanilla code on the XE6.

Nb. Procs	Generic Opt constraints comp. to	
	Bak XT4	Bak XE6
16	5.61	-3.83
32	-13.21	-5.53
64	-15.01	-6.68
128	-17.63	-12.02
256	-9.77	-15.95
512	-3.34	-17.55
Average	-8.89	-10.26

Table 2.7: Variation rate of different runs of generic optimised constraints_shake_vv on XE6 with Bak on XT4 and XE6

2.3.2 Modifications

The modifications to constraints_shake_vv mainly concern loops over *k* in $[1, ntcons]$. In the original there are a number of loops over all the possible constraints, and in each of these loops tests are performed to determine whether a pair of atoms is affected by this loop. Instead in the optimised version in the first loop a list is created of the atoms which are affected, and subsequent loops are only over this, shorter, list. In more detail (see the algorithm below) the new list, $lstval$, is of *i*, *j* and *k* for which $i = lstop(1, k)$, $j = lstop(2, k)$ and $(i > 0.and.j > 0).and.(i \leq natms.or.j \leq natms).and.(lfrzn(i) * lfrzn(j) == 0)$ so avoiding all these condiotns in later loops. k has been stored because some external function calls or arrays use k which, in those calls, corresponds to the *old k*. The k-loops became loops over over *k* in $[1, ntval]$ for which $i = lstval(1, k)$, $j = lstval(2, k)$ and the condition $(i > 0.and.j > 0).and.(i \leq natms.or.j \leq natms).and.(lfrzn(i) * lfrzn(j) == 0)$ is tested.

Results obtained for the no frozen atoms version of constraints_shake_vv show a speed up when compared to the generic version. This version is obtained from the previous optimised version when taking off all the *If* conditions over lfrzn. This version is not presented as the main changes are presented on the pseudo algorithm [6].

All those modifications are shown below in the algorithms [5] and [6].

Algorithm 5 Bak Version of constraints_shake_vv

```
while (.not. safe) .and. (icyc < mxshak) do
    icyc=icyc+1
    for k=1,ntcons do
        i=lstop(1,k)
        j=lstop(2,k)
        if (i > 0 .and. j > 0) .and. (i ≤ natms .or. j ≤ natms) then
            dxt(k)=xxx(i)-xxx(j)
            dyt(k)=yyy(i)-yyy(j)
            dzt(k)=zzz(i)-zzz(j)
        else
            dxt(k)=0.0_wp
            dyt(k)=0.0_wp
            dzt(k)=0.0_wp
        end if
    end for
    ...
    for k=1,ntcons do
        i=lstopt(1,k)
        j=lstopt(2,k)
        if (i>0 .and. j>0) .and. (i ≤ natms .or. j ≤ natms) .and. lfrzn(i)*lfrzn(j)=0 then
            ...
        end if
    end for
    ...
    if (.not.safe) then
        for i=1, natms ...
        for k=1,ntcons do
            i=lstopt(1,k)
            j=lstopt(2,k)
            if (i>0 .and. j>0) .and. (i ≤ natms .or. j ≤ natms) .and. lfrzn(i)*lfrzn(j)=0 then
                ...
            end if
        end for
        for k=1,ntcons do
            i=lstopt(1,k)
            j=lstopt(2,k)
            ! for all constrained particles, native and shared
            if (i>0 .and. j>0) .and. (i ≤ natms .or. j ≤ natms) .and. lfrzn(i)*lfrzn(j)=0 then
                ....
            end if
        end for
    end if
end while
```

Algorithm 6 Optimised Version of constraints_shake_vv

```
while (.not.safe) .and. icyc<mxshak do
    icyc=icyc+1
    ntval=0
    for k=1,ntcons do
        i=lstop(1,k)
        j=lstop(2,k)
        if (i>0 .and. j>0) .and. (i ≤ natms .or. j ≤ natms) .and. (lfrzn(i)*lfrzn(j) == 0) then
            ntval=ntval+1
            lstval(1,ntval)=i
            lstval(2,ntval)=j
            lstval(3,ntval)=k
            dxt(ntval)=xxx(i)-xxx(j)
            dyt(ntval)=yyy(i)-yyy(j)
            dzt(ntval)=zzz(i)-zzz(j)
        end if
    end for
    ...
    for k=1,ntval do
        i=lstval(1,k)
        j=lstval(2,k)
        k_old=lstval(3,k)
        listcon(0,k) ⇒ listcon(0,k_old)
    end for
    ...
    if (.not.safe) then
        for i=1, natms ... endfor
        for k=1,ntval do
            i=lstval(1,k)
            j=lstval(2,k)
            k_old=lstval(3,k)
            Change dxx(k) ⇒ dxx(k_old)
            Change dyy(k) ⇒ dyy(k_old)
            Change dzz(k) ⇒ dzz(k_old)
        end for
        for k=1,ntval do
            i=lstval(1,k)
            j=lstval(2,k)
            k_old=lstval(3,k)
            ! for all constrained particles, native and shared
            ! if (i>0 .and. j>0) .and. (i ≤ natms .or. j≤natms) .and. lfrzn(i)*lfrzn(j)=0
            ....
            end if
        end for
    end if
end while
```

2.4 *vdw_forces*

2.4.1 Results

The results of optimising the evaluation of the Van Der Waal's forces are shown in table [2.8].

Nb. Procs	Bak XT4	Bak XE6	Opt vdw
16	199.154	218.722	212.913
32	106.790	98.113	94.590
64	63.129	57.494	56.761
128	42.036	39.360	35.121
256	27.471	29.492	25.631
512	22.137	25.951	21.641

Table 2.8: Timing comparison of different runs with Bak on XT4, XE6 and optimised vdw_forces on XE6

The optimised version of *vdw_forces* shows a speed up of 8% in average over all runs.A peak is observed at 16.61% speed up compare to the Bak version on XE6 for 512 processes run.

Nb. Procs	Opt vdw comp. to	
	Bak XT4	Bak XE6
16	6.91	-2.66
32	-11.42	-3.59
64	-10.09	-1.27
128	-16.45	-10.77
256	-6.7	-13.09
512	-2.24	-16.61
Average	-6.67	-8.00

Table 2.9: Variation rate of different runs of optimised vdw_forces on XE6 with Bak on XT4 and XE6

2.4.2 Modifications

The vdw_forces routine includes an ***If*** condition over *ld_vdw* variable from vdw_module inside the main loop. As this variable is invariant through this loop the code has been optimised by taking the test outside of the loop and generating two versions of the code, one for which ***If*** and one where it is false.

2.5 Best combination

In this section I shall examine the effects of including all the above changes. Again I shall separate the no frozen atom and generic cases.

2.5.1 Results for no frozen atoms case

Here are presented the results obtained by using the modified routines for the non-frozen atoms case. A speed up can be observed for each number of cores.

Nb. Procs	Bak XT4	Bak XE6	Best Comb.
16	199.154	218.722	214.343
32	106.790	98.113	94.078
64	63.129	57.494	52.895
128	42.036	39.360	33.877
256	27.471	29.492	25.176
512	22.137	25.951	19.291

Table 2.10: Timing comparison of different runs with Bak on XT4, XE6 and best combination for non frozen atom cases on XE6

One can observe that the speedup is greater with the number of processes. On average, the best combination for no frozen atom cases is 10.18% faster than the Bak version on XT4, and 11.39% faster than the one on XE6. For 512 processes, the best combination version reaches a peak of 25.66% speedup compare to XE6.

Nb. Procs	Best combination	
	Bak XT4	Bak XE6
16	+7.63	-2.00
32	-11.90	-4.11
64	-16.21	-8.00
128	-19.41	-13.93
256	-8.35	-14.63
512	-12.86	-25.66
Average	-10.18	-11.39

Table 2.11: Variation rate of different runs of best combination for no frozen atom cases on XE6 with Bak on XT4 and XE6

2.5.2 Results for generic cases

In this part, all the cases are taken in account.

Nb. Procs	Bak XT4	Bak XE6	Best Comb.
16	199.154	218.722	214.409
32	106.790	98.113	92.137
64	63.129	57.494	53.182
128	42.036	39.360	33.556
256	27.471	29.492	24.347
512	22.137	25.951	20.363

Table 2.12: Timing comparison of different runs with Bak on XT4, XE6 and best combination for generic cases on XE6

One can observe that the speedup is slightly greater on average than the no frozen combination routines runs, a surprising result but the differences are small and when the above runs were performed large variations in runtime were observed. On average, the best combination is 10.31% faster than the Bak version on XT4, and 11.63% faster than the one on XE6. A peak is reached for 512 processes, with 21.53% speedup compare to XE6. Another one is noticed for 128 cores with a time reduction of 20.65% compare to XT4.

Nb. Procs	Best combination	
	Bak XT4	Bak XE6
16	7.66	-1.97
32	-13.72	-6.10
64	-15.76	-7.50
128	-20.65	-15.25
256	-11.37	-17.45
512	-8.01	-21.53
Average	-10.31	-11.63

Table 2.13: Variation rate of different runs of best combination for generic cases on XE6 with Bak on XT4 and XE6

Chapter 3

Conclusions

3.1 About DL_POLY

This work has concentrated on the serial optimisation of a number of routines in DL_POLY_3. The routines were identified by profiling and the work has resulted in a roughly 10-25% increase in performance dependent upon the number of cores in use. Given that the main driver for this project was ultimately the poor performance of DL_POLY on the XT6 compared to the XT4, to have the code now running on the XE6 appreciably faster than on the original architecture is a very satisfying result.

3.2 Acknowledgments

This project was funded under the HECToR Distributed Computational Science and Engineering (CSE) Service operated by NAG Ltd. HECToR - A Research Councils UK High End Computing Service - is the UK's national supercomputing service, managed by EPSRC on behalf of the participating Research Councils. Its mission is to support capability science and engineering in UK academia. The HECToR supercomputers are managed by UoE HPCx Ltd and the CSE Support Service is provided by NAG Ltd. http://www.hector.ac.uk