DL_POLY_3 Parallel I/O Alternatives at Large Processor Counts

Ilian T. Todorov[a] & Ian J. Bush[b]

a. STFC Daresbury Laboratory, Daresbury, Cheshire, WA4 4AD
b. NAG Ltd., Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR

Abstract

Two methods to deal with the I/O bottleneck when performing classical Molecular Dynamics runs on large MPI task counts are presented. We discuss the advantages and drawbacks of both methods, and present performance data for a typical problem size on both HPCx and HECToR. It is found that one method is markedly superior in terms of time to solution, despite an apparently large communication overhead. However, the better performing method use significantly more memory, and the both the implications of this and possible solutions are discussed.

This is a Technical Report from the HPCx Consortium.

Report available from
http://www.HPCx.ac.uk/research/publications/HPCxTR0806.pdf

1 Introduction

DL_POLY_3[1] is a general purpose package for classical molecular dynamics (MD) simulations developed by I.T. Todorov and W. Smith at STFC Daresbury Laboratory. The main purpose of this software is to enable the exploitation of large scale MD simulations on multi-processor platforms. DL_POLY_3[2] is fully self-contained and written in Fortran 95 in a modularised manner with communications handled by MPI. The rigorous and defensive programming style conforms to the NAGWare95 and FORCHECK95 standards and guarantees exceptional portability. Parallelisation is based on equi-spatial domain decomposition[3] distribution which guarantees excellent load balancing and full memory distribution provided the system's particle density is fairly uniform across space[2]. This parallelisation strategy results in mostly point to point communication with very few global operations, and excellent scaling[4,5].

However, for a practical MD run excellent scaling of the CPU bound portion of the code is only one part of the total solution; it is also necessary to save the results of the calculation to disk. As MD is a time-stepping algorithm, this must be done periodically, and for a typical calculation it is performed every 100-1000 timesteps. As discussed in an earlier technical report[6], it is this saving of data that has now become the bottleneck, especially at high processor counts, and in this report we discuss two possible solutions both of which have been developed since[7].

The remainder of this report is organised as follows. In the next section we discuss in more detail the problem to be solved and earlier methods and their drawbacks. In Section 3, we describe the methods under consideration in this report, and compare and contrast their potential advantages and drawbacks. Section 4, presents the results for the two methods, and the final section discusses these results and sets out possible future developments.

2 The I/O Problem

The main I/O[6] in DL_POLY_3 is, as is the case for most classical MD codes, reading and writing configurations representing frames of the time evolution of a modelled system. These are simply lists of the coordinates, velocities and forces acting on the particles that comprise the system. In DL_POLY_3, this has traditionally been performed using formatted I/O for portability; since the MD run itself may be done on HPC facility or commodity cluster whereas the analysis of the results is often done on a workstation at home or work. Thus the output resembles the following

```
Na+              1

    -184.5476802        -167.5198486        -185.5358832

    -2.881253622        -4.727721670         2.235042811

    -10840.17697        -5695.571326        -2740.726482
```

repeated many times, where the four lines are the atom's identity, position, velocity and the force acting upon it.

Although one may think this looks simple, there is one major complication as a consequence of domain decomposition scheme: As a parallel MD run progresses the original ordering of the atoms is scrambled. Thus while the second atom as input may originally reside on processor 0, as the run proceeds it may migrate through a number of other processors. The net result is that while it was natural at the start of the calculation to read it in as the second atom, at the end it may no longer be true that it is natural to write it out in the same place.

Of course, for the MD run itself this is not an issue. The forces acting on a given atom only depend on the types of atoms that make up its environment, and obviously any numerical labelling of those atoms makes no difference. However, many post-processing software tools rely on the order of the atoms in the output file(s) to be maintained. One example of such is visualisation software that displays the time evolution of the positions of the atoms in the run. Such software must be able to track a given atom from one output to another, and this is typically done by assuming that the position of a given atom within the output is conserved.

Thus for the user it is extremely convenient that the order of the atoms be conserved. On the other hand for the parallel application, it is extremely inconvenient! It implies that a reordering of the data must be performed, an operation that can be expensive for a distributed data application. This reorganisation can be achieved in one of two ways (or a combination of both)

1. Between the CPUs: The CPUs communicate between each other to restore the original ordering of the data
2. On the disk: By examination of the local data each CPU identifies where in the file the records for a given atom should be written, and simply writes to that record.

Historically, option 2 has been used by DL_POLY_3. This is made relatively simple by use of direct access files in Fortran, given the very regular format of the output files, and this is described more fully in[7]. Two methods were implemented and tested

1. SWRITE: Each processor in turn sends it data to processor zero, which deals with all the I/O by use of a direct access file
2. PWRITE: Each processor writes its local data to the appropriate records in the direct access file

As described in reference 7 both these have their drawbacks. SWRITE is obviously not a scalable solution, and further the available disk bandwidth on modern parallel file systems often cannot be saturated by a single processor. On the other hand while PWRITE may scale with processor count, the limited disk bandwidth may cause contention at the disk, and this is normally the case for large numbers of processors. However, this is not the only drawback of the PWRITE method – as the Fortran standard describes the behaviour of a serial code. Therefore, quite what should occur when multiple processes access one direct access file is not well defined, and in practice it is observed that the Cray XT3/4 series using LUSTRE

introduce “spurious” NULL characters into the file when the PWRITE method is used. This problem is ultimately due to incoherence between the CPU and disk caches.

Thus in this report we shall present two portable and potentially scalable solutions.

3 Parallel I/O Alternative

One solution to the portability problems associated with the PWRITE method is to use MPI-IO. We asserted in reference 7 that this is inherently non-portable, and if one treats it as an unformatted file this is true. However, it has been pointed out to us* that as a file created by MPI-IO is simply a byte stream one could create a file which is exactly the same as that a Fortran formatted file by use of Fortran internal writes to create the byte steam that corresponds bit by bit to a Fortran formatted record, and then using MPI-IO to write the record. This is portable from system to system, thus overcoming our previous objections†. This is the first of the two methods we present in this report, which we shall term MWRITE and which is, in fact, now the default method for writing in DL_POLY_3.

While this (all but) solves the portability problem of the PWRITE method, it still has the potential disk contention problems. It also does not solve the problem generic to all the methods that rearrange the data “on the disk”. Each atom is written individually, thus leading to very short I/O transactions and so potentially poor performance. Further, as the ordering of the atoms might be quite random the disk head may have to travel between transactions, also leading to poor performance. The second method we consider tries to address all these issues.

This method, which is ultimately derived from MWRITE and also uses MPI-I/O to write to the disk, allows for a subset of the processors to write the data. For this to occur, data must be gathered from a number of processors onto the I/O processors, so incurring a communication cost, and also a memory overhead as the data structures are now not fully distributed. Once the gather has occurred the data on the I/O processors is then reorganised across these processors into the same order as it was originally read in, again incurring a communications overhead, and also a compute overhead in the sort. This new method is, therefore, an example of reordering “between the CPUs”. As the data is now in order, it may be written many records at a time, thus ensuring good I/O performance. This new method we shall term MWRITE_SORTED.

Therefore, while the two methods reach the same end, they have somewhat different characteristics. We may classify them in terms of compute overhead, communication overhead, memory overhead and disk use efficiency. The MWRITE method has negligible compute, communication and memory overhead, but because the I/O transaction size is small the disk will not be used that efficiently. On the other hand MWRITE_SORTED uses the disk well, but has an appreciable memory and

---

***** by Lucian Anton at NAG Ltd. and David Tanqueray at Cray Inc., who were also kind enough to provide us with an implementation, for which we are very grateful.

† This is not strictly true. For instance which character(s) should be used as a record separator are not well defined. However in practice the portability problems are small, and certainly much less than those posed by the PWRITE method.

communication overheads, and also some compute overheads. However, this last term is small as sorting is a fast operation. These differences are summarized in the table below.

	Compute Overhead	Communication Overhead	Memory Overhead	Disk Efficiency
MWRITE	*Negligible*	*Negligible*	*Negligible*	*Poor*
MWRITE_SORTED	*Small*	*Large*	*Large*	*Good*

4 Results

Both MWRITE solution and a prototype of MWRITE_SORTED, as discussed above, were tested for production of a default dump of a single configuration at the end of a MD run on both HPCx[8], an IBM P575 cluster, and the HECToR[9], a Cray XT4 cluster. A system of 216,000 ions of NaCl was chosen as an average (user usage) size benchmark to run. The particle ordering in the system was not advantageous to any possible domain decomposition, so no particular writing scheme is favoured. Simulations of 500 time-steps with a single configuration dump at the end were carried out on a range of MPI task counts, and for MWRITE_SORTED for a range of dedicated writing processors. Only a single run per fixed number of MPI tasks and writers was carried out and neither of the computational systems was available for exclusive use. All shell environment variables were set to their defaults as in new user accounts. This last point is because, in our experience, the vast majority of users have little or no knowledge of the various I/O options available through environment variables, and simply run with the defaults.

Figure 1 shows the scaling of the two methods on HECToR for the benchmark, with varying numbers of writers for the MWRITE_SORTED method. Also compared is a run with no I/O. It can be seen that the MWRITE_SORTED method is much more efficient at large processor counts, the scaling being very close to that of the run with no I/O. Indeed, it can be seen that for MWRITE, though CPU bound portions of DL_POLY_3 are scaling well, the I/O completely degrades the parallel performance. Thus it is, perhaps unsurprisingly, how the disk is actually accessed that is the crucial aspect. In fact, even in the MWRITE_SORTED method the dominant step, in terms of time, is still the time to write the data to disk, this being typically 95% of the total time involved in the dump of the configuration. In Figure 1, it can be seen that the number of writers in the MWRITE_SORTED method does make a small difference.

In Figure 2, we present the time taken for the MWRITE_SORTED method as a function of the number of writers averaged across all processor count runs for that number of writers. It is assumed in this averaging that the time for the I/O is only a function of the number of writers, not the number of processors that the simulation is run upon. Given that as stated above typically over 95% of time taken for the write is in accessing the disk, this seems reasonable as it implies the communication time is almost negligible. As the disk is a shared resource the numbers are fairly noisy so we also use error bars to indicate the standard deviation in the measured times. The standard deviation is calculated under the same assumptions as the average. However, as the sample size is small, these error bars should not be taken too seriously, they simply give some indication of the variation observed.

It can be seen that the minimum time occurs at 64 writing processors, though given the flatness of the curve and the noise in the data that any number of writers from 16 to 128 is reasonable.

Figures 3 and 4 show the same data for HPCx. Very similar behaviour is observed. Again, the scaling DL_POLY 3 is very good and when using MWRITE_SORTED it is very close to the run with no I/O. Also as before the timings are quite noisy and there is a large range of writers which give good performance. However, for HPCx it appears that, given the data here, the optimal number of writers is 32.

In Figure 5 we show the achieved data transfer rates for the MWRITE_SORTED method. It can be seen that the transfer rate on HECToR are very good, with a peak of around 280 MBytes/s. However, on HPCx they are not so impressive, peaking at just below 40 MBytes/s. This is disappointing as the PWRITE method, which works correctly on HPCx, was shown in reference 7 to peak at 123 MBytes/s. However, it can be seen from the scaling curve that the new method, MWRITE_SORTED, is sufficiently efficient to allow DL_POLY 3 to scale well, so given that it is portable it must be the preferred method.

Finally, one set of benchmark runs were performed on HECToR on a much larger system to check the scalability with system size of MWRITE_SORTED. The system used was simply a doubling of the basic benchmark system in each direction, and so consists of 1,728,000 particles. Again 500 steps were performed, and then the final configuration was written. Only 64 writers were tested as this was found to be the optimal number for the smaller case. Figure 6 shows the measured scaling. Only the MWRITE_SORTED method was tested.

Again, it can be seen that the scaling is excellent, with or without I/O. In fact, for this larger benchmark the observed average transfer rate is even better than that achieved for the smaller case at over 400 MBytes/s, though not too much should be read into this; the data is again very noisy with a lower transfer rate of 238 MBytes/s and upper bound of 659 MBytes/s. In fact, the worst transfer rate was at the lowest number of processors! This may simply be a reflection of their being a larger potential for competition with other users when a smaller fraction of the whole machine is used.

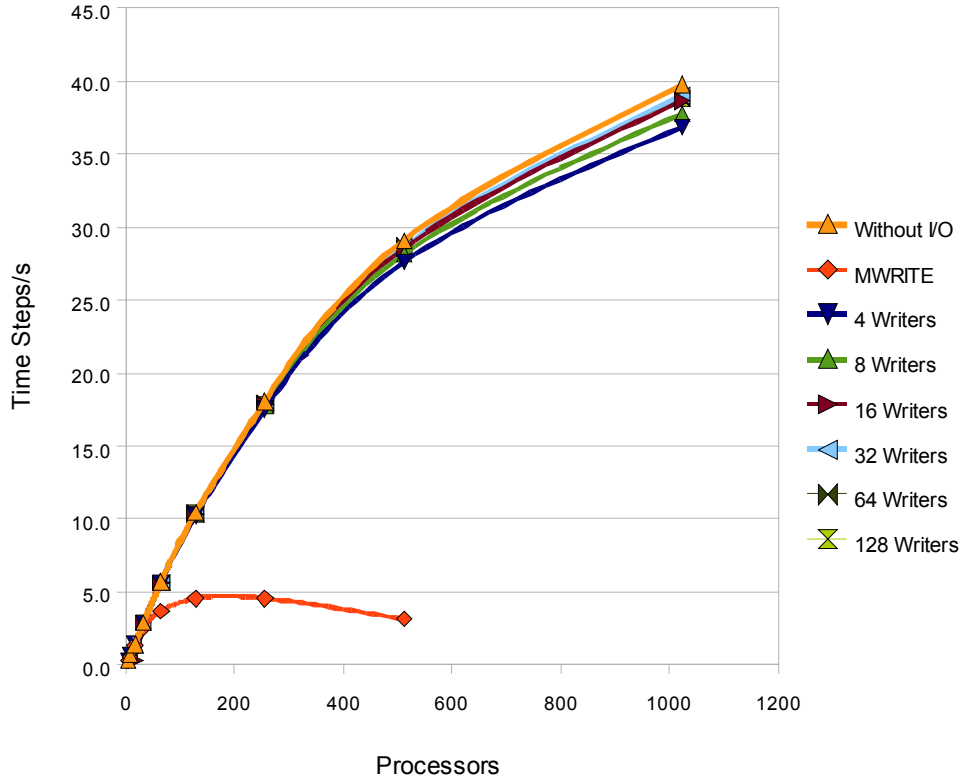

Figure 1: The scaling of DL_POLY_3 on HECToR

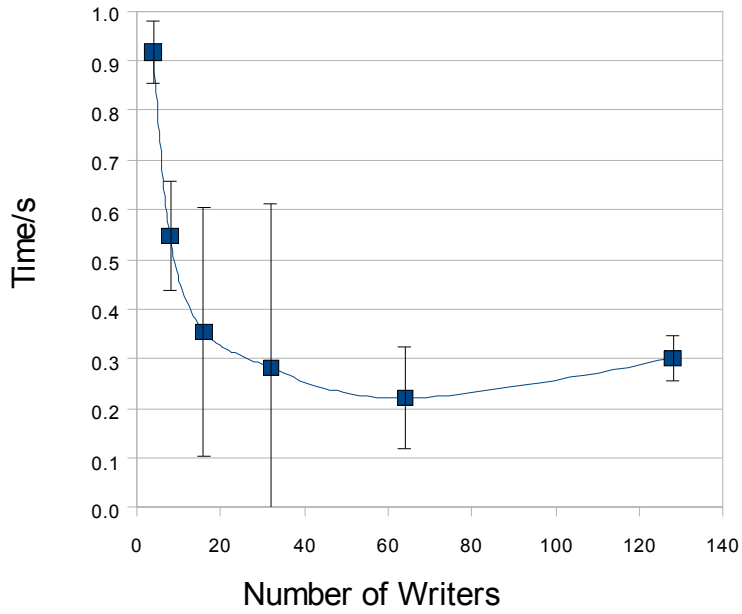

Figure 2: The time for the MWRITE_SORTED method as a function of writers on HECToR

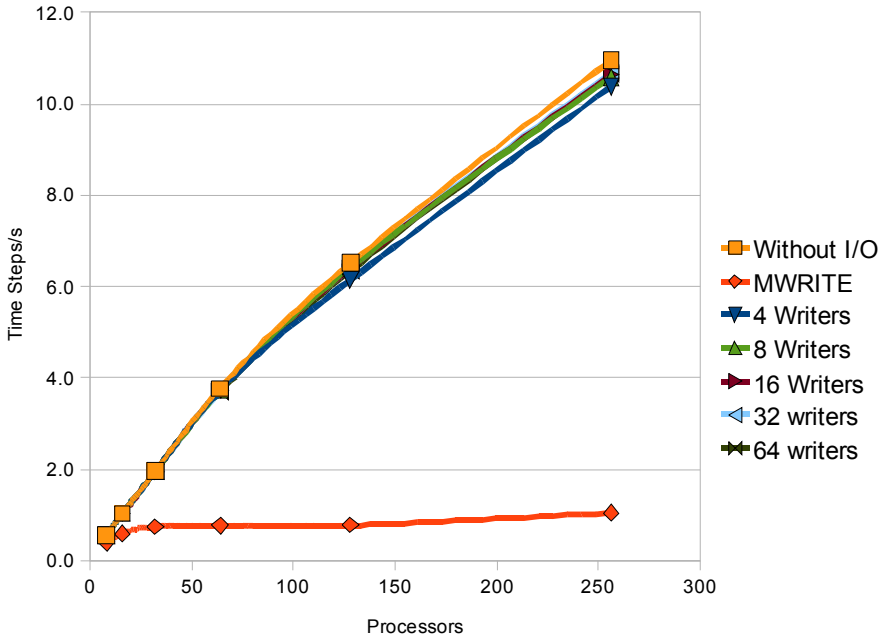

Figure 3: The scaling of DL_POLY_3 on HPCx

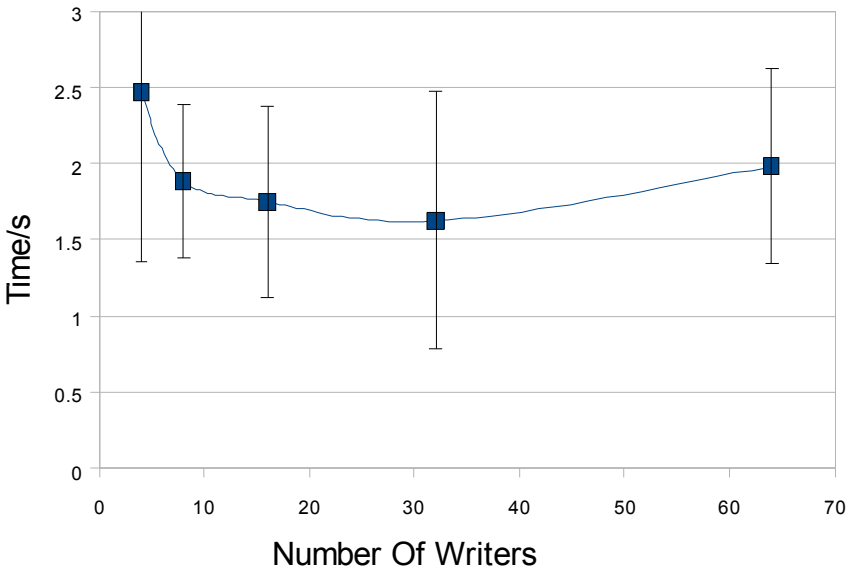

Figure 4: The time for the MWRITE_SORTED method as a function of writers on HPCx

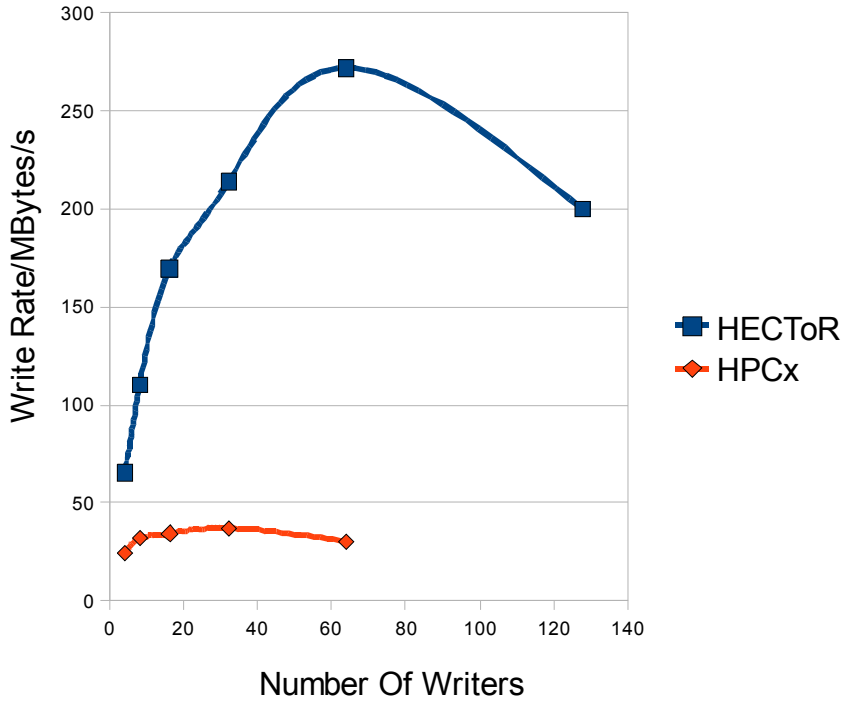

Figure 5: The data transfer rates for the MWRITE_SORTED method

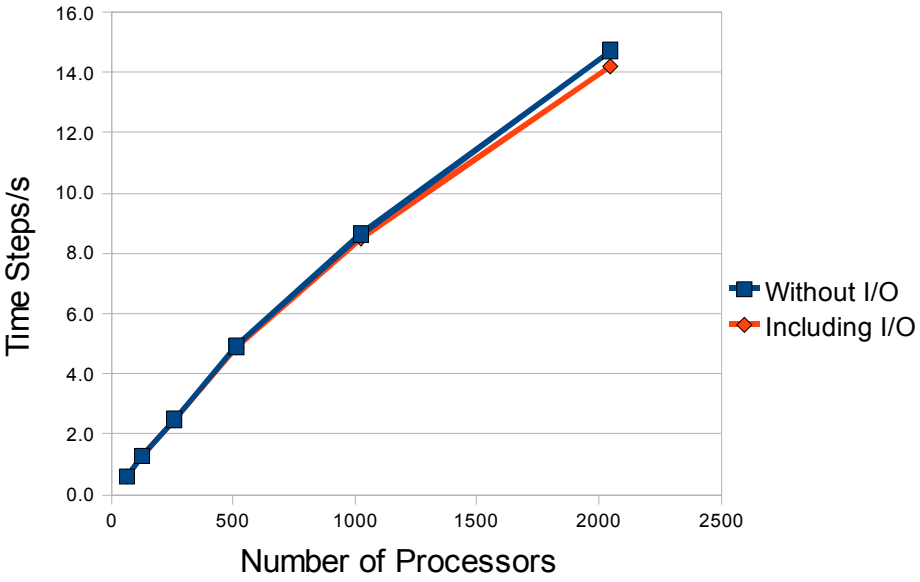

Figure 6: Scaling of the 1,728,000 particle benchmark on HECToR

5 Discussion and Future Work

It is clear that the MWRITE_SORTED method is much the superior in terms of time to solution, so much so that one might claim that at least for the system sizes and machines considered here the I/O bottleneck has been solved.

However, MWRITE_SORTED is not without its drawbacks, as mentioned above in Section 3. It is clear from the results that for such system sizes and machines the compute and communicate overheads are negligible as the time is dominated by the disk access. Nevertheless, the memory overheads will limit the size of system that may be studied, and should be considered further.

In the current prototype of MWRITE_SORTED the memory overhead per atom consists of (counting only those arrays that are functions of system size)

1. 3 `Character( Len = 8 )` variables
2. 3 default `Integer` variables
3. 10 "`Double Precision`" variables

or, for a typical implementation, 116 bytes. Thus, if one were to allow a buffer of, say, 64 MBytes on each of the I/O processors specifically for the writing of the data around 570,000 particles can be stored by each I/O processor. As a result, if there are 64 I/O processors the largest system that could be studied with a 64 MByte buffer per processor is approximately 36,000,000 particles.

While this would be a large MD run today, it is far from inconceivable, and indeed much bigger runs have been already been performed[10]. As machines in the near future will be able to perform such calculations fairly straightforwardly the memory overhead is potentially significant and needs careful consideration.

Two obvious methods for performing bigger runs are by using either or both of a bigger buffer or more I/O processors. These both have obvious drawbacks. A bigger buffer for the I/O means less memory for the main calculation, one of the main things we are trying to avoid. More I/O processors do not have this problem, but may reduce the performance due to contention at the disk. However, it is not necessary to hold all the particles that will be outputted by a given I/O processor all at one time. It is straightforward to identify the first 500,000 particles that will be written by I/O processor 0, and the first 500,000 that will be written by processor 1 etc. Thus, it is a relatively small complication to batch the output such that only a 64 MByte, or any other reasonable size, buffer is needed. As we have demonstrated good performance for processors writing many less particles than this the method should scale in terms of memory to much larger systems.

The immediate future work consists of two parts. The first is to develop a production version of the MWRITE_SORTED algorithm from the current prototype. This is simply an implementation of the ideas in the above paragraphs. It should allow the user to specify both the number of I/O processors and the maximum buffer size in his/her input, and have reasonable defaults for when it is not specified.

The second is to optimise the reading operations required by DL_POLY_3. Though this report has concentrated almost exclusively on writing the initial configuration must be read from disk, and it was noticeable when running the short tests above that this reading was now the bottleneck for this kind of runs. Essentially, the reverse of the hierarchical MWRITE_SORTED strategy can be employed:

- A subset of the processors read the batch of the data in.
- For this batch rearrange the data such that each I/O processor holds all the data for a given set of domains, i.e. no domain is split across I/O processors
- Send the particles in each domain to the processor responsible for it in one long message
- Move onto the next batch until finished

In the longer term the limitations of the method with system size should be examined. It is clear from Figures 2 and 4 that while MWRITE_SORTED has some scalability, it is limited. Therefore, for larger systems sizes increasing the number of processors will speed up the MD while the I/O will not improve. In fact, the beginnings of this can be clearly seen in Figures 1, 3 and 6; the lines with I/O only start to diverge from that with no I/O at the highest processor counts. From this study it is not clear where the limitations in the I/O scaling derive, it could be either in the hardware or the software. If the former, it will be hard to do more. However, if it is in the software it may be possible to improve the method further, one obvious possibility being to move to portable unformatted files such as NetCDF as this will reduce the number of bytes that need be written compared to the current method.

However, it is clear that for the size of systems studied today on both HPCx and HECToR the MWRITE_SORTED method described here solves the I/O bottleneck when performing classical Molecular Dynamics runs on large MPI task counts.

Acknowledgements

As noted above we would very much like to thank Lucian Anton of NAG Ltd for providing the initial implementation of the MWRITE method.

References:

[1] http://www.ccp5.ac.uk/DL_POLY/
[2] I.T. Todorov and W. Smith, 2004, *Phil Trans R Soc Lond*, A **362**, 1835
[3] M.R.S. Pinches, D. Tildesley and W. Smith, 1991, *Mol Simulation*, **6**, 51
[4] I.T. Todorov, W. Smith, K. Trachenko and M.T. Dove, 2006, *J. Mater. Chem*., **16**, 1611
[5] I.J. Bush, I.T. Todorov and W. Smith, 2006, *Comp. Phys. Commun.*, **175**, 323
[6] http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0707.pdf
[7] I.T. Todorov, I.J. Bush, A.R. Porter, *Proceedings of Performance Scientific Computing (International Networking for Young Scientists)* (February 2008, Lithuania), R. Čiegis, D. Henty, B. Kågström, J. Žilinskas (Eds.)(2009) Parallel Scientific Computing and Optimization. Springer Optimization and Its Applications, ISSN 1931-6828, Vol. 27, Springer, ISBN 978-0-387-09706-0, doi:10.1007/978-0-387-09707-7
[8] http://www.hpcx.ac.uk/
[9] http://www.hector.ac.uk/
[10] K. Kadau, T.C. Germann and P.S. Lomdahl, 2006, *Int. J. Mod. Phys.*, C **17**, 1755