Parallel I/O in the hmc_tm code

Albert Deuzeman and Siebren Reker

June 23, 2009

1 Introduction

Over the past year, the most common production lattice volumes used by the ETM collaboration have grown, and the proposed [1] lattice sizes for the near future show that this trend will not stop. As a consequence, the size of associated files such as gauge configurations or propagators will also grow drastically (see table 1).

Lattice volumes ($T = 2L$)	24	32	48	64	80
Configuration size (Gb)	0.382	1.21	6.12	19.3	47.2
Babel serial write time (s)	8.8(2)	27.8(2)	141.6(7)	450(2)	1105(7)

Table 1: Comparison of file sizes and serial writing times of the hmc_tm code on Babel.

Working with these files puts increasingly large demands on the computational prowess of the computer systems and the data handling capacities of storage facilities. Currently, the typical production environment for ETM ensembles is a Blue Gene/P (in IDRIS, Jülich and Groningen), with the MareNostrum system being used as well. Storage systems run GPFS in Jülich, IDRIS and Mare Nostrum, NFS in Groningen, and hardware configurations vary. In order to work with these ever larger lattices, ever larger numbers of compute nodes are needed. Because the computational part of the hmc_tm code used in current production runs scales well with the number of nodes used[1] and the newer machines are very big (Jülich will have 77 racks of BG/P), these bigger runs are actually feasible from a computational point of view. However, the I/O part has up to now always been serial. This is primarily because historically more advanced I/O was not supported on all machines used for production (BlueGene/L machines do not support parallel I/O for example). Since the time spent doing I/O is proportional to the file size and does not scale with the number of compute nodes, it becomes an ever bigger burden as lattice sizes increase. It was our goal to improve on this situation.

[1] Lattice QCD codes are currently rather unique in this respect, there are very few applications where the codes scale so well with the number of nodes, and this is one of the reasons why the lattice QCD community is such a big user of parallel computer power relative to other communities.

2 I/O in the hmc_tm code

The I/O section of the hmc_tm code is responsible for the following functionality:

1. properly ordering the data for writeout (it is usually stored distributed over all compute nodes, and not necessarily consecutive in memory)
2. little-endian/big-endian conversion of the data (storage is always big endian)
3. computing a CRC-32 checksum on the data
4. writing descriptive LIME-headers (format used for storing on ILDG)
5. writing all the data to a single file (actual writing of data)

In the original I/O routines, step 1 and 2 are done per node, then data is sent to node 0, which takes care of 3-5. While this single node is receiving all data site by site from the node which holds the information, all other nodes are waiting for it to finish. This idle time grows longer and longer times as lattice sizes increase. Some example writing times for configurations are listed in table 1 for illustration[2].

To improve the writing times, our new I/O routines parallelize or restructure each step listed above as follows:

1. Order the data properly locally (per node), rewriting of the loop over the whole lattice into a loop over only the local lattice
2. Was already properly parallelized
3. Locally compute a checksum, and the combine all local checksums into one global checksum using a fast parallel call (this can be done parallelized because of the type of checksum we use). This code existed, but was not used/tested.
4. Writing descriptive LIME-headers is difficult, since it is a decidedly non-parallel step, it is now handled by LEMON (see below)
5. Have one parallel LEMON call (i.e. an MPI_FILE_WRITE_AT_ALL call) write everything directly in its proper place in the file (this is where we get the big speed gain)

2.1 LEMON

LEMON is a parallel alternative to LIME, written by Albert specifically for the parallelized I/O in the hmc_tm code. For us, the main gain over LIME is the careful handling of shared and individual file pointers to take care of many different nodes writing in different locations at the same time, or all trying to write a header at the same location. More information on this will follow in a separate technical note.

[2]These times also include the steps 1-3 and were run on Babel, the BGP in IDRIS. Timings depend on various environmental factors beyond the control of the testing code, but these times are representative and reproducable.

3 Performance

To test the performance of parallel I/O on our production machines, we first wrote a small test package to see what performance gains we could expect. This package does not need to do any of the steps listed in the previous section, except for the parallel writing of data. The results for a file with the size of a $48^3 \times 96$ configuration on 4 different partition sizes on Babel are given in figure 3 at the end of this note. As always with these tests, there are external factors that are beyond our control, and they can heavily influence a measurement. Our goal here was to show that parallel I/O was indeed orders of magnitude faster then serial I/O, and that it did scale with the number of nodes, which it more or less does in this test. The full results from the testing code (figure 4, also at the end of this note) more clearly show a bit of the variability and scaling properties for various file sizes.

Many things in our testing code do not translate directly to the hmc_tm code. For example, the sustained writing speed we find in the serial tests is around 130 Mb/s. The sustained writing speed for the hmc_tm code is much lower, since for example checksumming and endianness conversion also have to be performed. The sustained writing speeds that we found for the original serial hmc_tm I/O code are shown in figure 1. For larger volumes, the speeds can be seen to decrease a bit, probably due to larger local volumes. Overall the speed is very consistent at a bit less than 43 Mb/s.

Replacing serial with parallel I/O in our new implementation, the picture changes drastically, as can be seen by comparing figure 2 with figure 1 (notice the different scale on the y-axis!). The writing speeds now scale with the number of I/O nodes, and in general are orders of magnitude faster, up to almost a 100 times faster for the biggest lattices on the large partitions: an $80^3 \times 160$ lattice is written on a 4 rack partition in 11.9 seconds instead of the roughly 1100 seconds it takes in the original serial setup.

3.1 Example: a current production run

This implies that for example in the $48^3 \times 96$ production run being run on Babel right now, the computation time for a trajectory can be reduced by about 2 minutes, which equates to about 4% for the current setup. This is obviously dependent on the running parameters and the partitions in use. 4% is a very conservative estimate and larger improvements over serial I/O should be expected for the planned (bigger) future runs.

4 Example: inversions

The job type where the main speedup is to be achieved are obviously the inversions. Inversions are usually run with smaller partitions, precisely because I/O being serial gives such a big performance hit and it is more benificial to run multiple inversions in parallel on smaller partitions. For example, for the $48^3 \times 96$ run, reading a source takes about 33 seconds, inversions done on a rack on Babel take roughly 110 seconds, after which the serial writeout takes another 73 seconds. With parallel I/O, this reading should take about 1 second, and the writing should take about 2 seconds, so the total time should go

from 216 seconds to 113 seconds for an almost 50% time reduction for the total inversion job. Code for parallel I/O for propagators is in progress, it is close to fullimplementation.

References

[1] http://www-zeuthen.desy.de/∼kjansen/etmc/internal/ibm.pdf

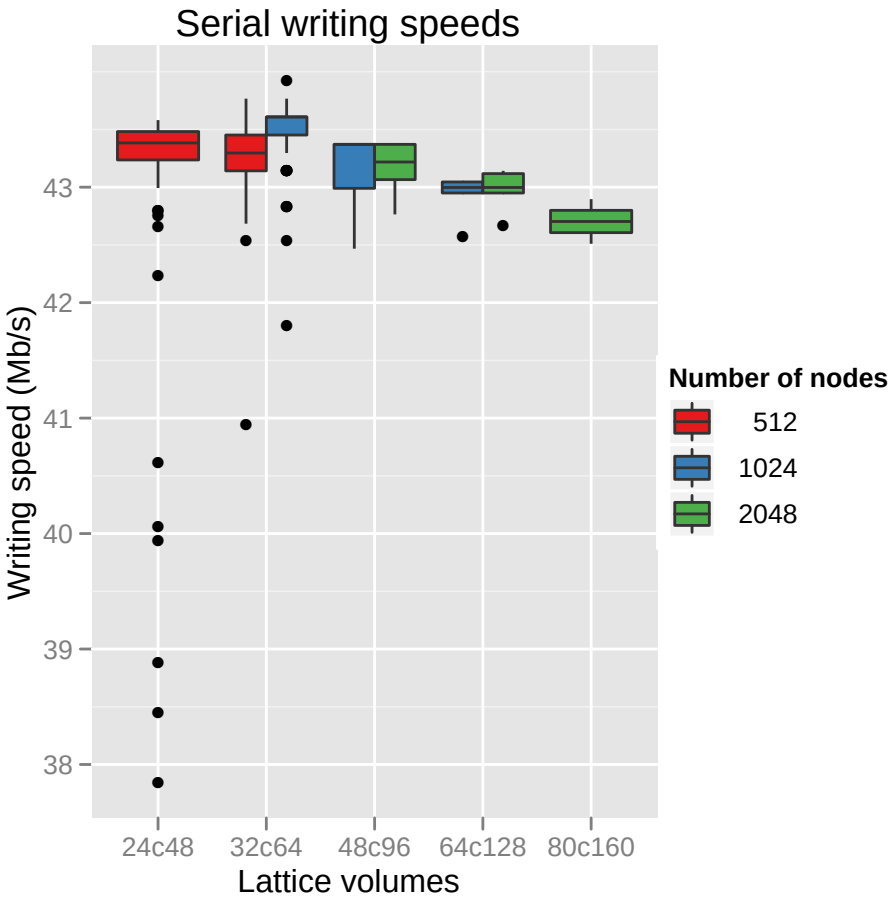

Figure 1: Serial writing speeds on Babel in the original hmc_tm I/O routines. The speeds listed here are not pure writing speed, but necessarily also include checksumming and communication. Because of the setup used, it is impossible to separately measure these steps. On the y-axis, the writing speed is given, the boxes show the median and first quartile distribution of the measurements, the black dots indicate outliers. Lattice volumes are grouped on the x-axis, with different colours being used to indicate different partition sizes. The reason for the outliers is filesystem usage by other processes. The fact that more outliers show in the 24c48 job is that that job has the most statistics.

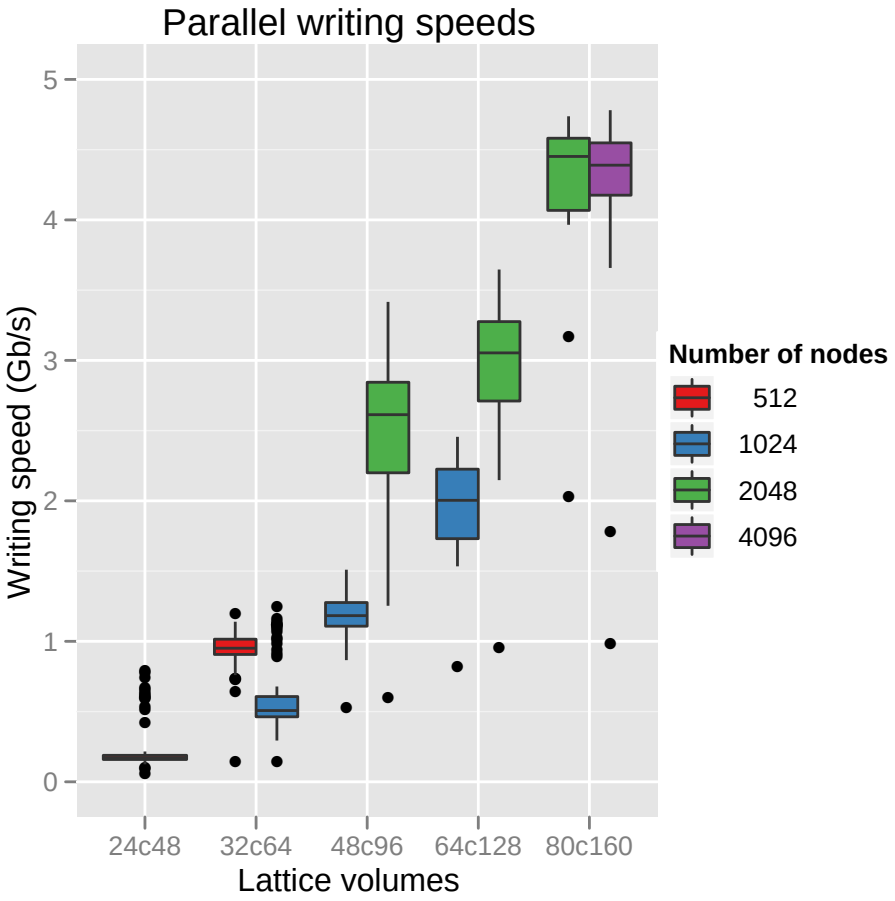

Figure 2: Parallel writing speeds on Babel in the new hmc_tm I/O routines. The speeds listed here are not pure writing speed, but also include checksumming and endianness conversion. These steps can be removed in the timing information to get more pure timing information. However to retain a fair comparison with the original routines, they are included in our timings. On the y-axis, the writing speed is given, (notice that for the figure on serial writing speed the axis is in Mb/s, while here it is in Gb/s). The boxes show the median and first quartile distribution of the measurements, the black dots indicate outliers. Lattice volumes are grouped on the x-axis, with different colours being used to indicate different partition sizes. Clearly visible is the increase of the writing speed for larger partitions, from red to blue to green to purple.

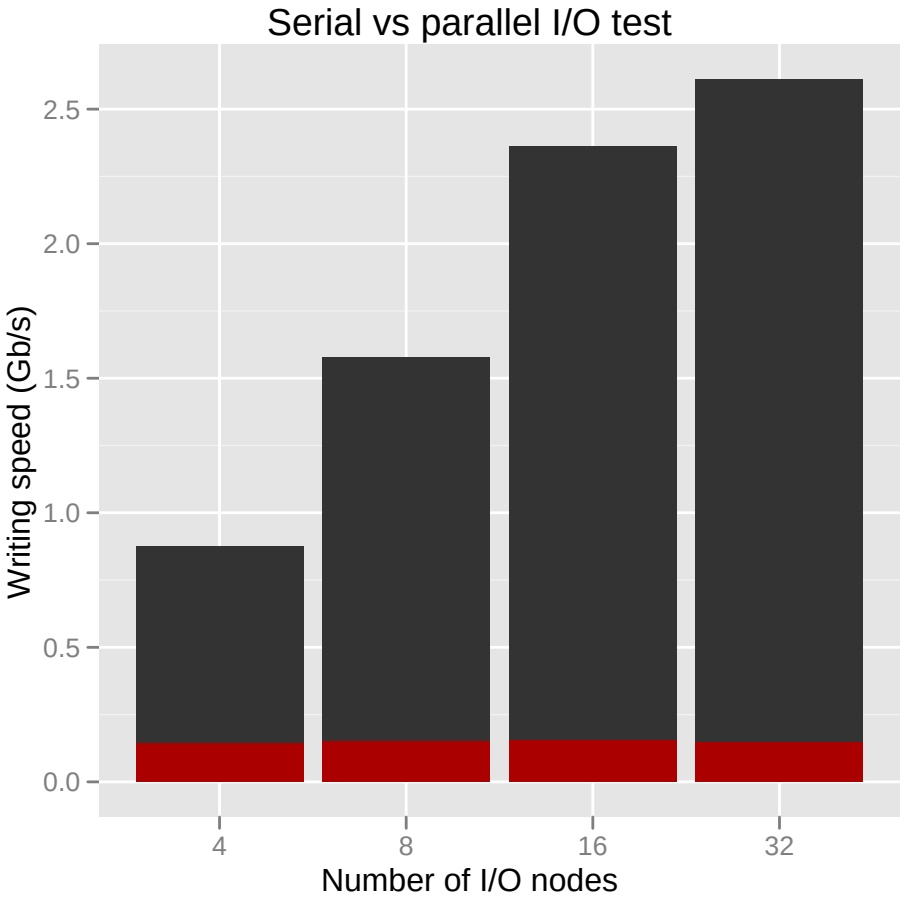

Figure 3: Results of a serial versus parallel test on Babel. A file with the size of a $48^3 \times 96$ lattice was written out on a 256 node partition (4 I/O nodes), a 512, 1024 and 2048 node partition. The writing speed in serial mode (red) and parallel mode (black) is shown. Optimistically and theoretically we would expect to see an exponential increase, but apparently we are hitting bottlenecks due to disk usage, file systems, hardware etc. The overall message remains clear however: parallel I/O is much faster.

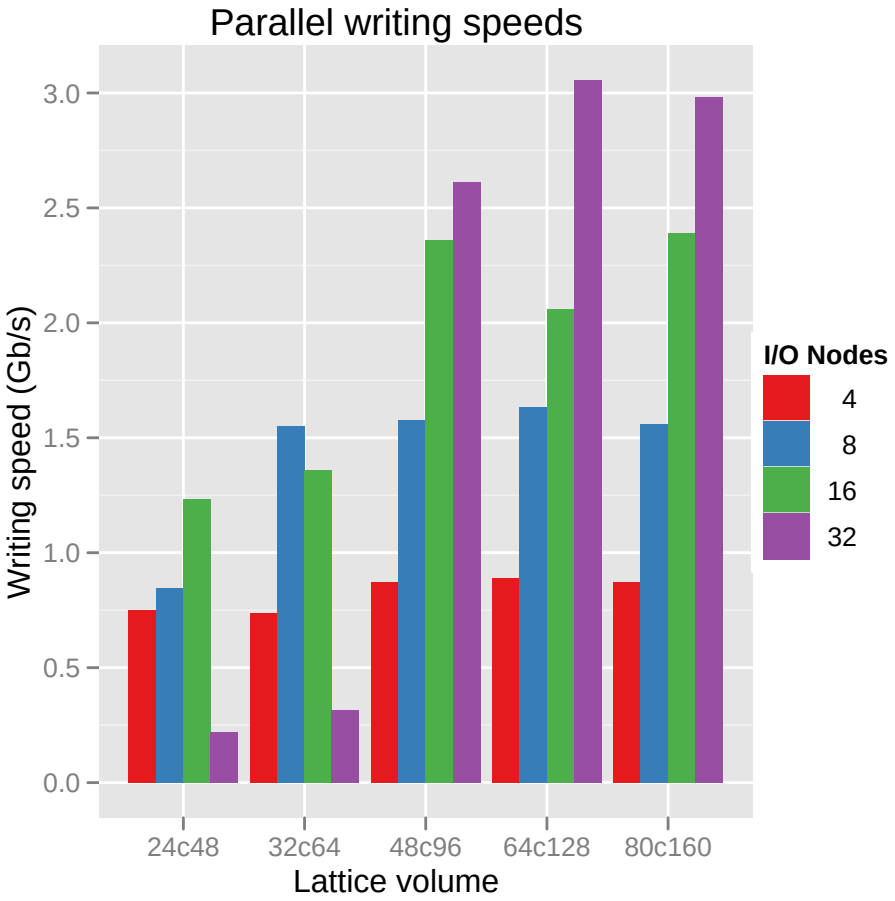

Figure 4: The complete results of the parallel test that we ran on Babel. Now only showing the behaviour of parallel I/O for different lattice sizes and volumes. We did not investigate the strange measurements for the small lattices on the large partition (32 I/O nodes), but one can imagine buffering not working as efficiently for such small local lattice volumes, or general file system activity due to other users. These files were written in about a second, and these bars are the result of only one measurement.