Adding Parallel I/O to PARA-BMU

Nick Johnson, Iain Bethune,
EPCC, The University of Edinburgh

October 1, 2012

Abstract

VOX-FE[1] is a voxel-based bone modelling suite. The solver part of the suite - PARA-BMU - currently uses only serial I/O routines which lead to poor scalability. We enhance the code by adding parallel I/O routines based on the netCDF[2] and HDF5[3] libraries. We demonstrate that this gives a reduction in file sizes of up to 190x and a reduction in wall-clock time of 7x at 512 cores. This paves the way for PARA-BMU to be used effectively on HECToR and allows users to perform research with much larger and more complex models than was previously possible.

Contents

1 Introduction

VOX-FE is a voxel-based finite element bone modelling suite developed by Prof. Michael Fagan's Medical & Biological Engineering group at the University of Hull. It is one of the demonstrator applications for the EPSRC-funded "Novel Asynchronous Algorithms and Software for Large Sparse Systems" project, and the core algorithms of VOX-FE are being redeveloped for increased scalability and functionality. The VOX-FE suite comprises two parts; a GUI for manipulating bone structures and visualizing the results of applying strain forces, and an MPI-parallelised Finite Element solver *PARA-BMU* which performs the computation required to solve the Linear Elasticity problem and calculate stresses and strains in the bone. Example applications would include computing the maximum principal strain in a human mandible (jaw bone) undergoing incisor biting, or understanding the stresses in an axially loaded femur.

Initail examination of the code revealed that plain ASCII files were being used for both input and output data. These were read and written in serial by a single process whereas the solver routines operated in parallel. This solution did not scale well to a large number of cores with the result that for systems with a very large number of finite elements, the runtime was dominated by I/O and MPI data exchange rather than by computation. Another side-effect of using plain text files is that file sizes grow extremely large as the problem size increases which increases transfer times to and from HECToR. Prof. Fagan's group intend to routinely study bone models with resolutions of over 100 million elements in the near future, where the input files are expected to be many GB in size.

2 Objectives

In this dCSE project, the primary goal was improve the scalability of the code by parallelising the I/O routines. To that end there were two objectives:

- Reduce file sizes by converting to netCDF-HDF5 formats with a target reduction of between 2 and 20 times.
- Increase disk I/O speed by using parallel netCDF routines with a target speedup of between 3 and 4 times minimum.

Currently, the code exhibits poor strong scaling: the input is performed in serial on the master process and the output is performed by each process but is serialised in a round-robin pattern. The effect on the overall performance of the code compared to the performance of the solver itself can be observed in Fig 1. With respect the purely serial case (1 MPI process), the maximum speedup is limited to $\sim$22. The theoretical scalability limit for this code is the number of z-planes present in the problem as the decomposition is one dimensional. Throughout this report, we use the larger of the two test cases supplied by Prof. Fagan's group, the *High res model* which has 884 z-planes comprising $\sim$29 million elements.

For backwards compatibility, it is obviously desirable to be able to re-use previous data files and for that reason, all serial I/O routines are preserved. In

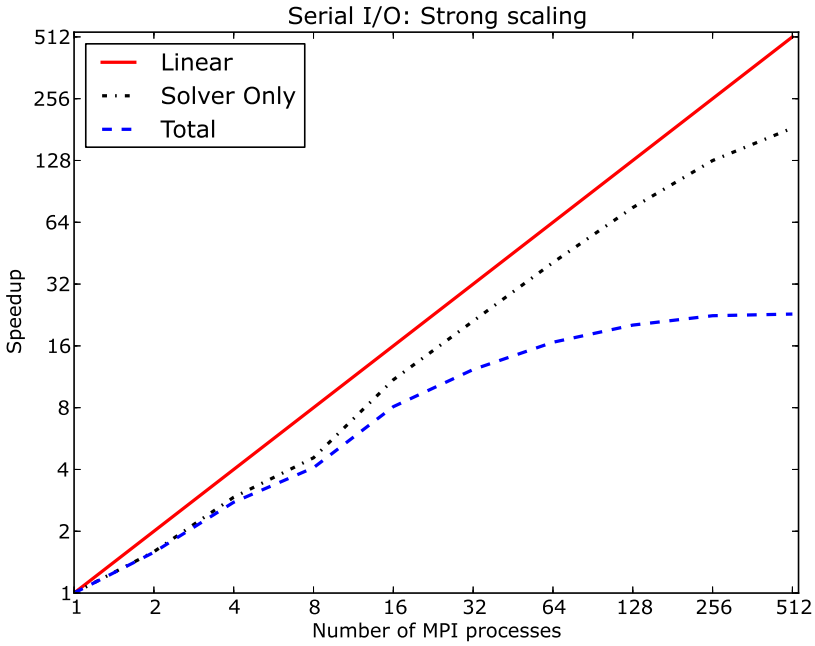

Figure 1: Strong scaling of PARA-BMU using only serial I/O showing ideal case (Linear), solver only (Solver) and complete runtime (Total).

addition, convertor utilities are provided to allow conversion of the old ASCII data files to and from the new format.

3 I/O schemes and optimization

To understand the I/O routines, we briefly review the data storage schema used in PARA-BMU.

Data is stored in a dynamically allocated array of type BYTE, a user defined type in C++ which is 8 bits in length and unsigned. Each array element represents a voxel, and its value the Hounsfield value of that region of the physical system (essentially a density value derived by CT scanning a physical bone sample). The data is organised as a 3D array with dimensions corresponding to cartesian co-ordinates. The solver itself operates on slices of data corresponding to z-planes in the system. The data contains many elements which have a value of 0 indicating that no material is present (i.e. empty space).

We now proceed to describe the current and new I/O routines in PARA-BMU.

3.1 Current Input

The existing input method consists of reading all data from an ASCII formatted text file into a buffer on the master process and then sending z-planes of data

to each worker process via MPI.

Each line of the input file, save for header information is one record and corresponds to a single voxel. It contains an element number, a type (the data value) and the aforementioned x, y & z co-ordinates of the voxel in the system. The data undergoes a mapping operation upon input which changes it from the values read from file to an internal representation as one of a defined set of materials.

Once read, a stencil operation is applied to the data which generates a distribution of certain-valued elements per z-plane. A load-balancing algorithm is used which generates, from the distribution, the number z-planes of data to be distributed to each worker process via MPI.

3.2 Current Output

The existing output routine uses a round-robin serialization to allow all processes to write their output values to a single text file.

Beginning with the master process, the file is opened and header information written to the file. The master process writes its data then signals process $P + 1$ with an MPI call to write its own data. Once this process has written its data, it signals the next process and the cycle continues until all processes have written their data. In addition to signalling the next process to write data, the renumbered node offset (a long integer) is also sent between processes. This value is necessary for the correct numbering of nodes in the output file. Each worker process must wait, idle, for the signal to write data and once finished must return to idle until all processes have completed their individual writes.

3.3 New Input

We have added a new input routine which can be selected in the script file with the arguements remaining identical to the serial case (see Section 5.1 for details). This routine can be described by Algorithm 1.

Each process (including the master process) now reads data. Any partial z-planes read are sent to process $P + 1$ along with a copy of the lowest z-plane which is necessary for the solver. The load balancing algorithm which is present in the serial case is not used as this would require complex re-distribution of the data once read.

3.4 New Output

We have added a new output routine which can be selected in the script file with the single arguement of the file to write the data to, as in the serial case. See Section 5.1 for details.
This routine is conceptually similar to the serial case except that all writes are now done in parallel. It is described by Algorithm 2.
In contrast to the serial output case, here, the renumbering offset is calculated before any data is written. Once it is calculated, an `MPI_Barrier` call is issued

Algorithm 1 New parallel input algorithm.

Synchronize MPI processes.
Open input file by call to netcdf routine `nc_open_par`.
Read N/P records into a private buffer where N is the number of records and P the number of processors.
Close file.
if Topmost layer in records buffer is incomplete **then**
    Send top-most partial layer to process P+1
end if
if Lowest layer in records buffer is incomplete **then**
    Receive incomplete layer from process P-1
end if
Translate records buffer into local data buffer respresenting voxel data values with transformations as per serial case.
Free records buffer memory.

Algorithm 2 New parallel output algorithm.

Compute renumbering offset.
Synchronize MPI processes.
Open output file by call to netcdf routine `nc_open_par`.
Write local data to file using netcdf routine `nc_put_vara`.
Close file.

to ensure synchronicity between processes prior to calling `nc_put_vara` which instructs processes to write (in parallel) to the output file.

Unlike the input files, parallel output files cannot be compressed at write time using netCDF, which is a limitation of the library. It is suggested to user that they make use of the netCDF operators to perform compression. See Section 5.3 for usage details.

4 Results

Measuring the I/O speed for comparison was found to be difficult on HECToR. CrayPAT, Cray's profiling tool did not handle well the case of netCDF/HDF5 API calls. Thus we resorted to using calls to the `MPI_Wtime` routine to measure the elapsed time for both serial and parallel I/O.

We show results for the case of 128 processes on 4 nodes, fully packed at 32 processes per node. Where times for each parallel process were not equal, we show the worst case result recorded. For the serial case, we show the elapsed time on the master process. We time only calls to read or write functions, ie `fprintf/fscanf` or `nc_put_vara/nc_get_vara` and not associated stores to memory, calls to transformation routines or counter increments, which are identical between versions.
As can be seen from both Tables 1 & 2, the relative I/O speed for the parallel case is much worse than that of the serial case, however, the absolute time is much lower and represents a significant speedup.

	Input		
	File size	Worst-case time	Approximate I/O speed
Serial	567MB	30s	18.9 MB/s
Parallel	2.9MB	0.4s	0.05MB/s

Table 1: Input file sizes and times for 128 processes, fully packed at 32 process/node.

	Output		
	File size	Worst-case time	Approximate I/O speed
Serial	2600MB	118s	22MB/s
Parallel	994MB	7s	1MB/s

Table 2: Output file sizes and times for 128 processes, fully packed at 32 process/node.

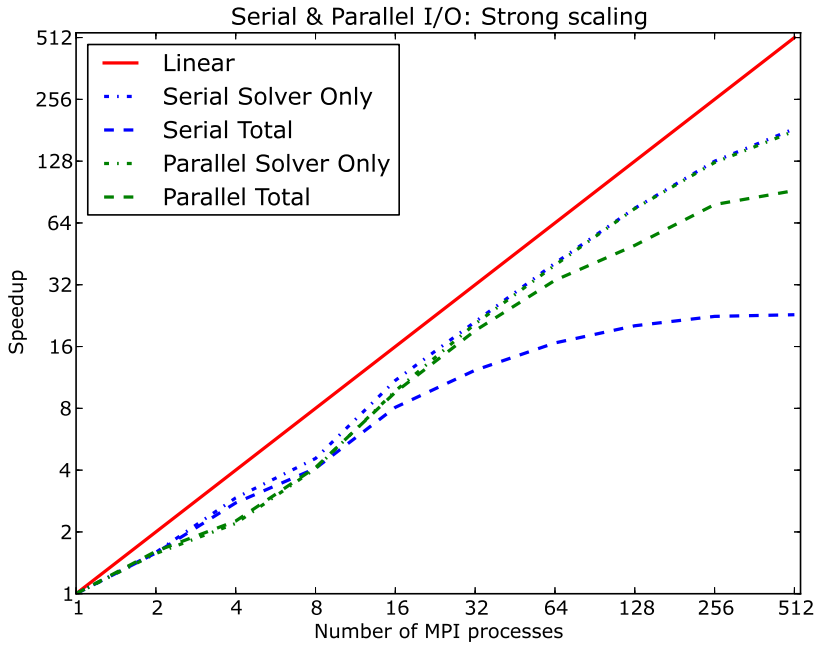

Figure 2: Strong scaling of PARA-BMU using serial and parallel I/O showing ideal case (Linear), solver only (Solver) and complete runtime (Total).

From Figure 2 we can see that the solver scales almost identically for the parallel and serial case. The total wall clock time for calculations using the new parallel I/O routines scales much better than those using the serial I/O routines although it is still not as close to linear as the solver alone, indicating that there may still be gains to be found by further optimising the parallel I/O. The speedup over a single core for serial I/O is $\sim$22 and for parallel I/O is $\sim$90. The speedup of the solver alone is $\sim$180.

5 Notes for users

5.1 Commands added to the scripting language

The new parallel loading routine can be invoked in a similar manner to the current serial routine.

```
LOAD_MCTSCANPAR <X> <Y> <Z> <filename>
```

Similarly for the new parallel output routine.

```
PARPRINT_X <filename>
```

These commands are interchangeable with their serial counterparts and one may elect to use only one of the pair, i.e. one may choose parallel input with serial output or vice-versa, as needed.

5.2 How to use the convertors

Two conversion programs are provided to convert from ASCII to and from the netCDF file formats.
The first program, `txt2hdf5` will convert an ASCII file to a netcdf format file. The type of file, either element data (PARA-BMU input) or displacement data (PARA-BMU output) is automatically detected. Invocation is thus:

```
./txt2hdf5 <txt_filename> <hdf5_filename>
```

This is a serial code which must be compiled for, and run in, the serial environment on HECToR.

Similarly, the second program `hdf52txt` will convert a netcdf format file to an ASCII file. Again, the type of file, either element data or displacement data is automatically detected. Invocation is thus:

```
./hdf52txt <hdf5_filename> <txt_filename>
```

Again, this is a serial code which must be compiled for, and run in, the serial environment on HECToR.

5.3 netCDF operators & compression

In order to compress the files generated by either the convertors or PARA-BMU itself, one should make use of the netCDF operators, NCO, and in particular the `nccopy` utility. A compression level of 9 (maximum) is recommended although it is known that this may not always give the best compression and users may wish to experiment. Invocation is:

```
./nccopy -d9 <input_filename> <output_filename>
```

6 Conclusion

We have shown that by using the netCDF libraries we were able to decrease I/O time and reduce the file sizes of both input and output files relative to the original implementation using ASCII case. We have merged these additions to the main trunk version of the code of CCPForge such that Prof Fagan's group may make immediate use of them. The impact of this work is that it is now feasible to use HECToR as a platform for running PARA-BMU rather than small-scale local clusters; this paves the way for bone modelling at unprecedented scale and accuracy.

7 Acknowledgements

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`

References

[1] Medical & Biological Engineering
University of Hull
VOX-FE: Voxel Based Finite Element Analysis
`http://www2.hull.ac.uk/science/medical__biological_eng/research/vox-fe.aspx`

[2] netCDF
`http://www.unidata.ucar.edu/software/netcdf/`

[3] HDF5
`http://www.hdfgroup.org/HDF5/`