Ludwig on ARCHER2: task-based execution ARCHER2-eCSE01-26

Kevin Stratford, EPCC

Prof. Davide Marenduzzo

July 2023

AN ARCHER2 ECSE PROJECT

Abstract. This report details a series of software developments in *Ludwig*, a code for complex fluids. These include improvements to general performance, message passing, and output to file. The common theme is the use of OpenMP to provide additional parallel where possible. Results of standard benchmarks using up to 4096 nodes are presented. **Keywords:** high performance computing, hybrid parallelisation, fluid dynamics.

Contents

Introduction

The aim of this proposal was to recast elements of *Ludwig* to follow a task-based approach. The rationale for this was to move away from fixed, static, domain-decomposition to a more flexible approach for not only larger systems, but for a more complex combination of physics.

It should be said at the outset that this approach turned to to have, at best, mixed results. The underlying resaon for this is that the task model available on ARCHER2 (OpenMP threads) does not seem to offer any advantage over a more standard loop-based work decomposition. In many cases, performanace is sufficiently degraded by using tasks that any implementation was of little practical worth. This negative result is disappointing in that the task model, in principle, has some attractive advantages. However, it may require a higher-level approach (such as a graph-based API) than the compiler/loop level addressed here.

The redeeming feature of the work was to concentrate effort on threads per se. While the task model was not found to be effective, there is certainly a case for using threads to give an efficient hybrid implementation. This has benefits in reducing overheads associated with large numbers of MPI processes. and in many ways is a natural picture for the multi-core architecture.

Some overall remarks

As a matter of policy, it has been decided that *Ludwig* will use `MPI_THREAD_FUNNELLED` as the model for the message passing interface (MPI) library. In particular, attempts to use `MPI_THREAD_MULTIPLE` have observed relatively poorer performance for standard problems, ascribed to the requirement that MPI make conservative assumptions about thread synchronisation to ensure correctness. It is therefore preferable that the application imposees thread synchronisation, as the application is best placed to make statements about the necessity of such synchronisation.

The report is split into four sections following the description of work set out in the original proposal. This is perhaps not the most logical order, and leads to a number of 'forward references' in the text. In particular, the reader should note that the updates to the underlying MPI halo mechanism were competed before the work on I/O. The work on I/O thus features the version of the code with all the relevant updates from the work undertaken under this proposal.

1 I/O tasks and usability

The existing parallel I/O facility for lattice quantities in *Ludwig* was based on a bespoke ANSI mechanism which was not decomposition independent, but was flexible in that it allowed a number of separate files to be written based on the Cartesian decomposition of the lattice. This required post-processing to reconstitute the complete system in a suitable order for visualisation and so forth. (Restarts where also restricted to the same number of processes so that files could be read correctly.)

A new I/O mechanism has been implemented which retains the advantage of the original in that the number of files can be controlled, but uses a standard MPI/IO mechanism to write individual files. Further, relevant data are aggregated to separate storage to provide a memory-order independent I/O mechanism and resulting file, and to admit the possibility of asynchronous

I/O. Additional I/O metadata in JSON format is now also provided to help describe the data for post-processing etc.

1.1 Data structures

There are a number of components of the new I/O mechnaism. The first is an abstract class which defines what functions must be implemented to perform I/O. There are:

```
  int io_impl_read(io_impl_t * io, const char * filename);
  int io_impl_write(io_impl_t * io, const char * filename);
```

stating that an implementation must provide functions to perform read and write operations for a given file. These are collective operations in MPI and are assumed to be synchronous (blocking) calls [1].

Concrete objects are brought into existence via a factory function

```
int io_impl_create(const io_metadata_t * metadata, io_impl_t ** io);
```

The `io_metadata_t` object encapsulates the necessary information required to return a specific implementation. At the time of writing, the only I/O implementation is that using standard MPI-IO functionality to write either ASCII or native binary files. However, it should be simple to add implementations for, e.g., HDF5 format in parallel.

The metadata object is itself composed of a number entities including:

```
typedef struct io_metadata_s {

  cs_t * cs;                   /* Keep a reference to coordinates */
  cs_limits_t limits;          /* Always local size with no halo */
  MPI_Comm parent;             /* Cartesian communicator */
  MPI_Comm comm;               /* Cartesian sub-communicator */

  io_options_t options;        /* User i/o options description */
  io_element_t element;        /* Per site data description */
  io_subfile_t subfile;        /* File(s) description */
} io_metadata_t;
```

The Cartesian co-ordinate system and the associated Cartesian communicator are provided, on which basis a new communicator for collective I/O (`comm`) can be generated. If output to more than one file is requested, then the parent communicator is split in a coarse Cartesian subdivision, and I/O takes place within a separate communictor for each file. This is described by the `io_subfile_t` object as required.

The number of files to use (more exactly, the file decomposition) is available from user input supplied by the `io_options_t` object. The default is a single file.

Any lattice quantity wishing to undertake I/O must be described by an `io_element_t` which will detail the number and type of data elements per lattice site. This allows a concrete I/O implementation to depend only on the aggregated representation.

1.2 Time evolution independent of I/O

Asynchronous I/O is possible via an abstraction at the level of the I/O implementation layer `write_begin` and `write_end` (no asynchronous read is available at present as this is probably less useful).

```
int io_impl_write_begin(io_impl_t * io, const char * filename);
int io_impl_write_end(io_impl_t * io);
```

1.3 Aggregation

The new I/O mechanism is based around the idea of an *aggregator* which provides storage for lattice quantities in a form which can be described to MPI-IO. Data quantities are first aggregated to the standard form before being written or read by an I/O implementation. This separate storage is also the basis for asynchronous I/O. The description of the data to the aggregator includes the number of elements per lattice site, the underlying data type, the lattice size and so on, which form a standard metadata object.

Lattice quantities in the code wishing to implement the new I/O functionality must provide aggregration (and 'dis-aggregation' on reading). For example, the code `field_t` providing scalar, vector and tensor fields implements

```
  int field_io_aggr_pack(field_t * f, io_aggregator_t * aggr);
  int field_io_aggr_unpack(field_t * f, const io_aggregator_t * aggr);
```

These are responsible for copying relevant data to and from the aggregator storage, and may be implemented using threads.

1.4 Output driver

The complete picture for I/O requires a driver. As an example, the `field_t` class provides a driver

```
  int field_io_write(field_t * f, int timestep, io_event_t * event);
```

which controls the aggregation step and the I/O itself, which delegated to the current implementation. The `io_event_t` object provides state to record relevant performance information, and state that may be related to asynchronous operations which must be completed by a separate interface call.

The time step is required to construct the appropriate file name.

1.5 Test for performance

The new MPI-IO implementation was tested on ARCHER2 in the context of synchronous output to a single shared file. This is probably the most challenging configuration to achieve, as there is no additional parallelism coming from multiple files.

The test is a standard benchmark involving liquid crystals for which there are 10 episodes of output for the liquid crystal order parameter (5 `doubles` per lattice site), and one episode of

	System size									
	128^3		256^3		512^3		1024^3		2048^3	
No. MPI process	`ofi`	`ucx`	`ofi`	`ucx`	`ofi`	`ucx`	`ofi`	`ucx`	`ofi`	`ucx`
2	0.3	0.3	0.7	0.5	—	—	—	—	—	—
4	0.2	0.2	1.3	1.2	—	—	—	—	—	—
8	0.2	0.2	2.6	2.1	—	—	—	—	—	—
16	0.3	0.3	2.9	2.8	5.3	4.8	—	—	—	—
32	0.5	0.5	1.1	1.2	10.0	9.4	—	—	—	—
64	0.4	0.5	1.5	1.6	16.4	12.9	—	—	—	—
128	0.3	0.4	1.2	1.4	18.1	21.7	28.9	23.3	—	—
256	0.3	0.4	1.2	1.4	3.2	3.5	31.6	29.3	—	—
512	0.2	0.4	1.1	1.4	2.4	2.6	30.6	26.9	—	—
1024	0.2	0.3	1.0	1.3	1.8	2.3	34.3	30.2	13.1	14.6
2048	0.2	0.2	0.8	1.1	1.8	2.3	3.6	6.6	17.7	16.7
4096	0.1	0.2	0.6	0.8	1.6	2.1	3.0	3.5	13.7	15.2
8192	0.1	0.1	0.4	0.5	1.3	1.7	2.7	17.3	14.9	fail

Table 1: Estimated aggregate output bandwidth (GB/s) for a standard benchmark run at a number of of different system sizes, and run with either `ofi` or `ucx` network. All measures use a directory with 12 stripes of the default size with a maximum of 96 MPI/IO aggregators (8 aggregators per stripe). There is no appreciable difference between the two network implementations (although `ucx` was observed to fail for some of the largest benchmarks). All runs use 64 threads/cores per MPI process. For problem sizes of 256^3–1024^3, there appears to be a clear limit to the number of processes that can support effective I/O. This is discussed further in the text.

output for the model state and the end of the run. The full state includes the lattice Boltzmann distributions (19 `double`s per site), the velocity field (3 `double`s per site) and the fluid density (1 `double` per site). The ten episodes of output for the liquid crystal order parameter represent a realistic frequency for diagnostic output which might be required for a production run. The overhead for the full state would probably be proportionally less in a longer production run, as it is typically required only once at the end of the run for checkpoint/restart purposes.

A range of system sizes is used: 128^3, 256^3, 512^3, 1024^3, and 2028^3. While the largest system size is not currently representative of production runs, the other system sizes are directly relevant to conceivable problems. For all system sizes, runs are made with 2 MPI processes per node (64 threads per socket) to the full extent of the machine (4096 nodes or 8192 MPI processes). This is clearly beyond the reasonable scaling regime for the smaller problems at the high-process count, although results are included for completeness. The larger problems are limited by available memory at the lower process counts [2].

Runs were repeated in a number of different I/O configurations, including the use of both OFI and UCX networks, and different numbers of MPI-IO aggregators per stripe. All runs used the maximum number of Lustre stripes available (12) and the default stripe size (1MB). Results obtained for 1, 2, 4, and 8 MPI-IO aggregators per stripe in all cases [3]. The results for 8 aggregators per stripe (96 aggregator processes in total) are shown in Table 1. The figure shown (GB/s) is the net reported figure based on 13 episodes of file output. It is not weighted by the size of the file. As considerable variability in the time for individual output events is observed, these figures should be treated with some caution; however, figures for OFI and UCX are broadly consistent. All the runs at a given number of processes were performed in the same batch job for a given number of MPI-IO aggregators. An additional run in each case was also performed without I/O to assess whether I/O was limiting overall performance.

A number of observations can be made about the results in Table 1.

1. Broadly, the larger system sizes are achieving higher net bandwidth to file. This is probably to be expected. If one compares these figures with the corresponding figures for 4 aggregators per stripe (data not shown), the bandwidth has saturated for system size 256^3, may have saturated at 512^3, but may not have saturated at 1024^3. However, further increases in the number of aggregators were not attempted.

2. There is a clear apparent "floor" (in the sense of going down the table) in I/O scaling as a function of process count below which I/O performance appears to collapse. This occurs moving from 16 to 32 processes for 256^3, moving from 128 to 256 processes for 512^3, and moving from 1024 to 2048 processes for 1024^3. This pattern suggests that the reason is common; however, the cause is still under investigation. This is important, because it affects the reasonable scaling regime (when run without I/O; see Section 2). This effect does not depend on the number of aggregators per stripe.

3. For the largest system size (2048^3), performance is severely hampered by the I/O, particularly related to the full state. However, for a system of this size, output to a single shared file should almost certainly be abandoned in favour of multiple files. This was not investigated.

In summary, the new I/O mechanism will replace the existing decompsoition-dependent version, aiding usability. The abstraction of the I/O data representation and handling should allow straightforward implementation of other formats such as HDF5.

2 Communication tasks and scaling

In this section we address the performance of the standard nearest-neighbour halo exchanges for lattice quantities.

A classic “trick” to manage three-dimensional halo exchanges is to perform three exchanges in turn, where each exchange is limited to one dimension. This obviates the need for explicit communication between diagonal neighbours in the Cartesian picture, and requires six messages per process in total. However, the shortcoming is that processes must synchronise after the exchange in each co-ordinate direction to ensure that the sides and corners are treated correctly. This imposition of order is potentially inefficient: it prevents the possibility that messages from other co-orrdinate directions can be handled first. It also means that the halo swap must take place as a single entity, and does not admit overlap of communication and computation.

This synchronisation has been eliminated in a new implementation of the halo exchanges: this requires a maximum of 26 messages (send and receive) for each process in three dimensions. We move on to discuss the role of OpenMP in the halo exchanges.

2.1 OpenMP for lattice Boltzmann distribution halo swaps

In a hybrid MPI/OpenMP picture, all parts of the computation should use threads wherever possible to maximise performance. We have made the policy decision at the outset [4] to adopt `MPI_THREAD_FUNNELLED`, that is the master thread is responsible for all message passing operations; strictly, the thread that called `MPI_Init()`. This means that all thread synchronisation decissions can be handled by the application itself. In particular, the additional complexity that is potentially introduced by `MPI_THREAD_MULTIPLE` is avoided, along with overheads associated with conservative synchronisation assumptions which are mandated in the MPI library itself.

The solution adopted is split into two parts, a send phase and a receive phase which can, in principle, be separated to allow potential for overlap of communication and computation.

The initial send phase follows the standard approach of issuing non-blocking receives for all incoming messages first. This allows that any incoming messages that have arrived can be handled as soon as possible. We then need to move the outgoing data to the relevant application communication buffer and issue a non-blocking send. Schematically, this is:

```
#pragma omp parallel
{
  for (int ireq = 0; ireq < nreq; ireq++) {
    lb_data_send(...); /* Pack data for outgoing message */
  }
}

for (int ireq = 0; ireq < nreq; ireq++) {
  MPI_Isend(...);      /* Non-blocking send for outgoing message */
}
```

The role of OpenMP here is to workshare the packing of individual messages. This allows all threads to work on all messages, including the largest messages. (Recall that the outoging messages are of significantly different sizes: $O(1)$ for the corners, $O(L)$ for the edges, and

$O(L^2)$ for the faces of a local domain of linear dimension L.) There should be no further thread synchronisation in the parallel region — worksharing for independent messages can be achieved with `nowait`.

The solution for the receive phase is, schematically:

```
MPI_Waitall(...);   /* Wait for all sends and revcs to complete

#pragma omp parallel
{
  for (int ireq = 0; ireq < nreq; ireq++) {
    lb_data_recv(...);  /* Unpack received message */
  }
}
```

The strategy here is to allow MPI to complete both receives and sends using `MPI_Waitall()`, and then use a similar single parallel region to unpack the incoming message buffers. The details of the packing an unpacking mechanism are discussed further in the following section.

It might be hoped that a strategy of the form

```
#pragma omp parallel
{
  for (int ireq = 0; ireq < nreq; ireq++) {
    MPI_Waitany(...);       /* Wait for a receive */
    lb_data_unpack(...);    /* Unpack the related message */
  }
}
```

might be advantageous. However, investigations revealed that the additional thread synchronisation required was harmful to performance compared with the approach outlined above. This extended to both OpenMP task-based mechanisms and standard worksharing mechanisms.

2.2 Reduced halo swap

It is possible to implement a reduced hlao exchange for the lattice Boltzmann distributions which excludes elements of the distribution with no component of discrete velocity in the relevant directional exchange. This is conveniently placed in the packing/unpacking routines which are, schematically:

```
#pragma omp nowait
for (int ih = 0; ih < nx*ny*nz; ih++) {
  ...
  for (int p = 0; p < lb->nvel; p++) {
    int propagates = ...;
    if (lb->full || propagates) {
      ... /* copy data */
    }
  }
}
```

This illustrates that, for each lattice site in the current halo region (index `ih`), a loop over each discrete velocity is required, and a criteria for whether a given velocity needs to be moved can be based on the current halo exchange direction. If a full halo exchange is required, all velocities propagate. but if not then only a subset are required. This reduces the size of the messages.

A previous implementation of the reduced halo swap used MPI derived data types to encode which elements of the distribution should be moved. However, this required a separate implementation for each velocity basis set, and limits scope for the use of OpenMP: all the copies must be performed internally by the MPI library.

2.3 OpenMP in halos for order parameter fields

A similar approach to the halo exchanges for other model quantities has been adopted. These exchanges include scalar, vector and tensor fields. Such halo exchanges always involve messages in all 26 nearest-neighbour directions in three dimensions, and no reduced mode is relevant.

A particular case of the halo exchange for the velocity field in the solution of the Beris-Edwards equation (standard benchmark SC16) provides an opportunity to investigate any possible improvement from the new implementation in terms of overlap of communication and computation. The picture for the “unitary” halo swap has been, schematically:

```
hydro_halo_u(hydro);               /* Velocity field halo */
...
beris_edw_h_driver(be, ...);       /* Molecular field computation */
beris_edw_update_driver(be, ...); /* Beris Edwards update */
```

The computation of the molecular field, which is relatively expensive, is independent of the velocity field which is only required at the point of the Beris Edwards update, which computes the dynamics for a liquid crystal order parameter. This provides scope for overlap of communication, schematically:

```
hydro_halo_u_post(hydro);          /* Start velocity field halo */
...
beris_edw_h_driver(be, ...);       /* Molecular field computation */
hydro_halo_u_wait(hydro);          /* Complete halo exchange */
beris_edw_update_driver(be, ...); /* Beris Edwards update */
```

Results for this permutation (not shown) were inconclusive. There seemed to be no identifiable consistent improvement within the error. The usual explanation for such a lack of improvement is that, while the overlap is sound is principle, in practive messages do not actually make progress until the wait stage, meaning that simply delaying the wait stage does not decrease the time taken significantly.

2.4 Benchmarks

A number of benchmarks are now presented which look first at the halo exchange specifically, and then more broadly at the performance of the final code complete with the updated MPI-IO implementation discussed in the previous Section. The benchmark problem is also the same as described earlier [5].

2.4.1 OpenMP in halo exchanges

Figure 1 shows a figure-of-merit — here the product of system volume in lattice sites and the number of iterations or timesteps per unit time per NUMA region — against the number of NUMA regions. This should be viewed as a rate of useful work per unit computational resource for which higher values represent better performance. This test was run with 1 MPI process per NUMA region (16 cores or 16 OpenMP threads) using a fixed problem size; the figure-of-merit is therefore a proxy for strong scaling in this case. The results compare the original halo exchange without OpenMP, and the updated implementation, along with an MPI-only run (16 MPI processes per NUMA region). The results are based on 100,000 halo exchanges, and errors are estimated by internal variability for the run in each case: it is estimated the the errors are no larger than the symbols in the Figure.

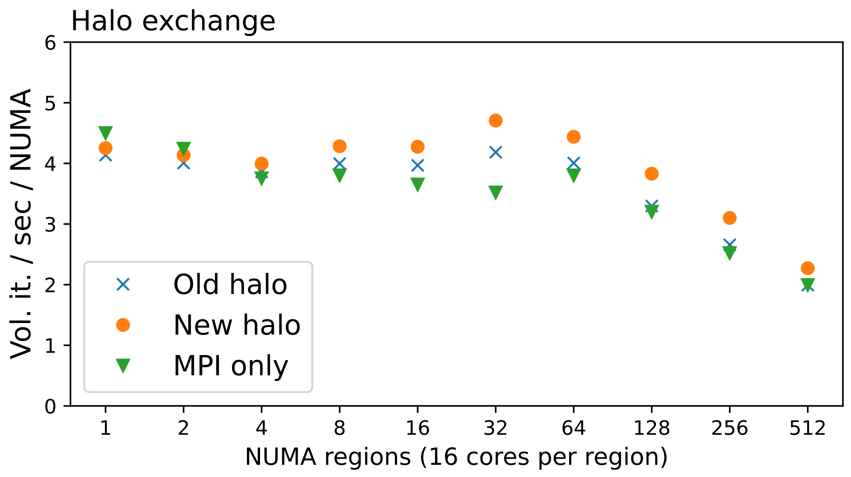

Figure 1: Performance as measured by volume iterations per second per NUMA region as a function of the number of NUMA regions for a fixed problem size. The new OpenMP implementation (circles) is compared with the original implementation without OpenMP (crosses). An MPI-only result (triangles) is also shown to illustration that this is not competitive with the hydrid approach. At this problem size (128^3) strong scaling is limited to around 32-128 NUMA regions (4-16 nodes). The errors in the results are estimated to be no larger than the symbols.

The results in Figure 1 show that the hydrid OpenMP approach improves the scaling in the important region — at the limit of strong scaling — compared with the MPI-only approach. The new implementation which includes OpenMP specifically in the halo exchanges themselves performs best.

2.4.2 Scaling

Having established that OpenMP halo exchanges are effective on a relatively small scale, larger system sizes can now be considered. Figure 2 shows an aggregated picture of strong and weak scaling using a similar figure of merit which is the system volume times the number of iterations achieved per second per node. This is an absolute measure which allows comprison on a linear vertical scale up to 4096 nodes (524,288 cores).

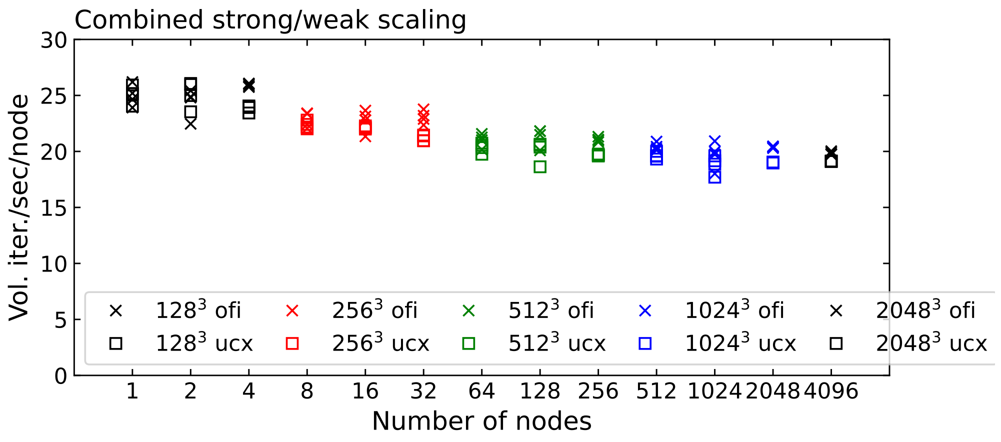

Figure 2: A view of both strong and weak scaling for a standard liquid crystal benchmark without I/O. A figure-of-merit — volume iterations per second per node — is plotted against the number of nodes for different system sizes: 128^3 (1, 2 and 4 nodes), 256^3 (8, 16, and 32 nodes), 512^3 (64, 128 and 256 nodes), 1024^3 (512, 1024, and 2048 nodes), and finally 2048^3 (4096 nodes only). All runs have 2 MPI processes per node. In all cases, runs were repeated using both OFI (crosses) and UCX (squares) implementations of the underlying communication libraries. Four repeats were run in each case, and all four data points are shown for both OFI and UCX (some are indistinguisable).

The standard liquid crystal benchmark as described in Section 1 is used. All runs were made using 2 MPI processes per node (one per socket) and 64 threads per MPI process; the largest number of MPI processes was thus 8192 (on 4096 nodes). Runs are made both with an without I/O; the case with no I/O is considered first. Runs are also made with both OFI (open fabrics interface) and UCX (unified communication X) as the underlying network interface layer.

For each system size, a range of node counts was used, and a subset are shown in Figure 2. (The full range of node counts is the same as that detailed in Table 1.) For each case, four repeats have been made, and all the data points are plotted. Both OFI and UCX runs are executed in the same batch job for given system size and node count, i.e., they use the same set of nodes. For a fixed system size, ideal strong scaling would be represented by a horizontal line in the Figure; weak scaling would maintain the horizontal line at the same level of performance. For example, for the 128^3 system size, strong scaling is maintained between 1 and 4 nodes (with 2 MPI processes per node, 4 nodes represents a favourable balanced decomposition into 8 cubes in three dimensions). There is a small but evident decrease in performance as the system sizes become larger at the node counts shown. Broadly, it can be seen that reasonable scaling at the level of 80% parallel efficiency is maintained across the range. There is a small but largely consistent improvement in performance using OFI comapred with UCX observed in these cases.

The corresponding case with I/O (specifically, output) is shown in Figure 3. The exact disposition of file output is described in Section 1, and all output is to a single file via MPI-IO. Again, both the runs with and without I/O have been executed in the same batch job for given system size and node count, along with the OFI and UCX cases. Four runs were performed at each system size, with differing numbers of I/O aggregators per stripe (12, 24, 48, and 96 aggregators). Again all four data points are shown individually.

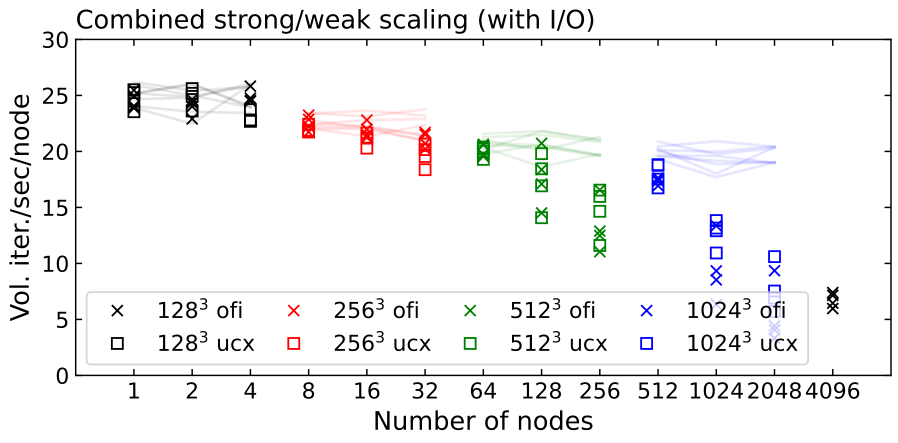

Figure 3: A consolidated picture of strong and weak scaling with output to a single shared file using MPI-IO. The figure-of-merit is plotted as a function of the number of nodes; all runs have 2 MPI processes per node. The system sizes are 128^3 (1, 2, and 4 nodes), 256^3 (8, 32, and 64 nodes), 512^3 (64, 128, and 256 nodes), 1024^3 (512, 1024, and 2048 nodes), and 2048^3 (4096 nodes only). Crosses are the results for OFI and squares are results for UCX network interfaces. The corresponding results for no output are shown by lines as a guide.

Figure 3 shows that the strong scaling is reasonable for the smaller system sizes, but is much more problematic for the larger system sizes when compared with the situation without output. As noted in the discussion of the aggregated output bandwidth rates recorded in Table 1, there is a notable failure in scaling beyond a certain limit for the larger problem sizes. The picture from Figure 3 is a more gradual degradation of perform than would be suggested from Table 1. The later does not correctly weigh the large configuration file written at the end of the run, which takes significant time in the largest problem sizes and nodes counts. As noted, the mitigation would be to split the output into a number of separate files. This has not been investigated here.

3 Kernel tasks and performance

This section presents some relevant comments on kernel performance and socket-level performance analysis.

3.1 Baseline: socket-level performance

The standard benchmark for a fluid-only liquid crystal problem is used again. The effectiveness of the threaded model is assessed at the socket level by undertaking a saturation exercise in which an increasing system size is run using fixed resource (one socket or 64 cores). The smallest system size is an $8\times8\times8$ cube, while the largest is 236^3. Intermediate systems sizes are not all cubic, so different permutations in (x, y, z) have been run to gauge the variation in time. Systems are run with either 4 MPI processes and 16 OpenMP threads per process, 2 MPI processes and 32 threads per process, or 1 MPI process with 64 threads. In addition, the influence of memory layout was selecting either a structure-of-arrays (SOA) ordering, or an array-of-structures (AOS) [6]. The results are presented in Figure 4.

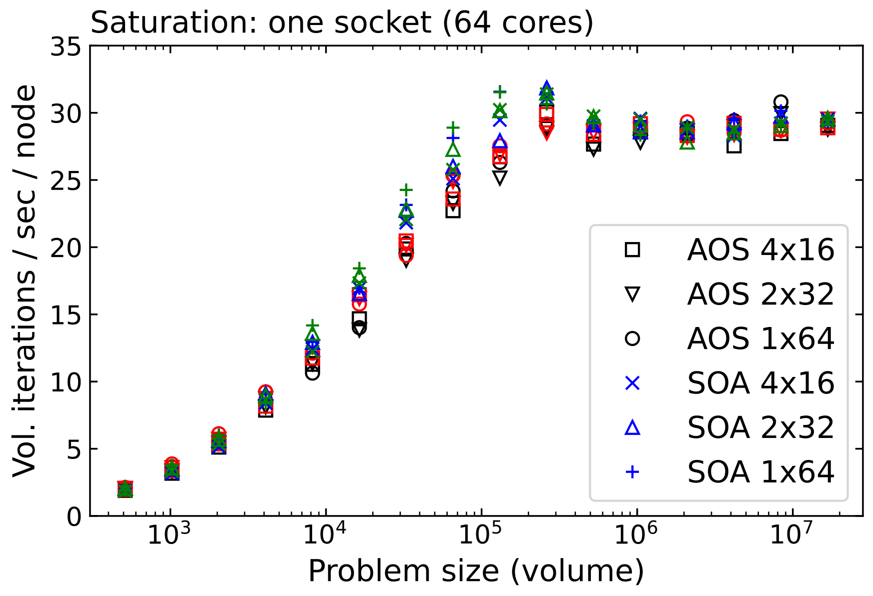

Figure 4: Saturation exercise for systems of increasing size run on a single socket with performance measured as the product of the system size and the number of iterations, divided by the execution time. A number of different cases are run: 4 MPI process with 16 threads each; 2 MPI processes with 32 threads each; 1 MPI process with 64 threads. Both array-of-structures (AOS) and structure-of-arrays (SOA) memory layout are considered. Different coloured symbols are different system aspect ratios at the same system size.

Broadly, as the system size increases, the figure-of-merit increases until a clear shoulder is seen at at around system size 10^5 lattice sites; beyond this point further increases in system size do not increase performance. The shoulder feature will be discussed presently. There is also clear evidence that the threaded implementation is efficient: there is little penalty in performance in increasing the number of threads per MPI process. In the region of the shoulder, a single MPI process is actually favourable for performance, particularly in the SOA case. This is the reason for the choice of 1 MPI per socket in the earlier scaling investigations.

The clear shoulder in the performance is ascribed to a cache effect. The L2 cache on ARCHER2 is 512 kB per core (L3 is 16 MB shared between 4 cores). The shoulder in the results can be equated to 128,000-256,000 lattice sites per 64 cores, or between 2000-4000 sites per core. If one estimates 160 bytes memory active per site (there are 20 doubles at least for D3Q19 lattice Boltzmann computation), then this is 324,000-648,000 bytes per core. One can therefore be fairly certain the shoulder is related to the capacity of the L2 cache. Further increases in system size are then limited by bandwidth to main memory (via L3 cache).

The exercise can be repeated for benchmarks with different memory requirements. For example, a different lattice Botlzmann basis D3Q27 increases the memory requirement per lattice site, and results in a lower figure of merit compared with D3Q19. However, the qualative features of the results (not shown) are the same, and the same conclusions are relevant.

One wider question that can also be addressed from the saturation plot is a quantative estimate for the limit of strong scaling. If one starts with a (large) fixed system size, strong scaling will move the size per computational resource to the left in the scaling plot. As an example, from Figure 4 the critical size is around 100,000-200,000 lattice sites, which is around 32x64x64—64x64x64 as a three-dimensional section of the system. For a system of 128^3, this would suggest the limit of strong scaling would be reached at 8 MPI processes (4 nodes if running 1 MPI process per socket). This is consistent with the picture obtained in Figure 2, as are all the results for larger systems. More qualitatively, this is saying that strong scaling must maintain effective L2 cache untilisation: if the local problem is too small, performance will degrade irrespective of message-passing overheads.

3.2 OpenMP `taskloop`

The question of whether OpenMP tasks can be used effectively in place of standard OpenMP worksharing for kernels is now addressed. The potential benefit of such an approach would be to allow the combination of relatively inexpensive kernels, and even serial code, to provide overall improvement in time-to-solution. This would represent a complement to the graph-based executaion models now offered by GPU APIs [7].

A typical replacement might be a loop with a fixed number of iterations implemented using

```
#pragma omp for
for (int index = 0; index < iterations; index++) {
  /* ... kernel body ... */
}
```

by the equivalent using a number of tasks which we set to be the same as the number of threads:

```
#pragma omp single
{
  #pragma omp taskloop num_tasks(nthreads)
  for (int index = 0; index < iterations; index++) {
    /* ... kernel body ... */
  }
}
```

As an exploratory step, the implementation was changed in a small number of central kernels (collision and progration) to use `taskloop`. The exercise described in the previous section was then repeated for the SOA memory layout. Reuslts are presented in Figure 5.

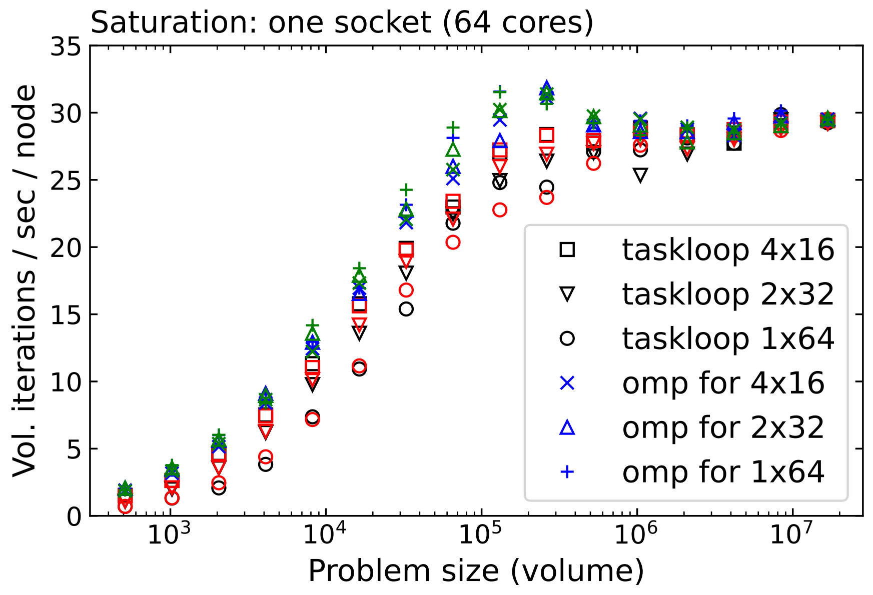

Figure 5: Single-socket saturation exercise for the standard liquid crystal benchmark comparing the performance of OpenMP `taskloop` with a standard OpenMP worksharing decomposition. Each data point represents a run with either 4 MPI processes and 16 threads each, 2 MPI processes with 32 threads each, or 1 MPI process with 64 threads. Additional coloured symbols are different aspect rations for a given system size.

The results in Figure 5 show that there is a clear performance degradation associated with the use of `taskloop`. This is most severe for the case with 1 MPI process and 64 threads. This is perhaps unsurprising, as data locality favours a fixed mapping of threads to array sections (allocated on the basis of first touch), and a static worksharing schedule is more likely to maintain such locality.

In addition to the performance question, use of `taskloop` would imtroduce a number of complications in the TargetDP [8] thread level abstraction implementation. The first is the additional `single` region introduced for generation of tasks, which is rather cumbersome. More seriously, thread-level reductions in kernels — where required — are much less straightforward in the task picture. Taking these factors together, it was decided to retain the standard static worksharing in kernels in the code.

3.3 Remediation for poorly performing kernels

Expanding the use of OpenMP forces the programmer to address all code with has a performance implication. This was found to be particularly important as the number of threads was increased to 64 per MPI task (using a complete socket) on ARCHER2. It was found that OpenMP was needed in all diagnostic routines, which might be executed at a frequency of once per 100 or 1000 time steps in a typical production run to reduce unnecessary serialisation.

Diagnostic routines involving globals sums, such as those used to computate conserved quanities including mass, momentum, and order parameter composition, required that sums be computated in the thread implementation. It was found useful to implement some of these as compensated sums to improve repeatability on different numbers of threads. In addition, it was found that the penalty for false sharing associated with accumulated sums was very noticeable on ARCHER2; appropriate padding can be used to eliminate this performance problem.

The liquid crystal anchoring gradient has been refactored to allow a more accurate anchoring condition consistent with threads; this involved refactoring to replace a number of functions used in different locations by `inline` versions. Kernels were added for an alternate force computation approach using the gradient of the chemical potential rather than the derivative of the stress.

3.4 Electrokinetics problems

A number of steps have been implemented to improve the usability and performance in the electrokinetic sector of the code. These include introduction of run-time options to select solver and stencil characteristics (which were previously compile-time options). Performance for problems involving charge is generally dominated by the time taken to solve the Poisson equation for the electric potential.

4 Accessibility, sustainability, and documentation

4.1 Accessibility

The addition of accessibility checks for the documentation website via continuous integration was investigated, but it was found that the available tools which do not charge money do very little beyond trivial checks. The idea was, unfortunately, abadoned. Accessibility checks remain manual.

4.2 Sustainability

Code quality via lgtm.com has been replaced by codeQL. The alert count has been reduced significantly during the course of the work, and the goal of zero alerts has been achieved in the duration of the work. The queries are `+security-and-quality`. New code quality alerts in the existing code are treated as they arise. Addition of new alerts from new code can be explicit avoided. A formal code coverage mechanism has beeen apoted for the unit tests.

The issue of an out-of-source build (e.g., using cmake) was omitted owing to lack of time.

4.3 Documentation

A significant expansion in the tutorial-level material in the documentation has accompanyied this work.

Acknowledgement

This work was funded under the embedded CSE programme of the ARCHER2 UK National Supercomputing Service (http://www.archer2.ac.uk).

Notes and References

[1] The code is C, and uses its own virtual function tables to provide polymorphism in the context of such interfaces.

[2] This may actually not be a problem of memory, but one of the failure of 32-bit integer addressing for very large local domain sizes in the current code. This will be addressed in a future release.

[3] The number of MPI-IO aggregators was set via, e.g.,

```
    export MPICH_MPIIO_HINTS=''*:cray_cb_write_lock_mode=2,
                               *:cray_cb_nodes_multiplier=8''
```

for 8 aggregators per stripe. This is the only difference between runs bar system size/process count.

[4] Initial investigation of `MPI_THREAD_MULTIPLE` suggested a small performance decrease even when running the original halo exchange implementation.

[5] All benchmarks were performed using the AMD compiler (`PrgEnv-aocc` on ARCHER2) either with version 3.0 or version 3.2.

[6] Array-of-structures is usually used for CPU architectures. The code allows a compile-time switch to structure-of-arrays to allow use on GPU architectures.

[7] Both CUDA and HIP now provide an API to manage work as the execution of a directed acyclic graph.

[8] A. Gray and K. Stratford, A lightweight approach to performance portability with Target DP, *International Journal of HPC Applications*, **32**, 288–301 (2018).