# Communication and I/O masking for increasing the performance of Nektar++
# eCSE 02-13 technical report

Rupert Nash, Simon Clifford, Chris Cantwell, David Moxey, and Spencer Sherwin

May 24, 2016

**Abstract**

This report summarises the work done and results obtained during this eCSE project to improve the Nektar++ spectral/hp element framework. The project targetted two main areas for improvement: I/O and threading.

For I/O we added support for the PETSc DMPlex sub-library for unstructured meshes. Unfortunately there were issues in the library and despite working with the developers, could not get this functional in the time available. We also added support for HDF5 for field data.

Regarding threading, we modified the Nektar++ threading library to allow multiple independent threads, providing a general solution to I/O and inter-process communication masking. We implemented an alternative conjugate-gradient algorithm that allows overlap between computation and communication. We investigated the shortcomings of some MPI implementations regarding how they wait for other processes.

## 1 Profiling and load imbalance

### 1.1 I/O performance

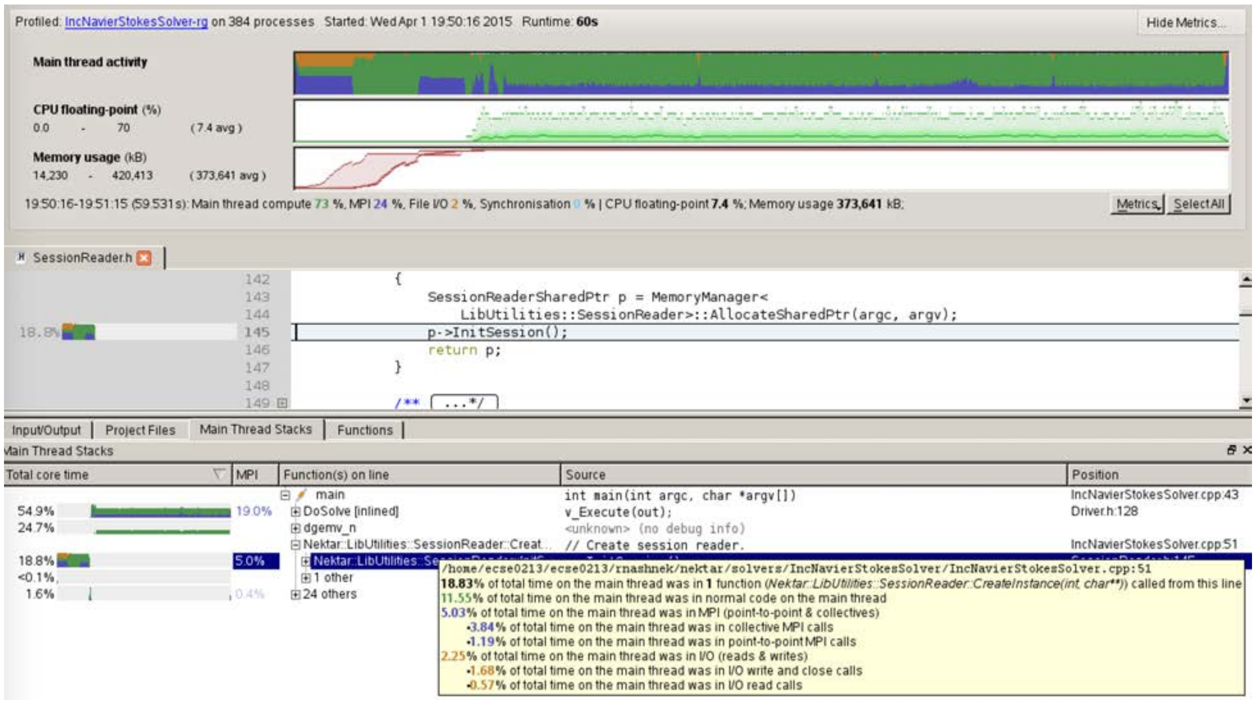
We selected two main ways of testing I/O performance. First, to run Nektar++ in its usual mode where it actively solves a set of equations and then writes the output, while being profiled by the MAP tool from Allinea. Second, to create a simple benchmarking executable that simply loads in an existing FLD file and writes it out a number of times recording the the time for each experiment. This can optionally be profiled with MAP also.

The first testing dataset is a mesh of a pair of intercostal arterial branches in the descending aorta, as described in Cantwell et al. [1] and available as supplementary material S6. This uses the incompressible Navier-Stokes solver (`IncNavierStokes`) with a Reynolds number of 300 and represents a typical medium-sized simulation for Nektar++. The mesh contains approximately 60,000 elements (prisms and tetrahedra).

This problem ran efficiently up to 32 nodes (768 cores) before the low number of elements per rank became problematic. In figure 1 we show an Allinea MAP profile of the program's execution on 16 node (384 cores). The total run time was 60s and, overall, 2% of this is spent doing IO operations. The profile naturally divides into two sections, a setup phase (the factory function `SessionReader::CreateInstance`) and an execution phase.

For this problem, 19% of run time is spent in initialisation, overwhelmingly in the main setup function `SessionReader::PartitionMesh()`. In figure 2 we show a MAP profile of this function only. This spends 37% of its time reading and parsing the input file with TinyXML, 26% of its time doing communication, 10% of its time *destroying* TinyXML objects, and 9% of its time performing filesystem metadata operations.

Writing a single complete checkpoint took approximately 75ms. Given that each timestep took on average 240ms, this is an acceptable overhead at this scale. Because the initialisation becomes such a dominant part


Profiled: IncNavierStokesSolver-rg on 384 processes Started: Wed Apr 1 19:50:16 2015 Runtime: **60s**

19:50:16-19:51:15 (59.531 s): Main thread compute 73 %, MPI 24 %, File I/O 2 %, Synchronisation 0 % | CPU floating-point 7.4 %; Memory usage 373,641 kB;

```
            {
                SessionReaderSharedPtr p = MemoryManager<
                    LibUtilities::SessionReader>::AllocateSharedPtr(argc, argv);
18.8%           p->InitSession();
                return p;
            }

            /** ...*/
```

Main Thread Stacks

| Total core time | MPI | Function(s) on line | Source | Position |
|---|---|---|---|---|
| | | main | int main(int argc, char *argv[]) | IncNavierStokesSolver.cpp:43 |
| 54.9% | 19.0% | DoSolve [inlined] | v_Execute(out); | Driver.h:128 |
| 24.7% | | dgemv_n | <unknown> (no debug info) | |
| | | Nektar::LibUtilities::SessionReader::Creat... | // Create session reader. | IncNavierStokesSolver.cpp:51 |
| 18.8% | 5.0% | Nektar::LibUtilities::Se | | |
| <0.1% | | 1 other | | |
| 1.6% | 0.4% | 24 others | | |

```
/home/ecse0213/ecse0213/rnashnek/nektar/solvers/IncNavierStokesSolver/IncNavierStokesSolver.cpp:51
```
**18.83**% of total time on the main thread was in **1** function (*Nektar::LibUtilities::SessionReader::CreateInstance(int, char**)*) called from this line
11.55% of total time on the main thread was in normal code on the main thread
5.03% of total time on the main thread was in MPI (point-to-point & collectives)
- 3.84% of total time on the main thread was in collective MPI calls
- 1.19% of total time on the main thread was in point-to-point MPI calls

2.25% of total time on the main thread was in I/O (reads & writes)
- 1.68% of total time on the main thread was in I/O write and close calls
- 0.57% of total time on the main thread was in I/O read calls

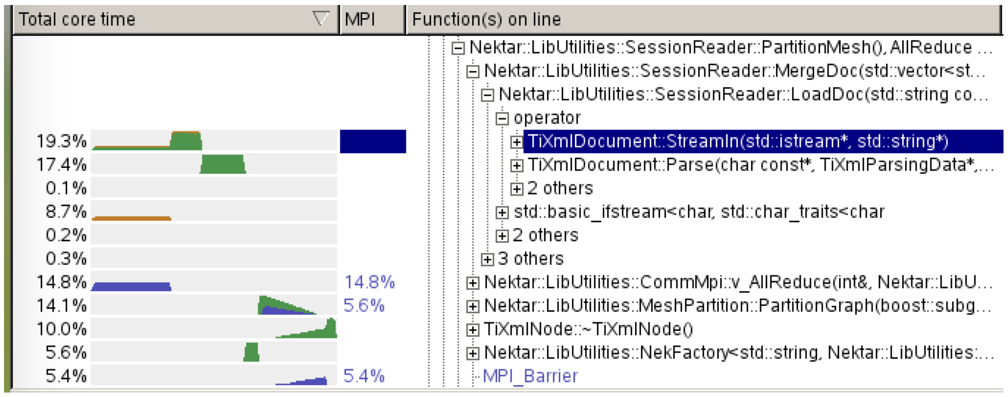
Figure 1: Allinea MAP profile of an `IncNavierStokesSolver` run on 384 cores

of the run time at larger scales (see below), we chose not to examine field output in in this way, but to use the benchmarking tool described below.

The second dataset is a more refined mesh, using curved elements, of aorta. It contains just over 150,000 elements. For this problem, we use the advection-diffusion-reaction solver (`ADRSolver`) to estimate mass transport. We ran this on 1 up to 64 nodes (24–1536 cores) under Allinea MAP. The timings for initialisation and solution are shown in figure 3. The solution time shows excellent strong scaling until 768 cores, when each core has less than 200 elements on average. At this point the system appears to still have good load balance, but the amount of computation is no longer enough to hide the communication required.

The initialisation shows a similar breakdown into constituent parts to the smaller problem, however at larger core-counts the time spent communicating become much more significant: increasing from 50% (24 cores) to 98% (1536 cores).

To better understand the performance on field data I/O we wrote a simple command-line tool to benchmark this. The tool, `FieldIOBenchmarker`, simply reads a Nektar++ field file and writes it back to the filesystem using Nektar's `FieldIO` class. We ran this tool on a complete checkpoint field from the large aorta problem above. The results of this are plotted in figure 4. The initial decrease in time is because each rank has a smaller amount of data to convert to XML and write. At larger core counts, the communication overhead of describing which elements will be written to each processor's output file becomes dominant.



```
Total core time        MPI     Function(s) on line
                               ⊟ Nektar::LibUtilities::SessionReader::PartitionMesh(), AllReduce ...
                                 ⊟ Nektar::LibUtilities::SessionReader::MergeDoc(std::vector<st...
                                   ⊟ Nektar::LibUtilities::SessionReader::LoadDoc(std::string co...
                                     ⊟ operator
19.3%                                  ⊞ TiXmlDocument::StreamIn(std::istream*, std::string*)
17.4%                                  ⊞ TiXmlDocument::Parse(char const*, TiXmlParsingData*,...
 0.1%                                  ⊞ 2 others
 8.7%                                ⊞ std::basic_ifstream<char, std::char_traits<char
 0.2%                                ⊞ 2 others
 0.3%                              ⊞ 3 others
14.8%                  14.8%     ⊞ Nektar::LibUtilities::CommMpi::v_AllReduce(int&, Nektar::LibU...
14.1%                   5.6%     ⊞ Nektar::LibUtilities::MeshPartition::PartitionGraph(boost::subg...
10.0%                            ⊞ TiXmlNode::~TiXmlNode()
 5.6%                            ⊞ Nektar::LibUtilities::NekFactory<std::string, Nektar::LibUtilities:...
 5.4%                   5.4%     MPI_Barrier
```

Figure 2: Detailed MAP profile of time spent in function calls during initialisation

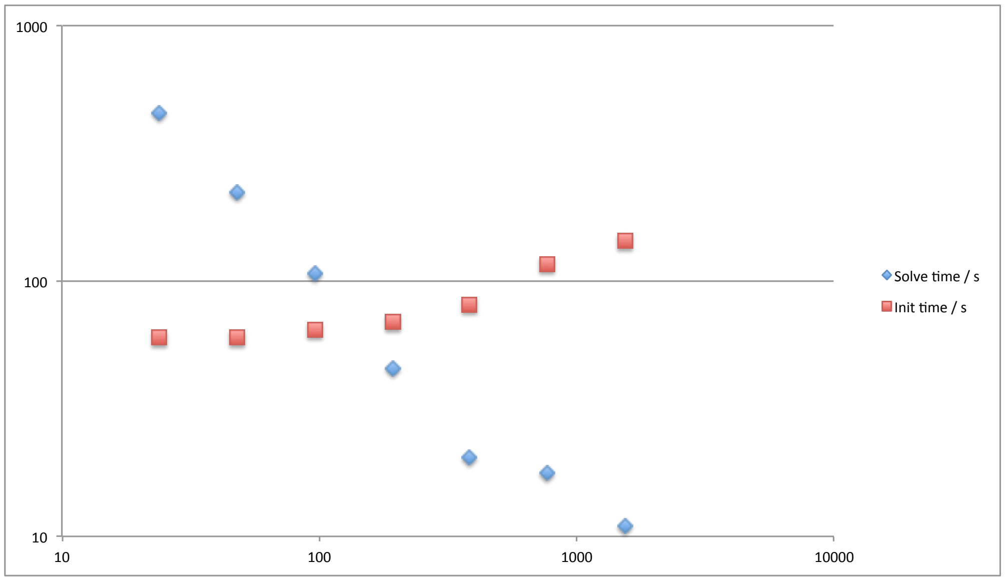


Figure 3: Scaling of ADRSolver on the large aorta problem. Blue shows the solution time and red shows the initialisation time. Number of cores is shown on the x-axis.

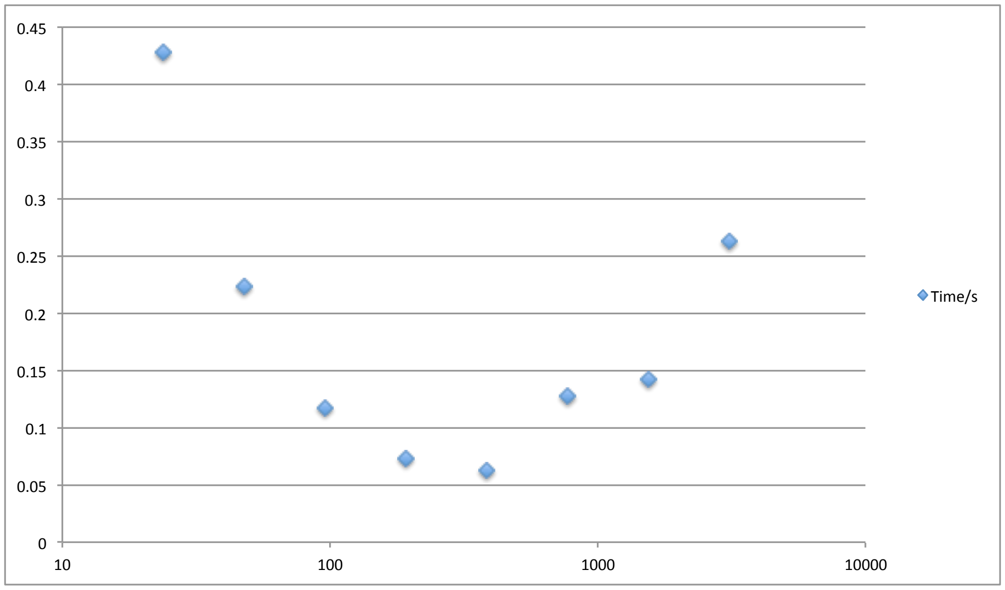


Figure 4: Initial time to write a checkpoint (150k elements). Number of cores shown on the x-axis.

# 2 Streamlining of XML

## 2.1 Mesh format

The Nektar++ XML format is flexible, but rather verbose and, like all XML-based formats, only supports sequential access. At the top level, the file begins with a `<NEKTAR>` tag, which must contain a `<GEOMETRY>` tag, specifying the dimensionality of the elements (`DIM`) and the space in which they lie (`SPACE`). The `<GEOMETRY>` element then contains the following child elements:

- `<VERTEX>` - this has a list of child elements `<V ID="$I">$X $Y $Z</V>`, where `$I` is a unique integer ID for the vertex, and `$X`, `$Y` and `$Z` are floating point numbers (the z-coordinate being omitted in a 2D space).
- `<EDGE>` - this has a list of child elements `<E ID="$I">$V1 $V2</E>`, where where `$I` is a unique integer ID for the edge and `<$V1>` and `<$V2>` are vertex IDs from the previous section.
- For 3D elements, `<FACE>`. This has a list of child elements, the type depending on whether it is a triangle (`<T>`) or a quad (`<Q>`). These are given an ID attribute similar to above and then list three or four (respectively) edges from the previous section. In the case of 2D elements, this tag is called `<ELEMENT>` and the next one is omitted.
- `<ELEMENT>` has a list of primitives defining the solution domain. The type can be tetrahedron (`<A>`), pyramid (`<P>`), prism (`<R>`), or hexahedron (`<H>`). These have an ID attribute similar to above and then list the required number of face IDs.
- `<CURVED>` has a list of all non-straight edges. Each child tag `<E>` has the following attributes: `ID` a unique integer ID; `EDGEID` the ID of the edge being curved; `NUMPOINTS` the number of points parameterising the curve; and `TYPE` a string identifying the interpolation scheme. Then, enclosed by this tag follow `NUMPOINTS` × `SPACE` floating point numbers specifying the point required, laid out as `X1 Y1 [Z1] X2 Y2 [Z2] ....`
- `<COMPOSITE>` Each child tag `<C>` specifies a union of previously defined objects, typically the elements and the boundary faces.
- `<DOMAIN>` this tag refers to one or more composites and defines the solution domain.

## 2.2 Plan

The profiling done above clearly showed that one of the key bottlenecks for simulation setup was communication done to distribute the input data, rather than the file reading in itself or the partioning with ParMETIS. Based on this and discussions with one of the PETSc contributors, we decided to use the PETSc `DMPlex` module [5, 6] to describe and (de)serialise meshes.

The DMPlex system takes a very flexible, abstract approach to describing a mesh, using a directed, acyclic graph, with vertices, edges, faces and cells all being nodes on the graph. An example of the graph for a single tetrahedron is shown in figure 5. Arbitrary data can be attached to any of these entities, such as coordinates to the vertices or curvature data to the edges. Using an external, widely supported library such as PETSc to manage this data on our behalf is a clear benefit in terms of maintenance, especially when the roadmap includes PETSc support for parallel mesh decomposition and loading. The DMPlex framework also has potential benefits in geometric preconditioning and mesh manipulation (by making easy queries such as "return all the cells that share a face with cell X").

The plan for implementation was to first create a DMPlex output module for the `MeshConvert` tool which is used to convert input data from other formats to Nektar's own format. Second, create a reader module for `MeshConvert` to ensure that the other operations (refinement etc) that it can perform are possible directly on the new mesh format. Third, implement a new `MeshPartition::ReadGeometry` method that can accept a DMPlex object rather than a TinyXML object.

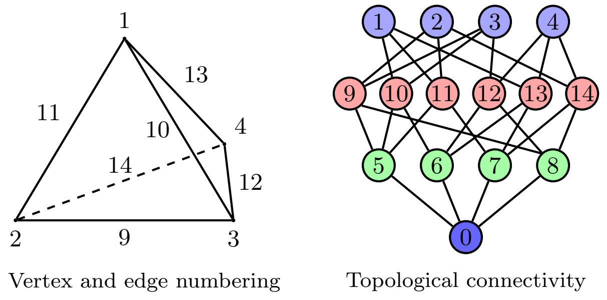


Figure 5: DAG-based representation of a single tetrahedron in DMPlex, from Lange et al. [5]

### 2.3 MeshConvert

The first step was to add CMake configuration options to add the necessary compilation and linking flags to use PETSc. Since the PETSc library requires initialisation, exactly once, before any PETSc API calls, we created a simple environment class that the user must instantiate near the start of their program (taking care to protect the call from any other threads). By taking a RAII (resource allocation is instantiation) approach and keeping track of which instance "owns" the PETSc environment through a flag, the environment can be cleanly shut down through a call to `PetscFinalize()` in the wrapper's destructor.

We chose to separate the mesh format into two parts: the core unstructured mesh data into the DMPlex format (i.e. the XML elements `<VERTEX>`, `<EDGE>`, `<FACE>`, and `<ELEMENT>`) while initially at least keeping the the Nektar-specific data (`<CURVED>`, `<COMPOSITE>`, `<DOMAIN>`) in the existing XML. The XML file will remain the top level file and will contain the filename of the new DMPlex file.

The tool has a simple, in-memory representation of the mesh that closely matches the Nektar++ XML format. This also maps reasonably easily onto the form required by DMPlex. However the DMPlex implementation must make two traverses through the mesh data structures. First one must set the size of the "chart" (the entire graph) and then set the so-called "cone" size of each entity, the cone being the set of lower-dimension entities that comprise it (e.g. the cone of a face is the set of its edges). Then an API call will allocate memory for storing the links of the graph. The second pass then sets the indices of the cone for each element, being careful to ensure the proper mapping from DMPlex's single global index space to Nektar's per-entity-type index spaces. Finally, the coordinates of the vertices can be added to the plex using a `PetscSection` object.

PETSc's documentation claims that DM objects can be serialised using PETSc's viewer objects, most interestingly using the HDF5 format. However while this worked on trivial test cases (e.g. a square subdivided into two triangles) this failed on more realistic cases with cryptic errors. Detailed debugging revealed that the DMPlex serialisation routines were specialised for meshes with only a single type of face and element and had not implemented the full range of element shapes needed.

We worked with some of the PETSc contributors to add the required functionality, but could not get this working in the time available. We took the decision to pause this until the DMPlex functionality is available.

## 3 Implementation of HDF I/O support

Nektar++ writes output as fields into an XML-based format. In parallel, this is written as one file per MPI rank, plus an associated metadata file describing which elements are written to which rank's file. This operation as well as field import is performed by instances of the `FieldIO` class. In write mode, this accepts a vector of `FieldDefinitions`, a vector of data vectors to write and some further metadata. The field definitions is a structure that describes the mapping between mesh elements with the expansion bases across them and the one-dimensional list of floating point numbers supplied.

The XML format begins with the usual XML declarations and a `<NEKTAR>` tag. This has a child `<Metadata>` that includes provenance data and solver-specific infomation such as the simulated time and other parameters. Then a number of `<ELEMENTS>` tags follow. Each one of these corresponds to a `FieldDefinitions` and vector of data pair. The field definitions are encoded as attributes of the `<ELEMENTS>` tag:

- `FIELDS="u,v,w,p"`
- `SHAPE="Prism"`
- `BASIS="Modified_A,Modified_A,Modified_B"`
- `NUMMODESPERDIR="UNIORDER:5,5,5"`
- `ID="24134,24170-24174,24186,..."`

The elements are output in groups of uniform shape and expansion basis. The IDs of all mesh elements are listed.

As part of this work, which required adding more parallel communication calls, we realised that the extant abstraction of MPI call (or in the serial case trivial no-operation equivalents) was relatively time consuming to add extra calls and argument types to, requiring a lot of boiler-plate. To ease this burden, the old member functions, which relied on C++ overloading to select the correct implementation, were replaced with templated equivalents. These use standard argument deduction coupled with a traits class template[1] to select the correct `MPI_Datatype`. Partial specialisations for `std::vector` and `Nektar::Array` objects were also added. With these changes, Nektar can support communication of all built in datatypes immediately and of user defined types by specialising the new `CommDataTypeTraits` class template.

## 3.1 Implementation

To enable multiple options for field I/O, we refactored the single `FieldIO` class into an abstract base class with two concrete subclasses: `FieldIOXml` and `FieldIOHdf5`. Instantiation was delegated to a factory object, in keeping with the Nektar++ style. All uses thoughout the code base were moved to the new factory. HDF5 files are, as one would expect from the name, hierarchical. So called *groups* can contain other groups or *datasets*. A dataset is a core object for storing data and must have both extent and type information associated with it. Both groups and datasets may have *attributes*, which are named objects that can be treated like datasets but they may only store small amounts of data.

Initially, we chose to make a fairly direct translation from the XML format to HDF5, leaving the file-per-rank plus an info file in place. As a first step, we designed a simple object-oriented wrapper around the HDF5 API. We were unable to use the included C++ bindings as this is explicitly not supported in conjunction with parallel operations. However the design was strongly influenced by this. We took a RAII approach, i.e. objects “close” their associated HDF object when they destruct to ensure easy management of resources. We implemented a minimal set of the HDF5 functionality anticipated by this work and added to it as needed.

We used an enhanced version of the type traits technique used for MPI to avoid a proliferation of member functions to (de)serialise data to attributes and datasets. This class had to have factory static member functions to create the HDF5 datatype objects needed as well as functions to perform any necessary type conversions.

To deal with the metadata, we wrote a simple abstraction for writing hierarchical attribute data (`TagWriter`) that has two operations: adding a child node or adding a named string attribute. The concrete implementations for XML (HDF5) mapped these concepts to creating a child element (creating a group) and creating child element with enclosed text (creating a string attribute).

The element data is then treated in a similar manner to the XML, creating a single group for each output item, the field definitions data being added as attributes of the group, with the exception of the ID data, which, due to its potentially large size, is added as a dataset. The data is naturally added as a dataset.

[1]see e.g. http://accu.org/index.php/journals/442

The deserialisation process also mimics the XML approach but mapped to HDF5 concepts. We had to implement iterator classes for both attributes and "links" (i.e. groups or datasets) to support this. A further complication was allowing the client code to open a file without having to know whether it was XML or HDF5 based. This was implemented through a factory function that inspects the first few bytes of the file for the HDF5 "magic number" and then creates the appropriate subclass.

## 3.2 Benchmarking results

Using the `FieldIOBenchmarker` tool described above, we performed scaling tests of the new implementation, using the larger aorta dataset (150k elements). In figures 6 and 7 we show the results of write and read performance, respectively, for both the XML and HDF5 formats. In both cases, the figures show comparable results. At low core counts, the time to completion decreases slightly for HDF5 and very significantly for XML. This is probably due to a roughly constant file access time (approx 50ms) combined with a decreasing amount of work per rank. Since the XML parsing/construction and (de)compression is much more demanding, this explains the large differences between the two formats. At larger core counts, both modes take significantly longer to complete the I/O. This is likely due to contention on the shared filesystem metadata servers.

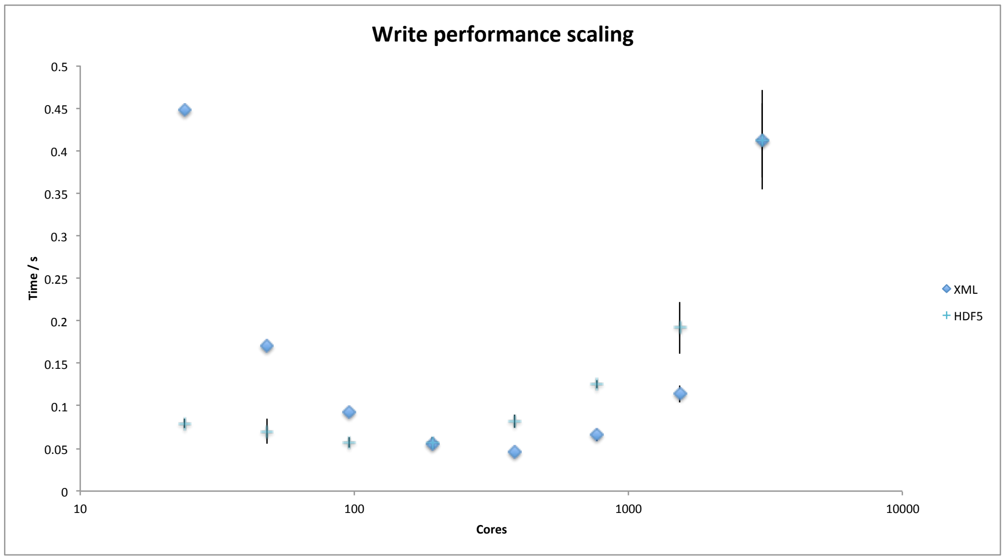


Figure 6: Average time (of ten tests) to perform a single field I/O write operation for the large aorta dataset.

## 3.3 Parallel file access

We next proceeded to implement parallel reading and writing of HDF5 files. The extension to this case initially seemed quite straightforward: for each field type, as each ranks knows how many elements and data values it must write, perform a collective operation to compute the offsets and the total size, then perform a parallel HDF5 write (by opening the file with the correct operations). In parallel HDF5, operations such as group and dataset creation are collective and require communication and possibly multiple processes accessing the metadata servers at once. This is known to be a bottleneck on some systems, so instead we chose to use a well-known patten and perform the file structure creation on a single rank in serial mode, before reopening the file collectively and performing the writes in parallel.

Unfortunately, we had not realised that, in general, the set of `FieldDefinitions` on each rank is not consistent across all ranks. Further, the field definition is very complex in order to allow Nektar's great

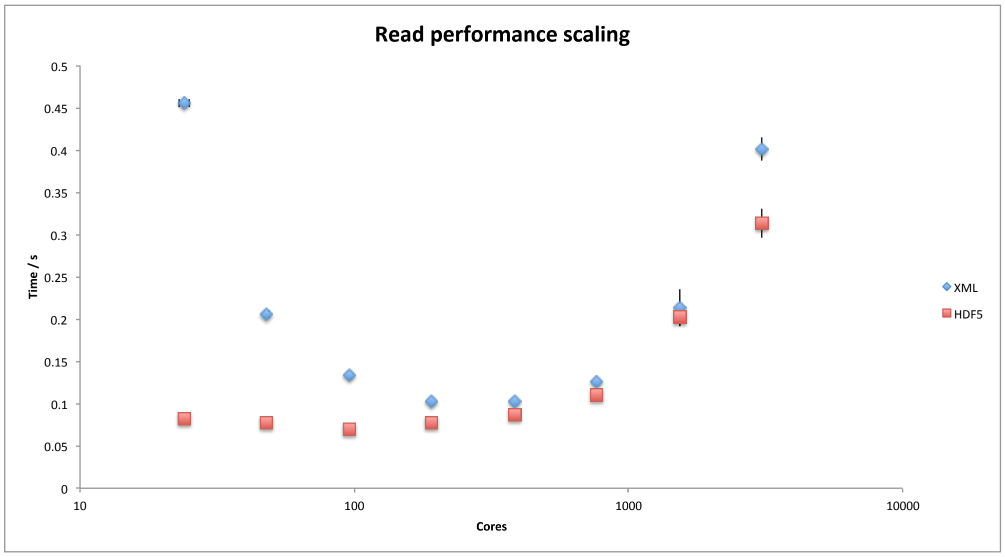


Figure 7: Average time (of ten tests) to perform a single field I/O read operation for the large aorta dataset.

flexibility. We attempted to implement the necessary reduction operation, to ensure that the root process had a consistent summary of all global field definitions, but could not produce a working implementation in the time available.

Since the end of this eCSE project, however, a European Horizon 2020 project, ExaFLOW, with EPCC and Imperial College among the partners has started, which includes effort to improve Nektar++'s scalability towards the exascale. As part of this, with have a preliminary implementation of this that is currently undergoing testing.

## 4 Masking of communication

The general problem to solve is to recover time (wall-clock time) that would otherwise be spent waiting for communications tasks, such as MPI message passing or I/O, to finish. Without masking the CPU is *waiting* [2] during communications; the goal is to have it continue to run code while the communication takes place. Subsequently there will come a point in the code where execution must stop until the communication is finished. For example, the result from an `MPI::AllReduce` is needed, or data must be read from a file that is being written to.

We chose to implement the communication and I/O masking using the previously implemented threading library in Nektar++. This provides a `ThreadManager` abstract class that creates long-lived instances of threads that pull tasks (instances of subclasses of `ThreadJob`) from a common job queue and execute them. A thread with no jobs to execute sleeps, consuming no CPU. A `Wait` method to synchronise the threads, so that all the `ThreadManager`'s child threads must finish before the master thread can progress is also provided. The concrete implementation uses the `boost::thread` framework. The library was modified to provide multiple named `ThreadManagers`, overseen and accessed through a Singleton `ThreadMaster` class. Each `ThreadManager` can schedule, queue jobs for, and wait for its threads independently of other `ThreadManagers`.

[2]Which may not be the same as idling.

## 4.1 Masking communication during the conjugate-gradient step

The thread library provides a general approach to overlapping computation with waiting for communication. Work is ongoing to find suitable parts of the code to apply the library, bearing in mind the caveats and pitfalls mentioned below. In some cases the original algorithms can be reformulated to allow greater overlap.

As an example, one of the iterative CG algorithms to solve $Ax = b$ used in Nektar++, due to Chronopoulos and Gear [2, 3], is shown in algorithm 1. It is a minor modification of the original CG algorithm that reduces the number of communications and improves memory management.

**Algorithm 1** Original Nektar++ CG algorithm

```
Input: $x_0, A, b, K$
 1: $r_0 \leftarrow b - Ax_0$;
 2: $q_{-1} \leftarrow 0, p_{-1} \leftarrow 0, \beta_{-1} \leftarrow 0$
 3: Solve for $w_0$ in $Kw_0 = r_0$
 4: $s_0 \leftarrow Aw_0$
 5: $\rho_0 \leftarrow (r_0, w_0); \mu_0 \leftarrow (s_0, w_0)$
 6: $\alpha_0 \leftarrow \rho_0/\mu_0$
 7: for $i = 0, 1, 2, \ldots$ do
 8:     $p_i \leftarrow w_i + \beta_{i-1}p_{i-1}$
 9:     $q_i \leftarrow s_i + \beta_{i-1}q_{i-1}$
10:     $x_{i+1} \leftarrow x_i + \alpha_i p_i$
11:     $r_{i+1} \leftarrow r_i - \alpha_i q_i$
12:     if $x_{i+1}$ accurate enough then quit
13:     Solve for $w_{i+1}$ in $Kw_{i+1} = r_{i+1}$
14:     $s_{i+1} \leftarrow Aw_{i+1}$
15:     $\rho_{i+1} \leftarrow (r_{i+1}, w_{i+1}), \mu_{i+1} \leftarrow (s_{i+1}, w_{i+1})$
16:     $\beta_i \leftarrow \rho_{i+1}/\rho_i$
17:     $\alpha_{i+1} \leftarrow \rho_{i+1}/(\mu_{i+1} - \rho_{i+1}\beta_i/\alpha_i)$
18: end for
Output: $x_{i+1}$
```

The inputs are $A$, $b$, and an initial guess $x_0$. $K$ is the preconditioner. The matrix $A$ might be represented sparsely or as a function that computes $y \leftarrow Ax$. In an MPI parallel job the matrix multiplies are effectively spread across the processors in a way that is highly parallel. However, the dot products at lines 16 and 17 are global reductions, and require that each processor contributes its data to the product. These then are synchronisation points in the algorithm. As each processor reaches this point in the code it must wait until the last processor reaches it before it may continue. Only in the unlikely situation that each processor has the same amount of work will there be no wasted time.

In algorithm 1 the two dot products can be performed at the same time (i.e. in one MPI global reduction), so there is only one synchronisation point. However, there is nothing that can be overlapped with this communication: $\beta$ and $\alpha$ are required immediately in the next step of the iteration.

An alternative algorithm from Ghysels and Vanroose [4] is shown in algorithm 2 (a discussion of it and similar algorithms is found at [7]). The inputs are $A$, $b$, an initial guess $x_0$, and the preconditioner $M$. There are now three global vector dot products per iteration (lines 3,4, and 7), however they are independent and can all be merged into a single communication. They can also be overlapped with the calculations on lines 5 and 6. The increase in computational effort to generate the vectors on lines 13 to 16 is ameliorated by observing that the two vector operations on each line can be performed in a single loop, reducing the number of memory accesses.

This algorithm was added to the Nektar++ library. At the start of the program a `ThreadManager` is initialised for this task, with one child thread. A `ThreadJob` subclass is created with the code implementing lines 5 and 6. To avoid complexity it was decided that all MPI calls (such as in lines 3, 4 and 7) would take

**Algorithm 2** Pipelined CG

```
Input: $x_0, A, b, M, \epsilon$
1: $r_0 \leftarrow b - Ax_0, u_0 \leftarrow M^{-1}r_0, w_0 \leftarrow Au_0$
2: for $i = 0, 1, 2, \ldots$ do
3:     $\gamma_i \leftarrow (r_i, u_i)$
4:     $\delta_i \leftarrow (w_i, u_i)$
5:     $m_i \leftarrow M^{-1}w_i$
6:     $n_i \leftarrow Am_i$
7:     if $(\|r_i\|_2 \,/\, \|r_0\|_2 \leq \epsilon)$ then quit
8:     if $(i = 0)$ then
9:         $\beta_i \leftarrow 0, \alpha_i \leftarrow \gamma_i/\delta_i$
10:    else
11:        $\beta_i \leftarrow \gamma_i/\gamma_{i-1}, \alpha_i \leftarrow \gamma_i/(\delta_i - \beta_i\gamma_i/\alpha_{i-1}$
12:    end if
13:    $s_i \leftarrow w_i + \beta_i s_{i-1}, r_{i+1} \leftarrow r_i - \alpha_i s_i$
14:    $p_i \leftarrow u_i + \beta_i p_{i-1}, x_{i+1} \leftarrow x_i + \alpha_i p_i$
15:    $q_i \leftarrow m_i + \beta_i q_{i-1}, u_{i+1} \leftarrow u_i - \alpha_i q_i$
16:    $z_i \leftarrow n_i + \beta_i z_{i-1}, w_{i+1} \leftarrow w_i - \alpha_i z_i$
17: end for
Output: $x_i$
```

place from a master thread. At the start of each iteration of the `for` loop the instance of the `ThreadJob` subclass is sent to the `ThreadManager` with references to $m_i$, $M^{-1}$, $w_i$, $n_i$, and $A$. The waiting thread is woken and immediately starts executing the code on lines 5 and 6. Meanwhile, the master thread continues, and starts the MPI calls for the communication steps of the global dot-products. For a short while many of the MPI processes will be mostly occupied in wait states freeing[3] them to work on the child thread. Once the master thread has finished the MPI communication it calls the `Wait` method, forcing synchronisation. The master will only progress once the child thread has finished executing (if it has already finished then the master continues immediately).

## 4.2 Issues with communication masking

Adding threading to any application brings a novel set of problems. Many are associated with data sharing: apart from the trivial "all reading" situation, if threads access the same memory locations data corruption may happen in non-deterministic ways. Similarly, one thread releasing dynamic memory (say by a C++ destructor) may cause a thread that is still accessing that data to segfault, read garbage data, overwrite (newly re-acquired) data, or continue normally. Libraries may be thread-unsafe, or may implement thread-safety by serialising all use. The C++ STL is (generally) thread-safe, but does not protect the programmer from using its tools in thread-unsafe ways. For example, pushing an element to the end of a `std::vector` may cause an automatic resizing of the vector, which will invalidate all pointers, references, and iterators to this vector held by other threads.

An unexpected problem is that many MPI implementations use *busy wait*. When an MPI process is waiting for other processes to respond in some way it may *sleep* and wait for the operating system to wake it when the response arrives. Alternatively it may *poll*, continually checking to see if the reponse has arrived. The former approach is kinder to other processes (and threads) on the system, as a process that is sleeping is not scheduled on the run queue, and so consumes no CPU. Polling, on the other hand, means that the CPU is 100% occupied in checking for the response and is only available to other processes or threads in a shared fashion. The current wisdom is that any serious MPI program will be running on a dedicated machine, where each processor has only one process allocated to it. By polling, the process will respond as

[3]But see the next section

fast as possible to any response, reducing latency. This means that with busy wait overlap is impractical. Some MPI implementations offer idle waiting (where the process sleeps, or polls a certain number of times, then sleeps) as an option. However, this has made gathering data on how beneficial the masking is difficult.

## Acknowledgement

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

## References

[1] C D Cantwell, D Moxey, A Comerford, A Bolis, G Rocco, G Mengaldo, D De Grazia, S Yakovlev, J E Lombard, D Ekelschot, B Jordi, H Xu, Y Mohamied, C Eskilsson, B Nelson, P Vos, C Biotto, R M Kirby, and S. J. Sherwin. Nektar++: An open-source spectral/hp element framework. *Comput. Phys. Commun.*, 192:205–219, July 2015.

[2] A T Chronopoulos and C W Gear. s-step iterative methods for symmetric linear systems. *J. Comput. Appl. Math.*, 25:153–168, 1989.

[3] James W Demmel, Michael T Heath, and Henk A van der Vorst. Parallel Numerical Linear Algebra. *Acta Numerica*, 2:111–197, 1993.

[4] P Ghysels and W Vanroose. Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm. *Parallel Computing*, 40:224–238, 2014.

[5] Michael Lange, Matthew G Knepley, and Gerard J Gorman. Flexible, Scalable Mesh and Data Management using PETSc DMPlex. *arxiv Condensed Matter e-prints*, May 2015.

[6] Michael Lange, Lawrence Mitchell, Matthew G Knepley, and Gerard J Gorman. Efficient mesh management in Firedrake using PETSc-DMPlex. *arxiv Condensed Matter e-prints*, June 2015.

[7] Fangfang Liu, Chao Yang, Yiqun Liu, Xianyi Zhang, and Yutong Lu. Reducing Communication Overhead in the High Performance Conjugate Gradient Benchmark on Tianhe-2. In *Distributed Computing and Applications to Business, Engineering and Science (DCABES), 2014 13th International Symposium on*, pages 13–18. IEEE, 2014.