Improving Load Balancing and Parallel Partitioning in Fluidity

Paul Woodhams[a], Jon Hill[b], Patrick Farrell[b]

[a]*HECToR CSE Team, The Numerical Algorithms Group Ltd., Oxford*

[b]*Earth Systems and Engineering, Imperial College London, London*

Abstract

Fluidity is a non-hydrostatic, finite element/control volume CFD numerical forward model used in a number of scientific areas; geodynamics, ocean modelling, renewable energy, and geophysical fluid dynamics. The applications cover a range of scales from laboratory-scale problems, through to whole earth mantle simulations. One of the unique features of Fluidity is its ability to adapt the computational mesh to the current simulated state: dynamic adaptive remeshing. When running on multiple processors the adapted mesh must be load balanced, which involves swapping elements from processor to processor. Zoltan is a library that performs such dynamic load balancing. Here, we document the steps taken to integrate Zoltan within Fluidity which extends the current functionality of Fluidity, thereby extending the range of science areas that it can be applied to. Although there is a small performance hit over the current customised load balancing library, this is only noticeable when the number of elements per process is small.

1. Aims and Objectives

The purpose of this project was to re-engineer the parallel anisotropic mesh adaptivity and load balancing algorithms of Fluidity (Piggott et al., 2009; Gorman, 2006) by incorporating Zoltan, a collection of data management services for unstructured, adaptive and dynamic applications. The aim of this was to improve the scaling behaviour of Fluidity and allow the adaptive remeshing algorithms (Pain et al., 2001) to be used in parallel, on any element pair, rather than being restricted to a single pair.

Zoltan includes a suite of parallel partitioning algorithms, data migration tools, parallel graph colouring tools, distributed data directories, unstructured communication services, and dynamic memory management tools. In addition to possibly improving scaling, the inclusion of Zoltan has also improved software sustainability, and add new functionality. This will deliver new performance capabilities that will prepare Fluidity for petaflop systems, such as those proposed by the PRACE project, when combined with the new capabilities will enable new science to be carried out.

In addition to allowing new features to be added to the adaptivity capabilities of Fluidity, the inclusion of Zoltan has allowed Fluidity to be coupled with other models such as atmospheric and ice-sheet models, a key part of future research, as these applications require a non-standard discretisations. Finally, a recent development in Fluidity was the addition of Lagrangian particles, which are free-moving particles in the flow and are used as either detectors or within agent-based modelling. This is now in need of parallelisation which, when considering possible contradictory

Email addresses: paul.woodhams@nag.co.uk (Paul Woodhams), jon.hill@imperial.ac.uk (Jon Hill)

load-balancing objectives between the mesh and the particles, is a non-trivial task. Load-balancing and parallel data migration algorithms for such purposes already exist in the Zoltan library (e.g. Rendezvous) and hence has aided in this objective.

The project was set out into six distinct packages of work with measurable deliverables. These were as follows:

1. To become familiar with the Fluidity code, build process and benchmark suite. A short report detailing the times for computation, communication, data migration and adaptivity for one of the benchmarks would be produced.

2. Adding the Zoltan solution for parallel adaptivity as a compile time option for Fluidity. Using the Zoltan interface to ParMETIS the functionality of the previous solution should be replicated and the Zoltan solution should pass all tests in the Fluidity test suite (unit, short, medium and long). Additional tests comparing Zoltan functionality against the previous solution as well as unit tests for the communications functionality of Zoltan should be completed. A short report detailing the implementation produced as a dCSE Technical Report.

3. Zoltan solution used as the default in all Fluidity development branches and future releases of Fluidity. All Zoltan development branches merged back into the trunk and all tests passing using the Zoltan solution. Documentation for all Zoltan options provided in the Fluidity manual and Fluidity options system, Diamond (Ham et al., 2009).

4. Investigate performance improvements for Fluidity using the various Zoltan options made available through the Fluidity options system, diamond. Aim is for a reduction of 15% or more in communication time for all HPC benchmarks when using the Zoltan solution compared against the previous solution. Aim for speed-up from 64 to 2048 processes to be increased by 15% using the Zoltan solution compared to the previous solution. All work to be documented in a short report. Code extensibility as important as code performance improvements but there are no deliverables related to this.

5. Final report including profiling results for the four HPC benchmarks completed.

6. Participate in and present updates on progress at the Fluidity HPC Workshops.

This report will begin by giving some background on Fluidity, along with the adaptivity and parallel adaptivity algorithms implemented in Fluidity. The Zoltan library will then be discussed. This will be followed by a description of the implementation and the profiling results from the HPC benchmarks run on HECToR will be presented. Finally the conclusion will give details of the outcomes from this project and how it has benefited HECToR users.

2. Background

2.1. Fluidity

Fluidity is an open source, general purpose, multi-phase computational fluid dynamics code capable of numerically solving the Navier-Stokes equations and accompanying field equations on arbitrary unstructured finite element meshes in one, two and three dimensions (Piggott et al., 2009; Pain et al., 2005). It is used in a number of different scientific areas including geophysical fluid dynamics (Mitchell et al., 2010), computational fluid dynamics (Hiester et al., 2011), ocean modelling (Pain et al., 2005) and mantle convection (Davies et al., 2011). It uses a finite

element/control volume method which allows arbitrary movement of the mesh with time dependent problems, allowing mesh resolution to increase or decrease locally according to the current simulated state (Fig. 1). It has a wide range of element choices, including mixed formulations. Fluidity is parallelised using MPI and is capable of scaling to many thousands of processors on the UK national HPC service, HECToR. Other innovative and novel features are a user-friendly GUI, Diamond (Ham et al., 2009) and a python interface which can be used to calculate diagnostic fields, set prescribed fields or set user-defined boundary conditions.

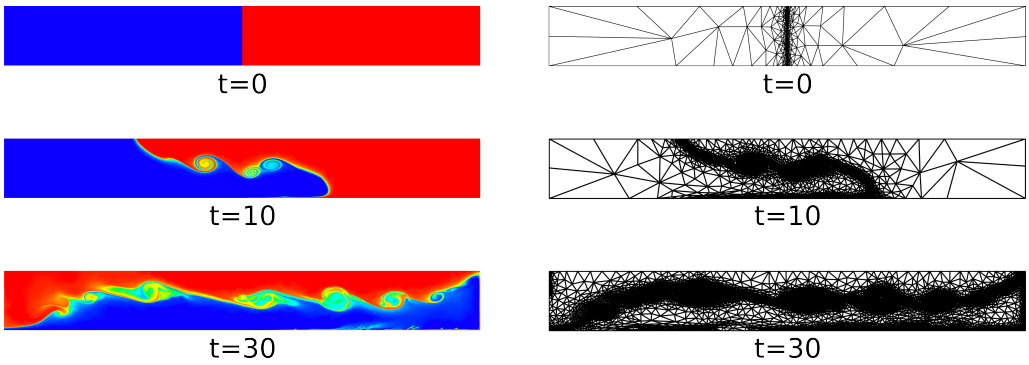

Figure 1: *Example of a Fluidity adaptive mesh simulation of the lock-exchange problem. The mesh is adapted to the temperature field (red-blue colour) and velocity (not shown). The mesh is refined where a sharp curvature exists, preserving this interface, whilst reducing overall computational load (Hiester et al., 2011)*

2.2. Adaptivity

Dynamic adaptive remeshing, hereafter known as adaptivity, is a process of adapting the mesh according to an error, which is a function of the current model state and user-defined functions. This allows the *a posteriori* metric to be used to alter the mesh as the simulation proceeds. Once *a posteriori* metrics have been computed, there are many possible ways of modifying the discretisation to achieve some error target. These include *h*-adaptivity, which changes the connectivity of the mesh (Berger and Colella, 1989); *p*-adaptivity, which increases the polynomial order of the approximation (Babuška and Suri, 1994); and *r*-adaptivity, which relocates the vertices of the mesh while retaining the same connectivity (Budd et al., 2009). Combinations of these methods are also possible (e.g., Houston and Süli (2001); Ledger et al. (2003)).

Fluidity uses a powerful combination of *hr*-adaptivity, since the meshes produced are not constrained by the previous mesh; therefore, this approach allows for maximum flexibility in adapting to solution features. However, this flexibility comes at a cost: guiding the adaptive remeshing procedure (choosing what mesh to construct), executing the adaptation (constructing the chosen mesh) and data transfer of solution fields (from the previous mesh to the newly adapted mesh) become more complicated than with hierarchical refinement.

The metric, M, on which adaptivity is carried out is calculated as follows. For each field, f_i to be included in the metric, metrics:

$$M_i = \frac{1}{\epsilon_{f_i}} |H(f_i)| \tag{1}$$

where ϵ_{f_i} is a user defined weight and $|H(f_i)|$ is the matrix formed using the absolute values of the eigenvalues of the Hessian matrix. The use of a tensor allows anisotropic directional information to be included and hence influence the adaptivity.

The adaptive algorithm carries out the following stages (also see Fig 2):

1. Construct the Hessian of the fields the user has specified for inclusion in the error metric, which depends on a number of user-defined parameters, such as minimum and maximum edge lengths.
2. Convert these Hessians into metric tensors (as above).
3. Merge the metric tensors from each field to provide a single metric.
4. Smooth the metric to prevent large jumps in mesh size, equivalent to mesh gradation.
5. Apply any other user-specified constraints on the metric, such as the maximum number of nodes allowed.
6. This metric is now passed to the adaptivity library which constructs a new mesh based on it.
7. Fields are then transferred from the old mesh to the new mesh via interpolation (Farrell, 2011).

More details of the algorithm and the methods employed can be found in the Fluidity manual and Farrell et al. (2009)

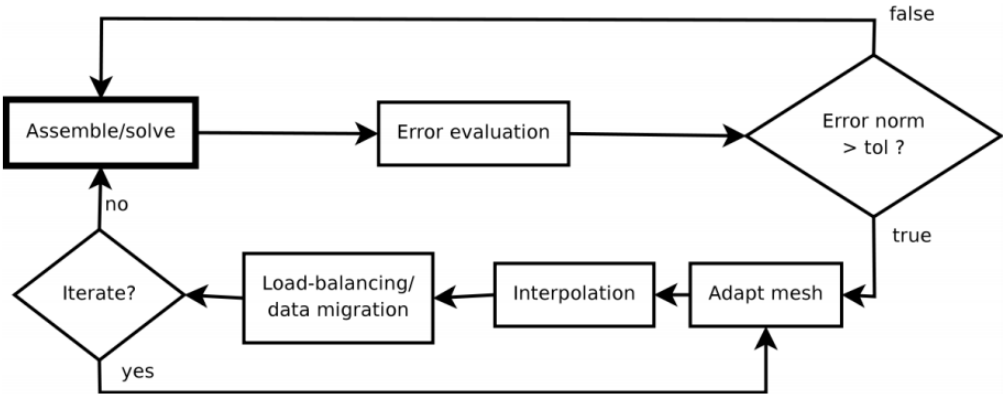

Figure 2: *The parallel adapt algorithm employed by Fluidity*

2.3. Parallel Adaptivity

The adaptivity method used in Fluidity as described in the previous section (2.2) is inherently a serial process. Parallelising the method itself across multiple MPI processes would lead to a large amount of complexity and interprocess communication. A review of methods is given in Farrell (2009). Instead the following approach is taken (Fig 3):

- Each process adapts their local mesh, excluding halo elements, according to the algorithm defined in the previous section.
- The mesh is re-partitioned with high edge-weighting applied to those elements below the element quality cut-off.
- Repeat the two steps above up to adapt iterations (default value of 3).
- Finally re-partition the mesh without applying edge-weighting to return a load balanced mesh.

This approach uses the serial adaptivity method previously described to adapt all the elements which are not in a halo. Those elements in a halo are also resident on another process so if they were adapted the same adaption would be needed on the remote process requiring communication. By locking the halo elements this communication is avoided but after the local adapt step the locked elements will not have been adapted and hence may be of poor quality. By re-partitioning with high edge-weights applied to the poor quality elements the partitioner should attempt to not cut those edges and hence the poor quality elements will be in the centre of a domain and can be adapted during the next local adapt iteration. The current load re-balancing is carried out by bespoke code, Sam Gorman (2006). Sam only works with piecewise linear elements and does not allow detectors or periodicity within the domain. It is a hard-coded solution, such that any new element pairs would need bespoke code written in order to perform the migration of data. However, it does have optimisations that are not present in Zoltan, such as the migration of element and field data simultaneously.

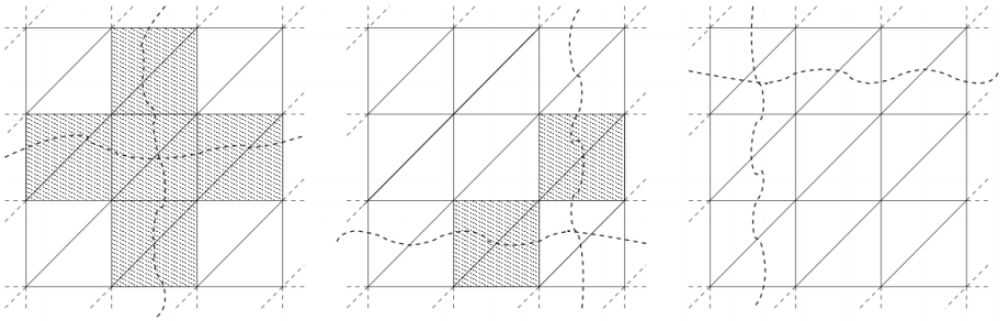

Figure 3: *2D mesh example where the graph partition is indicated by the broken wavy line: (right) sub-optimal restricted elements (hatched) forms a polyline; (centre) after repartitioning only isolated elements remain locked; (left) all elements have been visited.*

2.4. Zoltan

Zoltan is a library of routines for parallel partitioning, load balancing and data management (Devine et al., 2002; Boman et al., 2007). It is developed by Sandia National Laboratories as an open source library. Zoltan is also available as part of the Trilinos package, again from Sandia National Laboratories. Zoltan gives access to various different partitioning libraries; ParMETIS, PT-Scotch as well as its own graph and hypergraph partitioners (Catalyurek et al., 2009). Given these features it can be used as a replacement for Sam within the Fluidity code.

3. Implementation

The implementation makes use of the Zoltan functions for load balancing and for data migration; `Zoltan_LB_Balance` and `Zoltan_Migrate`. There are also some calls to the auxiliary function for inverting send/receive lists, `Zoltan_Invert_Lists`. In this section an outline of the implementation will be given as well as some more details on how the implementation was structured.

The implementation follows the basic outline below:

- Setup module variables
- Set Zoltan parameters
- Register Zoltan callback functions

- Load balance the nodes making up the mesh
- Migrate owned nodes
- Construct list of halo nodes still needed
- Invert the list of halo nodes needed to get send list
- Migrate the halo nodes
- Construct new mesh and other data structures
- Allocate fields on the new mesh
- Transfer the fields data for each element on the new mesh
- Clean up module variables

All of these steps were implemented in three Fortran module files; `Zoltan_integration`, `Zoltan_callbacks` and `Zoltan_global_variables`. The subroutine zoltan_drive from `Zoltan_integration` implements the process outlined above and it is this routine which is called from elsewhere in Fluidity. `Zoltan_callbacks` contains all of the callback functions that are provided to Zoltan. `Zoltan_global_variables` is setup to contain various Fluidity variables and data structures which are shared between `Zoltan_integration` and `Zoltan_callbacks`. It was unfortunate that global variables were necessary but due to the callbacks required by Zoltan having a fixed interface this was the only way to allow access to the required Fluidity data structures in the callback functions.

The auxiliary function `Zoltan_Invert_Lists` is very useful as it allows each process to just build a list of nodes or elements it needs. The list each process creates is the receive list required by the `Zoltan_Migrate` function and through using `Zoltan_Invert_Lists` the send list which is also required for `Zoltan_Migrate` can be obtained.

It is not feasible to give details in this report of how each of these steps was implemented as many require understanding of the Fluidity data structures. Further details can be found in the well documented source code [1].

3.1. Callbacks

As Zoltan is a general purpose parallel re-partitioning and data distribution library it is data agnostic. This means callback functions must be provided which Zoltan can query to retrieve the information it needs from the Fluidity data structures during the Zoltan library calls. There are four distinct sets of callback functions implemented in Fluidity; one set for load balancing and three sets for data migration, one for each of local nodes, halo nodes and fields.

It is not feasible to present full details of the implementation of each of the callback functions within this report. Instead a brief outline of the key callback functions will be presented here.

3.2. Load Balancing

When making a call to `Zoltan_LB_Balance` or `Zoltan_LB_Part` Zoltan is only concerned with the nodes making up the graph as well as how those nodes are connected, the edges. Four callback functions must be provided for these library calls:

[1] `https://launchpad.net/fluidity`

1. Counting the number of local nodes.
2. Listing those local nodes.
3. Counting the number of edges associated with each local node.
4. Listing edges for each local node (detailing the neighbouring node the edge connects to).

When providing the lists of nodes and edges a weight can also be provided. In this implementation non-uniform node weights are only applied when using an extruded mesh (an extruded mesh is one derived from a 2D surface mesh and extruded in the direction of gravity to produce a 3D mesh). Weighting is used more heavily for edges as it is key in the parallel adaptivity approach taken in Fluidity as described in section 2.3.

Edge weighting is applied in the callback function `zoltan_cb_get_edge_list`. This callback must provide for each edge of every local node the neighbouring node the edge connects to and which process currently owns the neighbour node. We also choose to provide an edge weight for each edge. The pseudo code (Listing 1) below details the implementation.

Listing 1: Edge weighting pseudo-code

```
for node = 1 to # local nodes
        store global id for each neighbouring node
        store owning process for each neighbouring node
        for element = 1 to # elements associated with local node
                determine element quality
                if (quality < local minimum quality )
                        local minimum quality = quality
        for nbor_node = 1 to # neighbouring nodes
            for element = 1 to # elements associated with neighbour node
                determine element quality
                        if (quality < neighbour minimum quality)
                                neighbour minimum quality = quality
                min_quality = min(nbor_min_quality, local_min_quality)
                if (min_quality < element_quality_cutoff)
                    edge_weight = ceil((1.0 - min_quality) * 20)
                else
                    edge_weight = 1.0


my_max_weight = max(my_edge_weights)
my_min_weight = min(my_edge_weights)
MPI_AllReduce(my_max_weight, max_weight, 1, MPI_FLOAT, MPI_MAX)
MPI_AllReduce(my_min_weight, min_weight, 1, MPI_FLOAT, MPI_MIN)


ninety_weight = 0.9 * max_weight
! avoid adjusting the weights if all elements are similar quality
if(min_weight < ninety_weight)
        for edge_weight = 1 to # total edges
                if (edge_weight > ninety_weight)
                        edge_weight = num_total_edges + 1
```

The implementation loops over all the local nodes and for each node records the global ID and owning process of all its neighbour nodes. The next step is to calculate the edge weight to apply to the edge between the local node and each neighbour node. The aim is to apply a high edge weight to poor quality elements. This is done by first determining the poorest quality element associated with the local node. Then for each neighbour node the poorest quality element associated with the neighbour node is determined. The minimum element quality for either the local node or the neighbour node is then used to calculate the weight applied to the edge between the local node and that neighbour node.

Once all of the edge weights have been calculated the nodes associated with the poorest quality elements then have their edge weights adjusted to a value one higher than the total number of edges in the simulation. This makes them uncuttable. Those edges with an edge weight calculated to be within 10% of the maximum edge weight on any process are given such a weight.

Edge weighting was implemented in such a way as to replicate the behaviour of the previous mesh repartitioning solution used in Fluidity. However, when using Zoltan, despite the edge weights being applied correctly, the mesh repartitioning sometimes failed to move the partition boundary away from the poor quality elements. This was found to be because Zoltan prioritises load balanced partitions over edge weighting and would sometimes ignore edge weighting to meet load balance criteria. The solution was to loosen the Zoltan parameter, `IMBALANCE_TOL`.

Within Zoltan priority is given to having well load balanced partitions. The amount of load imbalance tolerated in the system is controlled through the `IMBALANCE_TOL`. To determine the imbalance on each processor the weights of all the objects it is assigned are summed together to get its total load. The average load is calculated and from this the imbalance is computed as the maximum load divided by the average load. For example a value of 1.2 for `IMBALANCE_TOL` means that 20% imbalance is acceptable: that is no process should have more load than 1.2 times the average load.

The default `IMBALANCE_TOL` is 1.075 but for Fluidity this is changed to 1.5. The option has been made available through the Fluidity options system as `load_imbalance_tolerance`. This allows users to modify the value should they need to for their problem. This solved the problem with poor elements still being in a halo region after a repartition but added a further issue of empty partitions.

By loosening the `IMBALANCE_TOL` Zoltan would occasionally repartition the mesh in such a way that a process had no owned nodes. Fluidity assumes that no process will have an empty partition and hence this causes numerous problems throughout the code. It was not feasible to modify Fluidity to deal with empty partitions so instead a solution to prevent empty partitions was implemented. This was possible due to the implementation being split into distinct steps, first a load balance and then data migration.

The solution was to make a load balance call and then check for empty partitions before doing any data migration. The check for empty partitions was done by checking:

$$N_o + N_i - N_e \neq 0 \tag{2}$$

where N_o is the number of nodes owned, N_i is the number of nodes to import, and N_e is the number of nodes to export. This was possible as the `Zoltan_LB_Balance` returns both the number of nodes being imported to this process and the number of nodes it will be exporting.

If an empty partition would be created following the load balance then the load balance is attempted again but with the `IMBALANCE_TOL` tightened. This process continues until a partitioning with no empty partitions is found or the `IMBALANCE_TOL` can be tightened no further. In this situation a final load balance attempt is made with the edge weighting switched off and the `IMBALANCE_TOL` at 1.075. This solution prevents empty partitions being created and by going through this process before migrating the computational cost is reduced.

3.3. Data Migration

There are three calls to `Zoltan_Migrate` within Fluidity, each migrating a different set of data; nodes, halo nodes and fields. For each of these calls a different set of callbacks are provided to deal with packing and unpacking the different data being sent. Three callbacks need to be provided for each `Zoltan_Migrate` call:

1. The amount of storage needed for each graph node (mesh node, mesh halo node or fields data for an element).

2. How to pack data from Fluidity data structures into a Zoltan communication buffer.

3. How to unpack data from a Zoltan communication buffer into Fluidity data structures.

The Zoltan communication buffer is a byte array. This makes directly transferring data between it and the Fluidity data structures complicated as most Fluidity data is real. To simplify the process an intermediary real array is used for the packing and unpacking callbacks when migrating fields data. In the packing routine data is first packed from the Fluidity data structures into the real array. The Fortran intrinsic procedure transfer is then used to copy the packed real data into the integer Zoltan communication buffer. When unpacking the data is first copied from the integer Zoltan communication buffer into a real array and then data is unpacked from the real array into the Fluidity data structures.

3.4. Detectors

Detectors are either passive particles injected into the flow (Lagrangian detectors) or static probes within the flow. Lagrangian detectors can be used for a number of applications including calculating mixing within the flow or as agents in individual-based biological modelling. Both require high numbers of detectors to be injected into the flow. In a parallel-adaptive run the detectors must be moved along with their owning element when Zoltan deems that element should be moved to a new processor.

Within the Zoltan implementation detectors are dealt with in the callback functions for migrating the fields data. This is because each detector is associated with a particular element rather than a node and hence it is not known which process should own each detector until the transfer fields stage. There is also some work involved before and after the use of Zoltan to migrate the fields data to update certain data structures and to deal with corner cases.

Detectors are implemented as a Fortran derived type with all the information; position, name, element, type, id, etc. contained within the derived type. For the Zoltan implementation routines for packing/unpacking the detector information between the derived type and a real buffer were produced. Originally these were only used within Zoltan but they have since been adopted for use throughout the detector code in Fluidity whenever detectors are transferred between processes.

Before the fields are transferred the detector lists must be pre-processed. This is done because the detectors are stored in one or more unordered linked lists and in pre-processing them we avoid searching these lists during the packing callback. The pre-processing is done as a subroutine call from the callback function for determining the field data sizes. The pre-processing is done at this point as this is when the required list of elements being transferred from the process is available. The pre-processing consists of:

1. Determining the element associated with each detector.

2. If the detector's element is to be transferred, moving the detector from its current detector list to a list of detectors to be sent.

3. Keeping a count of the number of detectors to be sent with each element being transferred.

The pseudo code below details how the three pre-processing steps are implemented:

Listing 2: Detector re-distribution

```
for list = 1 to # detector lists
        for detector = 1 tp # detectors in list
```

```
Determine element the detector is associated with
for element = 1 to # elements being transferred
        Determine element universal number
        if (detector element = element universal number)
                Determine new element owner
                if (new element owner = current element owner)
                        Do not need to transfer the detector
                else
                        Move detector to list of detectors to pack
                        Increment number of detectors in the element
```

The implementation aims to avoid traversing the detector lists more than once. For each detector it is checked if its owning element is on the list of elements being transferred. If a match is found then the process which will own the element in the new partitioning is determined. If the new owner is the current owner then the element is only having its data transferred as it will be in another processes halo in the new partitioning. In Fluidity detectors are only dealt with by the process which owns the element containing the detector, so for halo elements detectors do not need to be transferred. If the new owner is not the current owner then the ownership of the element is changed in the new partitioning and the detector must be transferred. The detector is moved from its current detector list to a list of detectors being transferred and the count of the number of detectors in the element is incremented.

The pre-processing of the detector lists gives an array containing the number of detectors in each of the elements being transferred and an ordered list of the detectors to be transferred. The order in which detectors were added to the send detector list is the same as the order they will be removed from the list when the detectors are packed into the communications buffer with the other fields data. The pre-processing therefore allows the send detector list to be traversed serially during the packing process and avoids needing to search the send detector list for detectors which must be sent with each element.

The following modifications are made to the callback functions to add detector functionality to parallel adaptive simulations in Fluidity:

1. During field data sizes callback the pre-processing of the detector lists is carried out. The callback also ensures enough memory is available to pack all the detector information along with the element's field data using the count of detectors in each element calculated during the pre-processing.

2. During the field data packing callback, for each element loop over the number of detectors in the element (as calculated in the pre-processing) and remove each from the send detector list and pack them into the communications buffer.

3. During the field data unpacking callback for each element read the number of detectors being sent with it. Loop over the number of detectors transferred with the element checking to see if they are owned by this process in the new partitioning. If the process now owns the detector allocate a new detector in the unpacked detector list and unpack the detector data into it.

Unpacking the detectors within the unpacking of fields data callback also makes use of a temporary detector list, the unpacked detector list. This is so after the migrate the detectors which were not transferred can be updated in the normal lists before merging in the detectors that were transferred. Elements may be transferred to more than one process in the migrate phase. This is because it is not just owned elements data which is being transferred but halo elements data as well. When unpacking detectors each process checks the owner of the element containing the detector and only unpacks to the unpacked detectors list if it is the element owner in the new partitioning.

As was briefly mentioned above once the call to `Zoltan_Migrate` has been made and the field data has been transferred the detector lists containing detectors which were not transferred must be updated. Local element numbers change between the old and new partitioning and each detector is associated with a particular local element number so this value must be updated. This is done using two mappings:

- Between old local element number and the universal element number created from the old mesh before any data migration.
- Between universal element number and new local element number created from the new mesh after the node migrations and construction of the new mesh.

Each detector has the element number of the element that owns it first mapped the the universal element number and then from the universal element number to a new local element number. The new local element number is stored in the detector.

An additional complication when migrating detectors can be found at this point. A corner case exists where a detector does not get flagged as being in an element that is being transferred during the pre-processing of the detector lists, but it is also in an element which will not be owned (or even known about) on the current owning process in the new partitioning. In this case the detector is broadcast to all other processors until one accepts it as being within an element that processor owns.

Finally once the detector lists have been updated those detectors received when migrating the fields data are merged in from the unpacked detector list. Each detector in the unpacked detector list was transferred with the element owner set to the elements universal element number. The universal element number is mapped to the new local element number, the detectors element owner is updated and the detector is moved from the unpacked detectors list to the appropriate detector list.

3.5. Flredcomp

The Zoltan solution has also been used for the tool, `flredecomp`, provided with Fluidity. This tool is for redecomposing an input checkpoint to a new checkpoint using more or less processes. As the number of elements varies throughout an adaptive simulation, it is possible that the number of element exceeds the capacity of memory. In this case the simulation can be checkpointed, redecomposed onto a greater number of partitions and restarted. The tool is also useful for setting up strong scaling simulations.

When using Zoltan in `flredecomp` the same code as called from adaptivity is used but with the logical `flredcomping` set to true. This changes various options within `zoltan_drive`. The partitioning is carried out with uniform edge-weighting applied as we are only doing a single iteration with the goal being a load balanced final partitioning.

When called from `flredecomp` the implementation must use `Zoltan_LB_Partition` instead of `Zoltan_LB_Balance` as the Zoltan parameters `NUM_GLOBAL_PARTS` and `NUM_LOCAL_PARTS` are only used when using `Zoltan_LB_Partition`. Each of these must be set by every process with `NUM_GLOBAL_PARTS` the same for all processes but `NUM_LOCAL_PARTS` independent for each process. `NUM_GLOBAL_PARTS` tells Zoltan the total number of partitions to be created and `NUM_LOCAL_PARTS` tells Zoltan the number of partitions to be placed on the specific process. `Zoltan_LB_Balance` is in essence a specialised version of `Zoltan_LB_Partition` where `NUM_GLOBAL_PARTS` is the number of processes being run on and `NUM_LOCAL_PARTS` is one for all processes. For `flredecomp` the `NUM_GLOBAL_PARTS` is the number of processes the checkpoint should be decomposed onto. When scaling up `NUM_LOCAL_PARTS` is one for all processes

and when scaling down is one for all processes up the target number of processes and zero for all others.

4. Profiling

A single benchmark case was tested which represent a common use-case for Fluidity simulations. This is a backward-facing step CFD example (Fig. 4) on a piecewise-linear continuous Galerkin discretisation (P1P1). This discretisation is a common choice for CFD applications and also allows comparisons to Sam as a baseline. Other benchmark cases were either not adaptive, or if made adaptive, could not be performed using Sam.

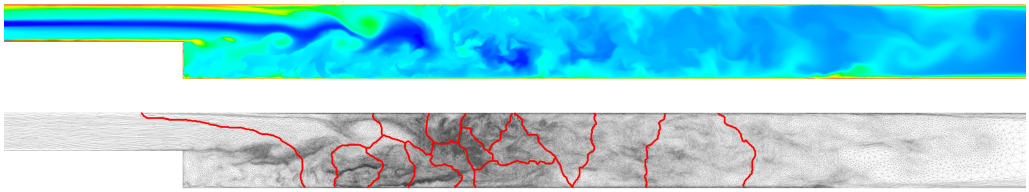

Figure 4: *Snapshot of a tracer field from the backward facing step benchmark (top) and the adaptive mesh (below) with domain partition boundaries shown in red.*

It was unfortunately not possible to use ParMETIS through Zoltan on HECToR for any of the profiling runs. This had previously been possible on HECToR but at the time the profiling runs were conducted there appeared to be a bug in Zoltan 3.5. Runs using ParMETIS through Zoltan failed due to requiring an extra callback which according to the Zoltan documentation should not be needed. This issue has been raised with the Zoltan developers. The code has previously been run on HECToR and other systems using ParMETIS through Zoltan 3.4 so this issue should be resolved in the near future.

It was also not possible to conduct profiling runs on HECToR which used PT-Scotch through Zoltan on more than 16 processes. This appears to be a problem with PT-Scotch which gives an MPI error when running on more than one core of the HECToR system. This issue has been raised as a bug on HECToR system.

The profiling results presented below are from runs on the HECToR Phase 3 system. The Phase 3 system is a Cray XE6 system, offering a total of 2816 XE6 compute nodes. Each compute node contains two AMD 2.3 GHz 16-core processors giving a total of 90,112 cores – a theoretical peak performance of over 800 Tflops. There is presently 32 GB of main memory available per node, which is shared between its thirty-two cores, the total memory is 90 TB. The processors are connected with a high-bandwidth interconnect using Cray Gemini communication chips. The Gemini chips are arranged on a 3 dimensional torus. Further profiling work on the Phase 3 system of the other test cases was not possible due to time constraints.

Figures 5 through 10 above show the profiling results for the small backward-facing step benchmark run on HECToR. This is a small benchmark which uses approximately 23,000 elements that is typically run on 32-64 processes. Here it has been run on up to 256 processes to test the different partitioners. In addition, dynamic adaptivity of the mesh was performed every two timesteps, rather than the usual 10 to 20. As such, the adaptive algorithm dominates the runtime of these benchmarks. In most problems this would not be the case and instead the assemble and solve phases would dominate the total runtime.

Figure 5 show the performance of the Zoltan implementation matching the Sam performance up to 8 processes. At >8 processes the overall runtime for Zoltan becomes greater than that for Sam.

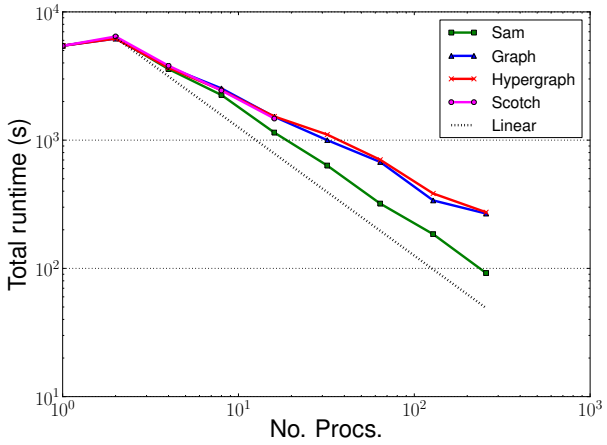

Figure 5: *Total run time for small backward facing step benchmark.*

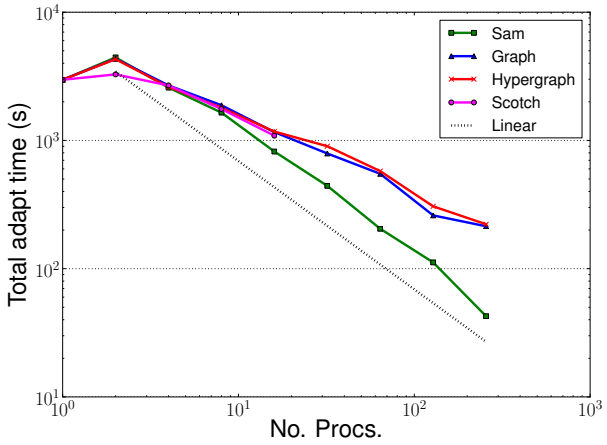

Figure 6: *Total adapt time for small backward facing step benchmark. Note that the increase in total adapt time mirrors the total run time (Fig. 5)*

This is due to the increased adapt time when using Zoltan (Fig. 6). It was expected that the time to complete the adapt phase in Zoltan may be longer than in Sam due to Zoltan being a general purpose solution. Sam only works for particular element types which allow the field data to be migrated at the same time as the nodes. The Zoltan solution requires an extra migration phase to migrate the fields data separately. Figure 10 show that the extra migration only impacts the Zoltan performance when the partitioner is being worked heavily, as happens when you have a small problem being divided up on to too many processes. The same increase is not seen in the serial adapt times (Fig. 9).

The hope was that the access to different partitioners, particularly the hypergraph partitioner, would improve the load balance for the assemble and solve phases and these sections would hence see a performance improvement. As can be seen from Figures 7 and 8 there has been no performance improvement in the assemble or solve phases.

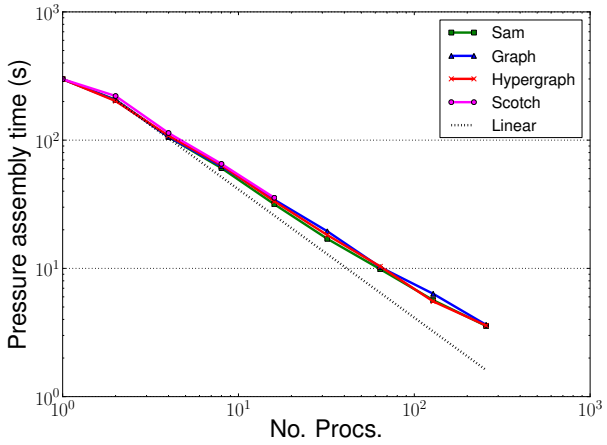

Figure 7: *Pressure assembly time for small backward facing step benchmark.*

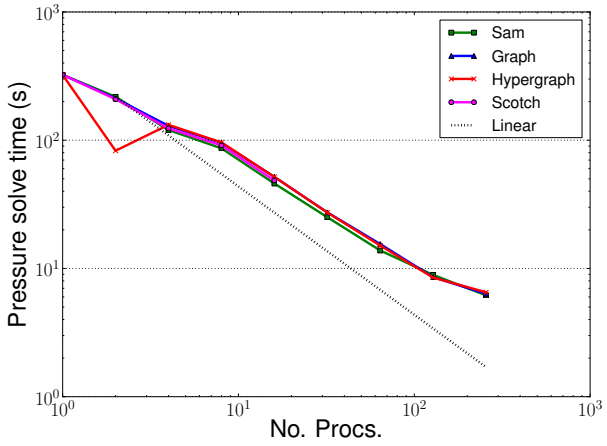

Figure 8: *Pressure solve time for small backward facing step benchmark.*

5. Outcome

The Zoltan based mesh re-partitioning solution was originally incorporated as a compile time option accessed using the flag `--with-zoltan` when configuring Fluidity. At this stage all but the short report on the Zoltan implementation and unit tests from M2 were complete. It was decided that detailing the implementation in a short report would not be possible. Unit tests for Zoltan are complicated as the functionality needs to run on full simulations to be tested. As such it was decided to not add unit tests but to add tests to the Fluidity test suite which compared the output from an adaptive run using Zoltan with the output from both a run with the previous implementation and a serial run. This is test `2d_circle_adapt_zoltan_sam` in the Fluidity test suite. To test the functionality of Zoltan on element types which could not be done by Sam a number of additional test were also added, such as `parallel_p0_consistent_interpolation` and `diffusion_2d_p0_adaptive_parallel`.

From Fluidity revision 3530, Zoltan has been the default solution for the Fluidity trunk, development branches and releases. All tests run as part of the Fluidity buildbot system are using the Zoltan build and run successfully with Zoltan. The previous solution, Sam, is now a compile time option accessed by configuring with the flag `--with-sam`. All options for Zoltan are fully documented and available through the Fluidity options package, diamond, with full details also given

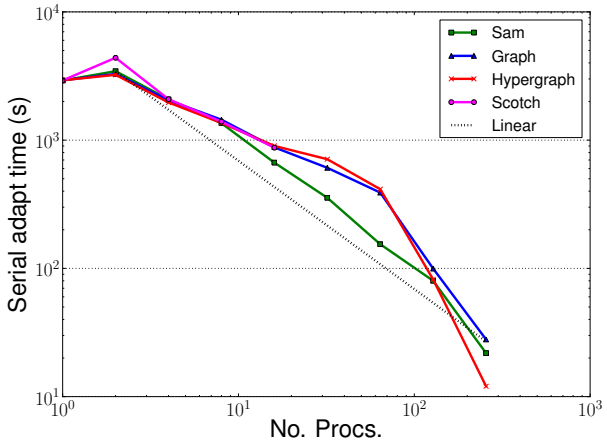

Figure 9: *Serial adapt time for small backward facing step benchmark.*

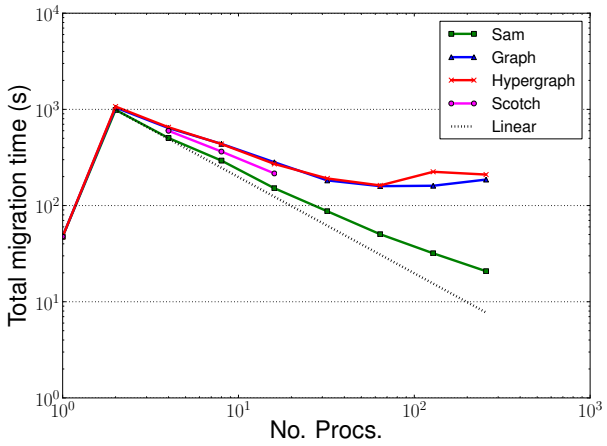

Figure 10: *Total migration time for small backward facing step benchmark.*

in the Fluidity manual. This meets all of the work in M3.

Section 4 shows the performance of the Zoltan implementation on one of the HPC benchmark cases: the backward facing step. Zoltan resulted in a drop in performance from 16 processors onwards, though at this number of elements per core and with the frequency of adapts it is not surprising that the performance difference between Zoltan and Sam becomes more pronounced. The expected performance increase in both the assembly and solve was not evident in this test case. However, other tests may show better performance, however at this stage it is not clear which category of simulation would show better performance and there is a limited number of test cases that can be used as a comparison between Zoltan and Sam.

M1 was to become familiar with the Fluidity source. This was achieved and the completion of the subsequent work packages shows this. The short report was not written as it was felt it was more important to continue with the other work. Initial profiling work was done but only for a single benchmark, the small backward facing step. This was because profiling work was completed as preparation for this dCSE and presented in the original dCSE proposal.

This work as well as aiming to improve the scaling performance of Fluidity was to improve the maintainability, extensibility and functionality of the repartitioning solution. The new Zoltan based implementation has already been extended to allow parallel periodic problems to be solved

by Fluidity. Detectors have also been implemented allowing them to be used within all parallel adaptive simulations as set out as the last element of WP3. The greatest benefit of the Zoltan implementation though is that it is general purpose. It allows any element type to be used in parallel adaptive simulations. This will enable Fluidity to be used for new science not previously possible. For example, modelling froths and foams using Fluidity benefits from a P0P1 discretisation and an adaptive mesh (Brito-Parada et al., 2011). Limiting this to serial simulations only is prohibitive and therefore the inclusion of Zoltan is enabling new science to be carried out.

Attendance and participation in the HPC workshops run by the Fluidity group was not possible as none were held during the course of this dCSE project. However details of the work were presented at the HECToR dCSE Technical Workshop on 4th and 5th October, 2011 and the slides from this presentation are available `http://www.hector.ac.uk/cse/distributedcse/technical2/`.

Zoltan has been installed as a module on HECToR available to all HECToR users. All partitioners (ParMETIS, PT-Scotch and Zoltan graph/hypergraph) are available when using this Zoltan module. A centrally installed copy of Fluidity using Zoltan is also available and details have been added to the Fluidity webpages detailing how to compile Fluidity on HECToR.

During the course of this project several times software or hardware upgrades to HECToR caused Fluidity to either fail to compile or fail to run. This often caused significant delays and it was decided to setup a build test. This is a serial job run once a day which attempts to checkout, configure and compile Fluidity with the results transferred to AMCG systems. These data are then automatically processed and used to update the status of the HECToR build on the Fluidity buildbot system. This means that compilation failures of Fluidity on HECToR can be noted earlier and fixed quicker. Despite being outside the original work plan this work was essential and very beneficial.

Fluidity is an open source project so this project as well as benefiting all Fluidity users is of wider benefit to all those interested in dynamic load balancing and adaptive mesh methods. The inclusion of Zoltan, whilst not increasing performance as hoped, has vastly improved the maintainability of Fluidity and enabled new areas of science to exploit the adaptive remeshing capabilities of Fluidity in parallel using all element types.

6. Acknowledgement

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

Babuška, I., Suri, M., 1994. The p and h-p versions of the Finite Element method, basic principles and properties. SIAM Review 36, 578.

Berger, M.J., Colella, P., 1989. Local adaptive mesh refinement for shock hydrodynamics. Journal of Computational Physics 82, 64–84.

Boman, E., Devine, K., Fisk, L.A., Heaphy, R., Hendrickson, B., Vaughan, C., Catalyurek, U., Bozdag, D., Mitchell, W., Teresco, J., 2007. Zoltan 3.0: Parallel Partitioning, Load-balancing, and Data Management Services; User's Guide. Sandia National Laboratories. Albuquerque, NM.

Brito-Parada, P., Neethling, S., Cilliers, J., 2011. The advantages of using mesh adaptivity when modelling the drainage of liquid in froths. Minerals Engineering , Corrected proofs.

Budd, C.J., Huang, W., Russell, R.D., 2009. Adaptivity with moving grids. Acta Numerica 18, 111–241.

Catalyurek, U.V., Boman, E.G., Devine, K.D., Bozda, D., Heaphy, R.T., Riesen, L.A., 2009. A repartitioning hypergraph model for dynamic load balancing. Journal of Parallel and Distributed Computing 69, 711–724.

Davies, D.R., Wilson, C.R., Kramer, S.C., 2011. Fluidity: A fully unstructured anisotropic adaptive mesh computational modeling framework for geodynamics. Geochem. Geophys. Geosyst. 12.

Devine, K., Boman, E., Heaphy, R., Hendrickson, B., Vaughan, C., 2002. Zoltan data management services for parallel dynamic applications. Computing in Science and Engineering 4, 90–97.

Farrell, P., 2009. Galerkin Projection Of Discrete Fields Via Supermesh Construction. Ph.D. thesis. Imperial College London.

Farrell, P., 2011. The addition of fields on different meshes. Journal of Computational Physics 230, 3265 – 3269.

Farrell, P.E., Piggott, M.D., Pain, C.C., Gorman, G.J., Wilson, C.R., 2009. Conservative interpolation between unstructured meshes via supermesh construction. Computer Methods in Applied Mechanics and Engineering 198, 2632–2642.

Gorman, G., 2006. Parallel Anisotropic Unstructured Mesh Optimisation And Its Applications. Ph.D. thesis. Imperial College London.

Ham, D.A., Farrell, P.E., Gorman, G.J., Maddison, J.R., Wilson, C.R., Kramer, S.C., Shipton, J., Collins, G.S., Cotter, C.J., Piggott, M.D., 2009. Spud 1.0: generalising and automating the user interfaces of scientific computer models. Geoscientific Model Development 2, 33–42.

Hiester, H.R., Piggott, M.D., Allison, P.A., 2011. The impact of mesh adaptivity on the gravity current front speed in a two–dimensional lock–exchange. Ocean Modelling 38, 1–21.

Houston, P., Süli, E., 2001. *hp*-Adaptive discontinuous Galerkin finite element methods for first-order hyperbolic problems. SIAM Journal on Scientific Computing 23, 1226–1252.

Ledger, P.D., Morgan, K., Peraire, J., Hassan, O., Weatherill, N.P., 2003. The development of an *hp*-adaptive finite element procedure for electromagnetic scattering problems. Finite Elements in Analysis and Design 39, 751–764.

Mitchell, A.J., Allison, P.A., Piggott, M.D., Gorman, G.J., Pain, C.C., Hampson, G.J., 2010. Numerical modelling of tsunami propagation with implications for sedimentation in ancient epicontinental seas: The lower jurassic laurasian seaway. Sedimentary Geology 228, 81 – 97.

Pain, C., Piggott, M., Goddard, A., Fang, F., Gorman, G., Marshall, D., Eaton, M., Power, P., de Oliveira, C., 2005. Three-dimensional unstructured mesh ocean modelling. Ocean Modelling 10, 5–33.

Pain, C.C., Umpleby, A.P., de Oliveira, C.R.E., Goddard, A.J.H., 2001. Tetrahedral mesh optimisation and adaptivity for steady-state and transient finite element calculations. Computer Methods in Applied Mechanics and Engineering 190, 3771–3796.

Piggott, M.D., Farrell, P.E., Wilson, C.R., Gorman, G.J., Pain, C.C., 2009. Anisotropic mesh adaptivity for multi-scale ocean modelling. Philosophical Transactions of the Royal Society A 367, 4591–4611.