4 Technical Report (publishable)

4.1 Abstract

In this project a standalone general-purpose numerical library, named libsupermesh, was created. This library significantly simplifies the process of writing new models, and adding new functionality to existing models, which make use of two different unstructured meshes. The library enables parallel supermeshing calculations which involve two different unstructured meshes with non-matching domain decompositions.

The software is available under the LGPL 2.1 license in public repositories [1], and its use is due for imminent merge into the Fluidity [2] project codebase.

4.2 Introduction

The unstructured finite element model Fluidity is capable of numerically solving the Navier-Stokes equations and accompanying field equations on arbitrary unstructured finite element meshes in one, two and three dimensions [2].

Models which use multiple non-matching unstructured meshes generally need to solve a computational geometry problem, and construct intersection meshes in a process known as supermeshing [3-4]. The algorithm for solving this problem is known [4-6] and has an existing implementation in the unstructured finite element model Fluidity, but this implementation is deeply embedded within the code and unavailable for widespread use. This project addresses these issues via the creation of a standalone general-purpose numerical library, libsupermesh, which can be easily integrated into new and existing numerical models.

Furthermore, for parallel calculations, the existing implementation in Fluidity assumes that the meshes used have domain decompositions which match perfectly. This limits access to this powerful numerical technique, and limits the scope of multimesh modelling applications. This project addresses the general problem whereby a numerical model may need to consider not only two non-matching unstructured meshes, but also allow the two meshes to have different parallel partitionings.

4.3 Construction of a general purpose supermeshing library (WP1)

Fluidity implements the local supermeshing algorithm described in [4] (see also [5-6]). This implementation includes an efficient advancing front based spatial searching algorithm, and a more broadly applicable R*-tree spatial searching algorithm using libspatialindex [7]. Element intersection is performed using code from the Wild Magic computer graphics engine, or in 3D using a highly optimised custom tetrahedron intersection code based on the “plane-at-a-time” clipping algorithm described in [8]. Additional intersection functionality is provided via integration with the CGAL computational geometry library [9]. The implementation is tightly integrated with Fluidity core data structures, which are

in general unsuitable for use with external codes. For example Fluidity mesh data structures include detailed information regarding discrete function space and numerical integration rules, which is inappropriate for external codes which support a different set of function spaces.

Implementation

The supermeshing code was extracted from Fluidity and placed in libsupermesh; a standalone open source library. The CGAL element intersection code is a largely unused experimental feature and was not transferred to libsupermesh. The Wild Magic code was replaced with polygon intersection via an implementation of the Sutherland-Hodgman clipping algorithm [10]. Fluidity data structures were removed and procedures were updated to use straightforward interfaces accepting fundamental data types as arguments. The interfaces accept arbitrary simplices as input, and return a local simplex supermesh. Auxiliary routines for intersection of simplices and convex cubical elements are provided. Further routines for the intersection of convex polygons, and for the intersection of tetrahedra with arbitrary half-spaces, are provided. Additional functions are provided for the division of convex hexahedra, square pyramids, triangular prisms, and convex polygons into simplices.

The existing advancing front algorithm intersection identification algorithm was re-written for efficiency. New quadtree and octree intersection identification code was written for 2D and 3D meshes respectively and are generally found to compete in terms of efficiency with (and are sometimes faster than) the advancing front intersection finder algorithm. The advancing front intersection finder requires certain properties of the meshes considered [5]. The new quadtree and octree intersection finders do not require this structure, and hence may be of use in more general problems. Fluidity provides an R*-tree intersection finder which uses libspatialindex [7]. The interface to libspatialindex was significantly updated. In particular it was identified that the implementation in Fluidity led to the heavy use of disk caching for larger problems. The implementation in libsupermesh was modified to use only memory caching.

Fluidity was modified to use the new libsupermesh library. This required minimal changes to Fluidity code.

Regression testing

A suite of regression tests for the libsupermesh library was created. The test suite performs several tests in order to validate that the library can correctly identify possible element intersections and create the local supermesh.

Fluidity was also part of the testing strategy of this project. Since Fluidity was modified to use the new libsupermesh library it was possible to use the Fluidity regression tests. The Fluidity project provides a large number of such tests (although only a subset use the supermeshing functionality). All Fluidity test cases were run with and without the libsupermesh library, and it was verified that all output was identical (to within expected small numerical errors).

Valgrind was used to test libsupermesh for memory leaks, and the code was compiled with the following GNU GFortran options enabled to verify correctness: "-O0 -g -Wall -fcheck=all -ffpe-trap=invalid,zero,overflow,underflow -finit-integer=-66666 -finit-real=nan -fimplicit-none"

Serial performance

Several benchmark calculations were performed in order to measure the performance of the new intersection finder and element intersector algorithms.

The 2D benchmarks take as input two quasi-uniform resolution unstructured triangle meshes of an equilateral triangle domain, A and B, with mesh B having roughly one half the element size of mesh A. The meshes were generated using Gmsh [11]. The resolution of the input meshes was increased until mesh B had 7.6 million elements. Figure 1 shows the runtime of the various 2D intersection identification algorithms and element intersectors, with measurements from Fluidity for comparison.

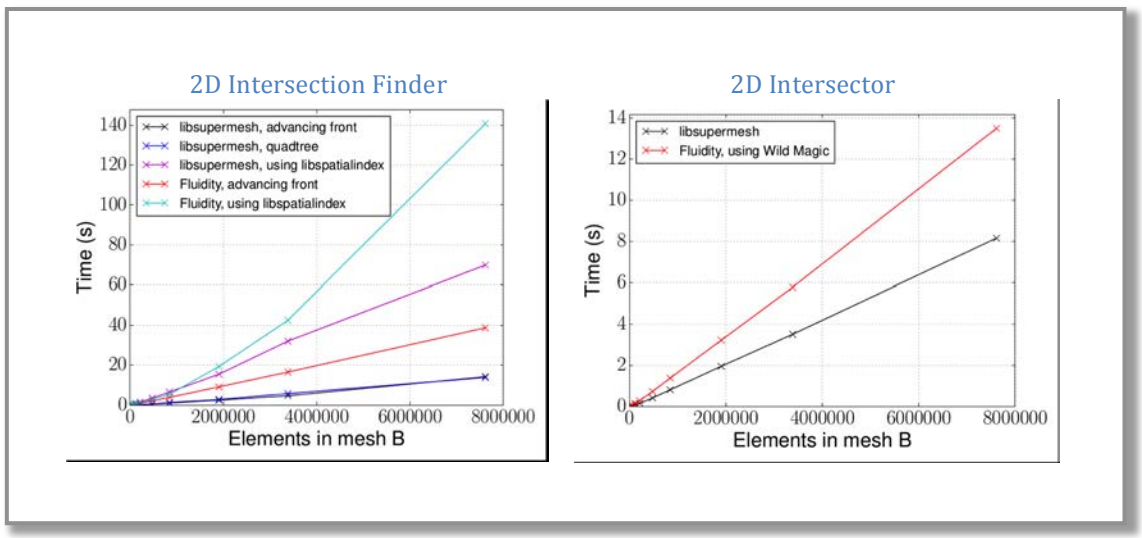

Figure 1 2D serial performance benchmark

For the R*-tree intersection finder the runtime in the largest case was reduced from 141 seconds with Fluidity, to 70 seconds with libsupermesh. The performance improvement is attributed to the switch from disk to memory caching for the larger cases considered. For the advancing front intersection finder the runtime in the largest case was reduced from 39 seconds with Fluidity, to 14 seconds with libsupermesh.

For the largest case libsupermesh took 8.2 seconds to intersect and calculate the area of the 2D supermesh; Fluidity took 13.5 seconds.

The 3D benchmarks take as input two quasi-uniform resolution unstructured tetrahedra meshes of a square pyramid domain, A and B, with mesh B having roughly one half the element size of mesh A. The meshes were generated using Gmsh [11]. The resolution of the input meshes was increased until mesh B had 4.1 million elements. Figure 2 shows the runtime of the various 3D intersection

identification algorithms and element intersectors, with measurements from Fluidity for comparison.

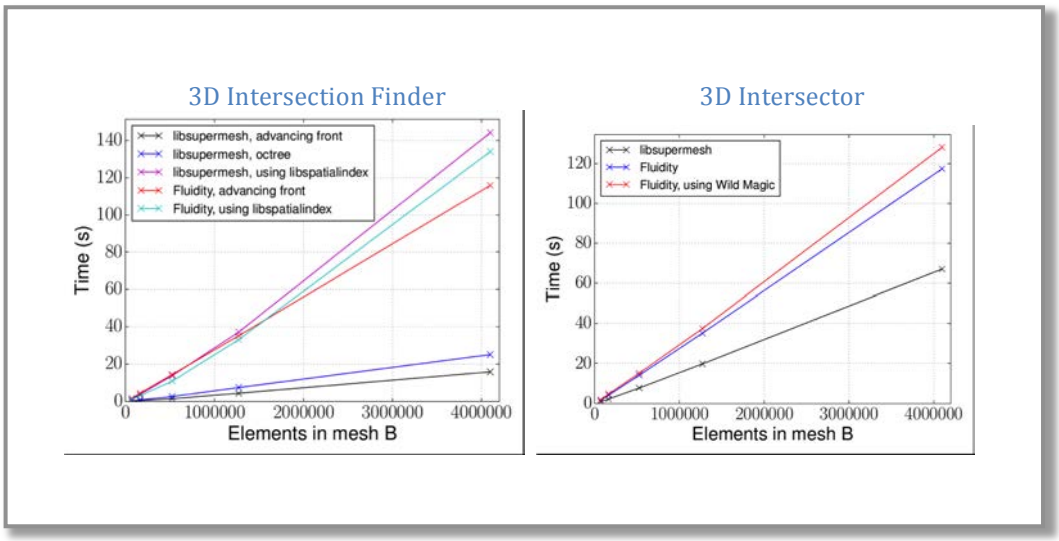

Figure 2 3D serial performance benchmark

The results show that when using the R*-tree spatial searching algorithm from libspatialindex the runtime of libspatialindex is not improved as compared with the performance of Fluidity. Since Fluidity and libsupermesh both use versions of the libspatialindex library for this, if disk versus memory caching is not an issue (for example if disk caching is not triggered in the Fluidity implementation) then no significant difference in performance is expected. For the advancing front intersection finder the runtime in the largest case is reduced from 116 seconds with Fluidity, to 16 seconds with libsupermesh.

libsupermesh took 67 seconds to intersect and calculate the volume of the 3D supermesh, whereas Fluidity took 117 seconds.

libsupermesh uses simple interfaces, whereas Fluidity uses complex internal data structures. The performance results measure only the runtime of the intersection finders and intersector functionality and disregard all other actions, including input and initialisation. Measuring the additional cost of allocating and deallocating Fluidity data structures would lead to further reduced Fluidity performance as measured in these benchmarks.

These serial benchmarks were performed using Fluidity 4.1.13 and libsupermesh version 1.0.0. The presented results are the mean of three measurements (from four runs with the first discarded in each case) performed on a system with 4 Intel i5-3470 CPUs and 16 GiB of RAM.

4.4 General parallelisation (WP2)

The second main objective of the project was to implement a parallel algorithm for the case where a numerical model may consider two non-matching unstructured meshes which have different parallel partitionings. In order to handle these cases the parallel Fluidity implementation assumes that the meshes

used have domain decompositions which match perfectly. While this property may be satisfied by parallel interpolation in adaptive mesh Fluidity calculations, it cannot be expected to hold in general.

Implementation

Constructing a local supermesh involves three key steps:

1. Identification of pairs of elements, one on each mesh, which intersect;
2. Generation of a mesh of their intersection (the "local supermesh");
3. The transfer of data onto this intersection mesh.

This algorithm directly parallelises, but only if the decompositions of the two meshes are perfectly aligned.

In the following the local mesh A refers to the local submesh (or partition) of mesh A stored on the current MPI process. Similarly the local mesh B refers to the local submesh (or partition) of mesh B stored on the current MPI process. The received mesh B refers to a submesh of mesh B received from a different MPI process.

The interface for the algorithm is very general, and the specific use case of interest is defined via three user provided callback procedures. The callback procedures will be discussed in sections 0, 0 and 0.

In order to generalise the algorithm to perform parallel supermesh calculations on meshes that are not perfectly aligned, the following algorithm was implemented:

1. Communicate the axis-aligned bounding boxes (AABBs) of all mesh A partitions and all mesh B partitions using all-to-all communication;
2. For each mesh A partition whose AABB intersects with the local mesh B AABB:
    a. Identify local mesh B elements whose AABBs intersect with the AABB of the mesh A partition;
    b. Obtain data associated with these elements, and communicate these data via point-to-point communication.
3. Construct the intersection meshes for local mesh A and local mesh B elements, and perform calculations on these intersection meshes;
4. For each mesh B partition whose AABB intersects with the local mesh A AABB:
    a. Unpack data communicated in step 2b;
    b. Construct the intersection meshes for local mesh A and received mesh B elements, and perform calculations on these intersection meshes.

4.4.1.1 Step 1

The initial step of the algorithm calculates the axis-aligned bounding box (AABB) of the local meshes (A and B). Once the local AABBs have been calculated, the library uses MPI all-to-all communication to distribute the bounding boxes across the whole domain. After this step all remaining communication is point-to-point. Once step 1 is complete all MPI processes know the bounding boxes of all mesh partitions.

4.4.1.2 Step 2

Each process runs a test on the bounding boxes of each mesh A partition, communicated in step 1, with the local mesh B partition. If the bounding boxes intersect then some of the local mesh B elements may intersect with some mesh A elements on a different process. At this point one approach would be to communicate all local mesh B data to the MPI process which holds the relevant mesh A partition. However, step 2a optimises this process and reduces the amount of data that need be communicated.

Similarly each process runs a test on the bounding boxes of each mesh B partition, communicated in step 1, with the local mesh A partition. If the bounding boxes intersect then some of the local mesh A elements may intersect with some mesh B elements stored on a different process. It is noted that some data will be received from this process.

4.4.1.3 Step 2a

In this step each local mesh B element bounding box is tested against the bounding box of the mesh A partition identified in step 2. The local mesh B elements which intersect with the bounding box of the mesh A partition are marked for sending. At this point it is not certain that the local mesh B elements actually intersect with any mesh A partition elements of the other process. However, this step nevertheless reduces the amount of data packing and MPI point-to-point communication required in the following steps.

4.4.1.4 Step 2b

In this step the local mesh B elements to be communicated to remote processes are known. A user provided callback function is used to create a packed array containing necessary user data. libsupermesh is oblivious to the amount of data and the associations between elements and data values. The user is responsible for providing a procedure which will return an array of data values based on the local mesh B vertices and elements which are tagged for sending. Once the data have been packed, libsupermesh creates a packed MPI message containing additional meta-data. The packed MPI message has the following format:

1. Number of elements (MPI_INTEGER);
2. Number of mesh vertices (MPI_INTEGER);
3. Connectivity of mesh vertices (flat array of MPI_INTEGER);
4. Coordinates of mesh vertices (flat array of MPI_DOUBLE_PRECISION);
5. Size of user supplied data (MPI_INTEGER);
6. User supplied data (MPI_BYTE).

The packed MPI message is sent to the relevant MPI process using point-to-point communication.

If during 4.4.1.3 Step 2a no suitable local mesh B elements are identified, an MPI message with the following format is sent:

1. Number of elements (equal to 0).

The receiver will not be aware that no mesh B elements actually intersected with the local mesh A AABB. Thus the receiver will think that a message is pending

and it could dead-lock. Sending an MPI message which will indicate that no elements intersect avoids this.

4.4.1.5 Step 3

This step handles the case where elements in the local mesh A and local mesh B intersect. If there is a local intersection, then the intersection meshes can be constructed and all relevant calculations can be performed using a user provided callback function (see step 4b).

4.4.1.6 Step 4

The receiver knows if the local mesh A AABB intersects with the mesh B partition. Thus, it is known whether the process will receive data from a remote MPI process. Furthermore, it is known which MPI process will send the message.

If the bounding boxes intersect, an MPI Probe call is initiated. An MPI Probe must be used because the size of the incoming MPI message is not known.

4.4.1.7 Step 4a

In this step an MPI packed message containing the remote mesh B partition data is received. The format of the message is the same as in 4.4.1.4 Step 2b. If the first MPI_INTEGER is equal to 0, then the message is discarded and remaining probing is performed.

If an MPI message is received where the first MPI_INTEGER is not equal to 0, the message is unpacked using MPI_UNPACK. Memory is allocated to hold the communicated user provided data, which is itself unpacked by a user specified callback function. The unpack user data is stored in user controlled memory space.

4.4.1.8 Step 4b

In the final step intersection meshes for the communicated mesh B elements and the local mesh A elements are constructed. The candidate intersection identification is performed using the libspatialindex R*-tree algorithm. The efficient advancing front based spatial searching algorithm cannot be used as it cannot be guaranteed that the considered meshes satisfy the properties required by this algorithm. Furthermore, the quadtree (2D) and octree (3D) algorithms were not used because they were completed late in the project. libsupermesh is unaware of the type of calculations that the user wants to perform. A callback function performs the calculations and, if required, stores the results in user controlled memory space.

4.4.1.9 pack_data_b callback procedure

This procedure is provided by the user and is called in 4.4.1.4 Step 2b (section 0). The procedure takes as input two integer arrays which correspond to the mesh B vertices and elements that will be communicated. The output of this procedure is a one dimensional array with a contiguous memory region which includes all data which will later be unpacked by the unpack_data_b procedure (see section 0).

4.4.1.10 unpack_data_b callback procedure

This procedure is provided by the user and is called in 4.4.1.7 Step 4a (section 0). This procedure takes as input two integers which correspond to the number of communicated mesh B vertices and element, and a one dimensional array with a contiguous memory region which includes mesh B user data, previously packed on a different process. Unpacked data is stored in user controlled memory space.

4.4.1.11 intersection_calculation callback procedure

This procedure is provided by the user and is called in 4.4.1.5 Step 3 (section 0) and 4.4.1.8 Step 4b (section 0). This takes as input several arguments:

- Element A vertex coordinates.
- Element B vertex coordinates.
- Intersection mesh C vertex coordinates.
- Mesh B vertices associated with element B in the local or received mesh B
- Index of element A in the local mesh A.
- Index of element B in the local or received mesh B
- Local (true or false), whether 4.4.1.5 Step 3 (section 0) [if true] or 4.4.1.8 Step 4b (section 0) [if false] is being performed.

The callback procedure performs calculations on each intersection mesh, storing the results in user controlled memory space.

Example

In this example two meshes (A and B) are going to be used. The elements of each mesh have been painted according to the MPI process that owns them. The blue elements are owned by MPI process 1, the green elements by MPI process 2, the pink elements by MPI process 3 and the red elements by MPI process 4. The full meshes are shown in Figure 3:

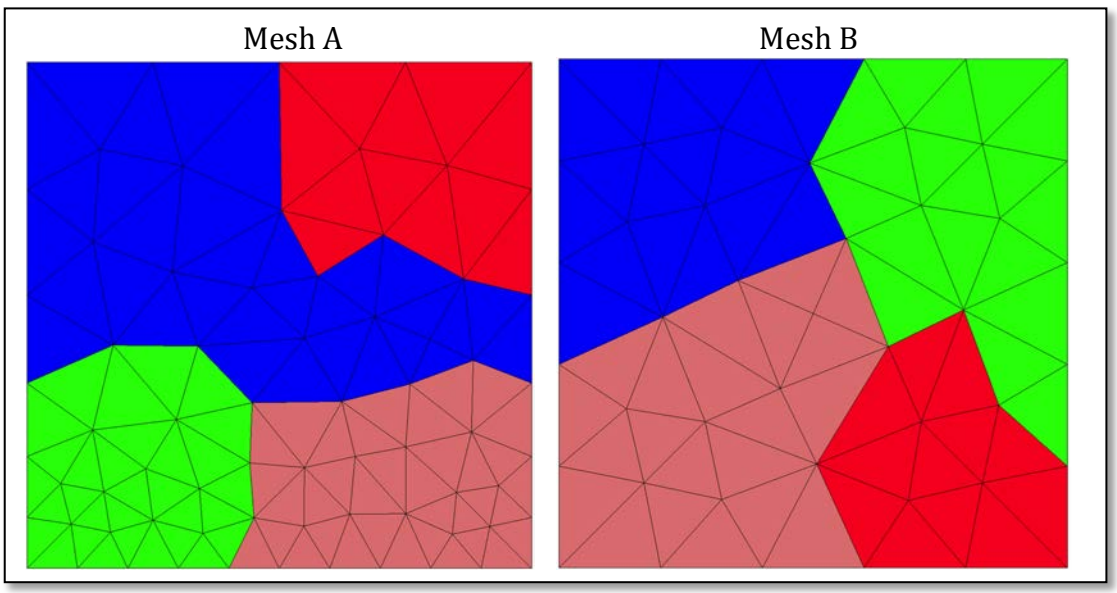

Figure 3 Complete Meshes

Each MPI process has an incomplete picture of the two meshes. For example, Figure 4 shows the view of both meshes by MPI process 1:

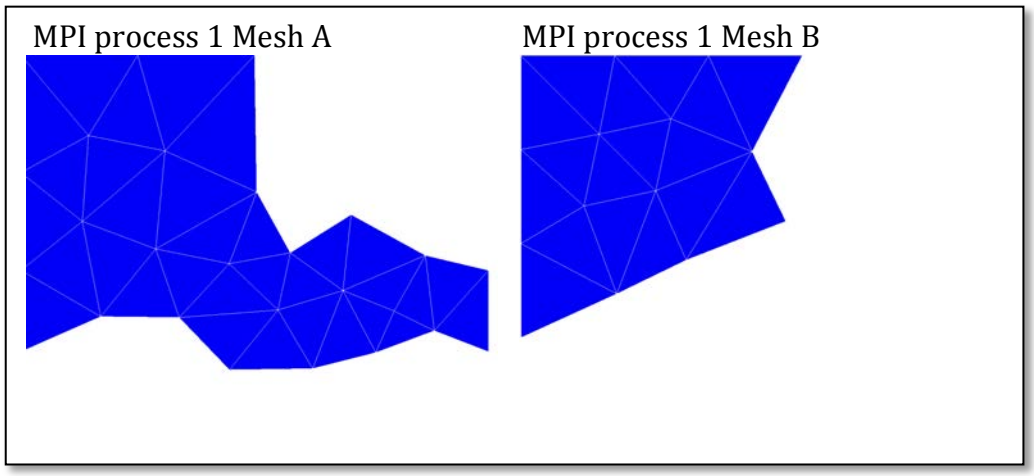

Figure 4 MPI process 1 View

For this example the local MPI process is MPI process 1 and the remote MPI processes are MPI process 2, 3 and 4.

The first step of the algorithm is to compute and communicate the axis-aligned bounding boxes (AABBs) of all mesh A partitions and all mesh B partitions using all-to-all communication. Figure 5 shows the view of each MPI process after computing the AABBs.

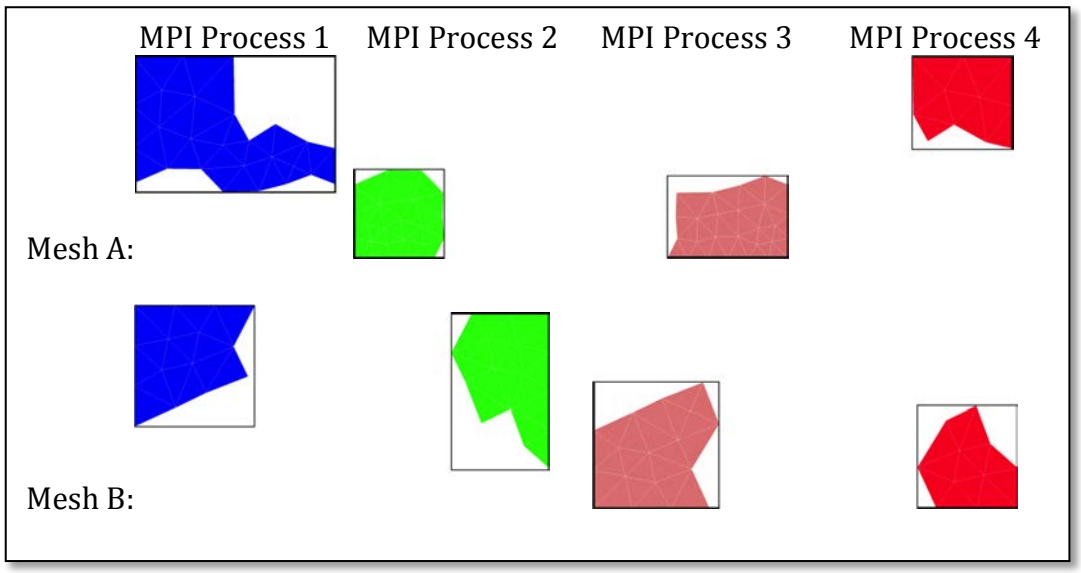

Figure 5 View of each MPI process

The next step is to communicate the AABBs using all-to-all communications. Figure 6 shows the view of MPI process 1 (local) after the all-to-all communication; whereas, Figure 7 shows the view of the remote MPI processes.

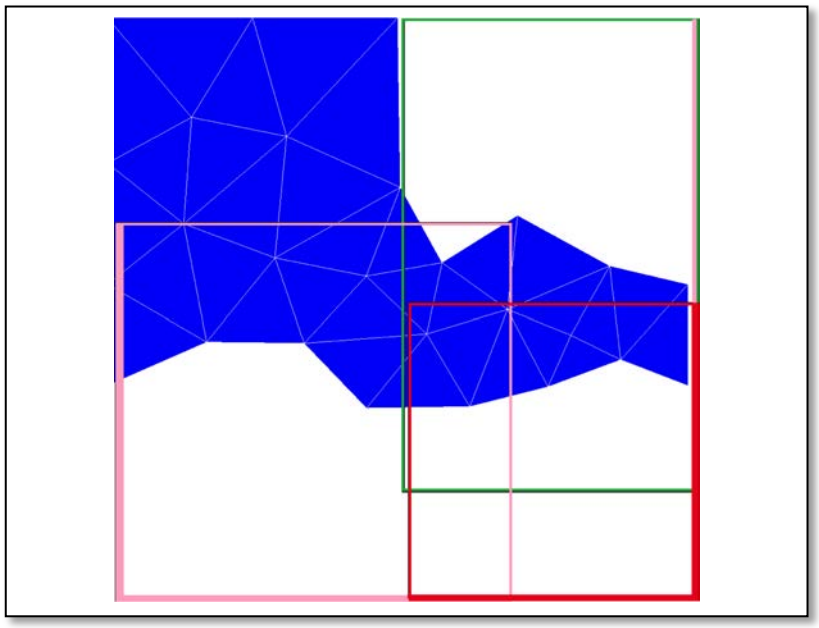

Figure 6 View of local process after all-to-all

Once MPI process 1 receives the elements and data from the remote processes it can construct the intersection meshes for local mesh A and received mesh B elements. Some of the received mesh B elements will not intersect (for example most of MPI process 2 mesh B elements do not intersect). However, at the end of this step MPI process 1 will have a complete view of the received mesh B elements that intersect with the local mesh A elements. Furthermore, all data associated with the remote mesh B elements have been communicated to MPI process 1. Thus MPI process 1 can not only construct the intersection meshes; MPI process 1 can also use the intersection mesh and perform calculations.

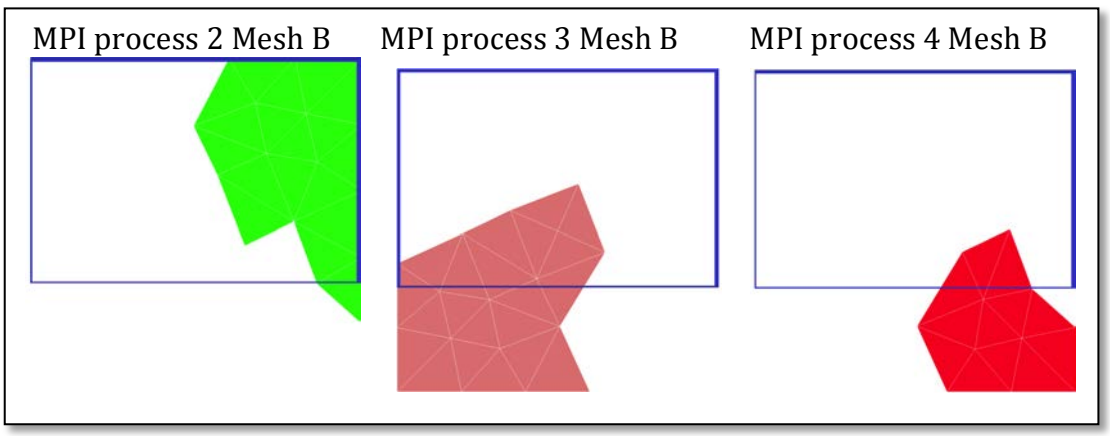

Figure 7 View of remote MPI processes

Overlapping of computation and communication

The algorithm allows the overlapping of computation and communication. 4.4.1.5 Step 3 computes the local supermesh using elements, vertices and data that are available to the MPI process. Not all MPI processes can perform this step. However, an MPI process that performs calculations in this step could perform the computation whilst the MPI library sends data to other MPI processes. Both approaches were implemented. However, during performance

testing an increase in runtime was observed when using overlapping. Therefore, this is available, but disabled by default.

Parallel performance

Several benchmarks were run in order to measure the performance of the libsupermesh library. The 2D and 3D benchmarks take as inputs two meshes (A and B) and construct an intersection mesh. For the 2D benchmarks mesh A is a triangle shaped mesh, whereas mesh B is a square shaped mesh. The meshes were generated using Gmsh [11]. Meshes A and B each consist of triangle elements with quasi-uniform resolution. Several benchmarks were run; however, only two sets of results (small and large) are presented:

- 33,065,204 elements for mesh A and 29,399,556 elements for mesh B;
- 297,512,852 elements for mesh A and 264,549,836 elements for mesh B.

The 3D benchmarks use a pyramid shaped mesh (as mesh A) and a cubed shaped mesh (as mesh B). The meshes were generated using Gmsh [11]. Meshes A and B use tetrahedral elements with quasi-uniform resolution. Several benchmarks were run; however, only two sets of results (small and large) are presented:

- 33,077,698 elements for mesh A and 27,301,039 elements for mesh B;
- 141,873,169 elements for mesh A and 128,459,529 elements for mesh B.

The first benchmark computes the area (2D) or volume (3D) of the mesh intersection region. Since the sizes and shapes of the two meshes are known, the area or volume of their intersection can be calculated, and used to verify correctness of the result using libsupermesh.

The second benchmark computes the intersection area or volume, and also the L^2 inner product of two piecewise linear continuous functions defined using a standard P1 Lagrange basis, computed using three point degree 2 quadrature in 2D [12], and four point degree 2 quadrature in 3D [13 section 8.8,14].

The third benchmark, performed only in 2D, computes the intersection area, and also the L^2 inner product of two piecewise quadratic continuous functions defined using a standard P2 Lagrange basis, computed using a local hard coded P2 element mass matrix.

Each benchmark was executed 5 times and the average of the last 4 runs is used (ignoring the first run). The following sections assess the strong scaling of the libsupermesh library. The following git commit version was used: 879f126b7a64a89463f2efd8caf013eea096a770. This version is not the same as the released version; however, no significant changes in the results are expected. The code was compiled using the GNU compiler (ver. 5.1.0) on ARCHER.

Some of the small cases were run on the Test and Development System (TDS); an infrastructure which includes the same hardware and software components as the main system. TDS nodes have the same processor and memory configurations as are used in the main system; however, the network topology is smaller.

4.4.2.1 2D – small mesh

Each benchmark was run first in serial and then in parallel. During testing the I/O performance issue discussed in section 4.3 was identified. The libspatialindex interface was modified so as to avoid this and a 4x performance improvement was observed for the small benchmark set (runtime reduction from 6,333 seconds to 1,019 seconds), in serial.

The Fluidity "fldecomp" tool was used to partition the meshes. Table 1 shows the distribution of elements of the two meshes.

Several smaller benchmarks were run in order to fine-tune the parallel job launcher (aprun) options and it was noticed that the code ran faster if the nodes were underpopulated. The benchmarks were thus run using 20 cores on each ARCHER node. Under-populating is a common practice on HPC systems. Furthermore, some test cases required a large amount of RAM. Under-populating nodes allows each MPI process to use more RAM.

Procs	Min		Max		Median	
	Triangle	Square	Triangle	Square	Triangle	Square
1	33,065,204	29,399,556	33,065,204	29,399,556	33,065,204	29,399,556
2	16,546,182	14,712,561	16,548,622	14,712,858	16,547,402	14,712,710
5	6,620,669	5,891,449	6,637,897	5,900,586	6,624,084	5,893,728
10	3,310,560	2,946,974	3,325,951	2,958,106	3,319,551	2,950,357
20	1,656,071	1,474,267	1,666,012	1,483,021	1,662,305	1,478,769
40	828,451	736,979	836,783	746,447	833,899	742,302
100	331,787	295,708	337,877	302,535	335,691	298,739
200	165,241	147,660	172,632	153,455	168,987	150,398
400	82,833	73,510	87,678	78,295	85,163	75,954
1,000	32,764	29,345	36,009	32,077	34,685	30,921
2,000	16,769	14,720	18,519	16,564	17,700	15,790

Table 1 Distribution of elements for the 2D small mesh benchmarks

The first 2D benchmark computes the area of the supermesh. Table 2 and Figure 8 show the runtime of the serial and parallel runs. All axes of Figure 8 are in logarithmic scale and the ideal speedup line is used as a reference point. Each benchmark verifies that the calculated area is correct. The calculated area varies between 49.9999999999911 and 50.0000000000057, with an exact analytical value equal to 50. This variance is acceptable and it is attributed to floating point errors.

Procs	Runtime (in seconds)	Speedup	Parallel Efficiency	MPI All-to-All (seconds)
1	407.6	1.00	100.0%	
2	298.4	1.37	68.3%	4.24785
5	195.0	2.09	41.8%	0.26500
10	97.6	4.17	41.7%	0.42673
20	45.2	9.02	45.1%	0.23855
40	23.7	17.21	43.0%	0.34014
100	8.5	47.95	47.9%	0.34432
200	4.0	102.15	51.1%	0.20742
400	2.0	202.42	50.6%	0.25765
1,000	4.2	96.40	9.6%	5.02010
2,000	0.7	551.36	27.6%	0.44537

Table 2 MPI all-to-all time for the 2D small meshes, intersection area calculation benchmark

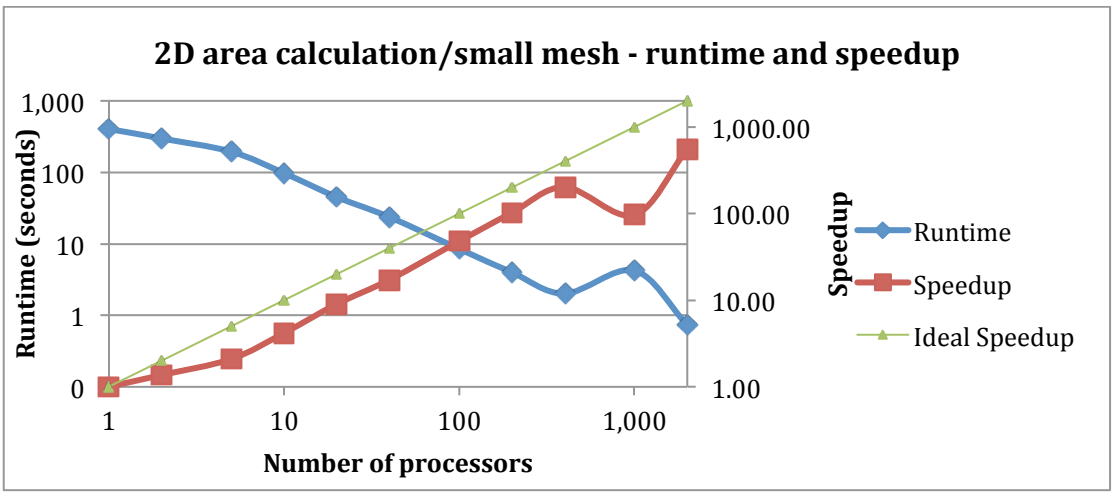

Figure 8 Runtime and speedup for the 2D small meshes, intersection area calculation

The drop in performance when using 1,000 MPI processes is caused by the MPI all-to-all communication. Table 2 shows the duration of the MPI all-to-all communication step. When we use 2 MPI processes the MPI all-to-all step takes a long time, due to the large amount of data that is sent across the network. However, as the number of MPI processes increases the cost of the MPI all-to-all call is reduced and it plateaus, thanks to the MPI implementation on ARCHER. The cost of the call (in seconds) reduces and stabilises at around 1/3 of a second. However, when 1,000 MPI processes are used the cost of call increases by an order of magnitude. This trend does not continue when the number of processes is increased to 2,000, as the call takes 0.4 seconds to complete.

Each ARCHER compute node has 24 Intel Xeon CPUs. 4 ARCHER compute nodes are bundled together forming an ARCHER compute blade (or 96 CPUs). 16 ARCHER compute blades (or 64 ARCHER compute nodes; or 1,536 CPUs) form an ARCHER chassis. Network communication (both latency and bandwidth) inside a compute blade is extremely fast. Communication between different chassis incurs a latency hit (since messages have to travel longer distances). The scheduler tries to allocate compute nodes which are located in the same compute

blade or chassis. However, when 50 ARCHER compute nodes are requested the scheduler might not be able to accommodate the request on one chassis. Therefore, the scheduler allocates a job on several chassis; network communication takes a hit.

Overall, the runtime of constructing and using the intersection mesh is reduced from 407 seconds to less than a second (using 2,000 MPI processes or 100 ARCHER nodes). The runtime includes all overheads and MPI communications. When using 2,000 MPI processes the library spends 0.4 seconds communicating using MPI all-to-all; 0.05 seconds sending and receiving data from other MPI processes; 0.15 seconds computing and using the local mesh using local mesh B data and 0.15 seconds computing and using the local mesh using remote mesh B data. The parallel efficiency of the code starts at 68% when using 2 MPI processes and drops to 27% when 2,000 MPI processes are used.

Figure 9 shows the runtime improvement of the P1 L^2 inner product benchmark using the small data set. The minimum number of MPI processes used in this benchmark is 2. All axes of Figure 9 use a logarithmic scale and the ideal speedup line is used as a reference point. Each benchmark verifies that the calculated area and integral are correct. The calculated area varies between 49.9999999999951 and 50.0000000000002, with an exact analytical value equal to 50. The calculated L^2 inner product varies between 833.333333333161 and 833.333333333338, with an exact analytical result equal to 833 and one third. This variance is acceptable and is attributed to floating point errors.

Using libsupermesh the time taken by the P1 L^2 inner product benchmark is reduced from 433 seconds to 1.5 seconds (when using 2,000 MPI processes or 100 ARCHER nodes). The parallel efficiency of the code starts at 85% when using 5 MPI processes and drops to 28% when we use 2,000 MPI processes.

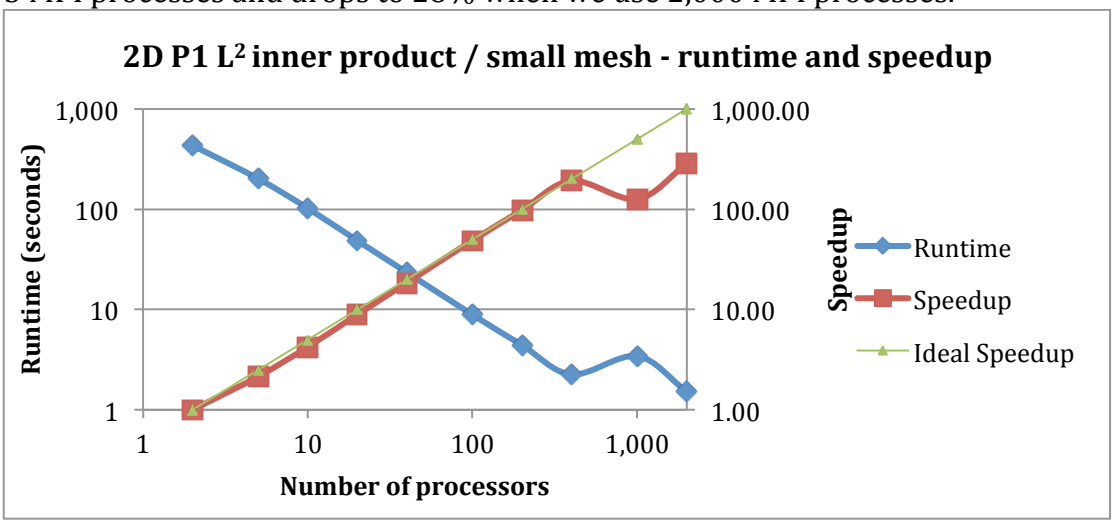

Figure 9 Runtime and speedup for the 2D small meshes, intersection area and inner product calculation

4.4.2.2 2D – large mesh

Table 3 shows the distribution of elements of the two meshes for the larger mesh case. The benchmarks are run using 20 cores on each ARCHER node. Thus, 500

nodes were used for the 10,000 MPI process runs and 10 nodes for the 200 MPI process runs.

Procs	Min		Max		Median	
	Triangle	Square	Triangle	Square	Triangle	Square
1	297,512,852	264,549,836	297,512,852	264,549,836	297,512,852	264,549,836
2	148,800,446	132,315,341	148,802,495	132,315,656	148,801,471	132,315,499
5	59,526,138	52,940,948	59,562,613	52,979,531	59,539,246	52,953,158
10	29,761,499	26,476,139	29,809,756	26,512,300	29,789,443	26,494,191
20	14,884,658	13,241,816	14,917,449	13,268,056	14,904,219	13,254,306
40	7,442,349	6,623,682	7,470,168	6,647,800	7,459,816	6,636,179
100	2,977,648	2,648,668	2,996,469	2,667,550	2,990,197	2,658,916
200	1,489,554	1,325,830	1,502,729	1,337,397	1,499,048	1,333,618
400	744,088	661,842	756,911	677,183	751,834	668,137
1,000	296,622	263,553	307,173	275,048	302,468	269,263
2,000	147,800	131,524	156,562	139,225	152,280	135,595
4,000	73,682	65,235	79,325	70,640	76,851	68,480
6,000	49,164	43,970	53,587	47,597	51,609	46,010
8,000	36,841	32,825	40,398	36,004	38,950	34,726
10,000	29,653	26,600	32,509	29,018	31,328	27,943

Table 3 Distribution of elements for the 2D large mesh benchmark

The runtime and speedup for the intersection area calculation are shown in figure 5. As long as the MPI processes each have several thousand elements to process the benchmark scales. However, as the number of MPI processes is increased the number of elements per process decreases and the benchmark starts to plateau.

The benchmark scales up to 8,000 cores. The runtime of constructing and using the supermesh is reduced from 6,980 seconds to 1.63 seconds (when using 10,000 MPI processes or 500 ARCHER nodes) or to 3.85 seconds (when using 2,000 MPI processes or 100 ARCHER nodes). Parallel efficiency starts at 75% and drops to 43% when 10,000 MPI processes are used.

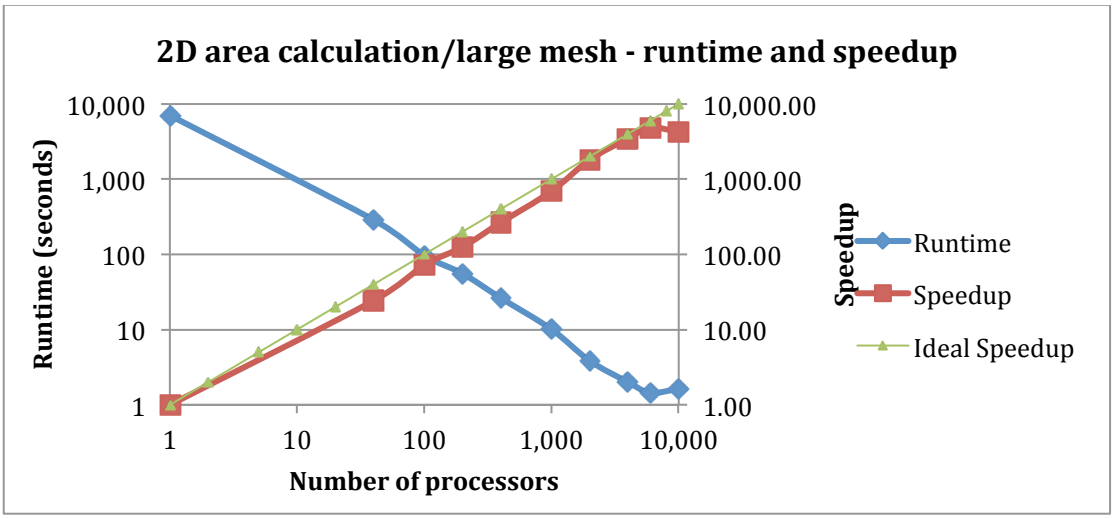

Figure 10 Runtime and speedup for the 2D large meshes, intersection area calculation

4.4.2.3 3D – small mesh

As in the section 4.4.2.1 2D case, each benchmark is run first in serial and then in parallel. The Fluidity tool "fldecomp" was used to partition the meshes. Table 4 shows the distribution of elements of the two meshes.

Procs	Min	Max	Median

	Pyramid	Cube	Pyramid	Cube	Pyramid	Cube
1	33,077,698	27,301,039	33,077,698	27,301,039	33,077,698	27,301,039
2	16,863,659	13,928,823	16,893,894	13,953,675	16,878,777	13,941,249
5	6,856,565	5,691,975	6,950,498	5,803,237	6,879,114	5,718,921
10	3,399,029	2,888,987	3,667,456	3,049,078	3,521,839	2,912,725
20	1,685,080	1,437,955	1,932,586	1,650,304	1,843,435	1,521,248
40	839,041	722,418	1,050,621	875,655	951,893	790,728
100	340,607	287,097	457,559	379,372	405,077	339,380
200	170,507	145,516	241,819	203,575	213,348	180,828
400	84,386	73,920	132,589	109,738	116,141	97,879
1,000	33,782	30,340	60,833	51,066	53,518	45,647
2,000	17,522	15,632	34,235	29,214	30,293	25,883

Table 4 Distribution of elements for the 3D small mesh benchmarks

The first 3D benchmark computes the volume of the supermesh. Figure 11 shows the runtime of the serial and parallel runs. All axes of Figure 11 are in logarithmic scale and the ideal speedup line is used as a reference point. Each benchmark verifies that the calculated volume is correct. The calculated volume varies between 333.333333332263 and 333.333333325764, with an exact analytical value of 333 and one third. This variance is acceptable and is attributed to floating point errors.

Overall, the runtime of constructing and using the supermesh is reduced from 1,695 seconds to 4.4 seconds (using 2,000 MPI processes or 100 ARCHER nodes). The runtime includes all overheads and MPI communications. When using 2,000 MPI processes the library spends 1.4 seconds communicating using MPI all-to-all; 0.15 seconds sending and receiving data from other MPI processes; 0.55 seconds computing the local mesh using local data, and 0.9 seconds computing the local mesh using remote data. The parallel efficiency of the code starts at 68% when using 2 MPI processes and drops to 22% when 2,000 MPI processes are used.

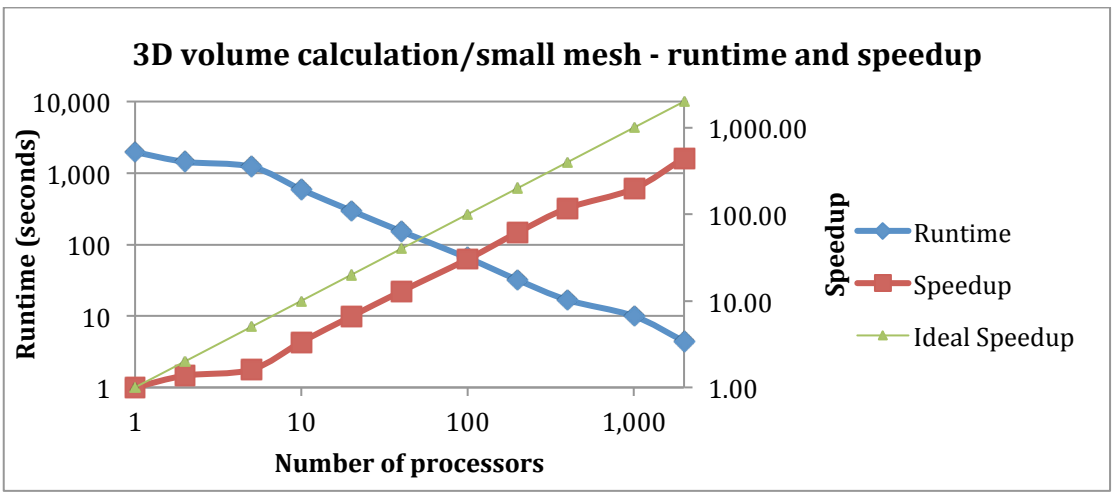

Figure 11 Runtime and speedup for the 3D small meshes, intersection volume calculation

Figure 12 shows the runtime and speedup of the 3D P1 L^2 inner product benchmark using the small mesh set.

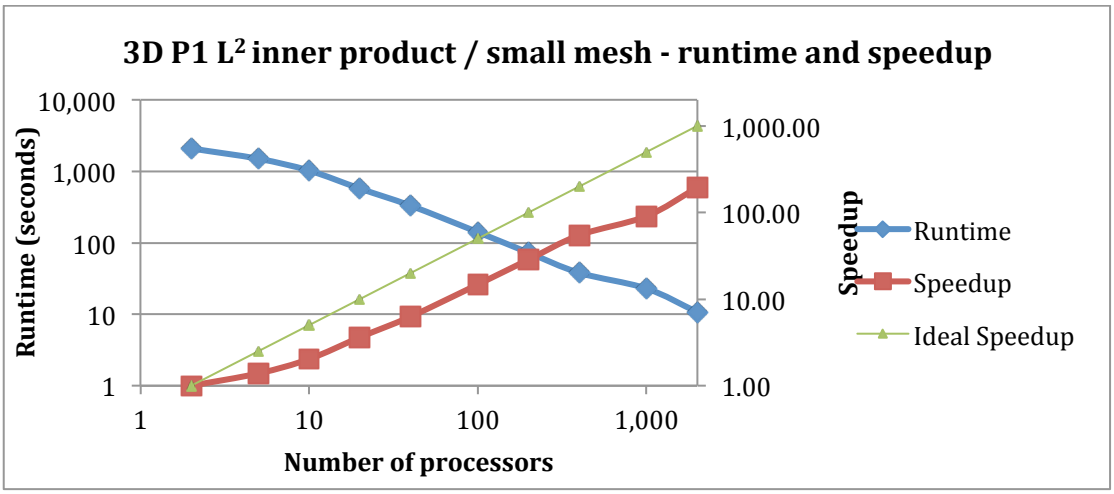

Figure 12 Runtime and speedup for the 3D small meshes, intersection volume and inner product calculation

The minimum number of MPI processes used in this benchmark is 2. It can be seen that the code scales up to 2,000 MPI processes. The parallel efficiency starts at 55% for 5 MPI processes and drops to 19% for 2,000 MPI processes. Total runtime is reduced from 2,082 seconds when using 2 MPI processes to 10.7 seconds for 2,000 MPI processes. The calculated value for the 3D P1 L^2 inner ranged between 12499.9999999224 and 12499.9999996795, with an exact analytical value of 12500. This variance is acceptable and is attributed to floating point errors.

4.4.2.4 3D – large mesh

Table 5 shows the distribution of elements for the two meshes used in the larger 3D case.

Procs	Min		Max		Median	
	Pyramid	Cube	Pyramid	Cube	Pyramid	Cube
1	141,873,169	128,459,529	141,873,169	128,459,529	141,873,169	128,459,529
2	71,762,140	65,084,706	71,863,664	65,091,104	71,812,902	65,087,905
5	29,065,505	26,415,340	29,115,706	26,609,193	29,078,719	26,442,965
10	14,474,160	13,302,577	15,220,962	13,860,999	14,778,191	13,365,003
20	7,227,813	6,661,998	7,871,941	7,188,064	7,581,408	6,869,904
40	3,623,315	3,357,265	4,117,087	3,742,773	3,879,448	3,508,224
100	1,442,679	1,323,823	1,742,331	1,579,347	1,615,741	1,470,771
200	720,006	673,832	914,791	838,839	839,927	766,798
400	349,144	333,555	484,535	441,758	441,138	401,968
1,000	143,653	134,480	210,185	191,851	193,496	177,947
2,000	72,545	67,484	116,447	105,620	105,155	96,491
4,000	32,718	34,066	63,090	58,524	57,733	53,059
6,000	23,544	23,648	45,902	41,592	40,957	37,627
8,000	16,822	18,486	37,066	34,415	32,194	29,640
10,000	13,901	14,403	30,493	27,837	26,744	24,645

Table 5 Distribution of elements in the 3D large mesh benchmarks

The first 3D benchmark computes the volume of the supermesh. Figure 13 shows the runtime of the serial and parallel runs. All axes of Figure 13 are in logarithmic scale and the ideal speedup line is used as a reference point. Each benchmark verifies that the calculated volume is correct. The calculated volume varies between 333.333333304712 and 333.333333333264, with an exact

analytical value equal to 333 and one third. This variance is acceptable and is attributed to floating point errors.

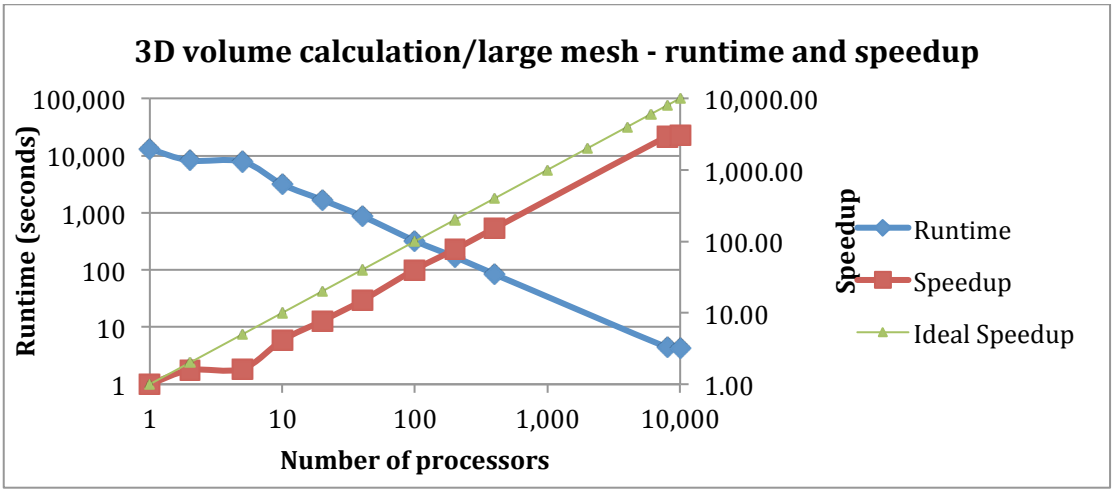

Figure 13 Runtime and speedup for the 3D large meshes, intersection volume calculation

It can be seen that the 3D benchmark scales as the number of MPI processes is increased. Runtime is reduced from 12,978 seconds to 4.3 seconds (when using 10,000 MPI processes or 500 ARCHER nodes). Parallel efficiency starts at 78% and it drops to 30% when 10,000 MPI processes are used.

Finally, Figure 14 shows the runtime and speedup of the 3D P1 L^2 inner product benchmark. The minimum number of MPI processes used in this benchmark is 2.

The 3D P1 L^2 inner product benchmark scales well. Parallel efficiency starts at 48% and drops to 26% when 10,000 MPI processes are used.

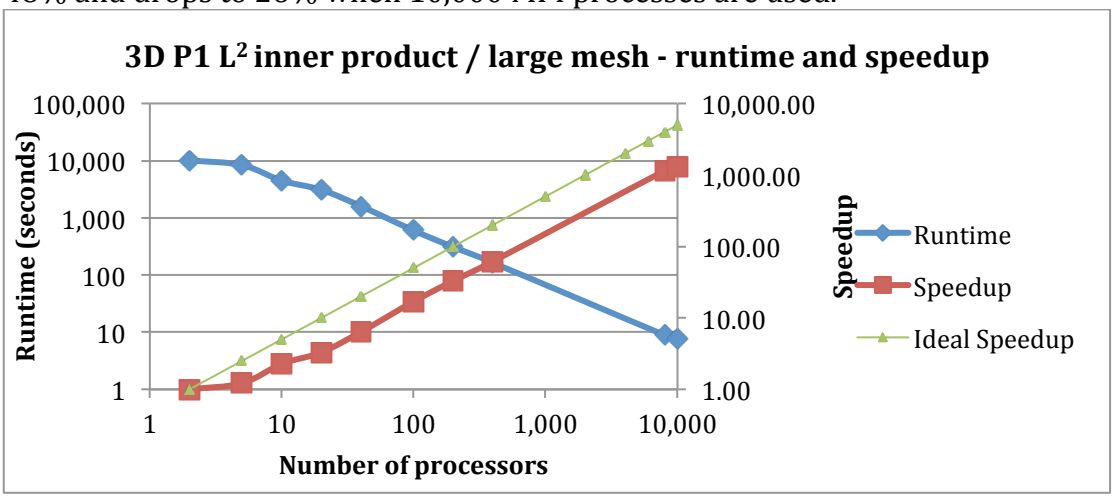

Figure 14 Runtime and speedup for the 3D large meshes, intersection volume and inner product calculation

4.5 Scientific Impact and Dissemination

libsupermesh has been released under the open source LGPL version 2.1 license [1]. The library includes a comprehensive set of regression tests, and a 49 page manual documenting the primary features. The open source license, combined

with the extensive documentation, will facilitate adoption in new and existing codes.

libsupermesh has been integrated with Fluidity [2], and this is expected to be incorporated into the main Fluidity source code repository.

4.6 Summary and Conclusions

This project has successfully produced a serial general purpose supermeshing library for non-matching unstructured meshes. Furthermore, an algorithm for parallel supermeshing, with non-matching domain decompositions, has been implemented in the library. The software is available under an open source license through public repositories. The library has been optimised and benchmarked. Benchmarking showed that it can scale up to 10,000 cores for a one hundred million degree of freedom problem with acceptable performance. Use of the standalone library has been integrated into Fluidity.

4.7 Funding Statement

This work was funded under the embedded CSE programme of the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk).
Development of libsupermesh was funded by ARCHER eCSE03-8, "Parallel supermeshing for multimesh modelling".

4.8 References

1. libsupermesh [online]. Available at: https://bitbucket.org/libsupermesh/libsupermesh [Accessed 14 May 2016]
2. Fluidity [online]. Available at: http://fluidityproject.github.io/ [Accessed 08 May 2016]
3. P. E. Farrell, M. D. Piggott, C. C. Pain, G. J. Gorman, and C. R. Wilson, "Conservative interpolation between unstructured meshes via supermesh construction", Computer Methods in Applied Mechanics and Engineering, 198, pp. 2632-2642, 2009
4. P. E. Farrell and J. R. Maddison, "Conservative interpolation between volume meshes by local Galerkin projection", Computer Methods in Applied Mechanics and Engineering, 200, pp. 89-100, 2011
5. M. J. Gander and C. Japhet, "An algorithm for non-matching grid projections with linear complexity", in "Domain Decomposition Methods in Science and Engineering XVIII", M. Bercovier, M. J. Gander, R. Kornhuber, and O. Widlund (editors), Springer Berlin Heidelberg, pp. 185-192, 2009
6. M. J. Gander and C. Japhet, "Algorithm 932: PANG: Software for nomatching grid projections in 2D and 3D with linear complexity", ACM Transactions on Mathematical Software, 40, pp. 6:1-6:25, 2013
7. libspatialindex [online]. Available at: http://libspatialindex.github.io/ [Accessed 14 May 2016]

8. D. H. Eberly, “3D Game Engine Design: A practical approach to real-time computer graphics”, Second Edition, CRC Press, 2007
9. The Computational Geometry Algorithms Library (CGAL) [online]. Available at: http://www.cgal.org/ [Accessed 23 May 2016]
10. I. E. Sutherland and G. W. Hodgman, “Reentrant polygon clipping”, Communications of the ACM, 17, pp. 32-42, 1974
11. C. Geuzaine and J.-F. Remacle, “Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities”, International Journal for Numerical Methods in Engineering, 79, pp. 1309-1331, 2009
12. D. A. Dunavant, “High degree efficient symmetrical Gaussian quadrature rules for the triangle”, International Journal for Numerical Methods in Engineering, 21, pp. 1129-1148, 1985
13. A. H. Stroud, “Approximate Calculation of Multiple Integrals”, Prentice-Hall, Inc., 1971
14. P. C. Hammer and A. H. Stroud, “Numerical integration over simplexes”, Mathematical Tables and Other Aids to Computation, 10, pp. 137-139, 1956