# Hierarchical grid generation and decompostion for HemeLB

Rupert W. Nash, Derek Groen

March 13, 2019

**Abstract**

We describe our work to improve the grid generation for HemeLB, a lattice Boltzmann based, highly parallel, computational fluid dynamics application. The software engineering challenges proved to be severe but we have created a tool that is accurate and consistent, parallel in two key parts of the process. For realistic problems using a 36 core machine we can now produce voxelisations of a factor of ten faster.

## 1 Introduction

This report summarises the work done during the eCSE project 03-13, "Grids in grids: hierarchical grid generation and decomposition for a massively parallel blood flow simulator" to improve the HemeLB computation fluid dynamics software [7, 8, 5].

Lattice Boltzmann methods work on a structured grid, typically a single resolution regular cubic (or square in two dimensions) Eulerian grid. The core problem we face is how to move from a Lagrangian triangulated representations of a surface to a voxelised, Eulerian grid of particles in an efficient and accurate way. Further, many boundary conditions for LBM require accurate determination of the cut distances, which are distances along the links between neighbouring points where intersection occurs.

A key problem faced in computational science that is not seen in the computer graphics field, where the majority of relevant published work is available, is the demand for higher accuracy so as not to introduce extra sources of error. This is particularly important for computation fluid dynamics problems where the flow is largely determined by boundary conditions.

Most methods for representing irregular geometrical objects use some sort of multi-block approach: typically either multiblock or tree-based. In the proposal stage, we restricted ourselves to tree-based approaches due to the planned objectives. There are two main types of structure: octrees and k-d trees.

Octrees start with a root node that represents a cube of space. A node can have children that represent a subspace of the parents space, divided into eight equal cubes, as shown in Figure 1. This has the advantage of relatively simple traversal and the possibility to deterministically assign an indexing to nodes in the tree.

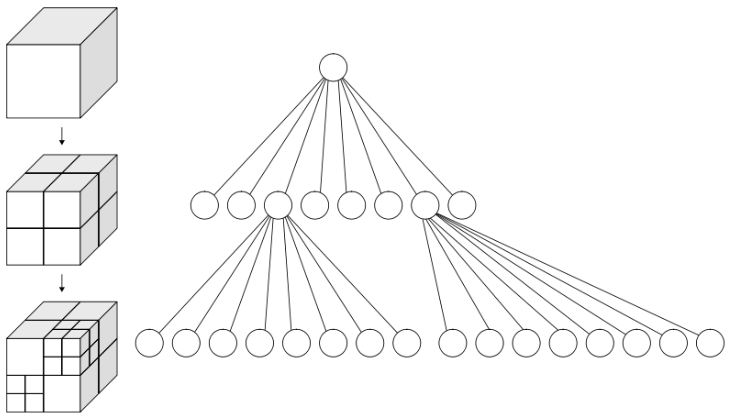

Figure 1: Schematic of a three-level Octree. (Image ©WhiteTimberwolf, [CC BY-SA 3.0], `https://commons.wikimedia.org/wiki/File:Octree2.svg`)

K-d trees are binary trees, where each node represents a cuboidal space. If a node has children, then the nodes space is split along a hyper-plane perpendicular to one of its dimensions and each sub-space assigned to one of the two children. Commonly the splitting dimension is cycled with each level down through the tree, but this is an implementation choice. K-d trees can be more optimal by some measures but are more complex to implement; we did not find any uses in the literature for voxelization methods.

A brief survey of the literature around voxelization a large number of approaches towards producing a surface voxelization, defined as identifying those voxels which through which any surface triangle passes. While this is not what we seek, from this we can, in principle determine which points are inside the domain and then fill the interior. The mostly widely cited method appears to be that of Huang et al. [6], which operates on a dense output voxel field. A number of other authors [13, 2] have extended this method to work using octrees to store the output voxelization. A few other widely cited methods [15, 10] have been published but they appear to be more complex to implement.

## 2 Setuptool

Here we discuss the improvements made to the HemeLB grid generation tool, which we refer to as the setuptool.

### 2.1 First implementation

We designed the workflow shown in table 1 to combine Huangs algorithm with a voxel octree in order to allow efficient shared-memory parallelism. We chose to use a test-driven style for development and to implement each step in order.

For preprocessing, we reused the existing code in python to a large extent and then passed the data into the C++ extension module inside VTK data structures as before. As part of this work, we replaced the setuptools old binary,

| | |
|---|---|
| Serial | Preprocess input triangulation |
| Serial | Prepare emtpy octree |
| Parallel over triangles | Sort triangles onto mid-level of tree |
| Serial | Merge trees |
| Parallel over subtrees | Apply Huang's algorithm to all triangles attached |
| Parallel | Flood fill interior |
| Serial | Write output |

Table 1: Planned workflow for setuptool

Pickle-based format (so called profile files, .pro extension) with a simple, human readable YAML profile file and created a small tool to convert old files to the new format.

We implemented a simple Octree template class in C++, using `std::shared_ptr` to hold the references to child-nodes in order to allow the subtrees to be efficiently shared across threads. This data structure allows easy growth of the tree at the cost of many dynamic memory allocations. The obvious mitigation of using memory pools wrapped in an allocator was planned for later in the project. We created a unit test harness using the CPPunit library[1] which is used by the main part of HemeLB.

A surface can have many thousands to millions of triangles which, in the VTK data structures used to read, clip and scale the domain, are not ordered in any way. Therefore sorting these spatially such that Huangs algorithm can be applied efficiently in parallel is needed. We decided to simply divide the unsorted list of triangles into contiguous chunks and for each chunk create an octree with the triangle IDs (an integer) assigned to a set at the middle level of the tree. We chose to use `boost::flat_set` here to allow fast merging of the IDs since it stores data in a single contiguous chunk and maintained in order. Each chunk is to be processed asynchronously using `std::async` in a thread and, since each thread will only write to its own private tree, scaling should be good. The resulting output tree was, as per the API of `std::async`, stored in a `std::future`. As these cannot be copied, we stored them in a `std::deque` which does not need to copy/move its contained elements. Thus launching the threads becomes straightforward:

```
std::deque<std::future<TriTree>> futures;
for (auto i = 0; i < ntasks; ++i) {
  decltype(n_tri) min_ind = i * tpp;
  decltype(n_tri) max_ind = std::min(decltype(n_tri)((i+1)*tpp), n_tri);
  auto triStart = triangles.begin() + min_ind;
  auto triEnd = triangles.begin() + max_ind;
  futures.push_back(
    std::async(
      std::launch::async, work_function,
      n_levels, tri_level, points, triStart, triEnd, min_ind
    )
  );
}
```

[1]http://cppunit.sourceforge.net/doc/lastest/index.html

The main thread can then inspect the resulting trees, merging them.

We then implemented Huangs algorithm for producing a surface voxelization and twelve unit tests. The implementation was relatively short at 216 lines. The basic process is:

- For each triangle:
  - Compute the bounding box in terms of voxel centres.
  - For each voxel in the bounding box check:
    - Is the point closer than $R_C = \sqrt{3}/2$ to a vertex?
    - Is the point within a bounding cylinder of radius $R_C$ along each edge?
    - Is the point between the two planes parallel to the triangle but offset by $t_{26}$ (as defined in Huang [6] Fig. 10)?
    - If any test is true, the point is a surface voxel.
    - Determine, using the triangle normal, whether the voxel is inside the domain (i.e. is fluid).
    - If fluid, check for an existing octree node and create an empty one if one does not exist.
    - Compute the cut distances from the point to the triangle along the 26 nearest neighbour directions. If the new cut distance is closer, store the new cut distance and the ID of the triangle.

This algorithm was time consuming to implement but thanks to the unit tests (12) we had some confidence that it was correct. The implementation in parallel was relatively straightforward: a queue of tasks was created as before for execution by a thread pool. Each task stores creates a `std::future` which can be checked for the value. The main thread awaits each, in turn, merging them into a final tree if there are any fluid leaf nodes.

The flood fill algorithm was, however difficult to implement in parallel. We decided to use a known-fast serial algorithm using a queue of sites that are fluid but have yet to have their neighbours tested. We planned to replace the simple queue with a multi-producer/consumer thread safety guarantee, such as `boost::lockfree::queue`, at a later stage to parallelise this.

For the initial output, we wrote a simple text-based format for fast debugging.

We then began integration testing with some test problems, such as a simple pipe and a sphere, failed. Detailed and time-consuming investigations showed that some links between sites were finding intersections with the surface which were being missed by other links. An illustration of a typical case is shown in figure 2. Assume that the black-filled point is correctly identified as being a fluid site. We then calculate the intersections with the triangles and the intersections marked in grey are correctly found. However for the vertically upwards link, no intersection between either the left or right triangles is found. In the left case there is indeed no intersection, but in the right case, an intersection should have occurred but was not. This area is highlighted in red and shown zoomed in on the upper right.

The reason for this is that when one is doing finite-precision mathematical operations (dot and cross products, as well as conventional multiplications, additions etc), there is of course some small but finite error. When it comes to

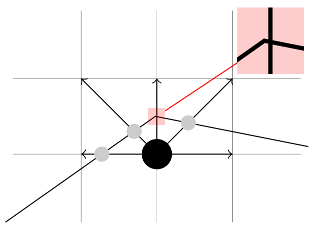

Figure 2: Illustration of a missed intersection. The point filled with black is correctly identified as being a fluid. Some intersections (grey) are correctly found. One, moving vertically upwards, is missed and is highlighted in red (see zoomed subfigure in top right).

determining if, for example, a line intersects a triangle, one must compare some computed quantity to a threshold. In the illustration shown in figure 2, this inequality fails to indicate intersection and the process now fails spectacularly.

When the flood fill algorithm inspects the neighbours of the black point, it sees no intersection when examining the link in the up direction and thus marks that exterior point as fluid. Our domain has sprung a leak! This causes the flood-fill step to fill the entire bounding cube at large computation expense and produces an invalid voxelization.

## 2.2 Second implementation

Given the issues around imprecision intersection detection, we decided to reimplement the surface voxeliser using CGAL (Computational Geometry Algorithms Library) [14] which offers computational geometry primitives such as line-triangle intersection etc.

The conversion to use CGAL was very challenging due to the sometimes complex ways the tests have to be constructed to ensure a consistent decision and eventually we abandoned using Huang's algorithm as too complex.

Instead we chose to use a CGAL data structure called `CGAL::AABB_Tree`: an axis-aligned bounding-box tree. Given a list of polygons, CGAL will build a tree structure that allows the user to perform intersections between the contained polygons and test primitives, such as the line segments we require.

We replaced Huang's algorithm with the following. First, construct a CGAL AABB seach tree and in parallel for each mid-tree level node with triangle data attached:

- For each grid point in the block
  - For of the 26 neighbouring points grid points
    - Compute all intersections
    - Sort the intersections by cut distance
    - Merge approximately coincident intersections
    - Store the closest intersection to the grid point
  - Classify the site by analyzing all links

        * Use the normal of the triangle to determine if the closest intersection implies we are inside or outside.
        * If inside get or create a node in the output subtree and fill in this link's data.
    - Check that all links implied the same side of the boundary
    - Store wall normal in site data.

These sites will be only those that are interior to the domain and have at least one of their lattice vectors intersected by the surface.

The reason for filtering coincident intersections is that CGAL will create multiple intersections for line segments that intersect very close to an edge or corner. We set a tolerance $\epsilon$ as a fraction of the displacement vector. We chose to merge multiple intersections on the same side of the boundary, arbitrarily picking a single triangle as the one intersected. In the case of two intersections being on opposite sites, we chose to discard them both, as in this case it represents a small concavity in the surface and can be ignored with only a very small error.

We chose $\epsilon = 10^{-3}$ after running a few experiments and finding this to be the largest value that did not cause problems. We also chose to track the number of leaf edge sites stored within the subdomain of each octree node. When adding a node to the subtree this means incrementing the counter at each level.

## 2.3 Improving output format

Outputting in a verbose text format was useful during development but is obviously not useful for geometries with many millions of sites. Inspired by PETSc's concept of a section we created a simple static data structure to represent the octree structure.

An $N$-level octree is represented by $N + 1$ arrays of nodes. A node is represented by eight unsigned integers giving the indices of its children, or a null value (bitwise negation of zero) to indicate that a given child is missing. At the leaf node level we attach extra data stored in flat arrays. Each leaf node stores the index of its data items and the number of them. We also compute the total number of fluid sites within each subtree of the octree which will be useful for reading the section tree later.

These trees are relatively simple to build in serial, using a simple visitor pattern for traversing the octree. The vistor simply tracks its current position as a path from the root of the tree and creates nodes on the way down the tree, computing summary data on the way back up.

This data structure is then very simple to serialise using HDF5.

## 2.4 Results

We tested the setuptool using two different domains: a straight cylinder of radius 3mm and a cerebral artery network reconstructed from a 3D angiography scan of a patient (approximate inlet radise 0.2mm). In figure 3 we show 3D renderings of the two geometries from the setuptool GUI. The green (red) planes mark the inlets (outlets).

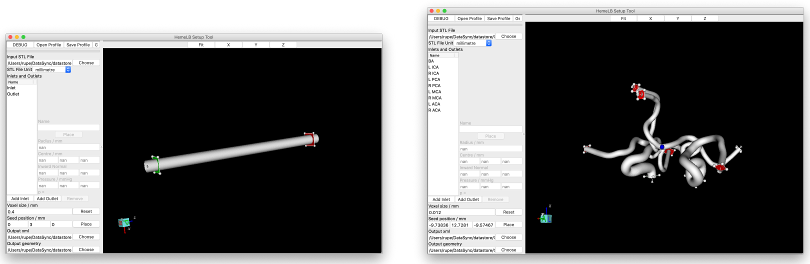

Figure 3: Visualisation of the pipe (left) and circle of Willis (right) domains used for testing.

For each of the two problems we started at the lowest resolution voxelisation that would give simulations of any accuracy and then refined by a factor of 2 until runs times became unmanageable. The resolutions used were:

**Pipe** : 0.4mm, 0.2mm, 0.1mm, 0.05mm, 0.025mm

**CoW** : 12µm, 6µm, 3µm, 1.5µm

We ran all the tests on a single node of Cirrus, which contains two 2.1GHz, 18-core Intel Xeon E5-2695 (Broadwell) CPUs and 256GBof RAM.

In figure 4 we plot the total run time for both versions of the setuptool and the two geometries versus the number of sites produced. In all cases, the run time scales approximately linearly with the number of output sites as one would expect since something is being created for each site.

For the pipe problem, the run times of both implementations are very similar however of the CoW, the new setuptool outperforms the old one by a factor of approximately ten.

To understand the performance better, we tracked the wall clock time of the different stages of the process. In figure 5 we show these, excluding the preprocessing and triangle sorting steps as these took a negligible fraction of the total. The results clearly show that the flood fill step is dominant, as it must visit every fluid site in the output in serial, much as for the old implementation.

## 3 Protopart

We have developed a standalone library, Protopart+PPStee[2]), which is able to apply and analyse a range of decomposition algorithms. The library takes HemeLB domain data as input, and performs the process of assigning individual blocks to computational cores using a user-specified algorithm from the options:

- Parmetis [12]
- PT-Scotch [4]
- Zoltan [3]

[2]`http://www.github.com/djgroen/protopart`

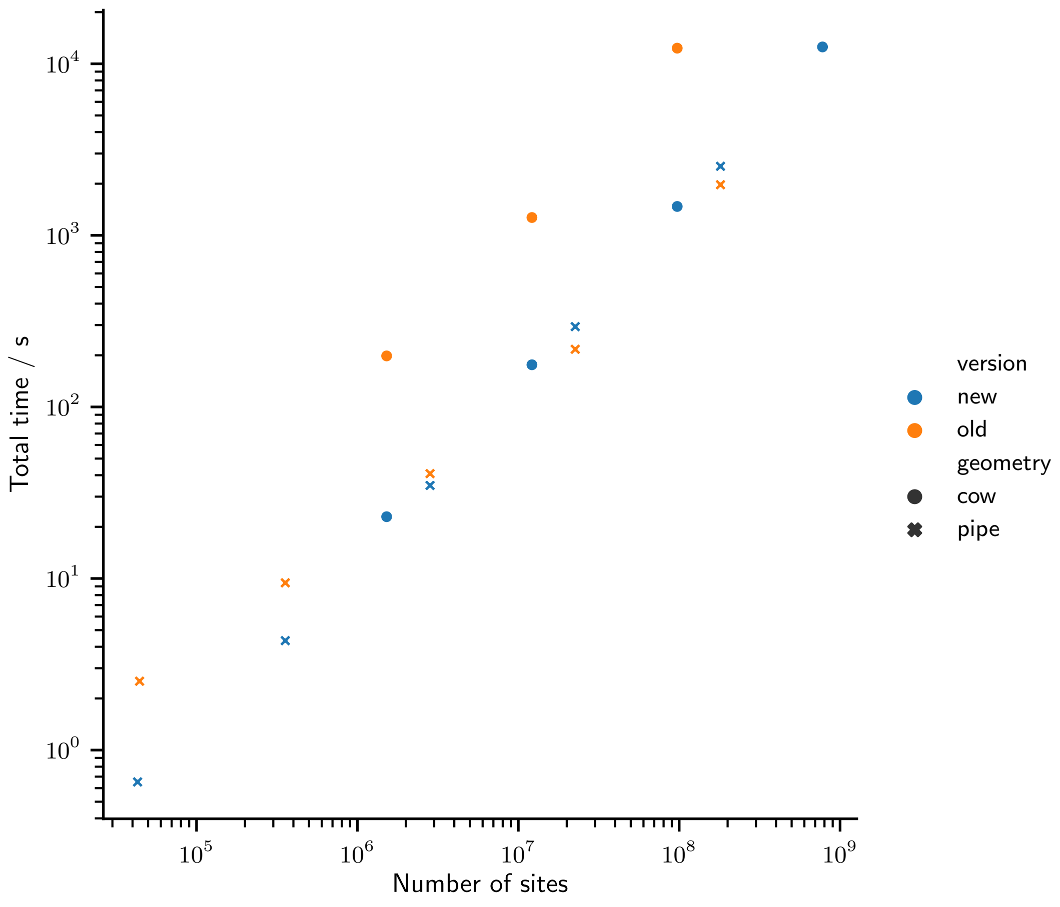

Figure 4: Total time for domain voxelisation with old and new implementations of the setuptool, indicated by yellow and blue respectively. We show results for two different domains, a simple pipe (crosses) and the circle of Willis (circles). Each domain is discretised with multiple resolutions giving different numbers of output sites ($x$-axis). All runs performed on a single node of Cirrus.

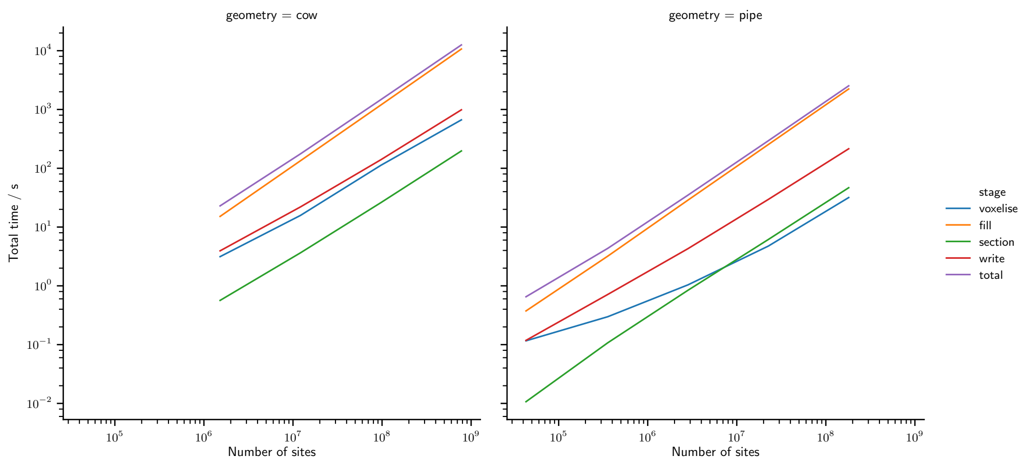

Figure 5: All runs performed on a single node of Cirrus.

Within Protopart with have created some tools for assessing the quality of domain decompostions produced. Given $P$ processes, let each domain $i$ have $N_i$ sites assigned and $L_i$ links from its sites to sites on another domain $j \neq i$. The quality metrics are

- The maximum number of lattice sites assigned to a domain scaled by the average, i.e., $\frac{\max N_i}{\bar{N}_i}$.
- The maximum number of neighbour links per domain scaled by the average number of neighbour links, i.e., $\frac{\max L_i}{\bar{L}_i}$.
- The largest number of connected neighbouring domains.

We have modified HemeLB to allow the user to provide the domain decomposition instead of computing one at simulation startup. In this way we can use Protopart+PPStee for quick diagnostics and testing, with immediate feedback on the quality. We can then feed this decomposition to HemeLB for actual simulations, optionally refining it with the preexisting decomposition library.

However, this implementation proved to be much more complex than expected, and our initial implementation led to errors for very large problems that we did not resolve during the project.

In collaboration with colleagues at UCL since the project finished, we have completed this work, incorporating the Zoltan [3] graph partioning library into HemeLB [11].

## 4 Improvements to HemeLB

We first implemented checkpointing within HemeLB. This reused the existing property extraction framework for IO. HemeLB controls output of data via an XML file, specifying the names of output files. Each file will contain a single, static set of grid points selected by a combination of simple selectors (whole geometry, surface points, near plane, near disc, near line segment). The fields written are configured by the XML file also, being selected from a list of options: pressure, velocity, shear stress, etc. We added a further option, "distributions", which writes the distribution functions $f_i$ that the lattice Boltzmann methods uses.

HemeLB also configures the initial conditions via the `<initialconditions>` element in the XML input file. Previously, the only allowed initial condition was constant pressure: i.e. the fluid in the domain was initialised to fluid at rest and local equilibrium with a constant pressure.

We first refactored the initialisation code into a separate class that was initialised with the data read from the XML (or sensible defaults). We then created a simple class hierarchy of an abstract base class with two concrete subclasses: the preexisting `EquilibriumInitialCondition` and the new `CheckpointInitialCondition`.

The new initialiser needs to read the extracted property file corresponding to a checkpoint. We had not implemented this in HemeLB, only in the post-processing tools written in Python, but having that as a reference made the implementation in C++ simple. Combined with a few new unit tests this was simple. However, the order of data within an output file depends on the domain decomposition and dealing with restarting on different decompositions, possibly on a different number of MPI tasks, was non-trivial.

Due to the overruns in the implementation of the setuptool, we restricted ourselves to restarting on exactly the same decompositions. Effectively this means using the same version of HemeLB, compiled with the same LB options, and run on the same number of MPI tasks. Then each task can easily (via the same code used for preparing to write the output) compute the offsets of the data it must read. This is read in with a collective MPI IO read and we then check that we have read in exactly the points we needed.

## 5 Conclusion

Implementing an accurate, consistent, parallel, efficient meshing tool is a very difficult software engineering task! We have created a tool that is accurate and consistent after much effort. Some key parts of the process are parallel and efficient and for realistic problems, using a 36 core machine we can now produce voxelisations a factor of ten faster.

We have implemeted tools for producing and analysing decompositions and added checkpoint restart capabilities to HemeLB.

### 5.1 Further work

**Improve memory locality for Octrees.** Since the most time-consuming part of the grid generation process, the flood-fill step, is heavily dependent on traversing the octrees, storing the nodes close to each other in memory will reduce the number of cache-misses. This can be done using the allocator abstraction used heavily by STL containers [1, 9].

An alternative would be to provide a different storage policy for the octree used as output. Since this data structure is used in an append-only mode, the overhead of using `std::shared_ptr`[3] can be avoided.

[3]A shared pointer is typically implemented as a pointer to a "control block" containing a

**Redesign the flood fill step using knowledge of octree.** The flood fill algorithm used creates a leaf node for every site that it visits. Allowing the fill to move across branch nodes that are entirely filled with simple bulk fluid sites will drastically reduce the cost in memory and run time. This would have the benefit of also reducing the cost of the section tree building step.

**Parallelise flood fill step** Currently this is serial, allowing multiple threads to work on this would be very advantageous although locking of the subtrees was a very difficult problem.

**Adapt HemeLB to directly read octree format files** This would remove a step in the simulation workflow and open the door to the improved flow settlement work originally proposed.

## 6 Acknowledgements

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

## References

[1] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] Jeroen Baert, Ares Lagae, and Philip Dutré. Out-of-core construction of sparse voxel octrees. In *HPG '13*, page 27, New York, New York, USA, 2013. ACM Press.

[3] Erik G Boman, xdc mit V xc7 ataly xfc rek, C xe9 dric Chevalier, and Karen D Devine. The Zoltan and Isorropia Parallel Toolkits for Combinatorial Scientific Computing: Partitioning, Ordering and Coloring. *Scientific Programming*, 20(2), 2012.

[4] C Chevalier and F Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Comput*, 34(6-8):318–331, July 2008.

[5] Derek Groen, James Hetherington, Hywel B Carver, Rupert W Nash, M O Bernabeu, and Peter V Coveney. Analysing and modelling the performance of the HemeLB lattice-Boltzmann simulation environment. *Journal of Computational Science*, 4(5):412–422, September 2012.

[6] Jian Huang, Roni Yagel, Vassily Filippov, and Yair Kurzion. An accurate method for voxelizing polygon meshes. In *Volume Visualization, 1998. IEEE Symposium on*, pages 119–126, October 1998.

[7] Marco D Mazzeo and Peter V Coveney. HemeLB: A high performance parallel lattice-Boltzmann code for large scale fluid flow in complex geometries. *Comput. Phys. Commun.*, 178(12):894–914, 2008.

reference count, a mutex or similar and the underlying pointer, hence any access to the logical target of the pointer requires following two pointers.

[8] Rupert W Nash, Hywel B Carver, M O Bernabeu, James Hetherington, Derek Groen, Timm Kr uger, and Peter V Coveney. Choice of boundary condition for lattice-Boltzmann simulation of moderate-Reynolds-number flow in complex domains. *Phys. Rev. E*, 89:023303, February 2014.

[9] Arthur O'Dwyer. An allocator is a handle to a heap. Talk at CppCon, 2018.

[10] Jacopo Pantaleoni. VoxelPipe. In *HPG '11*, page 99, New York, New York, USA, 2011. ACM Press.

[11] Alexander Patronis, Robin A Richardson, Sebastian Schmieschek, Brian J N Wylie, Rupert W Nash, and Peter V Coveney. Modeling Patient-Specific Magnetic Drug Targeting Within the Intracranial Vasculature. *Front. Physiol.*, 9:8225, April 2018.

[12] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.

[13] Michael Schwarz and Hans-Peter Seidel. Fast parallel surface and solid voxelization on GPUs. *ACM Trans. Graph.*, 29(6):1, December 2010.

[14] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.13 edition, 2018.

[15] Long Zhang, Wei Chen, David S Ebert, and Qunsheng Peng. Conservative voxelization. *Visual Comput*, 23(9-11):783–792, June 2007.