# A Pre-Processing Interface for Steering Exascale Simulations by Intermediate Result Analysis through In-Situ Post-Processing

Gregor Matura[a], Achim Basermann[a], Fang Chen[b], Markus Flatken[b], Andreas Gerndt[b], James Hetherington[c], Timm Krüger[c], Rupert Nash[c]

[a] *Distributed Systems and Component Software, Simulation and Software Technology, German Aerospace Center, Linder Höhe, 51147 Köln, Germany*
[b] *Software for Space Systems and Interactive visualisation, Simulation and Software Technology, German Aerospace Center, Lilienthalplatz 7, 38106, Braunschweig, Germany*
[c] *Center for Computational Science, University College London, 20 Gordon Street, London, WC1H 0AJ, United Kingdom*

**Abstract**

Today's simulations are typically not a single application but cover an entire tool chain. There is a tool for initial data creation, for partitioning this data, for actual solving, for afterwards analysis, for visualisation. Each of these tools is separated and so is the data flow. This approach gets unfeasible for an exascale environment. The penalty for every data movement is extreme. A load balance optimised for the solver most likely does not hold true for the complete run time. Compared to tweaking each part, our solution is more disruptive: Merge the tools used and thereby improve overall simulation performance. In this paper, we provide a first step. We combine pre-processing, simulation core and post-processing. We outline our idea of a general pre-processing interface steering the overall simulation and demonstrate its applicability with the sparse geometry lattice-Boltzmann code HemeLB, intended for hemodynamic simulations.

The tasks of the pre-processing tool developed start with partitioning and distributing simulation data. Main aspect is to balance computation and communication costs of the entire simulation by using the costs of each simulation part. Measurement of these costs for each simulation cycle makes possible further performance improvement: Data can be redistributed in between cycles in order to achieve a better load balance according to these costs. Additionally, the pre-processing interface developed offers the possibility of integrating extensions covering, e.g., an automated mesh refinement or fault tolerance awareness. Finally, we investigate the applicability of our interface within HemeLB.

*Email addresses:* gregor.matura@dlr.de (Gregor Matura), achim.basermann@dlr.de (Achim Basermann), fang.chen@dlr.de (Fang Chen), markus.flatken@dlr.de (Markus Flatken), andreas.gerndt@dlr.de (Andreas Gerndt), j.hetherington@ucl.ac.uk (James Hetherington), t.krueger@ucl.ac.uk (Timm Krüger), rupert.nash@ucl.ac.uk (Rupert Nash)

We exploit the integrated partitioning methods and especially consider latest HemeLB-specific in-situ result analysis and visualisation methods.

---

**Contents**

## 1. Introduction

A simulation can be logically divided into several parts, pre-processing, simulation core, and post-processing are examples. So far, i.e. up to the petascale regime, these parts are often separated strongly, e.g., by large IO operations or even by separation into different programs. Such cuts in the simulation work and data flow are not alone very expensive but become unbearable when core counts rise above hundreds of thousand; they have to be overcome if a simulation wants to perform in an exascale environment.
The data flow in a typical simulation cycle starts with a proper initialisation of simulation data. Primordial simulation data, like a mesh of the simulated object or coordinates of the blood vessel boundary, is read into the memory where it is further processed. It is obvious that this step is inevitable. Yet it will be necessary to improve the situation by reducing the number of IO operations where possible. This could be achieved, e.g., by loading only a coarse mesh or data set and extrapolate a fine version in the next part of the simulation cycle. After the initial read-in mesh manipulation techniques can be applied. It may be necessary to refine the mesh in particular spots to force a more accurate solution there. Or alter the shape of the mesh so it fits the real geometry better. Or general smoothing could be applied. However, in an exascale regime, this mesh manipulation has to be automated and thus relies on experience or measurements from former simulation runs.
When all simulation input data is prepared the solver could be applied if there would not be one hitch to it. After the fairly random initial read and all applied mesh manipulations the data may not be aligned properly, i.e., an immediate solving procedure could lead to an unequal utilisation of the available computing, memory and communication resources. For an exascale system, this situation becomes much more severe as, in the worst case, hundred thousands of cores are waiting for just one to complete. Hence, the most crucial task of pre-processing is to balance the load. Additionally, it is important to consider the complete simulation cycle with all of its components; if, for example, an in-situ visualisation is integrated that may impose uneven and additional work load on the system, this cannot be neglected.
The calculation core of large-scale simulations produces huge data sets. Conventional post-processing relies on output data which is stored on disk. For exascale simulations, it will be no longer possible nor efficient to write out visualisation data to disk. Instead, in-situ data visualisation and analysis are giving promising solutions for data post-processing. In-situ post-processing extracts and analyses simulation output on the fly, providing the simulation experts with possibilities to explore and modify simulation parameters at run time. Choices on data structure and visualisation techniques have to be co-designed with the pre-processing and simulation scientists, in order to minimise overall latency and to achieve efficient and interactive data processing and analysis.
In total, we identify a delicate and common point: All simulation parts cannot stay apart and have to grow together or an exascale system will not be exploitable to its full performance. Since every part remains responsible for

its particular task, and this should stay unchanged from a software engineering point of view, it is necessary to establish a communication of information between all parts of a simulation cycle. Once each part is aware of the others and capable to use those additional yet performance-vital information the simulation is enabled to use the full potential of an exascale system.
This is where our work starts: We establish an interface between core simulation, where a solution is calculated, and post-processing in order to enable efficient in-situ visualisation techniques. This helps to avoid expensive off-system operations to process visualisation data. An interface between pre-processing and other simulation parts, called PPStee, manages overall simulation load-balance which is its primary task. It can be extended by further steering capabilities covering other important exascale-relevant duties like automated mesh manipulation or fault tolerance.
Our primary test bed is the simulation code HemeLB. It is intended for hemodynamic simulations and was developed at the Centre for Computational Science at University College London. The core of HemeLB provides a lattice-Boltzmann solver for blood flow simulation with sparse geometries. Since the geometry of an artery, e.g., is read in, processed, simulated and illustrated, HemeLB is particularly suited for exemplarily optimising pre- and post-processing of a simulation.
In this paper we introduce a first prototype interface for pre-processing steering named PPStee and point out its integration within HemeLB and HemeLB's recent visualisation methods. After some remarks on previous work in section 2, we name vital points exascale computing imposes on simulations and their pre-processing and post-processing (see section 3).
In section 4, we present PPStee. This software integrates in the simulation cycle and is fed with graph or mesh data and various communication costs and work load parameters from all simulation loop components. It uses state-of-the-art partitioning libraries to provide an overall simulation load-balance and can be extended with further functionality like mesh manipulation methods or connection to a fault tolerance framework.
We sketch features and properties of PPStee and show advantages and disadvantages of its architecture. We illustrate the integration into a generic simulation work flow in terms of both data flow in combination with PPStee and actual implementation using a basic usage example.
Section 5 points out advantages and disadvantages of PPStee. The specific integration of PPStee into HemeLB and its latest in-situ visualisation methods is covered in section 6. We end with concluding words on the prototype software PPStee, its usage in HemeLB and some prospects.

## 2. Related work

### *2.1. Partitioners and pre-processing*

ParMETIS [1] is an MPI-parallel and highly scalable extension of METIS [2]. It provides algorithms for partitioning unstructured graphs and meshes as well

as for computing fill-reducing orderings of sparse matrices. ParMETIS implements a parallel multilevel k-way method for graph-partitioning and adaptive repartitioning. It is particularly suited for dynamic multi-phase simulations because of its multi-constraint graph-partitioning algorithm. [3]
Scotch's [4] purpose is to apply graph theory to scientific computing problems. It pursues a divide and conquer approach to achieve graph and mesh partitioning, static mapping and sparse matrix ordering. It implements programs and libraries performing these tasks. Scotch claims a running time linear in the number of edges of the source graph and logarithmic in the number of vertices of the target graph for mapping computations. PTScotch [5] uses the MPI interface to provide a parallel optimised version of Scotch.
The Zoltan [6] library focuses on load-balancing, data movement, unstructured communication, and memory usage difficulties in parallel dynamic applications. It provides an object-based interface to simplify its usage and imposes no restrictions on application data structures thus improving its flexibility. Zoltan implements various tools for parallel applications: a suite of parallel partitioning algorithms and data migration tools, distributed data directories and unstructured communication services, dynamic memory management tools. Two classes of parallel partitioning algorithms are implemented. Geometric methods cover recursive coordinate or inertial bisection, Hilbert space-filling curves and refinement tree partitioning. Topology-based methods are applied in the graph and hypergraph partitioning.

### *2.2. Visualisation and post-processing*

A few leading research groups are dealing with data post-processing for petascale and exascale simulations. A system study is presented by Childs [7] which discussed possible challenges and solutions for petascale post-processing. Ultra-scale visualisation has been the main research focus of the SciDAC institute [8], which aims at advancing visualisation techniques to enable knowledge discovery at extreme-scale. Among visualisation and simulation community, in-situ processing have raised great attention in the recent years. A new library has been introduced to *VisIt* by Whitlock et. al [9], which allows full featured in-situ visualisation.

### *2.3. Lattice-Boltzmann and HemeLB*

The Centre for Computational Science at University College London develops, amongst other projects, scientific applications of HemeLB in the field of blood simulation. The core of HemeLB provides a lattice-Boltzmann solver for flood simulation with sparse geometries [10]. A prepartitioning library ParMETIS is employed to decompose computational domains in order to allow parallelisation. Recent investigation of HemeLB performance has shown that it can scale well to at least 32 thousand cores with more than 81 million lattice sites. (cf. [10])

## 3. Major challenges of pre- and post-processing at exascale

### *3.1. Pre-processing*

The main target of pre-processing certainly is an overall simulation load-balance. All simulation costs must be included in the calculation of a load-balanced partition to guarantee best-possible system performance. Computational work load and communication costs do not only arise in the simulation core, i.e. the solver and its associated tasks, but also in other parts surrounding a simulation. Input data preparation, post-processing and (remote) rendering move closer to the simulation core and may be performed on the same computer system as the simulation core, especially in the exascale regime.
This has to be addressed properly by a tighter coupling of pre-processing within the simulation cycle. Pre-processing cannot be seen as an external pre-simulation component any more. Simulation-intermediate interaction is necessary.
To facilitate this new role of pre-processing on a contemporary exascale system, an exchange of information with the simulation core, post-processing and other simulation parts is needed. For this, suitable interfaces must be designed and installed. They take care of passing useful data between these components and introduce options of steering. This enables direct performance gains of each part because of better adaption to the input data.
As the simulation parts merge, new capabilities emerge. For example, the transition from a run-and-stop fashion to a repeating simulation loop provides immediately the possibility of an optimised repartitioning. Timing measurements of the old cycle can be used directly to improve the partition quality of the following cycle and additionally determine whether it is reasonable to redistribute the data in this cycle in the first place. In contrast to a non-interactive simulation, calculation costs may vary in between cycles and so has the partitioning.
For the pre-processing interface, a suitable data format has to be specified. Main constraint is a minimal amount of data yet sufficiently large to retain all details of the input data. This ensures both low communication times, when data is transferred, as well as a minimal memory footprint.
Based on this simulation input data and its layout, pre-processing should provide an algorithm properly adjustable to the unique simulation data structure and its needs. It may be necessary to compare different load-balancing methods in terms of scalability and performance of the resulting partitioning.

### *3.2. Post-processing*

A few issues pose challenges to post-processing at exascale:

- Data can not be stored on disk. post-processing will have to access the data directly from the simulation nodes and assign memory for it. What data to process and how to fit them into memory will be an issue at exascale.

- What kind of data structure will be the most efficient? Shall we follow the same data structure as used in simulation? On the one hand, using the

same data structure avoids copying or moving data around. On the other hand, this data structure can be inefficient for visualisation computations.

- Parallelism in post-processing will be a major challenge in providing in-situ post-processing. Whether we should use a load-on-demand approach or a static one heavily depends on the choice of the visualisation algorithms. How to optimise parallelisation and minimise communication latency will be another major issue.

### *3.3. HemeLB*

There is increasing evidence that the development of cardiovascular diseases, such as aneurysms, is caused by certain blood flow patterns. As intracranial aneurysms are quite common (1 - 5% of the entire population are affected), it is desirable to understand their formation and which particular aneurysms are at enhanced risk of rupture. HemeLB is an application for simulation of blood flow in the larger arteries. Its ultimate goal is to contribute to patient-specific treatment, e.g., real-time risk assessment of cerebral aneurysm rupture. To this end, HemeLB has to produce reliable predictions on the time scale of one hour. Additionally, due to the high resolution required in time and space, HemeLB has been designed as a highly parallelised algorithm.

## 4. PPStee: introduction

PPStee is an interface for pre-processing steering. It ships as a library and is implemented in the pre-processing phase of a simulation. Supplied with information on simulation data, its main purpose is to optimise the overall simulation load-balance starting with initial data distribution and not necessarily ending after visualisation of the simulation results.

### *4.1. Properties*

PPStee is built around various partitioning tools, namely ParMETIS [1], PTScotch [5] and Zoltan [6]. These are established and widely used libraries. They provide partitioning capabilities which are mostly congruent among each other. Each of them can be used to retrieve a decent load-balance for a simulation. Yet, they have been developed independently and use different approaches to compute the partitioning. This leads to a partitioning of different quality depending on the input data and therefore on the simulation. Now, PPStee offers an easy-to-implement possibility to swap the used partitioning tool by only slight changes in the code. Doing so, the obtained timings can lead to a better choice of partitioner for the simulation at hand and therefore directly to an improved load-balance.
The data format of PPStee orientates towards compatibility and a minimal footprint. The possibility of direct data access without additional copy operations and a minor overhead for internally used data improves memory consumption.

The overhead is not entirely needed but can save cost-intensive collective communications. These can occur if a conversion from one to another of the native partitioner data format is calculated. Nevertheless, the PPStee data format is designed to be capable of this conversion and to do it fast and cheap.
PPStee's main task is to balance the simulation load. This is not a new approach; as a matter of fact all mentioned partitioners do so. Yet, PPStee provides a disruptive feature: it incorporates different simulation stages by default. In this way, not only the computation costs of the simulation core are balanced. Other parts, like visualisation, can provide their calculation and communication cost parameters for load balancing in addition. These parts are naturally present in exascale simulations due to the unfeasibility of ex-situ processing of the huge data amounts. A true simulation-covering load-balance is gained.
The modular architecture and the flexible data format make PPStee easily adjustable. New partitioning tools, whether they are developed directly in PPStee or stand-alone, can be integrated with minor effort. Further kinds of stages can be added if the need arises. PPStee offers several places to introduce fault tolerance techniques. E.g., an extension to PPStee could take care of a redundant backup of the graph data which in turn is used to recover lost data if one processor dies. In addition, PPStee could prevent the usage of this processor and simulation can continue almost without delay. Furthermore, mesh refinement routines are conceivable which can alter the submitted graph data automatically or adjust it to the system's structure.

### *4.2. Integration into simulation work flow*

In general, PPStee does not allocate any memory (beside, of course, a tiny amount of private data) and is not responsible for any data movement.[1] The simulation keeps track of the graph or mesh data and the accessibility of this data throughout simulation lifetime. This way the integration of PPStee into an existing code is kept simple and least disturbing for the simulation's data flow. The responsibility for the data belongs completely to the simulation.
The work and data flow is illustrated in 1. The simulation core reads the initial data, i.e., usually some kind of geometry, from the file system and submits this data in form of a graph to PPStee. Additionally, it provides computational work load and communication costs it will use as (graph) weights where work load matches weights for the vertices and communication is mapped onto the edges. These weights can be estimated based on former simulation runs or precise prediction based on a proper investigation of the code. Also, a posteriori measurements should be used to improve the reliability of these figures.
Furthermore, all other simulation components should submit their weights, too, whenever possible. For example, this includes work load of any result data

[1] Obviously, more advanced features like mesh manipulation techniques break this general rule.

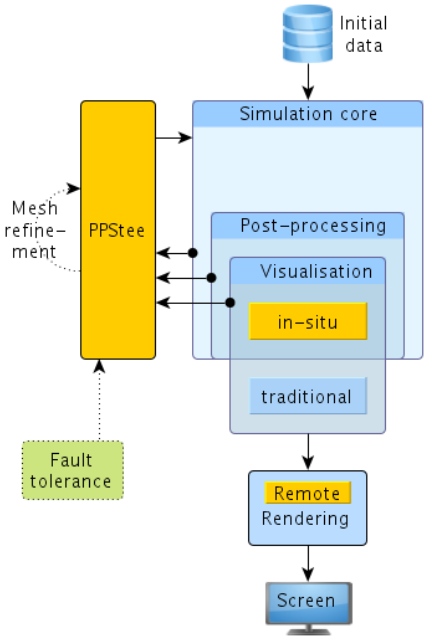


Figure 1: PPStee flow chart

analysis done in the post-processing phase and especially cost-intensive calculation and communication of in-situ visualisation. So, basically, every task done within the simulation code and executed on the cluster should provide its cost to guarantee all-over load-balance throughout the full simulation loop.
Finally, the simulation core retrieves a partitioning either directly after the initial submission of the graph and all corresponding weight estimates or, in subsequent cycles, triggers a calculation of an updated partitioning. This repartitioning should be based on timing and costs measurements of previous cycles and thus is better balanced than the initial partitioning. Due to the data responsibility, the simulation core compares the re-partition to the current partition and decides whether it is worth the effort to move the corresponding data.
In total, the result is a load-balance covering the complete simulation loop. For a single-loop simulation, this result requires estimates from former runs; for a simulation traversing multiple cycles, the result becomes even better using adaption and repartitioning.
Apart from this main data flow, some insertions are possible in the future. Mesh manipulation techniques could be applied after initial graph submission or between cycles. After initial graph submission, the mesh could be smoothed or used to generate a finer mesh. Between the cycles, the result of a previous data analysis in the post-processing component could trigger a refinement of spots in the mesh where this modification leads to a more accurate or faster solution.

Additionally, a fault tolerance framework could interact with PPStee and steer the distribution of data. If, for example, some nodes drop out graph and load data could be adjusted to the new cluster status. Obviously, a decent data backup and recovery mechanism would be required.

### *4.3. Basic usage example*

In this section we describe how to use PPStee based on an example implementation. We assume an existing code that initialises its data and then does a standard ParMETIS call,

```
// a ParMETIS call
ParMETIS_V3_PartKway(
  vtxdist, xadj, adjncy,
  vwgt, adjwgt,
  wgtflag, numflag, ncon, nparts,
  tpwgts, ubvec, options, edgecut,
  part,
  comm);
```

to retrieve a partitioning named `part`. Other partitioners can be used analogously.
We start by initialising a `PPSteeGraph` object with the graph data we have, i.e. `vtxdist` for the global vertex distribution and `xadj` and `adjncy` for the thread-local adjacency structure:

```
// get graph (as ParMETIS type)
PPSteeGraph pgraph = PPSteeGraphParmetis(MPI_COMM_WORLD, vtxdist,
    xadj, adjncy);
```

Next, we construct weights objects derived from the graph as these have to be compatible. We fill in weights for the computation and visualisation part. These weights denote the work load (vertex weights, `vwgt`) and communication time (edge weights, `adjwgt`) each simulation part needs.

```
// construct and set weights for computation
PPSteeWeights wgtCmp(&pgraph);
wgtCmp.setWeightsData(vwgt_c, adjwgt_c);
// construct and set weights for visualisation
PPSteeWeights wgtVis(&pgraph);
wgtVis.setWeightsData(vwgt_v, adjwgt_v);
```

Now, we establish an instance of the interface's main object and submit our graph and weights data.

```
// get interface
PPStee ppstee;
// submit graph
ppstee.submitGraph(pgraph);
// submit weights
ppstee.submitNewStage(wgtCmp, PPSTEE_STAGE_COMPUTATION);
ppstee.submitNewStage(wgtVis, PPSTEE_STAGE_VISUALISATION);
```

Finally, we trigger the calculation of the partitioning and get the desired partitioning.

```
// calculate partitioning
PPSteePart* ppart;
ppstee.getPartitioning(&ppart);
```

## 5. PPStee: analysis

### *5.1. Advantages*

PPStee's main advantage is the standardised partitioner access. Once PPStee's data structures are created and filled with the according graph data the partitioner is chosen arbitrarily. This introduces the option to independently change the partitioner used. Then, timing measurements and other tests can be used to reveal the best-suited partitioner for the simulation.
PPStee relies mainly on established external partitioning tools. Their mature and methodologically sound algorithms are used and provide partitioning at a state-of-the-art level. Additionally, PPStee's basic data containers can be used to manufacture a partitioning routine particularly tailored for the user's needs. Later, a direct integration in PPStee is possible.
PPStee comes with little programming overhead. If a partitioner is already implemented in a simulation the change to make use of PPStee is minimal. PPStee provides function signatures very similar to those native to the partitioners. All data structures can be kept and used as they are making data handover and reception of the resulting partition quite easy.
PPStee requires only a small amount of additional memory. PPStee uses only little auxiliary data for internal book keeping. The full graph data can be passed by reference thus keeping memory obligations at the simulation side. Data access is read-only; whether it can be freed afterwards depends on its usage: stage weights, e.g., should be kept alive if the simulation will do more than one cycle and thus needs a later repartitioning.

### *5.2. Disadvantages*

PPStee accesses only basic routines of the partitioning libraries although most of them provide more extended features which may improve the partitioning quality. This certainly is a side effect of the standardised access. On the other hand, this very access helps to indicate whether a further investigation of these extended features is reasonable and for which of the partitioners it should be done. Also, if a specific extended routine becomes crucial in the future it can be integrated in PPStee belatedly.
Another point to mention is the insertion of another software layer by PPStee. Although this should not negatively affect the simulation, it does increase the complexity and may lead to undesired or faulty behaviour which may get harder to track.

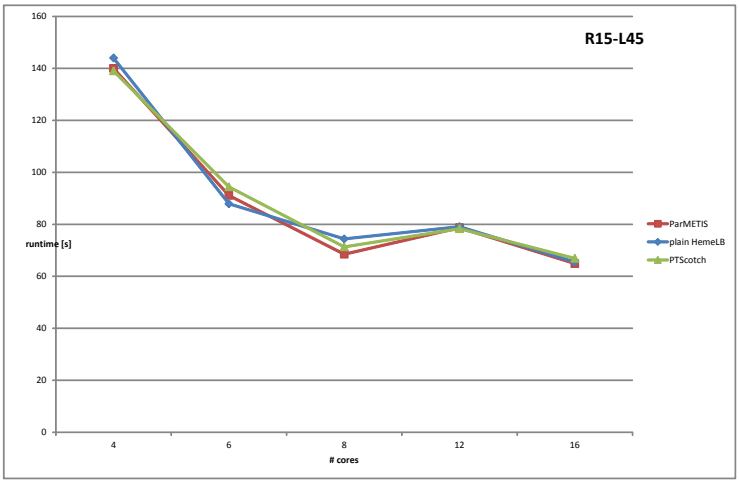


Figure 2: HemeLB runtimes for data set R15-L45 with plain HemeLB, HemeLB with PPStee using ParMETIS and HemeLB with PPStee using PTScotch

## 6. HemeLB and PPStee

The specific integration of the pre-processing steering interface PPStee into a mature code such as HemeLB is straight forward (cf. section 4). The integration is devided into two steps, bare code changes and some additional changes in the build system. In the first step, PPStee substitutes the partitioner call. HemeLB already uses the library ParMETIS to evenly distribute computational domains among all processes. Thus, we have all mesh data set up correctly and we save the trouble to assemble the arrays needed. We simply replace the ParMETIS call as shown in section 4.3 and make sure to include PPStee's header file instead of ParMETIS' header.
Additionally, we adjust HemeLB's build system according to the needs of PPStee. Since ParMETIS is already included, we add PPStee and all other partitioners we want to use to the dependencies of the build target HemeLB. For an update of include and library directory settings, the very entries for ParMETIS are well-used as guides. Put altogether, the code changes are small and it is easy to find the right places, in spite of the size of a project like HemeLB.
For a proof of concept, we have integrated and built HemeLB with PPStee and two partitioners, ParMETIS and PTScotch. Our first test system is a small one and provides an Intel Xeon E5520 with 8 real and 16 virtual cores. We used five different process counts between 4 and 16 and the minimal time of three runs for each data set. Figures 2, 3 and 4 depict the runtime results of the three different HemeLB data sets R15-L45, R15-450 and R30-L900, respectively.

In all three charts we find a close match of runtimes for all three methods, i.e. plain HemeLB (using ParMETIS), HemeLB with PPStee using ParMETIS and HemeLB with PPStee using PTScotch. There are slight performance gains and losses here and there; however, a general tendency with respect to a method or

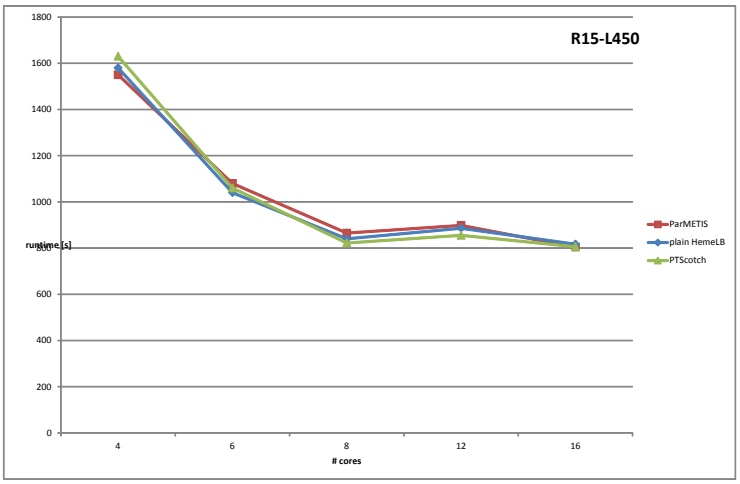


Figure 3: HemeLB runtimes for data set R15-L450 with plain HemeLB, HemeLB with PPStee using ParMETIS and HemeLB with PPStee using PTScotch

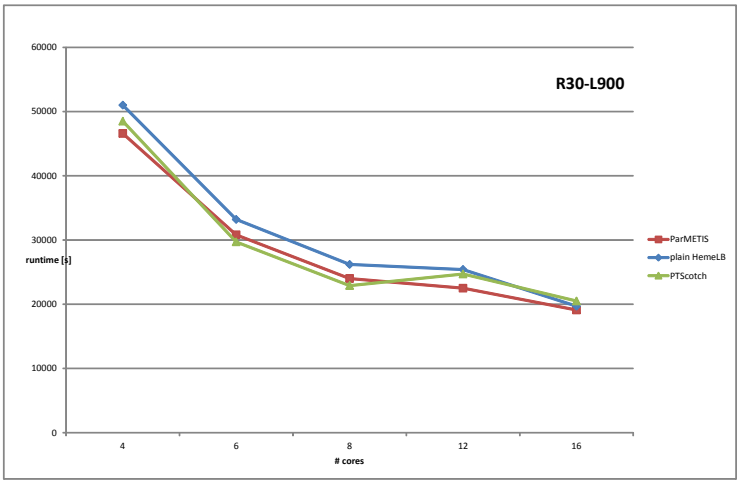


Figure 4: HemeLB runtimes for data set R30-L900 with plain HemeLB, HemeLB with PPStee using ParMETIS and HemeLB with PPStee using PTScotch

process count cannot be identified.
Concluding for low process counts, we see no penalty in runtime when using PPStee. On the one hand, this proofs that PPStee does not introduce any significant overhead in computation time. On the other hand, PPSTee offers the possibility for an improvement in runtime when using another partitioner or other data sets, in particular in case of very large data sets and extremely high process counts.

Post-processing in HemeLB aims at providing interactive flow visualisation techniques which demonstrate flow properties resulting from blood simulations. Common visual representations of fluids include: volume representation, where flow density or other scalar fields are mapped to volumes, line representations such as pathlines and streamlines which illustrate how flow is moving, and topology representations which extracts the hidden information in the flow fields. Depending on what the simulation experts want to see from their simulation output, special visual representations of the data can be implemented.
The challenge with implementing these methods into HemeLB lies in the parallelisation. There are two reasons why parallelisation in visualisation computations is tricky. First, geometry data is already decomposed in HemeLB at the pre-processing stage. Changing data structures or layout creates additional effort in moving data around. Second, simulation results are not stored on disk. When the simulation moves to the next time step, previous results are discarded and therefore no longer available. This poses a challenge in time dependent visualisation computation where information from multiple time steps is needed.
Current HemeLB post-processing allows the user to inspect the flow field at simulation run time. A steering client is implemented which assesses the simulation at the current time step, extracts the simulation output, and performs visualisation computations with the data. Simple examples such as volume rendering of the stress field and the cutting plane which demonstrate the flow field in a cross section are provided. With the current steering client, the user is able to interactively inspect the simulation at run time without writing out data or waiting until the simulation is terminated.
Based on the visualisation output, simulation experts can determine whether and how to modify their simulation process. However, how to steer and what to steer are questions that domain experts in simulation have to answer.

## 7. Conclusion

We introduced the new pre-processing interface PPStee that offers the possibility of balancing the load of all parts of a simulation. PPStee implements access to multiple partitioners and provides an easy way to switch the partitioner used. It supports costs of various simulation parts through weights per separate computation stage and still remains adaptable for future developments in partitioning algorithms.
PPStee is easy to integrate even in large-sized codes like HemeLB. With HemeLB data sets and test runs, we demonstrated that PPSTee introduces almost no

overhead in terms of simulation runtime.
Furthermore, we integrated an additional steering client into HemeLB for in-situ post-processing that extracts the simulation output and performs visualisation computations. It is compatible with PPStee and enables the user to interactively inspect the simulation at run time.

## 8. Future work

We are currently working at HemeLB experiments with large data sets on supercomputers in order to obtain runtime measurements of PPStee with thousands of cores. This will help to emphasise the negligible overhead of PPStee and the scalability of the data format used.
Meanwhile, the partitioning library Zoltan is being integrated into PPStee and the next version of PPStee will support Zoltan in addition to ParMETIS and PTScotch. This extension enables a comparison of three distinct partitioning algorithms so that simulation developers can identify the best partitioner for their specific data set.
Concerning post-processing, two important things are left on the agenda. First, the proposed visualisation techniques must be tested at a larger scale with big datasets, in order to answer the question if they are scalable or not. Second, based on the visualisation output, parameters of the simulation which require steering have to be identified and further steering functionality has to be implemented.
Finally, we intend to apply the pre-processing interface as well as the steering client to other simulation applications that make extensive use of pre-processing large input data sets as well as analyse and visualise huge result data sets.

## 9. Acknowledgement

The authors would like to thank all our colleagues from the CRESTA project for their support and for many lengthy and fruitful discussions without which this work would not have been possible. The support of the European Commisssion through the Seventh Framework Programme (ICT-2011.9.13) under Grant Agreement no. 287703 is gratefully acknowledged.

## References

[1] ParMETIS, parallel graph partitioning and fill-reducing matrix ordering. URL `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`

[2] G. Karypis, V. Kumar, A fast and highly quality multilevel scheme for partitioning irregular graphs, SIAM Journal on Scientific Computing 20 (1) (1999) 359–392.

[3] K. Schloegel, G. Karypis, V. Kumar, Parallel static and dynamic multi-constraint graph partitioning, Concurrency and Computation: Practice and Experience 14 (3) (2002) 219–240.

[4] F. Pellegrini, J. Roman, Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs, in: Proceedings of HPCN'96, Brussels, Belgium, LNCS 1067, Springer, 1996, pp. 493–498.

[5] C. Chevalier, F. Pellegrini, PTScotch: a tool for efficient parallel graph ordering, Parallel Computing 34 (6–8) (2008) 318–331.
URL `http://www.labri.fr/perso/pelegrin/scotch/`

[6] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, C. Vaughan, Zoltan data management services for parallel dynamic applications, Computing in Science and Engineering 4 (2) (2002) 90–97.
URL `http://www.cs.sandia.gov/Zoltan/Zoltan_phil.html`

[7] H. Childs, Architectural challenges and solutions for petascale postprocessing, Journal of Physics: Conference Series 78 (1) (2007) 012012.

[8] K.-L. Ma, In Situ Visualization at Extreme Scale: Challenges and Opportunities, Computer Graphics and Applications, IEEE 29 (6) (2009) 14 –19. doi:10.1109/MCG.2009.120.

[9] B. Whitlock, J. M. Favre, J. S. Meredith, Parallel in situ coupling of simulation with a fully featured visualization system, in: Proceedings of the 11th Eurographics conference on Parallel Graphics and Visualization, EG PGV'11, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2011, pp. 101–109.

[10] D. Groen, J. Hetherington, H. B. Carver, R. W. Nash, M. O. Bernabeu, P. V. Coveney, Analyzing and Modeling the Performance of the HemeLB Lattice-Boltzmann Simulation Environment, CoRR arXiv:1209.3972.