D5.2.6: Post-processing: tool evaluation and investigation with application data

WP5: User tools

Project Acronym	CRESTA
Project Title	Collaborative Research Into Exascale Systemware, Tools and Applications
Project Number	287703
Instrument	Collaborative project
Thematic Priority	ICT-2011.9.13 Exascale computing, software and simulation

Due date:	M39
Submission date:	31/12/2014
Project start date:	01/10/2011
Project duration:	39 months
Deliverable lead organisation	DLR
Version:	1.0
Status	Final for submission
Author(s):	Markus Flatken (DLR), Fang Chen (DLR)
Reviewer(s)	Xavi Aguilar (KTH), George Mozdzynski (ECMWF)

Dissemination level	
PU	*PU - Public*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	27/11/14	Initial draft	Markus Flatken (DLR), Fang Chen (DLR)
0.2	10/12/14	Revised draft taking into account reviewers' comments and suggestions	Xavi Aguilar (KTH), George Mozdzynski (ECMWF)
1.0	15/12/14	Final draft for submission	Catherine Inglis (UEDIN)

Table of Contents

1 Executive Summary

In this deliverable, we present the evaluation of our developed in-situ post-processing software infrastructure designed for enabling data exploration and post-processing towards exascale.

As the last deliverable for work package 5.2, we give a detailed review of the software architecture, and evaluate our in-situ post-processing algorithms during an on-going simulation. We focus on evaluating the time needed to prepare the in-situ operations, for extracting features, and for rendering. We investigate the scalability and interactivity of the proposed in-situ processing tools. All proposed methods have been evaluated using HemeLB as the core application for the in-situ analysis.

2 Introduction

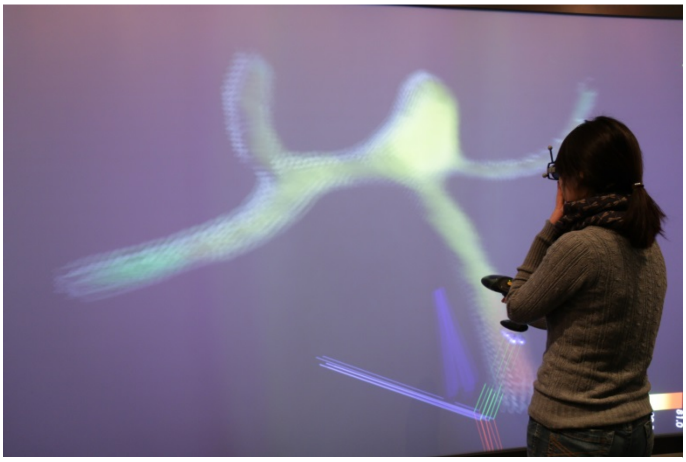

Figure 1: A scientist is interactively exploring the blood flow of an aneurysm during an on-going HemeLB lattice Boltzmann simulation

The ever-increasing compute capacity of HPC hardware systems enables scientists to simulate and explore physical phenomena with an incredible spatial and temporal accuracy. This high accuracy also leads to dataset sizes of many terabytes, petabytes and even exabytes if we think about the upcoming exascale area expected in 2018[1]. To obtain knowledge from these advanced simulation datasets an efficient analysis and visualization process becomes increasingly important but also difficult. In particular, explorative scenarios where the users continuously change input parameters to the analysis process pose tremendous challenges to the underlying compute and software system. These parameter changes demand a re-execution of all compute and data-fetching operations. Thus, the challenges are related to all steps of a classical visualization pipeline (as shown in Figure 2 on page 8):

1. Filtering: An operation which extracts meaningful features out of the huge quantity of raw data.
2. Mapping: An operation which maps the extracted features to visualization primitives such as points, lines or polygons.
3. Rendering: Generates a 2D image from the extracted primitives based on user defined camera parameters.
4. Display: A final stage which simply displays the generated 2D image on the screen.

A common way to speed up the filtering and mapping stages of this visualization pipeline is data parallelism. This technique is used massively by the two widely-utilised open-source visualization tools ParaView [2] and VisIt [3]. Thereby, each process executes at least the filtering operation only on a fraction of the complete dataset. Partial results are collected later on in a subsequent pipeline stage.

Due to the enormous spatial accuracy, high resolution displays are required to recognize tiny but significant details. While classical display systems often do not provide a sufficient resolution in all application cases, tiled-display systems have proven to support the visualization of small details with ultra-high resolutions up to tens or hundreds of mega pixels [4]. However, the local resources are often insufficient to process and render the still large amount of features extracted during the filtering and mapping operations. One solution is to use parallel rendering techniques. These parallel rendering techniques also use the data parallelism approach. Therefore, each processor or graphics processing unit (GPU) only renders a part of the total geometry. An early classification of these parallel rendering techniques is given in [5]. Today two major frameworks are common to develop parallel rendering engines, namely the IceT [6] and Equalizer framework [7]. IceT is a sort-last rendering solution for tiled-display systems. The main advantage of the sort-last rendering is that this approach is scalable with respect to the size of data, but the major bottleneck is the required network bandwidth to combine all partial images to a final result. Due to this limitation this technique often prohibits interactive frame rates necessary when using e.g. virtual environments and tiled-display systems.

Therefore, hybrid rendering methods which render with varying frame rates on local and remote resources have been developed. Moreland et al. presented an image-based rendering approach which generates local images with the help of a remotely generated unstructured lumigraph [14]. Wagner et al. presented an image-based rendering approach optimized for hybrid rendering using tiled-displays [11]. Noguera et al. applied this technique to navigate through large terrains on commodity mobile devices [15]. The terrain closely located to the user is rendered on the mobile device while the terrain far away is rendered remotely into a sky box which is then sent to the mobile client. Thus, hybrid rendering is an important key feature to guarantee interactivity on the user's front-end which would seem to be unfeasible just using sort-last rendering at high image resolutions.

Another critical fact which often prohibits interactive exploration in a conventional post-processing setup is the limited data I/O bandwidth. The simulation data has to be initially loaded from disc which requires a significant amount of time. Furthermore, when running an exascale simulation it is also questionable whether storing the complete dataset is possible or even necessary. In-situ processing, compared to conventional post-processing, allows scientists an early inspection and analysis of an on-going simulation and enables domain experts to obtain first insight into their running simulation process and intermediate results. Such in-situ processing has the advantage of keeping data in memory, avoiding the storage of large raw data to disk, providing on-the-fly analysis, and eventually preventing early failures in the simulation process.

In this deliverable we present our distributed and scalable software infrastructure, which provides distributed in-situ data processing, feature extraction and interactive exploration at the user's front-end. Therefore, we couple a large-scale cluster system executing the filtering and mapping operations concurrently within each simulation process, a remote GPU cluster for parallel rendering and a high resolution tiled display system for presentation (depicted in Figure 1). In contrast to common scientific visualization frameworks we use the hybrid rendering approach to guarantee interactive frame rates. We integrated and extended an existing post-processing application to allow direct visualization of an on-going HemeLB simulation. Finally, an interactive user front-end has been developed which enables scientists to directly interact with the visualization of a running simulation, gain insight, and make decisions.

The aim of this deliverable is to present an evaluation of the software-system and its algorithms and tools which we designed and developed for exascale in-situ processing applications. In section 2, we explain the overall system and software architecture. In Section 3 we evaluate the performance of our approach based on real world scenarios and in Section 4 a conclusion and outlook is given.

3 System and Software Architecture

Figure 2 proposes our chosen software architecture which can either consist of a two- or three-tier architecture depending on the available hardware resources and the overall size of the simulation data.

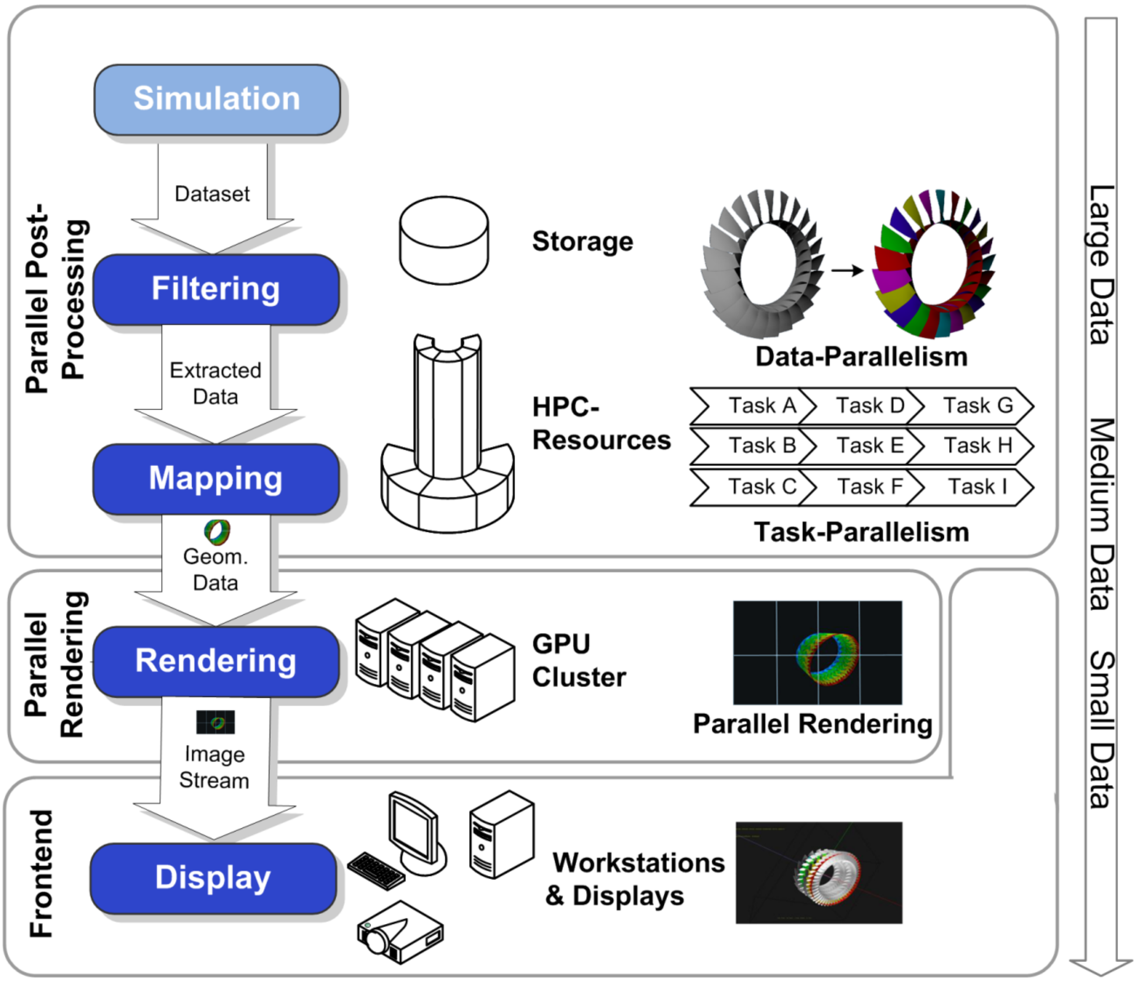

Figure 2: Distributed post-/in-situ processing infrastructure. The visualization pipeline is either distributed in a two- or three-tier fashion. The parallel post-processing running on an HPC hardware system extracts important features, mapped to geometric primitives. These primitives are transferred to a GPU cluster for parallel rendering. A front-end application only receives an image stream from the parallel rendering application. When using a two-tier architecture, the rendering and display stage is executed locally.

The system contains of three major parts. First, in-situ data processing and parallel feature extraction takes place on a large-scale cluster system executed concurrently with the ongoing simulation. Next, depending on the size of the extracted features as well as the complexity of rendering algorithms, rendering is performed either on a smaller-scale GPU-cluster or directly on the local front-end machines. Thirdly, the resulting images are displayed either on a single desktop or in a virtual environment accompanied by different types of user interaction techniques.

In such a distributed software infrastructure, communication and data transfer bandwidths are essential factors to enable interactivity and scalability. The data must be exchanged with minimal latency between different software layers, applications, and resources. To increase the bandwidth between the first and the second tier of our proposed software infrastructure, the data transfer is parallelized with a *m* to *n* connection between the involved resources. *m* denotes the number of processes doing the filtering and mapping operations on the simulation data and *n* is the number of processes performing the rendering step. The transfer between the second and third tier uses a *1* to *1* connection. Therefore, the master process of the parallel rendering application sends the images to the front-end application. The overall size of data is reduced during each of the pipeline stages, so that each subsequent stage is able to handle and process the incoming data.

The user, interacting at the front-end application, is able to adjust the parameters for the feature extraction process which demands a data exchange with the simulation solver and a re-execution of the specified filtering and mapping algorithms. Each process sends the extracted feature results (geometry data in our case) directly to the rendering stage. This data streaming approach enables a quick user response time; the time expired until first results are visible to the user. This is an advantage to common scientific visualizations tools which typically wait until all processes finished processing. Each rendering resource receives some extracted features and prepares them for rendering. When the rendering process is finished, the image including its colour and depth buffer is transmitted to the front-end application where it is finally drawn to screen. Both buffers are required due to our image-based hybrid rendering approach. To enable a consistent local and remote view position and direction, camera updates are continuously sent to the rendering stage of the visualization pipeline.

3.1 Parallel Post-Processing

Our integrated post-processing application is a parallel program based on the Viracocha [8] middleware layer, which itself is based on the message passing interface (MPI). We extended this framework to be more flexible for new algorithms and optimized scheduling development. Furthermore, we enhanced the software with interfaces allowing for data exchange between the simulation and the post-processing application and thus enabling in-situ processing. As depicted in Figure 3, this application consists of a single ***scheduler*** receiving messages via a TCP/IP socket from a front-end application and *n* ***worker*** instances doing the processing. On top of each worker, an ***algorithm layer*** allows developers to integrate their own algorithms for specific problems such as stream-surfaces, topological methods or data compression algorithms. Furthermore, each worker has their own ***data manager*** and ***streaming interface***.

The Viracocha framework itself consists of three libraries:

- **ViracochaBackend**: A library which provides the development of parallel post-processing or in-situ processing applications.
- **ViracochaFrontend**: A library for building a front-end application. It includes the TCP/IP sockets for communication with the parallel post-processing back-end and techniques to handle received data.
- **ViracochaBase**: A library which includes basic data items and message types. This library is also used by the other two libraries.

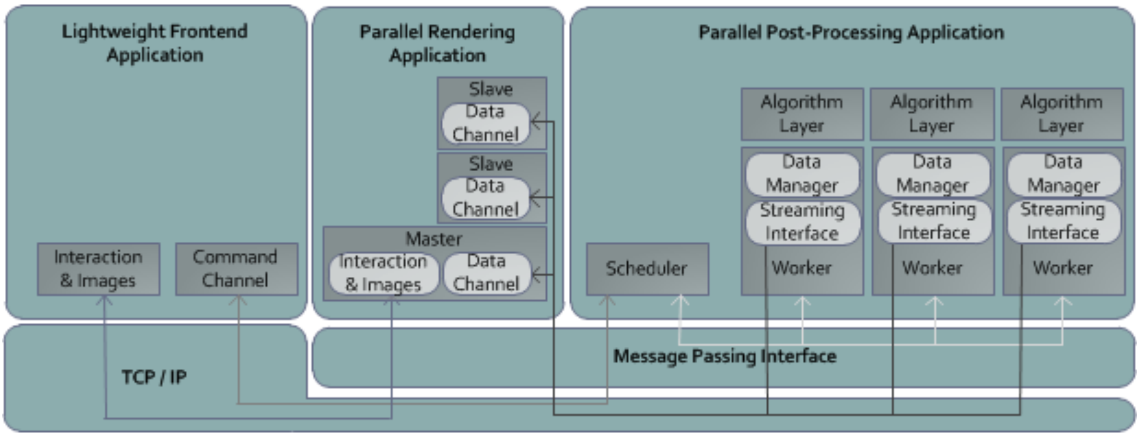

Figure 3: The interactive front-end application can send algorithm requests to the scheduler of the parallel post-processing application. The scheduler decomposes and distributes the work to the given number of workers via the message passing interface. These workers execute the requested algorithm. The algorithm can use the data manager functionalities to handle data I/O and resulting data can be transferred to the parallel rendering application via the streaming interface. Interaction commands and images between the lightweight front-end application and the parallel rendering application are transferred via TCP/IP.

The Viracocha framework requires all algorithms, strategies, and message types used later on in the distributed post-processing application to be implemented and registered to Viracocha. This ensures that the application is able to understand incoming messages from the interactive front-end. The post-processing application is then capable of creating and starting an algorithm with a given parameter set, or of terminating a running job and releasing resources.

When a compute message is received, the ***scheduler*** decomposes the job into small independent work items and pushes them into a so-called ***work table***. A single ***work item*** contains the algorithm id, data parameters, and algorithm parameters for execution. For each registered algorithm specific strategies are required to handle incoming compute requests. For this, Viracocha distinguishes between three different processing aspects the user can assign strategies to:

- **Decomposition Strategy:** This aspect specifies how the job is decomposed into smaller work items. They are all stored separately into the work table.
- **Assignment Strategy:** This specifies the strategy to assign work items of the work table to idle workers for processing.
- **Reduction Strategy:** This aspect is optional and specifies how to collect data after the processing is finished.

The ***streaming interface*** of Viracocha is used by algorithm developers to transfer the extracted features directly to the parallel rendering application. This decreases latency by avoiding the reduction operation and improves response times due to the immediate transfer of partial results. Furthermore, this streaming is performed in a concurrent thread via TCP/IP sockets which leads to an additionally increased performance due to task parallelism of computation and network transfer. A problem that can occur when transferring this data to the rendering nodes is that the workload and size of data can be unevenly distributed over the rendering nodes. This imbalanced data distribution can lead to varying processing times on the rendering nodes. As a consequence, the rendering performance will decrease. Therefore, Viracocha allows the integration of user-defined distribution strategies, which define how data is transferred to the parallel rendering processes. We have implemented and used a static distribution over the given number of rendering nodes to have data roughly balanced over the rendering instances.

As I/O performance is often a bottleneck when processing large-scale datasets, it is important to provide a data management mechanism to the developers. Therefore, we have implemented and integrated a ***data manager*** interface that can be used by algorithm developers to load, cache, or prefetch specific chunks of data. As the file-system's bandwidth in comparison to the aggregated network bandwidth of *n* compute nodes can be poor, we have integrated not only a local but also a global caching system into the ***data manager***. This functionality allows ***workers*** to access data also cached on other ***workers***. To dispatch the needed information, all ***workers*** share their cache information with the other ***workers*** as soon as the work table is completely processed. To make this approach more adjustable, the ***data manager*** component of Viracocha is configurable. It allows the cache size on the ***workers*** to be specified, as well as whether local or global cache access is enabled.

3.2 Coupling Viracocha with HemeLB for In-Situ Processing

Since the Viracocha framework was originally designed for classical post-processing purposes we first extended this framework and in a second step we integrated it into the HemeLB source code to allow for in-situ processing. The major obstacle was the possibility to exchange data between both applications. To allow this data exchange we had to extend the Viracocha data manager with a HemeLB specific data loader. Since HemeLB does not use a grid-based solver, and available algorithms within Viracocha are based on VTK [12] requiring a grid as input data, we had to do a conversion step. Therefore, we chose to create an initial unstructured grid where every quadric cell within the grid represents exactly one lattice within the HemeLB geometry. Depending on the lattice type this could be easily adapted to multiple cells per lattice. This initial grid creation is only executed once during the HemeLB initialization phase. This also means that an extra amount of memory per MPI process is allocated for the in-situ processing. The next task was to update the field values on the vertex points of each cell. Since cells are usually connected and share vertices on the corners, we calculate an average value on these vertices based on all input values. The field update is only performed when a user requests a feature extraction. This is done to avoid disturbing or impacting on the simulation behaviour unnecessarily. The next task was to synchronize both applications. The field data cannot be updated while HemeLB is actually changing values on the lattices. Therefore, a mutex exploited by HemeLB and the Viracocha data manager thread is used. The mutex is locked in the HemeLB main simulation loop before every iteration step and unlocked afterwards. This means data is only exchanged when a HemeLB iteration is finished.

With the previous enhancements Viracocha is able to access the HemeLB simulation data safely and use it for successive analysis algorithms. As mentioned in section 3.1 Viracocha further needs strategies defining how to deal with incoming feature requests. Therefore, we implemented a specific decomposition strategy which exactly creates one work item for each worker. The assignment is done via an already existing first in first out assignment strategy.

The last functionality to add was that both applications had to run concurrently within the same process space. Therefore, either the Viracocha scheduler or one of the workers are executed as a separate thread within each HemeLB process. Since both applications use MPI communication mechanisms we had to create a communicator specifically for Viracocha. This ensures safe communication but also requires an MPI implementation that allows fully multi-threaded MPI communication.

Figure 4 depicts the HemeLB integration, communication and data flow. The Viracocha master is executed as a thread inside the HemeLB master process. Each Viracocha worker is executed within a HemeLB slave. The data is exchanged via the Viracocha data manager. This uses the HemeLB data reader to read the lattice information and to update the field values.

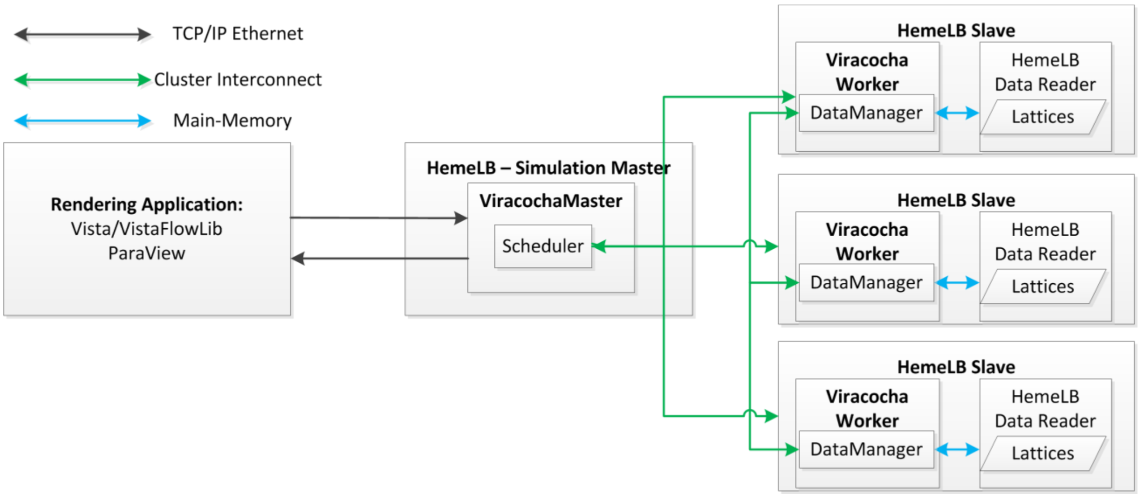

Figure 4: Communication and data flow: Every Viracocha master and worker instance is embedded into HemeLB as a concurrent thread. The Viracocha scheduler is responsible for receiving user requests from the front-end and initiating the algorithm execution on the Viracocha worker instances. The data manager within the workers has direct access to the simulation data in main-memory and is able to communicate over the cluster interconnect with similar worker threads to exchange data when required.

A user can send feature extraction requests to the Viracocha master. The scheduler executes our developed decomposition strategy and assigns the created work items to the idle workers. The workers use the data manager interface to access the HemeLB data. As soon as the features are extracted, results are streamed to the rendering application for immediate presentation on the user's front-end.

3.3 Parallel Rendering System and Front-end Application

Features extracted from a large-scale simulation in fact are much smaller than the raw simulation data itself but they can still be large. The second-tier parallel rendering system has to be efficient enough to take the data from the network stream, to deserialize the raw packages to VTK data objects, and use them to update the rendering data. The challenges, however, are high compute demands for the real-time processing of the incoming geometry data and huge memory requirements for the rendering. This was the main motivation for the design of a three-tier architecture with a high-performance rendering stage as the second tier. The rendering application is based on ViSTA [9], ViSTA FlowLib [10] and the IceT image compositing library. ViSTA is used because of its flexible interaction and display opportunities. ViSTA FlowLib is an add-on to enable time management of unsteady simulation data and provides rendering methods specially designed for flow visualization. However, the core purpose is parallel rendering managed by the IceT framework. It is capable of dividing the compute and memory requirements to a scalable number of rendering nodes, reducing the load of each single node.

To facilitate efficient data throughput, all core tasks of the parallel rendering application are performed concurrently. Besides the main rendering thread, a network thread and a pool of processing threads are executed (cf. Figure 5).

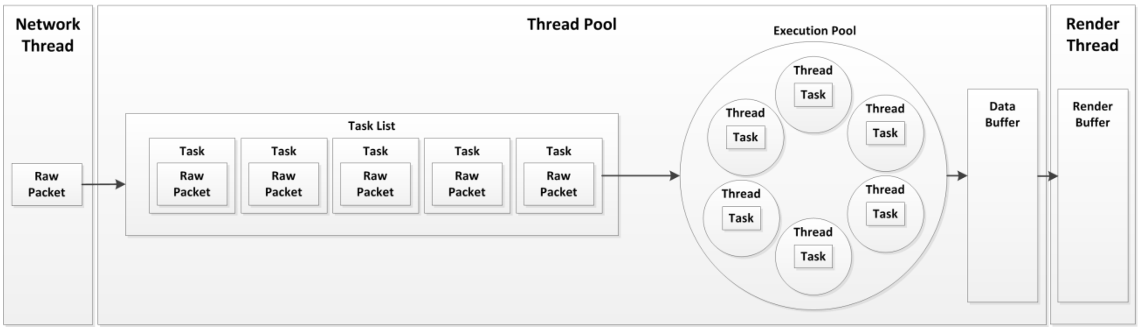

Figure 5: When the network thread has received enough raw data packets, it adds a new task to the task list of the thread pool for execution. An internal execution pool consisting of a defined number of threads executes these tasks in a first in first out order. Each thread in the execution pool deserializes the raw data to a VTK data object and appends its data to the data buffer. The render thread verifies for updates within the data buffer and possibly updates the render buffer.

The network thread listens to incoming data packages. If a defined count of packages is received by this thread, it adds a new task to the task list of the thread pool. A user-defined number of threads are responsible for processing these tasks. Every task has to deserialize the raw buffers to a VTK data object, append these data objects internally and finally append this data with the already present data in the data buffer.

The last step needs to be locked because multiple threads may want to add data to the buffer. Before the main rendering thread generates the next image, it verifies whether an update is already available in the data buffer. If so, it tries to lock the render buffer, copies data from the data buffer to the render buffer, and finally marks it to be ready for GPU upload and rendering. If the data is already locked by another thread of the pool, updating is postponed until a lock is possible. Due to this double buffering technique, processing and rendering is only locked for a minimal amount of time. The thread pool approach also has the advantage that the post-/in-situ processing application on the first tier can concentrate just on the extraction process while usually idle processors of the parallel rendering application are used for subsequent data preparation. Thus, the presented software architecture is able to exploit today's multi-core processors on rendering systems much more efficiently.

If the parallel rendering application has to create images for a tiled display, it has to handle not only one but multiple rendered images. The number of images is specified by the number of so-called viewports. The main thread within an IceT process now generates one 2D image per defined viewport. Therefore, it determines whether the data is visible on the viewport and calculates the colour and depth values for each pixel. If the rendering operation is finished on every node, IceT performs a conclusive composition task. During this composition task all images are collected by IceT to generate one final image per viewport. To combine a set of images per viewport to one final result, the depth values per pixel are compared. The colour of the pixel with the foremost depth value is used. When the images are generated they are sent to the front-end applications for presentation.

Since the pixel transfer for compositing the final image is costly and often prohibits interactive frame rates, a hybrid rendering approach [11] is integrated into our setup. This approach enables rendering with different frame rates on the local front-end and the remote parallel rendering application. The local front-end application guarantees interactive rendering of a context geometry, e.g. the wall boundary of an artery or menus for interaction, while the extracted features are rendered with a lower frame rate on the parallel rendering application.

Since the remote image is always delayed the camera position of the remote and local image may differ slightly. For this reason we implemented a hybrid rendering solution which combines local and remote images by an image-based rendering (IBR) approach. The IBR technique allows the re-use of obsolete remote images in order to generate images for the current view and therefore hides the latency of the requested remote image. To compose the re-adjusted remote image buffers with the locally rendered image of the context geometry, depth buffer comparison is used. The result of the composed remote and local image is depicted in Figure 6. This compose operation is directly executed on the GPU using the OpenGL Shading Language (GLSL).

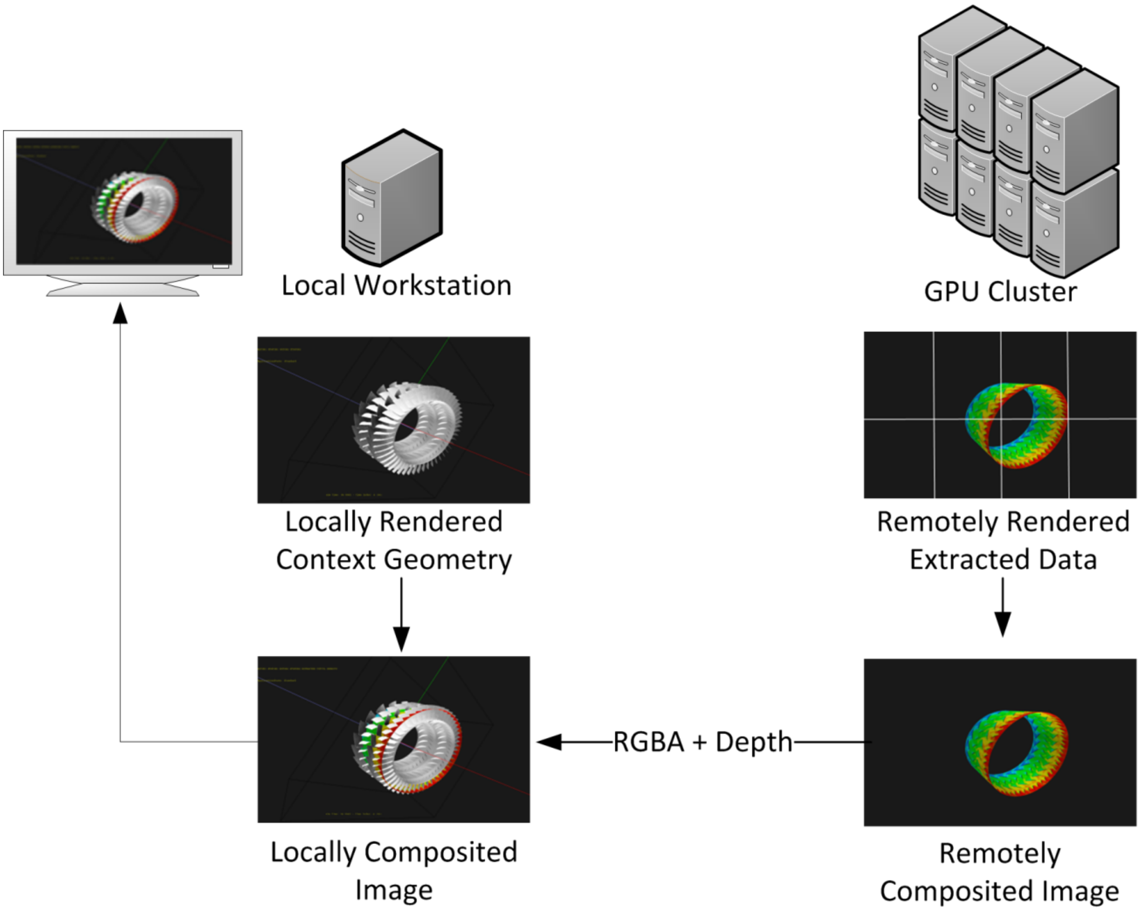
Local Workstation
GPU Cluster
Locally Rendered Context Geometry
Remotely Rendered Extracted Data
RGBA + Depth
Locally Composited Image
Remotely Composited Image

Figure 6: Our hybrid rendering approach. Remotely-rendered images are composed on a GPU cluster and transferred to the front-end. Here, they are combined with the local context geometry according to the pixels' depth values.

4 Evaluation

In this section we present the results of our performance evaluation. We first describe the used hardware infrastructure including their resources, followed by an evaluation of the in-situ feature extraction. Finally we benchmark the performance of the data preparation and rendering, the second tier of the software infrastructure.

4.1 Hardware system

This subsection describes the hardware systems on which the performance evaluation was performed.

4.1.1 Archer Hardware system

The ARCHER cluster system is based on a Cray XC30 supercomputer. This 5920 node supercomputer has 118,080 cores and is supported by a number of additional components including e.g. a high-performance parallel file-system. The compute nodes are connected together by the Aries interconnect. Every compute node contains two 2.7 GHz, 12-core E5-2697 v2 (Ivy Bridge) series processors. Each of the cores in these processors can support 2 hardware threads (Hyper threads). The used compute nodes on ARCHER have 64 GB of memory shared between the two processors.

4.1.2 DLR GPU-Cluster

The used GPU cluster consists of four high-end visualization workstations. Each of these workstations has two Intel(R) Xeon(R) X5670 hexa-core processors with hyper-threading support, 48 GB main memory, and three NVIDIA Quadro 6000 graphics cards with 6 GB DDR5 graphics memory. The interconnect is a 40 Gbit QDR-Infiniband network.

4.2 Datasets

We used two different bifurcation (bifurcation of vessels) datasets for our evaluation setup. The bifurcation geometry is a section of an intracranial vasculature model that has been constructed from multiple rotational angiography scans of a patient with an intracranial aneurysm treated at the U.K. National Hospital for Neurology and Neurosurgery. Starting from this geometry we created the two bifurcation datasets for the HemeLB simulation. The small dataset consists of 3.5 million lattice sites while the large bifurcation dataset includes 24 million lattice sites.

4.3 Evaluation of the In-Situ Processing Performance

In this Section we present the results of the performance evaluation with regard to the in-situ processing layer. As presented in Section 3.2, to perform the in-situ feature extraction we initially create an unstructured grid during the HemeLB initialization phase. This step adds an additional overhead to the regular simulation runtime. Therefore, we measured the costs and speedup of this initial grid creation process with increasing number of CPU cores. For the two bifurcation datasets (depicted in Figure 7) we can see that this approach scales with a super linear speedup.

Every time a point is inserted into the grid we have to check whether a point is already present at this spatial location in order to avoid duplicated points. As this operation is memory-constrained, we believe that this is the reason for the super linear speedup. Nevertheless, a more detailed study on this effect has to be made.

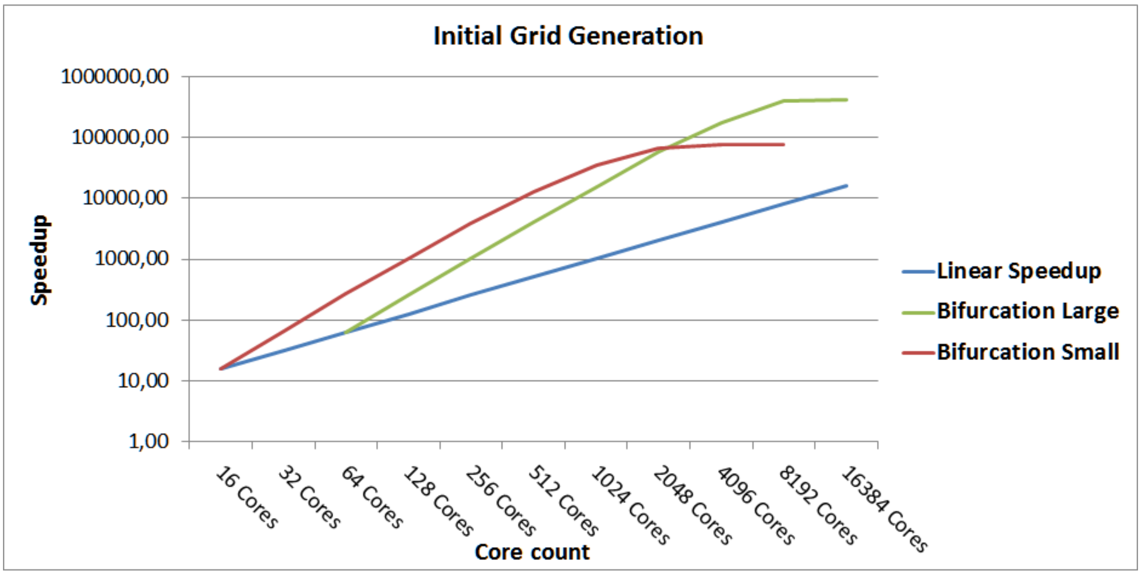

Figure 7: Achieved speedup for the initial generation of an unstructured grid from the HemeLB lattices. The speedup is super linear until some point of saturation.

For the small bifurcation dataset the scaling stops at a speedup of 75,360 using 4096 cores. This is the maximum achieved speedup for the initial geometry generation operation. Using larger core counts does not further decrease this time. Using the large bifurcation dataset the maximum achieved speedup is 421,800 using 16,384 cores, but the scaling already stops at 8192 cores.

Figure 8 depicts the results of a surface extraction algorithm. This algorithm represents an example of the feature extraction operation and can be exchanged easily. The algorithm has been executed every 100th HemeLB time step. Based on 50 extractions during 5000 time steps we calculated the speedup of this algorithm to show the scaling limits of the feature extraction.

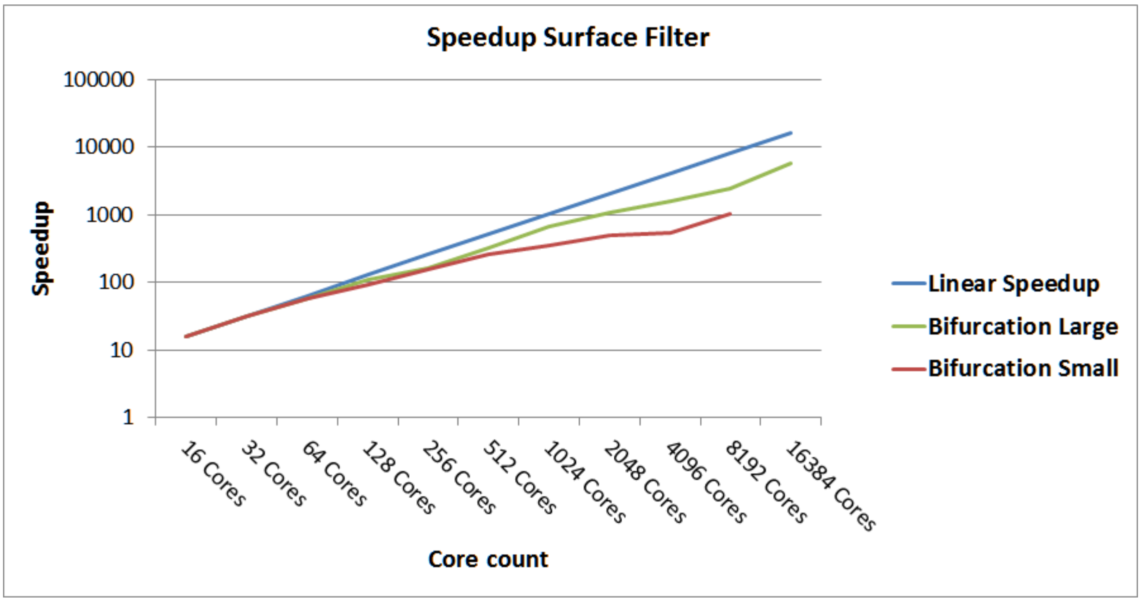

Figure 8: Measured speedup of the in-situ surface extraction with increasing number of CPU cores. The extraction operation was executed concurrently with the simulation and therefore stole CPU time from the simulation.

One should keep in mind that the extraction process is executed concurrently with the running simulation and therefore the runtime is heavily dependent on the simulation's CPU usage. The maximum achieved speedup for the small dataset was 1160 using 8192 cores which is poor, but looking at the overall simulation speedup (depicted in Figure 9) including the extraction operations it shows that the overhead for the feature extraction only harms the simulation slightly. The speedup on the large bifurcation dataset was 6650 using 16,384 cores. For the small dataset, the decomposition to 16,384 cores was not possible.

Figure 9 illustrates the speedup based on the total runtime of the simulation with and without the in-situ extraction of the surface features. The HemeLB partitioning and initial geometry generation operations are neglected. The results show that the additional extraction process does not influence the behavior of the simulation performance significantly. The total overhead for extracting features on the small dataset is on average 2% and for the large dataset 4.2%.

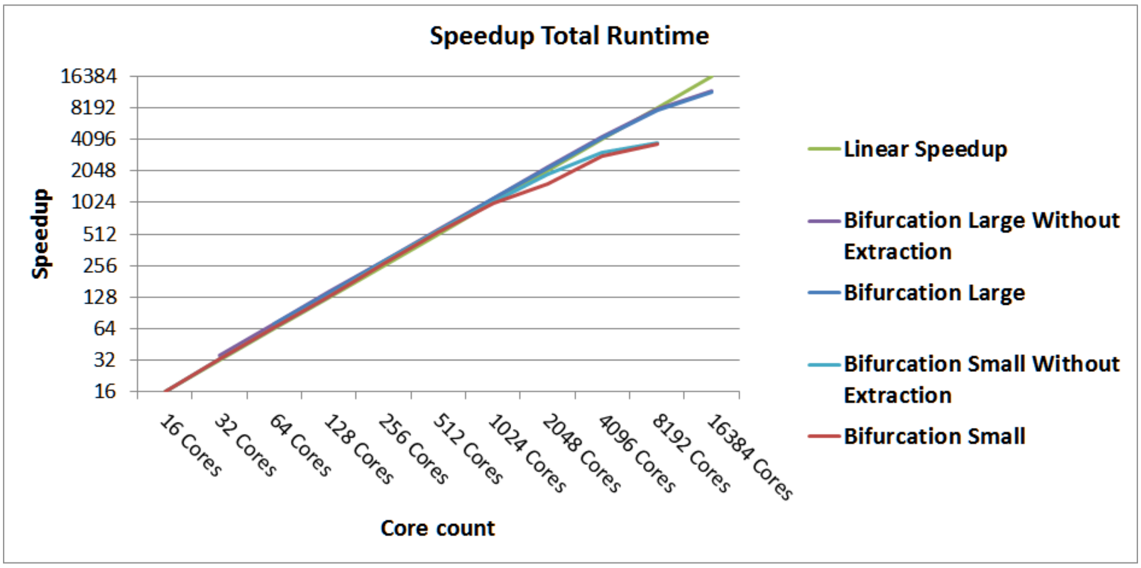

Figure 9: Speedup of the total simulation. The benchmarks show that the concurrent feature extraction only slightly influences the simulation.

Figure 10 depicts the efficiency of HemeLB with and without additional extraction of features. It shows that the efficiency with the extraction operation is nearly the same as without extraction of features. This indicates that the concurrent execution of post-processing only influences the simulation to a minimal extent. Furthermore, the figure depicts that HemeLB scales in a super-linear fashion at lower core counts and loses efficiency when the lattice count per core drops below 5000. The same behavior has been described in [13].

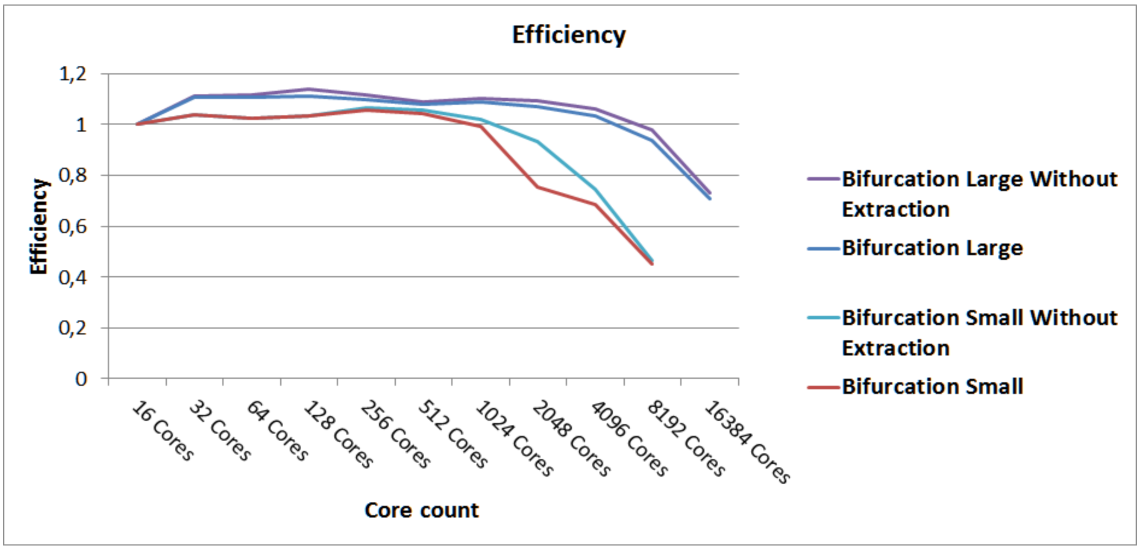

Figure 10: Efficiency plot of the two test cases with and without the extraction of features. The results show that HemeLB in conjunction with our post-processing approach suffers only a tolerable loss of efficiency.

4.4 Evaluation of the Multi-Threading & Rendering Performance

As mentioned in Section 3.3 real-time processing of incoming features is essential to enable interactivity during the exploration process. Figure 11 depicts the achieved geometry append bandwidth for input data with 1300 vertices per fraction. Figure 12 shows the bandwidth for input geometry with 11,000 vertices per fraction. Each fraction is appended to the output geometry. The benchmark has been performed on a single workstation of the GPU cluster with different numbers of threads.

With increasing vertices in the output data the bandwidth drops due to location of duplicated points. For the small bifurcation dataset the extracted surface consists of 1.5 million vertices with a serialized buffer size of 120MB.

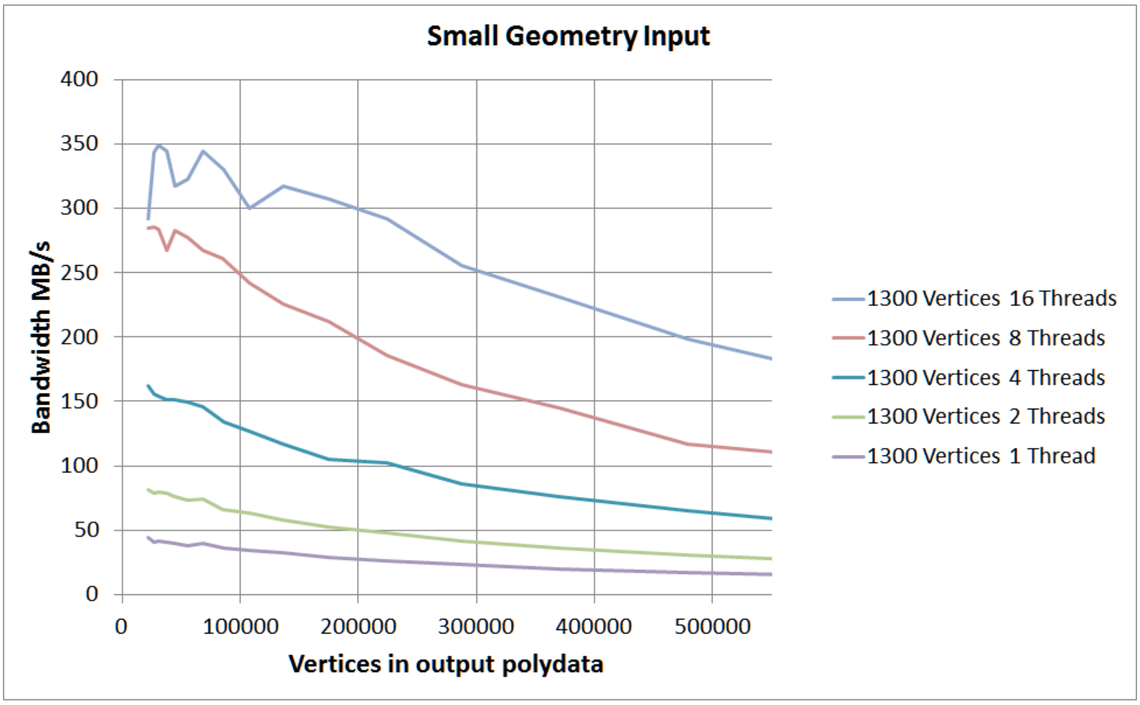

Figure 11: Append bandwidth using a single node of the GPU cluster with different numbers of processing threads.

Using 2048 cores on the small dataset a maximum output bandwidth of 1.2GB/s could be achieved. To prepare this data amount in real-time on the GPU cluster we would need around 5 nodes each using 16 threads. The maximum vertex count per output data using 5 nodes is 300,000 and the bandwidth per node is at around 250MB/s and therefore 1250MB/s in total.

For the large bifurcation dataset the extracted surface consists of 22 million vertices and a total buffer size of 1.4 GB. Using 4096 cores, features are extracted with a bandwidth of 6.4 GB/s. Since each fraction of the large dataset has roughly 11,000 vertices we would need at least 6 nodes with 16 threads. Using this approach the number of nodes can easily be adapted to the required bandwidth for real-time processing.

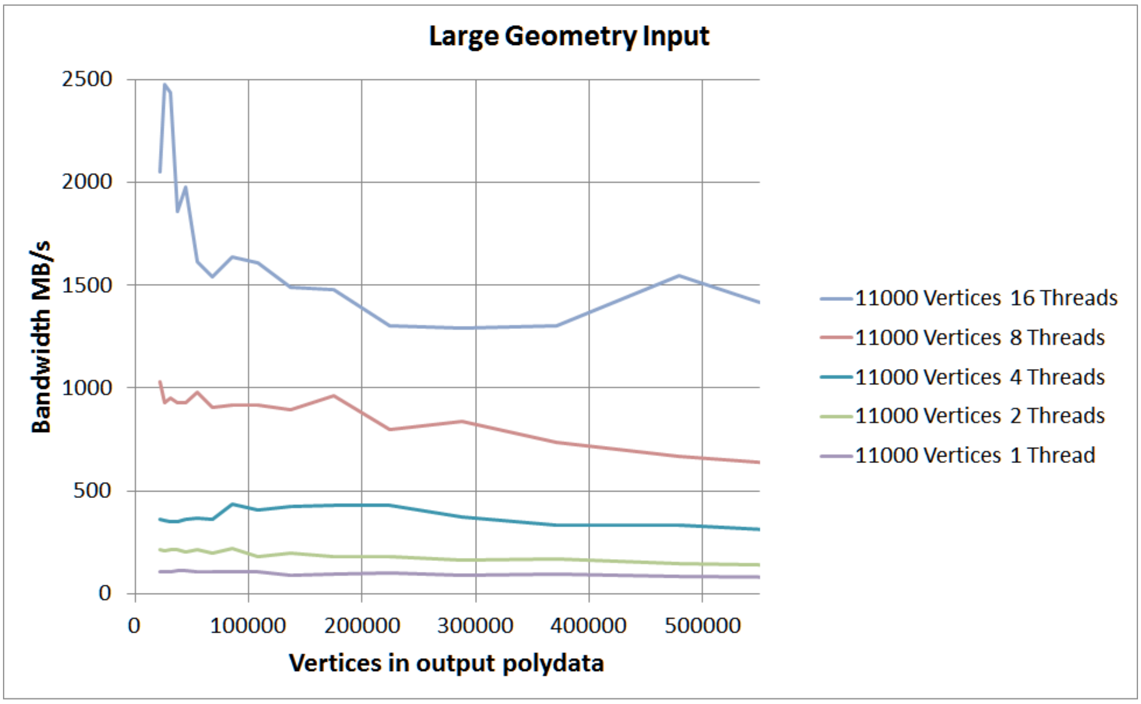

Figure 12: Append bandwidth using a single node of the GPU cluster with different number of processing threads.

One major limitation when performing parallel rendering is the huge compositing overhead due to the massive pixel transfer. This often prevents true interactive frame rates when using high image resolutions or multi-tiled display systems. A state of the art approach is to decrease the resolution while interacting. While this approach is very promising for desktop systems it is sub-optimal for immersive environments where the user position is changed frequently. In this way, the technique will always render with a lower resolution as the displays can provide and block artifacts visible to the user.

Figure 13 shows the achieved remote frame rates on our rendering system when using different display configurations. E.g. full HD resolution using one tile produces a maximum frame rate of about 17 frames per second which is much too slow for interactive environments. This frame rate will further drop by half when rendering stereoscopic image pairs. We could just achieve this 17 frames/s because of pipelining, where the rendering and image transfer are performed as parallel tasks. Due to our hybrid rendering approach, the local frame rate is always above 60 frames per second while the extracted geometry data is rendered remotely with the frame rates shown in

Figure 13. Therefore, the hybrid rendering technique enables us to be interactive on the front-end. It allows for smooth camera movements. Due to the image warping approach, extracted features are always rendered at full resolution with a minimal error.

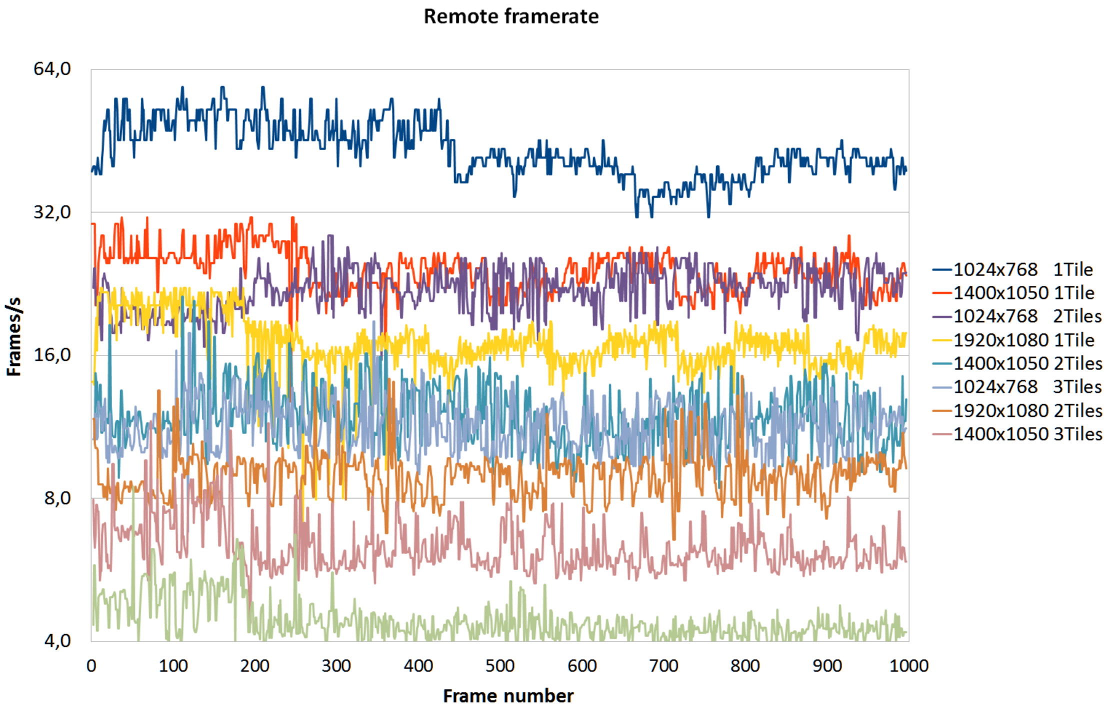

Figure 13: Frame rates using different number of tiles and resolutions.

5 Conclusion and future work

In this deliverable, we have evaluated our in-situ processing system using the HemeLB simulation code as the driving application. We demonstrated the usability and interactivity of the developed software infrastructure by benchmarking the speedup of the first tier in-situ processing layer followed by time and bandwidth measurements of the rendering application. The measurements demonstrate that the in-situ processing application integrated into HemeLB scales up to many thousands of cores depending on the size of the simulation. We could also show that the extraction process itself does not influence the HemeLB simulation scaling. The initial grid generation will scale up to 421,800 cores using the large dataset. Above this core count a further speedup will not be achieved and the data converting step has to be revisited. Features could be extracted within 60ms on a simulation dataset with 22 million lattice sites. This means features could be extracted nearly interactively.

The hybrid parallelized rendering application is capable of performing real-time processing of incoming features. It allows the rendering of massive quantities of extracted features. To avoid slow frame rates using the sort-last rendering approach we integrated a hybrid rendering approach into the software architecture. With this hybrid rendering approach interactivity could be guaranteed during user exploration.

In order to further improve the scaling behaviour of the in-situ processing application, the in-situ algorithms have to use the HemeLB data structures directly instead of transferring data from lattices to grid cells. Furthermore, the integrated Viracocha post-processing framework with a single scheduler has to be enhanced by optimized scheduling approaches allowing for higher parallel efficiency in case of extremely large numbers of processes.

6 References

[1] Ashby, S.: *The opportunities and challenges of exascale computing* (2012)

[2] Moreland, K., Thompson, D.: *From cluster to wall with VTK*. In: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics, PVG '03, p. 5. IEEE Computer Society, Washington, DC, USA (2003)

[3] Laboratory, L.L.N.: Visit user's manual (October 2005)

[4] Moreland, K.: *Redirecting research in large-format displays for visualization*. In: Large Data Analysis and Visualization (LDAV), 2012 IEEE Symposium on, pp. 91-95 (2012).

[5] Molnar, S., Cox, M., Ellsworth, D., Fuchs, H.: *A sorting classification of parallel rendering*. In: ACM SIGGRAPH ASIA 2008 courses, SIGGRAPH Asia '08, pp. 35:1-35:11. ACM, New York, NY, USA (2008)

[6] Moreland, K., Wylie, B., Pavlakos, C.: *Sort-last parallel rendering for viewing extremely large data sets on tile displays*. In: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics, PVG '01, pp. 85-92. IEEE Press, Piscataway, NJ, USA (2001)

[7] Eilemann, S., Makhinya, M., Pajarola, R.: *Equalizer: A scalable parallel rendering framework*. Visualization and Computer Graphics, IEEE Transactions on 15(3), 436 -452 (2009)

[8] Gerndt, A., Hentschel, B., Wolter, M., Kuhlen, T.,Bischof, C.: *Viracocha: An efficient parallelization framework for large-scale cfd post-processing in virtual environments*. In: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, SC '04, p. 50. IEEE Computer Society, Washington, DC, USA (2004)

[9] Assenmacher, I., Kuhlen, T.: *The ViSTA virtual reality toolkit*. In: Proceedings of the SEARIS Workshop, IEEE VR 2008. Shaker Verlag (2008)

[10] Gerndt, A., Hentschel, B., Wolter, M., Kuhlen, T., Bischof, C.: *Viracocha: An efficient parallelization framework for large-scale CFD post-processing in virtual environments*. In: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, SC '04, p. 50. IEEE Computer Society, Washington, DC, USA (2004)

[11] Wagner, C., Flatken, M., Chen, F., Gerndt, A., Hansen, C.D., Hagen, H.: *Interactive hybrid remote rendering for multi-pipe powerwall systems*. In: C. Geiger, J. Herder, T. Vierjahn (eds.) Virtuelle und Erweiterte Reality at, 9. GI-Workshop Virtuelle und Erweiterte Realität at, pp. 155-166. GI-Fachgruppe VR/AR, Shaker Verlag (2012)

[12] Schroeder, W., Martin, K.M., Lorensen, W.E.: *The Visualization Toolkit (2nd Ed.): An Object-Oriented Approach to 3D Graphics*, Kitware Inc.,1998.

[13] Groen, D., Hetherington, J., Carver, H.B., Nash, R.W., Bernabeu, M.O., Coveney, P.V.: *Analysing and modelling the performance of the HemeLB lattice-Boltzmann simulation environment*, Journal of Computational Science, Volume 4, Issue 5, pp. 412-422, September 2013.

[14] Moreland, K., Lepage, D., Koller, D., Humphreys, G.: Remote rendering for ultrascale data. Journal of Physics, Volume 125, Number 012096, 2008.

[15] Noguera, J.M., Segura, R.J., Ogyar, C.J., Joan-Arinyo, R.: Navigating large terrains using commodity mobile devices. Computers and Geosciences, Volume 37(9), pp. 1218 – 1233, 2011.