D5.1.1 – Pre-processing: analysis and system definition for exascale systems

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:	M6
Submission date:	31/03/2012
Project start date:	01/10/2011
Project duration:	36 months
Deliverable lead organisation	DLR
Version:	1.4
Status	Final
Author(s):	Gregor Matura (DLR), Achim Basermann (DLR)
Reviewer(s)	Jan Åström (CSC), Harvey Richardson (UEDIN)

Dissemination level	
<PU/PP/RE/CO>	*PU - Public*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	09/03/2012	First version of the deliverable	Gregor Matura (DLR)
1.0	09/03/2012	Revision	Achim Basermann (DLR)
1.1	10/03/2012	Full paraphrase	Gregor Matura (DLR)
1.2	12/03/2012	Second revision	Achim Basermann (DLR)
1.3	13/3/2012	Review, track changes	Harvey Richardson (CRAY UK)
1.4	20/03/2012	Accepted language alteration suggestions, implemented both reviews (with track changes)	Gregor Matura (DLR)
1.4	20/03/2012	Check of final version after internal review	Achim Basermann (DLR)

Table of Contents

Index of Figures

1 Executive Summary

As a first step to develop a proper pre-processing for massively parallel simulations, one has to evaluate the current status of various pre-processing tasks and how they perform on and interact with a possible exascale system. Only if this information is at hand it is possible to derive effectively a road map on how and where further effort has to be expended to achieve exascale-ready pre-processing techniques at the end of the CRESTA project.

Covering prior initialisation, disk I/O and mesh partitioning we summarise the outline of pre-processing. Up until now most work was put into mesh partitioning where currently multilevel methods dominate due to their computing efficiency and thus are omnipresent. All partitioning tools under closer inspection, ParMETIS, PTScotch and Zoltan, use multilevel methods. Therefore all these libraries share the uncertainty of scaling on more than several thousands of cores. Consequentially an improvement of these methods is reflected in the requirements on future pre-processing to achieve load-balancing in an exascale environment.

This affects all three primarily reviewed co-design applications HemeLB, OpenFOAM and Elmfire. Yet these applications also claim focus on other pre-processing issues. All three of them enforce a tight coupling of pre- to post-processing since they require methods like computational steering or proper visualisation. Further at least OpenFOAM depends on a starting value retrieved in an initialisation phase that optimally would be merged with any pre-processing activities. These requests to pre-processing translate into demands on future investigation and attention.

Future exascale systems have to provide faster communication lanes for a quicker redistribution phase, rapid disk I/O for proper initialisation, more memory for post-processing controlling parameters and for in-situ post-processing itself, faster computational units for possibly bypassing memory shortage with additional calculations.

As these deduced system definitions are sort of unspecific, required interfaces can be named explicitly. Pre-processing should be tuneable to accept a start value or calculate one by itself and forward it to the main computation. It has to be responsive to parameters originating in post-processing as to steer a real-time simulation or to adapt load-balancing to calculations and memory consumptions that will be performed during post-processing. Additionally pre-processing can be significantly improved by taking the actual system architecture, in particular the memory and communication hierarchy, into account so that load-balancing can be adjusted accordingly. These measures will give rise to a high-performance pre-processing that solves current scaling bottlenecks and is ready to be suitable for exascale systems.

2 Introduction

In principle every step performed prior to the main computation can be called pre-processing. Partitioning of the simulation data is surely the most widely known and widespread action realised. Further tasks can be counted among the pre-processing as well. The prime algorithm may demand a start value. Thus an initialisation may require coarse results from an internal or external pre-solver to grant appropriate starting conditions. Depending on the specific simulation the necessary disk I/O to get the data into memory may be counted, too.

A main aim of the pre-processing phase can be to support and accelerate the main computation phase. As already mentioned, the simulation may not start at all without an initial value of some kind. Even if this is not the case a decent start value can accelerate the solution process significantly. Last but not least, in a parallel environment the work load of the computation has to be analysed. Uniformly distributed computations guarantee satisfying load balance among the computational units in order to decrease wall clock time.

This is achieved by tracking all computational and communication costs. A graph is set up which traces the hardware structure of cores, sockets, nodes. Weights in the graph represent the costs. Now a partitioning of this graph yields a proper load-balance. For this task different methods can be applied. The most widely used is the multi-level approach. It is depicted in detail in section Partitioning in Pre-processing3 and offers a state-of-the-art good and fast partitioning.

For a first step self-implementation of this partitioner is not needed. Various ready to use tools are already available. Some of them are stand-alone partitioners like ParMETIS and Scotch. Others are integrated in a framework also covering pre-conditioners and solvers. They are all reviewed in section 4 along a check where and how they are used in CRESTA co-design applications.

These tools can be configured by some parameters. Before this can actually happen the special requirements of the CRESTA applications have to be recognised. Section 5 gathers these together with general ones and answers the following questions: What is lacking in the currently used pre-processing phase? Where has it to be enhanced? Once this is known possible forecasts can be made.

Section 6 depicts an outlook on hardware favourable for pre-processing. Desirable system definitions are derived from the requirements. These help to build future exascale systems that supply well designed resources for pre-processing needs.

Hardware, however, is only one aspect. Software interfaces have additionally to be manufactured. These should provide a better coupling between various simulation components. Initialisation, computation and post-processing are connected to the pre-processing that is focused on here. Section 7 names suggestions for these necessary interfaces. This completes the review on pre-processing.

2.1 Purpose

The purposes of this deliverable are as follows:

- Gather knowledge on existing pre-processing methods
    - In general
    - Application-specific
- Study (dis)advantages of algorithms being used as partitioners
- Analyse current status and
- Put together a list of requirements of pre-processing regarding the challenges of exascale computing
- Furnish particulars on favourable additional system resources
- Provide information on interfaces needed in future

3 Partitioning in Pre-processing

3.1 Multi-level Method

Within pre-processing, partitioning plays the most central role. It is a crucial factor for overall performance. There are already quite useful tool kits around (cf. Section 4) addressing this task. All of them usually apply multi-level methods to achieve the partitioning.

As a first step, the computational problem usually is represented by an undirected graph. Vertex weights in the graph stand for computational costs, edge weights for communication costs. Partitioning of such a graph is an NP complete problem and hence only heuristics and approximation algorithms are a suitable solution approach. Among the various partitioning algorithms available multi-level methods mark the state of the art.

A multi-level method consists of different stages. Each stage starts with a coarsening phase in which the current graph is reduced by collapsing vertices and edges. The now coarsened graph can be partitioned more easily. Thereafter the obtained partition of the coarsest graph determines the partition of the next finer graph. In a recursive way this leads to fast execution times reflecting the divide-and-conquer principle. The quality is tuned very well by designing and applying appropriate coarsening and refinement algorithms.

This method is advantageous for small core counts. Yet one has to keep in mind some drawbacks regarding scaling. In the first recurrences there is sufficient parallelism. The solution of the coarsened graph, however, is carried out only on a handful of cores. This results in waiting times, many cores idle. Furthermore communication between the stages puts constraints on the calculation time. Coarsening information from the former level is shared. The solution on the coarse graph is distributed. This affects almost all cores. So this communication stresses the network rather completely and simultaneously.

4 Tools

4.1 ParMETIS

4.1.1 Overview

ParMETIS is described succinctly by the introduction on the Karypis Lab website:

"ParMETIS is an MPI-based parallel library that implements a variety of algorithms for partitioning unstructured graphs, meshes, and for computing fill-reducing orderings of sparse matrices. ParMETIS extends the functionality provided by METIS and includes routines that are especially suited for parallel AMR computations and large-scale numerical simulations. The algorithms implemented in ParMETIS are based on the parallel multilevel k-way graph-partitioning, adaptive repartitioning, and parallel multi-constrained partitioning schemes developed in our lab." [1]

4.1.2 Application in CRESTA

In CRESTA ParMETIS is used in HemeLB for static partitioning. Computed lattice sites information is redistributed based on the ParMETIS outcome. Here the partitioning is very successful at least up to a thousand of cores. The major bottleneck arises in its aftermath: Redistribution of the data is very expensive regarding communication costs (see 5.2.1 for further details).

4.2 Scotch

4.2.1 Overview

Scotch's [2] 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 uses the MPI interface to provide a parallel optimised version of Scotch.

4.2.2 Application in CRESTA

In CRESTA PTScotch is used indirectly within the OpenFOAM framework. There it couples to and invokes methods supplied by PTScotch.

4.3 Zoltan

4.3.1 Overview

The official webpage provides a brief and accurate description of Zoltan:

"The Zoltan library is a collection of data management services for parallel, unstructured, adaptive, and dynamic applications. It simplifies the load-balancing, data movement, unstructured communication, and memory usage difficulties that arise in dynamic applications such as adaptive finite-element methods, particle methods, and crash simulations. Zoltan's data-structure neutral design also lets a wide range of applications use it without imposing restrictions on application data structures. Its object-based interface provides a simple and inexpensive way for application developers to use the library and researchers to make new capabilities available under a common interface." [3]

4.3.2 Application in CRESTA

In CRESTA Zoltan is not explicitly used. Though as a part of the Trilinos Project [4] it may be beneficial to use it. The Trilinos framework is package oriented and provides various direct and iterative solvers for sparse systems of linear equations. They are accompanied by different preconditioners that also include multigrid and ILU-type methods. The Trilionos default partitioner is Zoltan. PTScotch and ParMETIS can be integrated. Hence Trilinos may supply a capable software suite for all pre-processing and even simulation tasks. It also ensures possibilities to test several algorithms against or in conjunction with each other.

5 Requirements

5.1 General

The list of general requirements to the pre-processing of a simulation will now be discussed. The following topics are treated in order of significance: runtime load-balancing, computational steering and other post-processing issues, simulation initialisation.

5.1.1 Load Balance

The most crucial part in a parallel computation is to guarantee a maximally possible load-balance. That is, in a parallel environment computation time on each core has to be equal for a certain part of the simulation. If this premise is not matched some of the cores have to wait for others. This may happen to many of them in unfortunate cases or if calculation is not tuned well. This increases the overall simulation time unnecessarily. Obviously it gets even worse when increasing the core count of the system.

Evidently load-balancing is tightly coupled to the most intense part of the simulation, the computation itself. The type of the main algorithm is crucial here. It has to be inspected thoroughly. Once this is done the data needed for the computation can be distributed. All cores should now have almost the same workload to handle. Thereby any other constraints like running short of memory are to be avoided.

Partitioning is done very often and there are very good tools around. There are, however, exascale-related concerns. Surely the most significant concern is the unclear scaling to huge system sizes of these tools. In particular multi-level partitioning methods cause scaling problems on massively parallel systems. This has to be investigated further: Computation partitioning should couple tightly to the computation algorithm.

On the other hand any further information on the whole simulation chain has to be taken into account in load-balancing. In-situ visualisation techniques can significantly alter particular computation times and memory needs. Result output methods can be in need of compression. This again requires additional computation time and memory. Even a prior initialisation for the simulation can carry suitable information. This can and should be exploited to enhance load balance.

Last but not least all aforementioned points are strongly related to the structure of exascale systems. At the moment we already find a varying landscape of systems that additionally have a very heterogeneous hierarchy. There are various counts of different nodes. Each of them contains a number of sockets with several cores. Sometimes a significant count of GPGPU cards is present per node. Future development of the general structure is unclear by now. For example first petaflop systems like Tianhe-I rely on GPGPUs but it is an open debate whether more GPGPUs in a system are practicable. So even significant changes can occur and have to be considered. Pre-processing should take this into account. Load-balancing has to be adaptable to these yet to come system hierarchies.

5.1.2 Post-processing Related Requirements

Post-processing itself puts some constraints on pre-processing as well. In the post-processing part of WP5 (cf. [5] CRESTA Deliverable 5.2.1) techniques like computational steering are investigated. They influence in reverse pre-processing directly or indirectly via load-balancing issues.

Computational steering analyses the result of a simulation. Examples are hot spot detection in the mesh during the simulation, physically not possible results or not wanted outcomes. Hot spots usually trigger a finer analysis via mesh refinement. Penalty functions treat undesirable simulation properties. To really steer the simulation there has to be a possibility to feed these findings into the pre-processing phase. Either the simulation has to be completely restarted, or the simulation may continue but with an adapted schedule of changed partitions and calculations. Here the pre-processing must be sensible to do additionally tasks like mesh adaptation, decrease of time steps, etc.

Occasionally a visualisation task follows the simulation. This task may result in extra calculations or memory needs. Pre-processing has to be able to address these issues, e.g., by reserving memory for later post-processing and not only for the simulation.

Post-processing is done in every simulation in some way (at least output of one result like a residual). Yet it gets more important on huge systems. Computational steering defines finer meshes only in critical regimes to avoid unnecessary computations; Real-time visualisation knows what the user wants to see. Hence it tells where data is needed and especially where not. This qualifies to discard lots of unneeded data and therefore may lead to a significant performance gain.

5.1.3 Initialisation Inherited Requirements

Some algorithms or whole solution chains need a start value. This holds especially in exascale computing where a solution calculated on a coarser grid is sometimes required as a starting point for the finer grid calculation. It is possible to directly integrate this functionality into the pre-processing, but it may also be desired to do this completely detached in an earlier initialisation phase. In both cases pre-processing has to be adapted accordingly. Either this initialisation has to be supported directly by some pre-processing. An internal mechanism could cover this controlled by additional initialisation parameters. Or initialisation data has to be accepted and processed.

5.2 Application-Specific Requirements

5.2.1 HemeLB

As the name indicates HemeLB is a Lattice-Boltzmann code and therefore scale independent at first. In a primary step (cf. A.1 HemeLB Setup Pipeline) the geometry (e.g. of the treated vessel) is matched onto the lattice. Detection ensures whether the sites are located inside, outside or contain walls and have to be calculated. Of course, this has to be done in parallel due to the overall lattice size. Now information on the sites workloads is available but spread arbitrarily. ParMETIS is used to partition and redistribute the relevant data to achieve a decent load-balance. Subsequently the solution can be obtained. Results like the stress on a vascular wall can be written to disk.

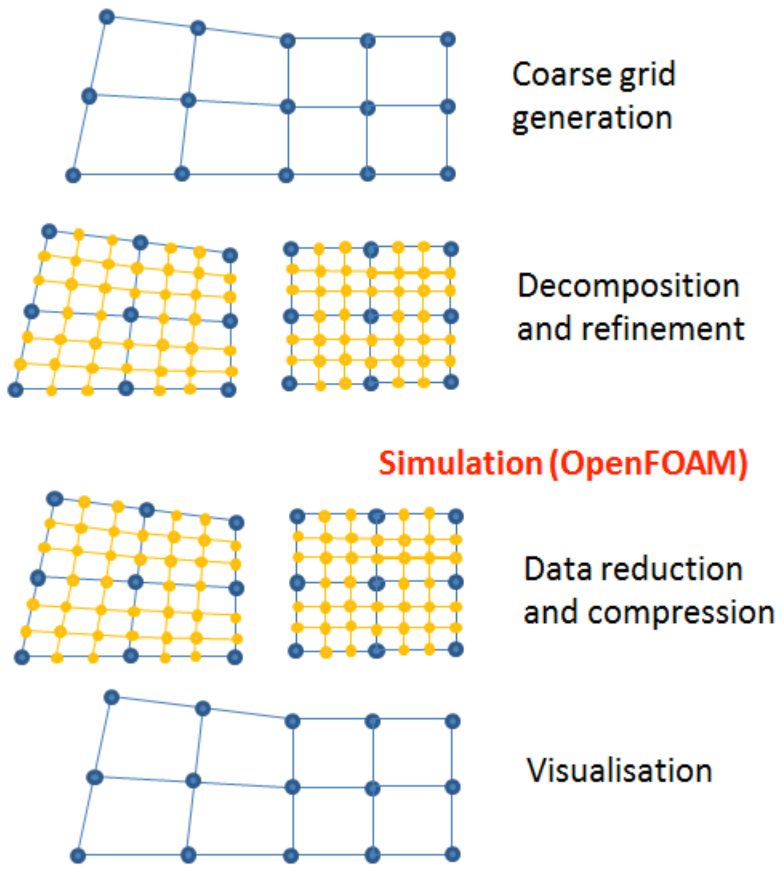
Unlike CFD codes lattice Boltzmann methods are fundamentally highly efficient in parallel. Within HemeLB the very solution time reflects this fact. The domain decomposition done by ParMETIS is also quite good in a load-balancing point of view. Furthermore HemeLB shows no problems with the partition so far, i.e., up to several thousand cores.

Nevertheless problems arise especially with an eye towards exascale. The domain decomposition works quite well. Yet it is surrounded by bottlenecks. First, the parallel I/O in the early matching phase has to be watched closely. It already contributes somewhat to the overall simulation time. Secondly, a shift from the on-going static domain decomposition to an adaptive one should be envisioned. This simplifies a future multiscale assembly of various HemeLB simulations. Besides that the most significant part, especially in overall wall clock time, is the redistribution of the data right after partitioning. Here much effort should be put in so that communication time drops significantly. A smarter way of distributing the data in the first place is the key. Likewise a co-optimisation of the whole pre-processing part starting from geometry generation could be favourable.

Last but not least it is pretty unclear how ParMETIS behaves moving on to exascale systems. Scaling on bigger system sizes has to be monitored closely. The underlying multi-level algorithms indicate bad behaviour regarding this issue. In future this could be tested on larger systems like JUGENE which are supplied e.g. via PRACE preparatory access.

5.2.2 OpenFOAM

OpenFOAM is a CFD code that typically performs Large Eddy Simulation. One starts with the creation of a mesh (cf. Figure 1) of the case to simulate, e.g., a hydraulic machine. This mesh is converted to a compatible format and then put into OpenFOAM. Here an internal mesh refinement is performed. A partitioner based on the PTScotch library is triggered which provides a domain decomposition for load-balancing. Thereupon OpenFOAM solves the system, reduces and compresses the data. Finally a visualisation takes place. The solution requires a start value. Prior to the whole chain a fast solver achieves this approximate solution based on a coarser grid. Furthermore the post-processing produces a huge amount of compressed data that is subsequently written to disk. For the specific test case of a hydraulic machine in 360 total time steps



about 90 TB data have to be handled.

Figure 1: Schematical view of the overall work process of OpenFOAM

Up to now OpenFOAM scales to thousands of cores. The main limitations to higher scalability are expensive interprocessor communication operations, in particular all-to-all reduction operations. We need to pay close attention to this scaling behaviour when considering moving to the exascale realm. Within the OpenFOAM setting scaling properties of the partitioner tool are not yet explored in the exascale regime.

Additionally, it is foreseeable that mesh refinement cannot be obtained completely and globally in advance. Rather than that, methods to combine the domain decomposition with a non-global hierarchical automatic mesh refinement have to be explored. They could bypass huge communication costs of an initially present extremely fine mesh. This, of course, is a major task of future exascale pre-processing.

Also, computational resources should be identified and reserved. So load-balancing is ensured in the post-processing stage, too. There, compression and in-situ visualisation exhausts computing power and memory while in turn reducing further communication.

5.2.3 Elmfire

Elmfire simulates the turbulent flow of the plasma inside of the fusion test reactor ITER. The modified kinetic Poisson equation for the electromagnetic field is solved for the charged particles. They in turn are responsible for the electromagnetic field.

The Elmfire code is time step oriented. In each of them particle properties are evaluated. Positions, velocities and alike determine the overall electromagnetic field. Afterwards the movement of the particles can be calculated. After, e.g., a hundred time steps the current status of the plasma can be written out.

At first the available particle data of all (up to several millions) particles is distributed randomly to the cores to disposal. This data is deduced from an earlier run time step or by a proper initialisation. The next step is computing the electromagnetic field and sharing this back to all cores. It is determined primarily by interprocess communication. The calculation of the new particle positions and velocities stays local. Hence it can be done very well in parallel. Yet the next time step requires again the former huge communication costs to share the new particle data. The amount of data moving around is vast. Estimates are obtained by outputting the result of one time step: for current simulation parameters this adds up to 40 TB. This output is done to get the results from computation. Visualisation of any kind takes place separately.

Elmfire´s scalability to about 30,000 cores on JUGENE may serve as a first estimate. However for an enhanced scalability the main application request is a proper visualisation. It ideally happens in-situ and hence circumvents the massive storage burden. Here pre-processing is applicable only indirectly: prepare possible data that is needed for post-processing.

Apart from this a decent pre-processing can help to increase scaling to more than the current some hundred cores. The electromagnetic interaction demands an all-to-all communication due to its long (in fact infinite) range. Nevertheless a (more) structured way of distributing the particles on the system would be beneficial. For this one could utilise the fast communication paths provided by the underlying hardware communication network. By paying attention to neighbourhoods the particles are aligned. As a result communication patterns are shorter and more optimal.

5.2.4 All applications

Since it is not present in any form in any of the codes fault tolerance gets a prominent position as a requirement here. Fault tolerance is a huge challenge in exascale environments. Flawless hardware is unachievable both in theory and practice. Errors at various stages of a simulation will certainly occur. How to deal with this issue is still an open question. So it is also unclear to what extent hardware or software has to deal with this.

Up until now fault tolerance is absent in all CRESTA applications at hand. However this is not due to ignoring this issue but rather because of a lack of specific treatment. Application developers, like in the case of HemeLB, are aware of missing fault handling. They claim fault tolerance to be a particular requirement for future applications at all, and especially in an exascale realm.

6 System Definition

In terms of pre-processing the most important issue regarding exascale systems surely is adaptation. The various requirements (cf. 5 Requirements) reflect this fact. An exascale setting will presumably be not only huge but also very heterogeneous. Pre-processing has to be flexible. So the application can exploit as much performance as possibly provided by the system at hand. Nevertheless propositions can be made in which manner pre-processing can benefit from additional system resources.

There is some doubt that the current partitioning algorithms scale well to more than several thousands of cores as HemeLB simulations suggest. So future methods may demand adaptation. For example a shift of resources is considerable. Instead of saving and sharing intermediate results additional recalculations may be performed. This would help to decrease the burden on memory size and communication speed.

Memory needs will increase as core counts increase, e.g., future algorithms can benefit of more halo data. This should be organised in the pre-processing phase. In-situ visualisation has to be considered likewise. It may require a decent amount of additional memory space. Computational steering algorithms store extra local observables. Pre-processing accounts for these and plans memory allocations ahead.

Most importantly various faster communication lines should be mentioned. Applications at large and pre-processing in particular take a lot of advantage from these. Potent Disk I/O is vital. Reading in matrices of very large sizes or writing back the results of a simulation require this. Faster memory lanes on the different levels are particularly useful. Core-local calculations run faster. Properly controlled intranode communication provides a benefit over regular network speeds. And last but not at all least the internode communication speed has to be addressed. Computation site counts increase within systems. This leads inevitably to an exponential increase in the amount of communications between the nodes. The distance of the communication within the network also rises. Therefor a lower latency and a higher throughput of the communication network will certainly affect application run-time significantly.

7 Interfaces Needed

Additional system resources as mentioned in the previous chapter 6 are indirectly related to pre-processing. On the other hand explicit interfaces can be named which get more and more vital for huge core counts and push forward overall application performance. Basically they can be summarised as interfaces providing additional information for the pre-processing and in this regard they are software components. The matching implementation schemes at disposal which exploit the steering parameters can lead to an entirely well load-balanced application.

7.1 Initialisation to Pre-Processing

For a solver that needs a proper initialisation, this has to be incorporated by the pre-processing. Either the pre-processing gets a premade result which it has to read in and distribute with the other data in an appropriate way, or the initialisation can be built-in. Controlled by run-time parameters pre-processing could launch a condensed computation, e.g., on an automatically coarsened grid, to receive a first starting value. This way initialisation can take place self-acting in short and optimised time without having the user to provide an explicit initial value. An interface should be capable of dealing with both variants of initialisation.

7.2 Post-Processing to Pre-Processing

In exascale simulations post-processing is inevitable. Huge amounts of computational results cannot be analysed manually so computational steering or an automatic evaluation has to take place. Prior to this a visualisation, most likely in-situ, may be required and performed. Since this puts various constraints on load-balance, memory use and computation it has to be addressed by pre-processing thus requiring an interface. Computational steering can supply information on hot spots, so indicating a point for mesh refinement in a real-time simulation. Observables of physical entities or flow parameters or even compression of the local results may be computed additionally thus disturbing the load-balance. In-situ visualisation will certainly be in need of extra resources. Pre-processing has to be accessible through a related interface, customisable by its parameters and adaptable to these changes.

7.3 System Information

The partitioning and distribution of computation data is sensitively connected to the system structure like local computational power and communication lanes. Pre-processing must take this into account to achieve a good load-balance and thus a good performance. Therefor accurate knowledge of the underlying system hardware is vital. Since exascale hardware is assured to be heterogeneous this fact has to be considered even more and hence an interface to retrieve hardware information is unavoidable for extraordinary efficiency. It has to provide intelligence on various system aspects like speed of cores, speeds of interconnects, the overall hierarchical structure and much more.

8 References

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

[2] Scotch, *Software package and libraries for sequential and parallel graph partitioning, static mapping, and sparse matrix block ordering, and sequential mesh and hypergraph partitioning,* http://www.labri.fr/perso/pelegrin/scotch/

[3] Zoltan, *Data-Management Services for Parallel Applications,* http://www.cs.sandia.gov/Zoltan/Zoltan_phil.html

[4] Trilinos, http://trilinos.sandia.gov/index.html

[5] CRESTA Deliverable 5.2.1, *post-processing: analysis and system definition for exascale systems*

Annex A. Further Information

A.1 HemeLB Setup Pipeline

A.1.1 Geometry File Creation

The main HemeLB executable requires 'geometry' files, which give a description of the sparse domain within which the simulation should take place. This geometry file is generated from a polygon-based description of the surface of the domain by a separate program. This program is a serial code, and is C++ driven by a python harness, using VTK libraries. (http://www.vtk.org/). The geometry file consists of a number of 'blocks', which correspond to cubic sections of the problem domain. For those blocks that contain any fluid sites, the block description lists, compressed via gzip, the coordinates of these sites within the block, and information as to whether the site is near a wall, inlet or outlet.

A.1.2 Seed Decomposition

The main HemeLB parallel executable loads the headers of the geometry file to determine the number of blocks. Each block is then assigned to a process using a simple serial decomposition, based on growing connected domains until a target domain size is reached. Each processor then calculates, based on the blocks it needs, which neighbouring blocks it also needs to know about.

A.1.3 Geometry Loading

A subset of the cores is used to do the reading of the blocks from the geometry file, configured to optimise reading parallelism versus file contention. The list of needed blocks for each processor is communicated to the cores that will do the reading. Each reading core then reads its share of the blocks using MPI-IO and sends the compressed block-information to the cores that need it. These cores parse the information, and store the site information for their blocks.

A.1.4 Parmetis Decomposition Refinement

Now that a representation of the geometry at site level exists, distributed over the processors, this is used to produce a connectivity graph for the sites. This is sent to ParMETIS to produce a new, refined, site-level domain decomposition. From this information, each processor re-calculates the blocks that it needs (those that contain sites it needs, or those in its neighbouring sites halo), and this information is used to re-load the geometry again for this second decomposition, repeating once again all the steps in phase 3. The geometry is now correctly loaded, and simulation can proceed.