D5.1.2 – Pre-processing: data format (hierarchical, regions of interest) and algorithms definition

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:	M12
Submission date:	30/09/2012
Project start date:	01/10/2011
Project duration:	36 months
Deliverable lead organization	DLR
Version:	1.0
Status	Final
Author(s):	Gregor Matura (DLR), James Hetherington (UCL)
Reviewer(s)	Jason Beech-Brandt (CRAY), Jan Astrom (CSC)

Dissemination level	
PU	*PU - Public*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	30/08/2012	First version of the deliverable	Gregor Matura (DLR), James Hetherington (UCL)
0.2	06/09/2012	Document revised	Achim Basermann (DLR)
0.3	07/09/2012	Minor additions	Gregor Matura (DLR)
0.4	24/09/2012	Review revision	Gregor Matura (DLR)
0.5	24/09/2012	Document slightly revised	Achim Basermann (DLR)
1.0	27/09/2012	Final version of the deliverable	Gregor Matura (DLR)

Table of Contents

1 EXECUTIVE SUMMARY 4

2 INTRODUCTION 5

2.1 PURPOSE 6

3 CASE STUDY: PARMETIS 7

3.1 DATA STRUCTURES 7
3.1.1 Format of the input graph 7
3.1.2 Format of the partitioning array 8
3.2 ALGORITHMS 8
3.2.1 Main computing algorithm: multilevel k-way 8
3.2.2 Use cases 9
3.3 PARMETIS AND EXASCALE 10
3.4 OTHER PARTITIONING TOOLS 11

4 CASE STUDY: HEMELB 12

4.1 INTRODUCTION 12
4.2 GEOMETRIES 12
4.2.1 Site 12
4.2.2 Link 12
4.2.3 Blocks 12
4.3 FILE MODEL 12
4.3.1 Header 12
4.3.2 Blocks 12
4.3.3 Site and links 13
4.4 GEOMETRY CLASS MODEL 13
4.5 LATTICE DATA CLASS MODEL 13
4.6 PRE-PROCESSING ALGORITHMS 13
4.6.1 Overview 13
4.6.2 Domain decomposition 1: initial decomposition 14
4.6.3 Parallel I/O: reading cores 14
4.6.4 Domain Decomposition 2: Optimised Decomposition 14
4.7 HEMELB AND EXASCALE 15

5 INTERFACE TO PRE-PROCESSING 17

5.1 POST-PROCESSING NEEDS 17
5.2 COMPUTATION NEEDS 17
5.3 ADDITIONAL NEEDS 17

6 REFERENCES 19

Index of Figures

Figure 1: A sample graph (without weights); [2] 7

Figure 2: Array content of distributed CSR format for sample graph using 3 processors; [2] 7

Figure 3: Multilevel k-way graph partitioning; [2] 8

Figure 4: Computational mesh for a contact-impact simulation. The surface elements are lightly shaded. [2] 10

© CRESTA Consortium Partners 2011

1 Executive Summary

With exascale computing, pre-processing becomes ever more important in order to increase overall performance and thus to lower costs. In this deliverable we focus on the usage and usability of partitioners in an exascale environment. Specifically, the data structures used, the underlying algorithms and coupling to other parts of simulation codes are tackled.

A distributed CSR format combines several advantages. Here data distribution information of the graph to be partitioned is spread among all processes. Graph data needed for the local calculation is kept local. However duplicated data on all cores is at a minimal level. This results in a light-weight yet complete graph data layout. The memory burden is therefore minimised, perfectly matching exascale demands.

State-of-the-art partitioning algorithms - like the multilevel k-way method described here – already produce quite good load balance in a reasonable amount of computing time. Additionally they have several features inevitable for exascale. Repartitioning is able to incorporate an old partitioning to gain a calculation time benefit. Mesh or graph refinement tasks get faster and thus feasible.

Native workload and communication patterns are default parameters for the algorithms. However they can also be adjusted in various ways. Multiple simulation phases and diverse constraints can be considered by passing corresponding weights. For example, this includes distinction of surface or inner cells. It is particularly useful for all sorts of particle-in-cell codes or contact-impact simulations which already cover a large portion of exascale codes.

Especially important is another aspect: The algorithms are mainly weight-driven. As such they can be tuned for heterogeneous system architectures with ease. Regarding exascale this surely is very convenient as it is unclear what future systems will look like in terms of hardware detail.

Lastly the demand for a closer coupling of pre-processing rises. Techniques like computational steering and remote rendering get integrated further into the simulation and alter load equation(s). The primary goal of an overall load balance can only be reached by forwarding vital information on to pre-processing. An interface to do so is needed.

2 Introduction

By definition, exascale systems perform very large simulations. In fact, the amount of calculations is tremendous. Underlying meshes get finer and finer. Resulting graphs have many vertices and edges. Particle numbers rise. Input and output data masses.

This mere size introduces various risks to proper simulation execution. Here a failed load balance is of particular interest, not only from a pre-processing point of view. Simulation lifespan is directly proportional to overall costs. Now, if one processor unnecessarily sets back calculation of hundreds of thousands of cores even for a short time real money is wasted.

Here partitioners' responsibilities are clearly formulated: Calculated partitioning must be particularly good in the sense of load balancing, although it has to be computed in a reasonable amount of time. Pre-processing shall not dominate the whole simulation. Overall a fast and good partitioner is fundamental, as well as the data structures and algorithms used within.

Pre-processing requires various pieces of information. Input data with graph or mesh details is processed. Computational methods must be considered. Post-processing might afford other constraints. These data flows have to be minimal. Hence, data structures must be kept reasonably small to facilitate every data exchange. Ease of handling these structures is essential as too much effort on decoding again increases overall computing time.

In general, algorithms' requirements are fairly obvious: The primary objective is a well-balanced load during the whole simulation run. Nevertheless, some conditions should be mentioned. In an exascale environment simulation steering will be useful. This of course results in repeated simulation cycles and therefore repeated partitioning may be necessary. Algorithms which are able to benefit from the consideration of old or estimated partitionings are preferred. They have to be adjustable in many forms: Adaptive mesh refinement will alter partitioning constraints in between simulation cycles. Multiple stages of computation might be necessary but the costs might not be determinable at simulation start. Heterogeneous architectures of exascale systems must be matched. Fault tolerance mechanisms may shut down parts of hardware.

Another important aspect was already mentioned and concerns coupling of simulation parts. Until now, simulation phases are divided in most codes, and pre-processing and post-processing is not accounted for in the core computation. Demand for computational steering, however, increases (see [1]). This forces coupling between core computation and pre- and post-processing. Here, too, the main advantage is a well-balanced overall simulation which in turn results in decreased costs.

Links between simulation parts therefore have to be implemented. Regarding pre-processing an interface should manage the information flow. Core processing measures or estimates workload and memory needs, and passes them to the pre-processor. Post-processing gathers the same values and calls the interface as well. Now pre-processing is provided with individual data for the complete simulation and is therefore able to calculate an efficient partitioning.

In this deliverable we first take a close look at the partitioning tool ParMETIS (see section **Error! Reference source not found.** and [2]). Data structures needed in ParMETIS calls are examined for suitability. The main algorithm of ParMETIS is based on a multilevel k-way partitioning method and is described in subsection **Error! eference source not found.**. Here we do not focus on the well-established core, we rather search for possibilities to adjust it to pre-processing. The specific requirements of an exascale simulation are covered separately and in detail in subsection **Error! eference source not found.**.

In section **Error! Reference source not found.** we proceed with an investigation of emeLB [4]. We describe the current state of pre-processing within this CRESTA co-design application. Geometries and the file model used are tackled before we get to an

explicit description of algorithms. Initial reading and decomposition, partitioning calls and redistribution strategy are detailed. Afterwards, HemeLB's pre-processing is analysed with respect to its exascale capabilities.

Finally section **Error! Reference source not found.** summarises the gathered aspects of pre-processing through the description of a required interface. Needs for a proper interface arise in different parts of a simulation and are addressed accordingly.

2.1 Purpose

The purposes of this deliverable are as follows:

- Investigate ParMETIS as an exemplar for a partitioning tool, regarding
  - the data structures used and their exascale potential, as well as
  - the algorithms implemented and their usability within exascale computing.
- Investigate HemeLB, as an example of a co-design application, in terms of its pre-processing use and needs
- Narrow down specifications of an interface to pre-processing

© CRESTA Consortium Partners 2011

3 Case study: ParMETIS

3.1 Data structures

3.1.1 Format of the input graph

In ParMETIS, the structure of the graph is represented by a compressed storage format. As every processor holds a disjoint part of the graph locally, we describe first this local and thus serial part of the format.

The adjacency structure of the locally present `n` vertices is encoded in the two arrays `xadj` and `adjncy`. The latter one holds a list of vertex numbers describing to which vertex the current vertex under consideration is connected to. Hence it represents the edges of the graph. This strategy obviously leads to a double counting of the edges, so this number is denoted by `2m`.

The former array separates the vertices under consideration. Therefore it holds a list of indices pointing into `adjncy`. At every one of these indices a new vertex and its edges starts in the `adjncy` array.

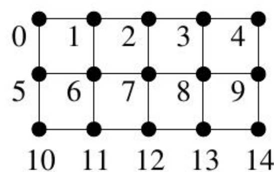

Figure 1: A sample graph (without weights); [2]

Figure 1 depicts an example graph (for a start without weights, see below), while Figure 2 shows the array content of the CSR format distributed across 3 processors. As we can see on processor 0 the first 2 (=2-0) entries of `adjncy` lists the two edges (1 and 5) of vertex 0. Because of 5-2=3 the next three edges (to 0, 2 and 6) belong to the next vertex 1 and so on.

Processor 0:	xadj	0 2 5 8 11 13
	adjncy	1 5 0 2 6 1 3 7 2 4 8 3 9
	vtxdist	0 5 10 15
Processor 1:	xadj	0 3 7 11 15 18
	adjncy	0 6 10 1 5 7 11 2 6 8 12 3 7 9 13 4 8 14
	vtxdist	0 5 10 15
Processor 2:	xadj	0 2 5 8 11 13
	adjncy	5 11 6 10 12 7 11 13 8 12 14 9 13
	vtxdist	0 5 10 15

Figure 2: Array content of distributed CSR format for sample graph using 3 processors; [2]

Furthermore ParMETIS allows weights to be provided for both edges and vertices (they are not shown in the example, but accompany `xadj` and `adjncy` in a straight forward manner). Here `vwgt` holds the *n* vertex weights linked to the vertices in `xadj`. `adjwgt` is as long as `adjncy` and contains the corresponding edge weights.

So far the described structure is fine for a serial representation of the graph. To get a parallel extension we distribute the vertex and edge arrays, `xadj` and `adjncy` respectively, equally to all processors. Then, every process needs only one additional piece of information: which process holds how many vertices is stored in `vtxdist`.

© CRESTA Consortium Partners 2011

Present on every processor, `vtxdist` can be used to determine whether a vertex or edge is local to the processor or not. This completes the format.

3.1.2 Format of the partitioning array

The output of ParMETIS partitioning routines is the array `part`. Its size equals the number of local vertices and after the routine returns, it holds the target processor for every local vertex. This fact also implies that ParMETIS is only responsible for calculating the partition, but not for distributing data. Thus it is necessary to communicate data correspondingly if one or more array values do not match the local processor number.

Furthermore ParMETIS does have built-in routines for repartitioning graphs or meshes. Old partitionings have to be supplied and are conveniently passed via `part` in the routine calls. So these routines obtain the old distribution structure from `part`, compute the new one and store it back to `part`.

3.2 Algorithms

3.2.1 Main computing algorithm: multilevel k-way

The main computing algorithm within ParMETIS is based on a multilevel k-way partitioning method. Whether (re-)partitioning of a graph, adaptation of a mesh or similar operations are performed, almost every routine uses this algorithm by some means or other for its basic computation. Of course, this is hidden behind the respective routine call for user's convenience.

The algorithm is split in three phases. First, a series of steadily coarser graphs is constructed from the input graph (see Figure 3). The initial partitioning is computed on the coarsest level. Since this is also the smallest graph the calculation time is small, too. The final phase consists of refinement steps of the partition for each finer graph. The result is a fast and high quality partitioning of the finest, i.e. original, graph.

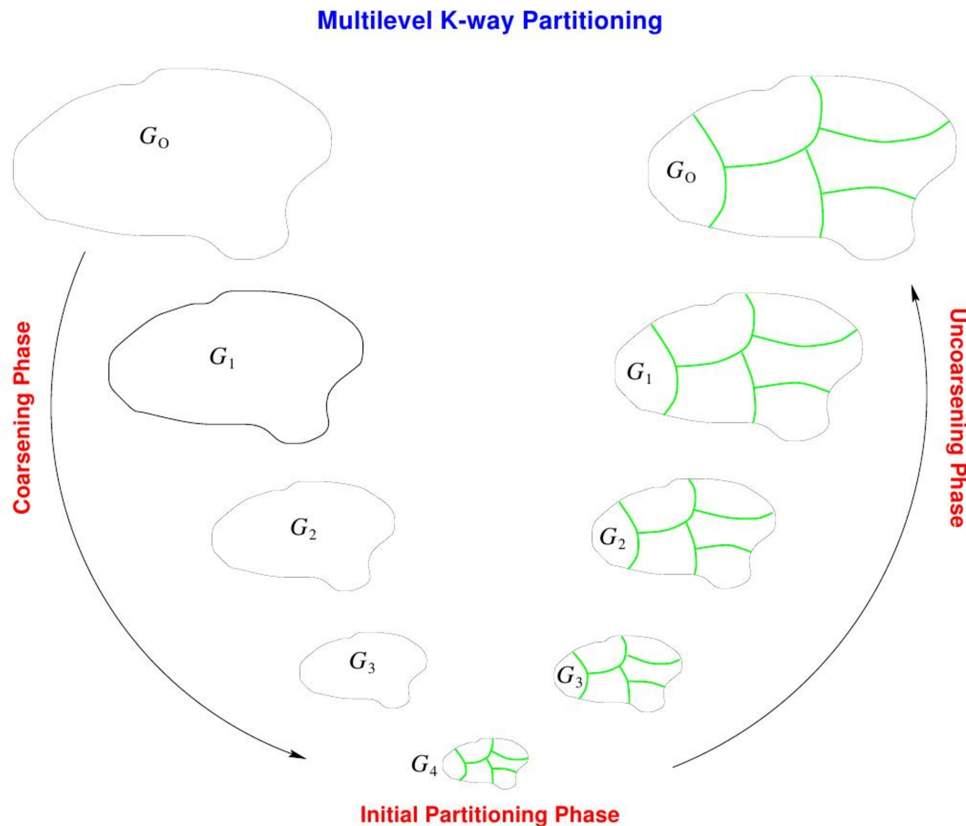

Figure 3: Multilevel k-way graph partitioning; [2]

© CRESTA Consortium Partners 2011

3.2.2 Use cases

Computing partitionings of unstructured graphs or meshes is the main use case of a partitioning library like ParMETIS. Nevertheless ParMETIS offers more options covering additional functionality tightly coupled to the needs of applications using partitionings.

3.2.2.1 Adaptively refined meshes

Some applications need to alter the underlying meshes continuously. Although a new partitioning could be computed from scratch it can be significantly faster to do so by taking into account the old one. For that reason ParMETIS provides a routine to get a new partitioning for a refined mesh.

There are two competing methods to do so. The first one tries to diffuse computing load away from highly loaded sub-domains. Opposite to that, the second method computes a new partitioning and tries to map this onto the old one while minimising redistribution costs.

To determine which method is currently the best one the routine accepts the so called ITR factor (see [3]) as a parameter. This factor describes the ratio between the time required for performing the inner-processor communications incurred during parallel processing, and the time to perform the data redistribution associated with balancing the load. The ITR factor can be estimated by division of two times measured during a simulation run. Namely, these are the time for all inter-processor communications that have occurred since the last repartitioning and the time for data redistribution associated with the last repartitioning/load balancing phase.

The ITR factor is passed to ParMETIS' routines. By its nature it is a single metric describing the quality of the repartitioning which technically is a multi-objective optimisation problem. Thus it is a convenient technique to decide what the best type of method for the repartitioning is.

In general, repartitioning is assumed to be done after the refinement. ParMETIS, however, also offers the possibility to do this in advance as well. If the degree of refinement of each element can be estimated beforehand these values can be used as weights of the vertices. If these weights are available, repartitioning and so redistributing can take place preceding the refinement. Depending on the refinement, communication for redistributing data can be significantly reduced.

3.2.2.2 Partitioning for multi-phase

Another feature of ParMETIS is given by the optional use of multiple weights. Say we have separated stages in our simulation, e.g., first particle movement followed by force computation based on stationary sites. Then, a partition has to optimise load balance over the full simulation, taking all stages into account. Workloads for each stage can be passed as further vertex weights and will be included in the partition calculation. Communication costs in between stages can be incorporated, too.

This technique of multiple weights per vertex or edge is not limited to separate stages. Likewise it can be used for multi-constraint graph partitioning problems. As an example, a contact-impact simulation indicates the possibilities. The dashed partitioning in Figure 4 shows only a balance of the number of mesh elements. On the other hand the solid partitioning balances the number of mesh elements as well as the number of surface elements.

© CRESTA Consortium Partners 2011

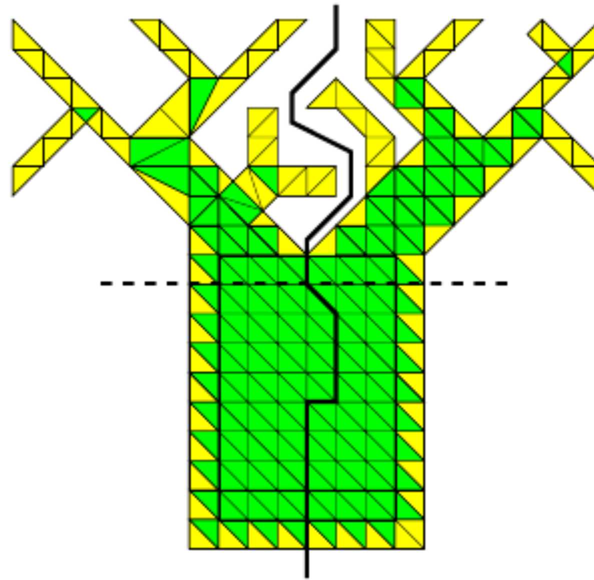

Figure 4: Computational mesh for a contact-impact simulation. The surface elements are lightly shaded. [2]

3.2.2.3 Heterogeneous computing architectures

Most of ParMETIS' partitioning routines also accept another optional parameter called `tpwgts`. It stores weights for every computing (sub-)domain. Therefore it can be used to reflect a varying computing power across a whole computing network. Hence a complex, heterogeneous cluster can be considered within every partitioning.

3.3 ParMETIS and exascale

Regarding exascale aspects, ParMETIS' data structures have advantageous properties. The information stored is minimal in the sense that it only includes information that is needed. Furthermore it can be computed simply from data each processor has to have anyway (or even is already present in the required form). It helps to keep memory requirements low thereby helping to address the predicted growing gap between higher computational power and weaker memory (i.e., comparatively lower speed and smaller size) per node in future exascale systems.

In addition the small memory requirement of ParMETIS' data structures enables a possible copy of the current partitioning. In the next cycle the simulation benefits from this copy as a new partition can be built upon the old one. This repartitioning saves not only computation time for the new partitioning;data redistribution costs in terms of inter-process communication might be lowered, too.

As for algorithms, ParMETIS provides some valuable features. Exascale simulations are evolving an increasing need for steering possibilities in some form. Real-time visualisation enables the user to spot regions of interest. An automated post-processing technique like feature extraction can mark these regions as well. Here the simulation can react immediately and adaptively refine the underlying mesh. ParMETIS' repartitioning functionalities then help to lower the redistribution costs and so increase overall performance.

Exascale simulation codes will have cost-intensive cycles, within which care should be taken to avoid any bottlenecks. They will tend to include the whole computational chain from pre-processing, through calculation, to post-processing, in order to get a direct response. So at least post-processing has to be included in load-balancing besides core calculation itself. This is a major employment of multi-phase partitioning.

As exascale systems become available the variety of codes will increase. Even now CRESTA co-design applications show different code structures with different calculations. So in addition to weighting core and post processing for an overall good load-balance, the main computation itself may unravel into various stages. Further

constraints may complicate the situation. The possibility for adapted multi-phase and multi-constraint partitioning will be of particular importance.

At present, the specifics of the architecture of exascale computing resources is still unclear. Heterogeneous architectures could play a key role. ParMETIS is already able to accommodate for varying computing power at different domains, although capability to include costs for heterogeneous network communication is missing.

3.4 Other partitioning tools

So far we have focussed solely on the partitioning tool ParMETIS. Another widely-used tool is PTScotch [5]. Here we briefly describe main differences between those tools.

PTScotch uses ParMETIS-compatible data structures. There are distributed arrays for local vertex and edge information and one array containing global vertex distribution information. They equal exactly the ones used in ParMETIS. Additionally, PTScotch maintains additional structures which can be constructed directly from the three main arrays mentioned above. A simple conversion is possible. In fact, PTScotch already provides compatibility routines to ParMETIS.

Regarding algorithms, the setting changes a little bit. Some algorithms used in ParMETIS and PTScotch differ in their basic method. The kind of routine calling, however, the resulting output format and provided additional functionality (see section 3.2.2) are again similar. In principal, this leads to similar behaviour in usage and implementation.

Overall these partitioning tools operate very similarly. Data structures are compatible and convertible. Code integration is almost identical. This opens up a possibility to cover different partitioning tools in one piece of software. In turn, this interface to partitioners could be a consistent footing to implement other needed functionalities (see section 5).

For dynamic applications, which require re-partitioning for load balancing, Sandia National Laboratories developed the Zoltan library [6]. Zoltan is a collection of data management services for unstructured, adaptive and dynamic applications such as adaptive finite-element methods, particle methods, and crash simulations. It includes a suite of parallel partitioning algorithms, data migration tools, parallel graph colouring tools, distributed data directories, unstructured communication services, and dynamic memory management tools. Zoltan's data-structure neutral design allows it to be used by a variety of applications without imposing restrictions on application data structures.

4 Case study: HemeLB

4.1 Introduction

HemeLB [4] has a complex task in loading the structure of the sparse domain in which the simulation will take place, distributing this information to each cooperating process, and deciding which processes will be responsible for which elements of the simulation domain. This *pre-processing* stage is discussed here, focusing on the data-structures and algorithms used.

The input to this process is a HemeLB *geometry file*, describing the domain. The output is a memory structure on each process describing the simulation sites that each process is responsible for, and, on each process, the identity of the processes who manage neighbouring sites.

The purpose of this section is not a detailed description which can be engineered against, but to specify the state of the art of HemeLB pre-processing, as a basis for analysis of potential improvements towards enabling HemeLB for the exascale. A sister document provides a similar overview of post-processing in HemeLB (see [1]).

4.2 Geometries

In this section, we introduce the *semantic* model used by HemeLB to describe the simulation domain, a *geometry.* Specific *syntactic* realisations as data files and memory structures are described in later sections.

4.2.1 Site

The basic unit of HemeLB geometry is the site, a location where Lattice-Boltzmann simulation will take place. The information for a site consists of its location in the domain (a coordinate triple) and an indication of whether it is a fluid site which should be simulated or a solid site outside the simulation domain.

4.2.2 Link

Each site also includes content regarding the nature of the link to each neighbouring site. Which sites are considered neighbours varies, and will be discussed below. Link information consists of whether the link crosses outside the simulation domain, how far along that link the solid/fluid boundary lies, and, if the link crosses outside the simulation via an open boundary (fluid inlet or outlet "*iolet*"), the identifier of the specific iolet.

4.2.3 Blocks

A block is an 8x8x8 cube of sites.

4.3 File model

We now turn to an overview of the syntactic realisation of the geometry used to represent it as a file on disk. This is an XDR binary representation. Engineering details can be found in HemeLB developer documentation.

4.3.1 Header

The file begins with a header. This first provides filetype and version information, and then gives the size of the problem domain in blocks, and transformation information relating the block/site coordinate system to the spatial domain.

4.3.2 Blocks

There then follows a table of information describing each block, giving the number of fluid sites in the block, and the compressed and uncompressed sizes of the block data, which varies from block to block. Data for all blocks in the cuboid domain is given, blocks being striped with z changing most frequently, thus block coordinates can be deduced. These vary, as we will see.

© CRESTA Consortium Partners 2011

If a block contains no fluid sites, no data is recorded for it, and this is reflected as a zero in the header table. Otherwise, for each site, solid or fluid, site data is given, compressed using zlib.

4.3.3 Site and links

An unsigned integer is recorded, indicating whether the site is solid (0) or fluid (1). If a site is solid, no further data is given. Otherwise, a representation of link data is given for each link to all neighbours, using a 26-member 3-D Moore Neighbourhood.

4.4 Geometry class model

HemeLB contains a set of classes used to represent the geometry information during parsing and analysis of the geometry file. These classes (GeometrySiteLink, GeometrySite, GeometryBlock, and Geometry) provide a simple de-serialised representation of the geometry file, each using STL vectors of the contained elements. (A geometry contains a vector of blocks, a block one of sites, and a site one of links.)

In this representation, the model is only partially sparse: either a block contains an empty site vector, if it is empty of fluid sites, or it contains a site vector with a site object for all 8^3 sites. Similarly, a fluid site contains either 25 link objects, or none. Site and block coordinates can thus be inferred from their location in the vector. On all processes, the full set of block objects is instantiated, but a block will contain no sites on processes where it is not needed.

In addition, the geometry site model also contains, for each site, the rank of the process to which that site is allocated by the domain decomposition.

4.5 Lattice data class model

The Geometry class model, being a simple object representation of the geometry, is not appropriate to efficient computation. HemeLB therefore uses a second representation of the same geometry information for the Lattice-Boltzmann and visualisation components. This representation uses a flyweight pattern: a LatticeData object contains vectors, separately, of each piece of data that might be stored about a site. A Site object contains a single data field, an index into these arrays, and a series of access methods, providing an object-oriented reflection of this flat data. Only the links in the Boltzmann lattice subset of the Moore neighbourhood selected for this build at compile time are used.

This representation is not used in the pre-processing stage – the LatticeData constructor takes a Geometry object as a parameter, and builds this representation. This representation will therefore be discussed further in the post-processing sister document. We will not discuss how the Geometry Class Model is transformed into the Lattice Data Class Model.

4.6 Pre-processing algorithms

4.6.1 Overview

Having provided an overview of the data structures used in HemeLB we now turn to the algorithms used to build the Geometry Class Model from the File Model. The simplest parsing of the file format into the basic objects is straightforward, and we will not discuss it in detail. However, several more complex tasks must be addressed: we must efficiently load a large file, in parallel, into multiple processes (parallel I/O), we must determine which sites belong to which processes (domain decomposition), and we must ensure that the sites and block information is made available to those processes which need it (information distribution).

The process is broken down as follows: we make an arbitrary initial decomposition of blocks to each of several processes. Then, we use a subset of processes to read the blocks, and distribute the data to the processes which need them. Next, we re-decompose using the now loaded site information, and finally, we re-distribute the data according to the new decomposition.

© CRESTA Consortium Partners 2011

4.6.2 Domain decomposition 1: initial decomposition

Domain decomposition suffers from a “bootstrap problem”: We cannot efficiently load and analyse the geometry without some distribution of the geometry across parallel processes, and we cannot determine an appropriate decomposition of the geometry without loading and analysing it.

We therefore make an initial decomposition which does not require detailed information about the geometry, only an indication of size of the cuboid bounding box of blocks, and whether a given block is wholly solid. This information can be obtained using only the headers of the geometry file.

This initial decomposition is carried out by the `BasicDecomposition` class, and uses a simple domain-growing algorithm. Improvements to this algorithm are unlikely to be of significant benefit, as it is only used to get some starting parallelism for block reading and is thrown away after being used as a seed for the geometry-aware second decomposition.

This algorithm is carried out on all processes, with no parallelism: the same answer is obtained on all processes. For each process, we start a domain, and grow it by adding neighbouring blocks, using neighbours based on the Lattice-Boltzmann lattice subset of the Moore neighbourhood. Once the domain contains enough blocks to give that process a fair share, we start a domain for the next process. Blocks which contain no fluid sites are not assigned, but are treated as barriers to the growing domain. It may therefore be necessary to start more than one growth region to achieve the required number of blocks per process.

4.6.3 Parallel I/O: reading cores

We assume that the geometry file resides on a parallel file system, with the file capable of being accessed in pieces by multiple processes. Rather than load from the file, the blocks and sites assigned to it by the domain decomposition, we choose to limit reading from files to a limited subset of cores, known as *Reading Cores.* Information from the file appropriate to each process is then passed to the relevant processes through appropriate messages. This approach has been seen to allow parallelism in the reading of large files, while at the same time avoiding slow-down due to excessive numbers of processes contesting for the file. The optimum proportion of total processes used to read the file depends on the file system being used and many other factors, can be set at compile time for HemeLB, and is an ideal target for autotuning, which we do not yet do.

Since we use a subset of processes to read the file (see below) these processes must learn which blocks contain sites used on which processes, so that they can send this information to those processes after it has been read. The class `Needs` handles this through a series of MPI Gathers to each reading core, gathering first the number of blocks needed on each process, then the identities of those blocks. Then, as blocks are read on each reading core, the compressed data is sent from reading process to needing process.

4.6.4 Domain Decomposition 2: Optimised Decomposition

Having loaded a full description of the geometry, we can now make a proper domain-decomposition. We use the ParMETIS library to do this. The resulting decomposition is on the level of individual sites: sites from a single block could be allocated to multiple processes. The task of HemeLB here is to transform the Geometry Class Model of the geometry into a graph-based representation suitable for use by ParMETIS. The ParMETIS library used for the optimised domain decomposition requires its own representation of the geometry, using a different semantic model, representing the geometry as a set of nodes and arcs.

In this data structure, not represented as classes but as a series of arrays created in order to call ParMETIS, LB sites become nodes, and the links become arcs. Only links in the Lattice-Boltzmann lattice subset are used. Each process is provided only part of the graph, based on the blocks assigned to it by the initial decomposition. ParMETIS

© CRESTA Consortium Partners 2011

uses global site IDs to reconcile this data between processes. In preparing to call ParMETIS, therefore, we must traverse all sites and links in the blocks assigned to a given process, and, where a link joins two fluid sites, include it in the arrays used to call ParMETIS.

ParMETIS returns its results in a partial form, returning to each process, the new processor rank for each of the sites provided to it on that process on input. Where this differs from the initial assignment, therefore, HemeLB must inform the new assignee of that new assignment. These changes of assignment between the initial and optimised decompositions are called *Moves.* Following the call to ParMETIS, therefore, a series of MPI calls (all to all and point to point) is used to share this information, and reassign the sites to appropriate places.

Following this, it is then necessary to re-load the geometry information for sites and blocks following the new assignment. Potentially, this data could be transferred between blocks. Instead, for simplicity, the whole data-file is re-parsed using the same code as following the initial decomposition, repeating the Needs sharing process, the reading on the Reading Cores, and the transfer of compressed data from reading process to needing process.

4.7 HemeLB and exascale

Regarding exascale aspects initial decomposition done in HemeLB marks a good starting point. Not only one core is doing the whole work of file reading for input data, but not every core has to access the file system either. So in total there is no I/O bottleneck and not too much fragmentation. This can be suitably adjusted to a variety of file system and communication network architectures on clusters.

The specific type of initial distribution ensures a well-balanced memory load. No data is duplicated, so no memory is lost. Every processor gets its fair share, there is no overburdened one.

The amount of gather and scatter calls to and from the reading cores is not avoidable but adjustable. According to the specific exascale system and its ratio between file system performance and network communication speed the number of reading cores can be configured to locate the optimum.

This optimisation gain pays out twice. As soon as ParMETIS calculates the final partitioning HemeLB does the whole reading chain again, but now corresponding to the final partitioning. This procedure saves coding time and lines. Nevertheless it might be advantageous to include a proper redistribution of data rather than a new file read. The performance of the file system of the particular exascale hardware could be unfavourable compared to that of the communication network. In this case redistributing via gather and scatter results directly in shorter pre-processing time.

Following initial decomposition ParMETIS is responsible for a proper partitioning. For a Lattice Boltzmann code computation time is equal on inner and boundary sites, respectively. Hence only neighbouring information is needed for partitioning, and so this already qualifies for a good load-balance for the core calculation.

Another aspect, however, might have to be taken into account. So far uniform load-balance is almost guaranteed for the computation. Possible computation time for visualisation may alter this situation. Depending on the method used in post-processing overall computation time can increase significantly on only a part of all cores. If for example a certain point and direction of view is selected some parts of the model may remain unseen. Inside these parts additional calculations are then not necessary. Cores assigned to these parts end up with a much lower work load. In total this could lead to even a very poor load-balance. This should be considered carefully.

To sum up, pre-processing as described is done only once in HemeLB. From the Lattice Boltzmann solver point of view there is no need for repartitioning because load stays the same for every processor. Additionally solving dominates the computation time. Thus HemeLB offers sufficient parallelism for efficient execution on exascale

systems. However this only holds true for HemeLB as is. The situation changes if post-processing comes into play. This can shift work load significantly as detailed above. Repartitioning will very likely be needed.

5 Interface to pre-processing

The HemeLB case study shows a clear need for an adaptive pre-processing. Methods used in post-processing can have a serious impact on load balance, yet core computation remains constant. It is likely, however, that this will not be true for other applications. These will have an intrinsic need for repeated repartitioning and redistribution. Likewise, post-processing will also impose further constraints.

This results in a strong demand for an interface supplying the pre-processing phase with additional performance relevant information. It should take care of passing information on work and memory load of the simulation parts to the pre-processing stage. Load balance can then be adjusted appropriately. Important aspects of this interface are discussed below.

5.1 Post-processing needs

Volume rendering and streak lines are techniques which can be performed fully in-situ (see [1]). Communication between processors is not needed. Thus, only local computation time increases. Particle tracing, on the other hand, demands additional communication overhead.

In the graph representation discussed in section 3 for the ParMETIS case study, computational costs for volume rendering and streak line techniques can obviously be considered by vertex weights while edge weights can describe communication costs used in particle tracing. The data structures from section 3 are well suitable to store these additional weights. Thus, once provided to pre-processing, the partitioning library of choice incorporates this information. A better load balance, also for post-processing, can be achieved.

Measurement of these weights is straightforward and can be performed every cycle. Estimations for the very first cycle of a simulation can be obtained from former runs. Weights are then passed to the interface. This sounds simple; however, there is other post-processing information, e.g., a real-time change of view. How this complex information should be passed along is not that clear. Here more investigation is needed.

5.2 Computation needs

Unlike HemeLB with its Lattice Boltzmann ansatz, other applications provide a variety of calculation methods. Multiple stages are possible, and communication may be needed in between. Other constraints also come into play. Therefore, computation and communication time change distinctly in the course of the simulation. Repeated repartitioning is required to obtain a good load balance at all times.

Here, again, calculation time and communication costs can be mapped to vertex and edge weights, respectively. Both quantities have to be monitored and passed to the interface. For an exascale simulation, this will be quite challenging. Here, forwarding of the gathered information on weights is not the main problem. Data structures mentioned earlier are pretty capable of doing this. Measurement – or a good prediction - of both quantities may be problematic. It requires decent code adaptation, yet is necessary to achieve well-balanced load for the simulation.

5.3 Additional needs

An open issue of the interface layout is its scope. The interface has to couple core computation and post-processing to pre-processing. This already covers almost the whole simulation. Here another particular aspect of exascale software comes to mind: Huge clusters will have to provide fault tolerance and a corresponding framework, toolbox or interface. Started before the simulation, it will do its work until the last line of code. Hence it will also cover the entire simulation run. A combination of both interfaces is imaginable as both of them exchange information among different parts of the simulation and different parts of the system.

A major benefit of linking fault tolerance and pre-processing interfaces surely is uniformity. One framework for both interfaces enables synergies which otherwise would not be there. Adaptive restarting with old data but recent partitioning information due to hardware failure is one example. Instant repartitioning and redistributing after a loss of a specific core or network connection would be possible.

This would not come easily; the main drawback in this technique concerns the amount of code change needed. Fault tolerance and steered pre-processing will have their individual impact in coding which cannot be bypassed. In order to really make use of synergies in a combined framework more effort will have to be put into the code.

© CRESTA Consortium Partners 2011

6 References

[1] CRESTA Deliverable 5.2.2, *post-processing: data format (hierarchical, multi-resolution) and algorithms definition*

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

[3] K. Schloegel, G. Karypis, and V. Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Proc. Supercomputing 2000*, 2000.

[4] Mazzeo, MD and Coveney, PV (2008) HemeLB: A high performance parallel lattice-Boltzmann code for large scale fluid flow in complex geometries. COMPUT PHYS COMMUN , 178 (12) 894 - 914. 10.1016/j.cpc.2008.02.013.

[5] 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/

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

© CRESTA Consortium Partners 2011