Dynamic Load Balancing for Large-Scale Multiphysics Simulations

Niclas Jansson[1(✉)], Rahul Bale[1(✉)], Keiji Onishi[1], and Makoto Tsubokura[1,2]

[1] RIKEN Advanced Institute for Computational Science, Kobe, Japan
{leifniclas.jansson,rahul.bale}@riken.jp
[2] Department of Computational Science, Graduate School of System Informatics, Kobe University, Kobe, Japan

Abstract. In parallel computing load balancing is an essential component of any efficient and scalable simulation code. Static data decomposition methods have proven to work well for symmetric workloads. But, in today's multiphysics simulations, with asymmetric workloads, this imbalance prevents good scalability on future generation of parallel architectures. We present our work on developing a general dynamic load balancing framework for multiphysics simulations on hierarchical Cartesian meshes. Using a weighted dual graph based workload estimation and constrained multilevel graph partitioning, the required runtime for industrial applications could be reduced by 40% of the runtime, running on the K computer.

Keywords: HPC · Load balancing · Multiphysics · BCM

1 Introduction

Load balancing is an essential component in today's large scale multiphysics simulations. With an ever increasing amount of parallelism in modern computer architecture, it is essential to remove even the slightest workload imbalance. As it could severely impact application's scalability. Traditionally, load balancing is seen as a static problem, closely related to the fundamental problem of parallel computing, namely data decomposition. Data is often decomposed either offline by a preprocessor or online in the initial steps of a simulation. This decomposition is typically performed with respect to the underlying discretization of the computational domain, with the aim of evenly distributing the cells, for example tetrahedra or hexahedra in unstructured meshes, or blocks in the case of Cartesian block structured meshes.

However, such a decomposition assumes that the workload for each cell is uniform. For certain problems this is true, but for a large class of problems it is not, for example, reactive flows, where the cost of computing the chemical reactions is different depending on the species concentration in a cell. Another example is when immersed boundary methods are employed. There, the cost of computing one cell will be different depending on whether the cell is cut by a surface or not, and also whether the geometry is stationary or moving through the domain.

E. Di Napoli et al. (Eds.): JHPCS 2016, LNCS 10164, pp. 13–23, 2017.
DOI: 10.1007/978-3-319-53862-4_2

In this paper we present our work on developing a generic dynamic load balancing technique for the Building Cube Method (BCM) [5], suitable for large-scale multiphysics problems. Our method is based on the load balancing framework used in DOLFIN HPC [3], a framework for automated scientific computing. The rest of this paper is organized as follows. In Sect. 2, we present the theory for static load balancing and discuss its limitations. Section 3 extends this theory to dynamic load balancing, with the introduction of workload modeling and re-partitioning schemes. In Sect. 4 we evaluate the performance. We present a discussion of the predictivity of the load balancing framework in Sect. 5 and, lastly, give conclusions and outline future work in Sect. 6.

2 Static Load Balancing

In parallel computing, the idea of data decomposition or static load balancing is simple, namely divide the workload evenly across all the workers. This can be formulated as a partitioning problem.

Given a set of cells $\mathcal{C}$ from a domain $\mathcal{T}$, the partitioning problem for p workers can be expressed as, find p subsets $\{\mathcal{T}^i\}_{i=1}^p$ such that:

$$\mathcal{T} = \cup_{i=1}^p \mathcal{T}^i \quad \text{and} \quad \mathcal{T}^i \cap \mathcal{T}^j = \emptyset, \quad i \neq j \tag{1}$$

with the constraint that the workload:

$$W(\mathcal{T}^i) = |\{\mathcal{C} \in \mathcal{T} \mid \mathcal{C} \in \mathcal{T}^i\}|$$

should be approximately equal for all subsets.

Solving Eq. 1 can be done in several ways. The least expensive, geometric methods, such as space filling curves [1] only depend on the geometry of the domain. These methods are fast, but do not take into account the topology, hence the data dependencies between different cells in the domain are not optimized. For Cartesian meshes such as BCM, neglecting the consideration of data dependencies is less severe. All the cells have the same amount of neighbors, and if the decomposition method tries to assign cells which are close to each other to one worker (in the geometrical sense), data dependencies will automatically be approximately balanced. However, if woarkload is not uniform across cells, or if data dependencies between the cells are assymetric, we have to resort to graph methods in order to solve Eq. 1.

Graph methods do not solve Eq. 1 directly, instead the following k-way partitioning problem is considered: Given an undirected graph $G = (V, E)$ with nodes V and edges E, split V into k subsets $\{Q_j\}_{j=1}^k$ with the constraint that the number of nodes in each subset should be roughly equal, and the number of edges cut should be minimized. If we model the computational work by V and the data dependencies in the domain by E, we see that this method will balance both the computational work and the dependencies. Furthermore, if we instead consider a weighted graph G and add the constraint that the sum of all weights should be roughly equal in all subsets Q_j, the method can then, by allowing multiple weights in the graph, handle a non uniform workload.

3 Dynamic Load Balancing

In order to perform dynamic load balancing, two components are needed: First, a way to evaluate the workload, and second, a way to decompose the data with the constraint to even out the workload. Using the graph based methods from Sect. 2 we can compute new constrained partitions of our computational domain. But the challenge is to be able to evaluate the current and future workloads, and decide if load balancing is needed.

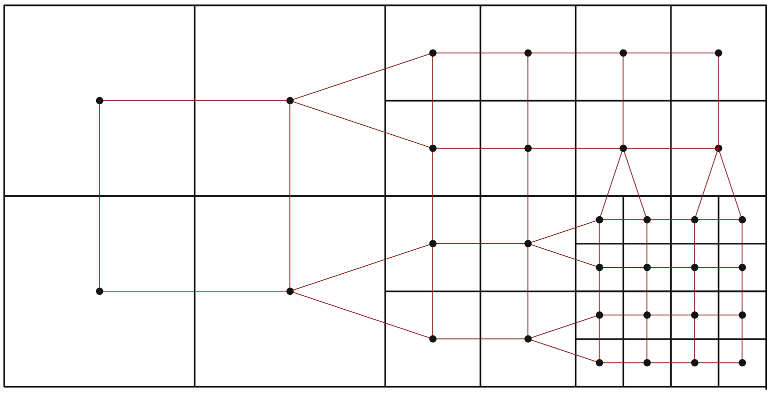


Fig. 1. Example of the dual graph of a BCM mesh.

3.1 Workload Modeling

We model the workload by a weighted dual graph of the underlying Building Cube mesh (see Fig. 1). Let $G = (V, E)$ be the dual graph of the mesh, with nodes V (one for each cube) and edges E (connecting two nodes if their respective cubes share a common face), q be one of the partitions, and let w_i be the computational work (weights) assigned to the graph. The workload of partition $q \in \mathcal{T}$ is then defined as:

$$W(q) = \sum_{w_i \in w_q} w_i$$

Let W_{avg} be the average workload and $W_{\max}$ be the maximum, then the graph is considered imbalanced if:

$$W_{\max}/W_{\text{avg}} > \kappa \tag{2}$$

where κ is the threshold value determined depending on the current problem and/or machine characteristics.

To model a simulation's workload, we finally have to assign appropriate values to the graph's weights w_i. In order to have a fine-grained control over the workload, we let each node have j weights $w_i^{v_j}$, representing the computational work for the given node, and each edge k weights $w_i^{e_k}$, representing the communication cost (data dependencies between graph nodes). The total weight for a given graph node is then given by

$$w_i = \sum_j w_i^{v_j} + \sum_k w_i^{e_k}$$

For a typical simulation, we always assign the number of grid points in each cube to $w_i^{v_1}$ and the size of the halo (number of grid points to exchange) to $w_i^{e_k}$ for each of the graph edges connecting to node V_i. One or several more weights are later added to the graph node to model the additional computational cost of chemical reactions or immersed bodies. Additional weights can also be added to the edges, but we limit the present study to model only the halo exchange cost. The graph is finally partitioned by a graph partitioner, with the weights as an additional balancing constraint. Thus, new load-balanced partitions are obtained, as illustrated in Fig. 2.

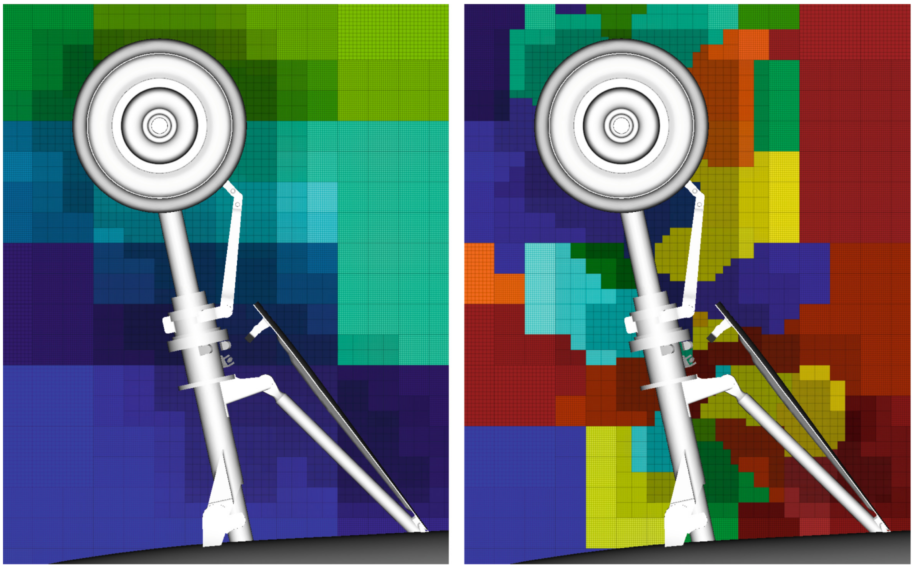

(a) Unbalanced (Z-ordering). (b) Balanced wrt. geometry.

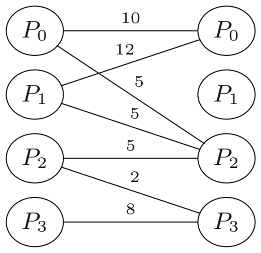
Fig. 2. Load balancing wrt. immersed geometry and fluid cells, colored by MPI rank.

3.2 Intelligent Remapping

For transient problems where the computational cost changes rapidly, it might not be feasible to load balance as soon as the workload changes. Therefore, the framework tries to minimize the data flow in two different ways. First, it uses the threshold value κ to filter out small workload fluctuations. Second, if load balancing is necessary, the algorithm tries to minimize data movement as much as possible.

Given a set of new partitions $\mathcal{T}'$ from an already partitioned domain $\mathcal{T}$, if the new partitions are assigned such that a minimal amount of data has to be moved from $\mathcal{T}$ to form $\mathcal{T}'$, we have achieved our goal of minimizing data movement. This can be solved using a method referred to as intelligent remapping.

Given an imbalanced workload. New partitions $\mathcal{T}'$ are computed using a constrained graph method. The result is then placed in a matrix S, where each entry S_{ij} is the number of graph vertices in a partition $\mathcal{T}^i$ which would be placed in the new partition $\mathcal{T}'^j$. The goal is to keep as much data local as possible, hence the maximum row entry in S is kept local. This can be achieved by transforming S into a bipartite graph (Fig. 3), with edges e_{ij} weighted with S_{ij}, and solving the maximally weighted bipartite graph problem (MWBG) [6].



$$S = \begin{pmatrix} 10 & 0 & 5 & 0 \\ 12 & 0 & 5 & 0 \\ 0 & 0 & 5 & 2 \\ 0 & 0 & 0 & 8 \end{pmatrix}$$

Fig. 3. Example of a weighted bipartite graph and its corresponding matrix S.

Solving this problem is known to be expensive with a cost of $O(V^2 \log(V) + VE)$, where V and E refer to nodes and edges in the bipartite graph. In [6] it was shown that the bipartite graph problem can be solved in time $O(E)$ using a heuristic algorithm based on sorting the matrix S and using a greedy algorithm to reassign the partitions. But in the worst case $E \sim P^2$, where P is the number of processes used to run the simulation, this linear heuristic also quickly becomes too expensive to solve. In [4] we decreased complexity of the heuristic algorithm to $O(P)$ using parallel binary radix sort.

The heuristic algorithm assigns the largest (unassigned) partition from a sorted list generated from the similarity matrix S (row-wise). In [6], S was gathered onto single core and sorted in serial using a binary radix sort. Since the matrix is of size P x P, where P is the number of cores, sorting quickly becomes a bootlneck at scale. Therefore, in [4] the heuristic was modified to perform the sorting in parallel using byte sorting parallel radix sort.

When we combine graph-based data decomposition methods, workload modeling using a weighted dual graph and intelligent remapping, we arrive at the general dynamic load balancing framework, as expressed in Algorithm 1.

Algorithm 1. Dynamic load balancing framework.

```
for each partition $q \in \mathcal{T}$ do
    $W(q) = \sum_{w_i \in w_q} \left( \sum_j w_i^{v_j} + \sum_k w_i^{e_k} \right)_{w_i}$
end
$W_{\max} \longleftarrow ComputeGlobalMax(W)$
$W_{\text{avg}} \longleftarrow ComputeGlobalAverage(W)$
if $W_{max}/W_{avg} > \kappa$ then
    $\mathcal{T}' \longleftarrow ComputeNewPartitions(\mathcal{T})$
    $S \longleftarrow ConstructMatrix(\mathcal{T}')$
    $\mathcal{G} \longleftarrow SolveMWBG(\mathcal{T}')$
    $\mathcal{T} \longleftarrow RedistributeData(\mathcal{G})$
end
```

4 Performance Evaluation

The load balancing framework presented in this paper has been implemented in the multiphysics framework Cube, developed at RIKEN AICS. Cube is based on the Building Cube Method and uses different kinds of immersed boundary methods to represent complex geometries. The framework is written in Fortran 2003, and uses a light-weight object-oriented approach for extensibility. The framework uses a hybrid MPI + OpenMP parallelization, in which each rank is assigned a set of cubes and thread parallelization is performed on per cube level basis, with two-dimensional slices in the z-direction of each cube. For scalability Cube uses parallel I/O in the form of MPI-I/O, and ParMETIS for graph partitoning.

To evaluate the performance of the load balancer, we used Cube to solve two different incompressible flow problems on the K computer and compared the total execution time for performing a fixed number of time steps for both an unbalanced (no load balancing) and a balanced case (using load balancing) on various numbers of cores. For both problems we used the QUICK scheme for the convective terms and an unsteady multigrid solver for the pressure. Time integration was performed using a second-order Crank–Nicolson method.

4.1 Immersed Boundary Method

A distributed Lagrange multiplier immersed boundary method [2,7] in Cube was used to represent the complex geometries (Fig. 4) in the numerical experiments. A Lagrangian-Eulerian approach is used in the implementation of the immersed boundary method because Lagrangian description is a very accurate method of representing complex, mobile immersed bodies (IB). In this approach, an Eulerian description is used to solve the equations governing the fluid motion, whereas a Material or Lagrangian description is used to represent the immersed body. The immersed body is discretized in to a discrete set of Material or Lagrangian points. The interaction between the fluid and the immesrsed

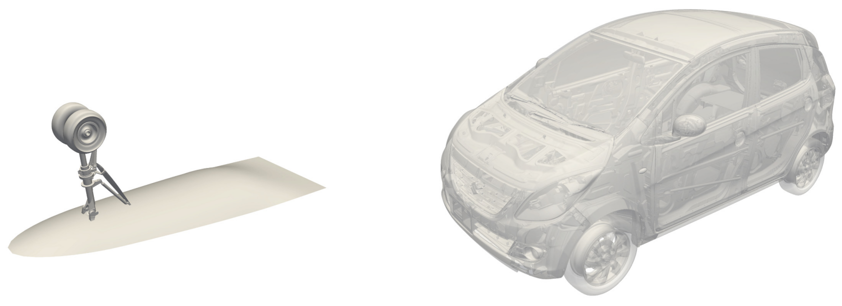

(a) Nose landing gear geometry. (b) Full car model.

Fig. 4. Geometries used to evaluated the performance of the load balancing framework.

body is enabled through interpolation operators such as the smoothed Dirac delta function, inverse distance interpolation, or trilinear interpolation. In this work we use the smoothed Dirac delta function for the interpolation between Lagrangian-Eulerian domains.

A spatial decomposition approach is employed to discretize the combined Lagrangian-Eulerian system, wherein the Lagrangian domain is discretized on the basis of the Eulerian domain decomposition. For a given rank, this ensures data locality between Lagrangian and Eulerian domains, avoiding MPI communication for Lagrangian-Eulerian interpolation.

4.2 IB Workload Modeling

In the load balancer, the weights were assigned as described in Sect. 3.1, with the additional immersed boundary cost added to $w_i^{v_2}$, modeled as $\gamma \cdot n_{\text{particles}}$, where $n_{\text{particles}}$ is the number of Lagrangian particles. The choice of the parameter γ is not trivial and it depends on the relative number of Lagrangian-Eulerian interpolation operations for a given purely Eulerian stencil operation. The interpolation between Lagrangian and Eulerian meshes involves $\sim 2\text{n}^3$ operations for a given Lagrangian particle. Here, n depends on the type discrete delta function, e.g., for a 3-point delta function $n = 4$. n^3 could be a good candidate for the cost parameter γ. But, Lagrangian-Eulerian interpolation is required only once every time step, whereas purely Eulerian stencil operations depend on iterative processes such as solution of the Poisson equation. If N_{p-iter} is the number of Poisson solver iterations in one time step, then one could choose $\gamma = n^3/N_{p-iter}$. Therefore, γ would depend on the type of discrete delta function and the type of Poisson solver, but for most cases $n^3/N_{p-iter} \sim \mathcal{O}(1)$. Thus, we choose γ in the range of 1–4 for immersed body applications. It is to be noted that γ is application dependent, and a informed choice has to be made for its value.

4.3 Load Balancing Threshold κ

For the present analysis, a load imbalance threshold, $\kappa = 1.05$, is chosen, i.e. the load balance is triggered if there is an imbalance of 5% or more. In the two applications we consider, flow around a vehicle and nose landing gear, the immersed geometries are stationary, so load balancing is triggerd only once during the simualtion. Thus, κ only determines when load balancing and data redistribution is triggered; it has no influence on how the balanced or unbalanced cases perform, consequently it does not affect the overall runtime of the simulation. In more dynamic cases, such as applicaitons with rapidly moving IBs, the simulation runtime will be affected by the choice of κ. A small value of κ will frequently trigger data redistribution. Which will increase the overall simulation runtime. Thus, for dynamic applications, a parameteric study of the effect of κ would be necessary in order to choose an optimal value of κ.

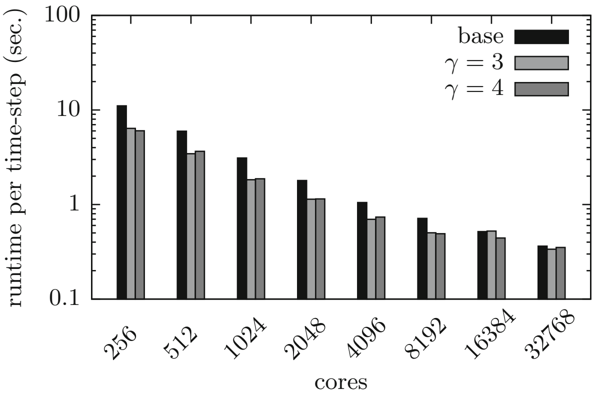
4.4 Nose Landing Gear

The first problem is based on the nose landing gear (Fig. 4a) case from AIAA's BANC series of benchmark problems. Our setup uses a mesh consisting of 48255 cubes, subdivided into 16 cells in each axial direction, and the landing gear consists of $0.5M$ surface triangles.

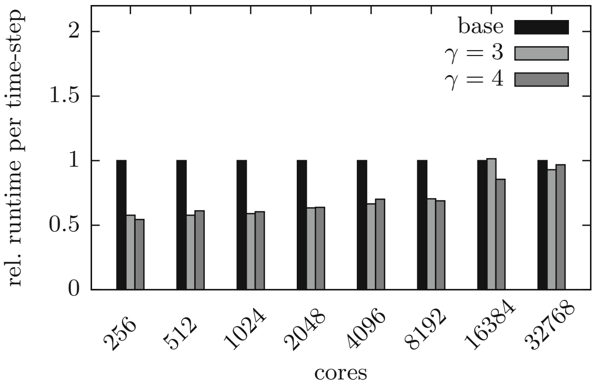
In Fig. 5a we present the time required to perform one timestep for the unbalanced case and for the balanced case when γ is set to 3 and 4, respectively. From the results we can observe that using the load balancer results in approximately 60% resuction in the unbalanced runtime. As the number of cores increases, the gains of load balancing diminishes. This is most likely due to the fact that when using a relative small model, such as the landing gear, and few cubes in the mesh, the initial data decomposition will (for large core counts) will result in more partitions around to the geometry and indirectly balance the workload automatically. A value of $\gamma = 1$ resulted in a runtime that was approximaltely equal to the unbalanced case. This indicates that values of 3 and 4 for γ are reasonable choices for load balancing nose landing gear type geometries. The lack of Lagrangian communication cost in the model could also affect the result. Figure 5a shows the relative runtime of both load balanced cases normalized by the unbalanced runtime.

4.5 Full Car Model

As a second example we simulate the flow past a full car model (Fig. 4b). The numerical methods used for this problem are exactly the same as for the landing gear benchmark. We use a mesh consisting of 38306 cubes with 16^3 cells per cube and a car model consisting of 12.5 M surface triangles.



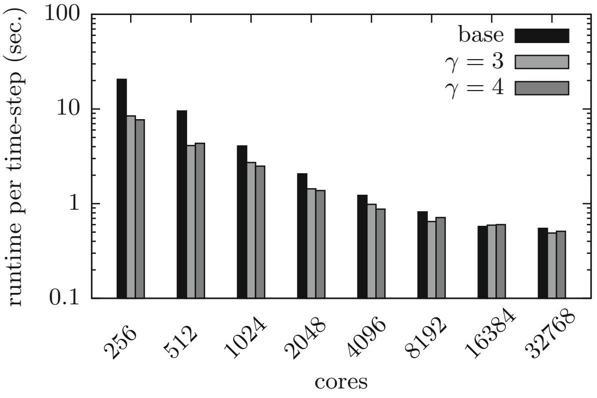
(a) Runtime per time-step.



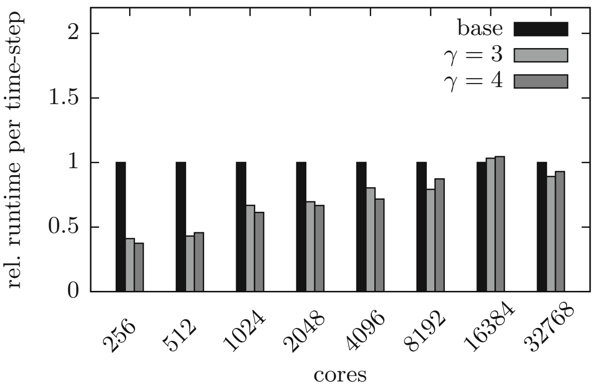
(b) Runtime per time-step normalized by the runtime of the unbalanced case.

Fig. 5. Runtime per time-step and relative runtime for the nose landing gear.

The runtime per timestep presented in Fig. 6a shows the results of the balanced case with $\gamma = 3$ & $\gamma = 4$ and the unbalanced case. The trends for the runtime in all the cases are similar to those of the nose landing gear case. We can see that for all the tested core counts the runtime is improved except for the 16384 core case. In the best case, which is 256 cores, the runtime of the balanced case reduced to 40% of the unbalanced case. The relative runtime is also plotted in Fig. 6a.



(a) Runtime per time-step.



(b) Normalized runtime per time-step.

Fig. 6. Runtime per time-step and relative runtime for the full car model.

5 Discussion

A key aspect of dynamic load balancing techniques is the prospect of load balancing not just current workloads, but future workloads as well. The evaluation of future workloads and ability of the load balancer to address such workloads depends of the type of applications. The applications considered in our work have static workload. The evaluation of future workloads is relevant to dynamic applications. For some dynamic applications it may be possible to evaluate, predict and address a future imbalance. Examples of such applications are simulations with moving geometries, simulations of spray dynamics, and others. When the velocity of an immersed geometry is known, the future location of the geometry and its workload can be evaluated in advance and addressed when necessary. In applications of spray dynamics, the rate of new spray particle injection and the average trajectory of the bulk particles can be used to predict the future workload and balance it accordingly. If these dynamic applications are coupled with adaptive mesh refinement (AMR), the ability to predict the future workload will be all the more useful. When AMR is in use, the overall workload of the system changes with time. In cases where the workload increases due to creation of new mesh cells, the ability to predict the future workload can reduce the cost of data redistribution. When the location of new mesh cells is known, the future workload can be evaluated in advance and the data redistribution can be carried out before the creation of cells to reduce the data redistribution cost. Conversely, when location of mesh cells to be destroyed, resulting in workload reduction, is known, the data redistribution can be deferred until after the cell destruction to reduce the redistribution cost [6].

6 Summary and Future Work

In this work we have investigated the feasibility of using dynamic load balancing techniques to improve the performance of multiphysics simulation using the Building Cube Method. Our results show that the runtime could be reduced by almost a factor of two fifth when using the load balancer. In the current study we have limited ourselves to flow problems, but we want to stress that the load balancing framework is generic and could be applied to any type of workload, as demonstrated when we incorporated the cost of Lagrangian particles in the workload modeling. Future work includes fine tuning of the workload modeling, especially focusing on the computational and communcation cost of, e.g., chemical reactions and Lagrangian particles.

Acknowledgments. This work was supported through the computing resources provided on the K computer by RIKEN Advanced Institute for Computational Science.

References

1. Bader, M.: Space-Filling Curves. Texts in Computational Science and Engineering, vol. 9. Springer, Heidelberg (2013)
2. Bhalla, A.P.S., Bale, R., Griffith, B.E., Patankar, N.A.: A unified mathematical framework and an adaptive numerical method for fluid-structure interaction with rigid, deforming, and elastic bodies. J. Comput. Phys. **250**, 446–476 (2013)
3. Jansson, N.: High performance adaptive finite element methods: with applications in aerodynamics. Ph.D. thesis, KTH Royal Institute of Technology (2013)
4. Jansson, N., Hoffman, J., Jansson, J.: Framework for massively parallel adaptive finite element computational fluid dynamics on tetrahedral meshes. SIAM J. Sci. Comput. **34**(1), C24–C41 (2012)
5. Nakahashi, K.: Building-cube method for flow problems with broadband characteristic length. In: Armfield, S.W., Morgan, P., Srinivas, K. (eds.) Computational Fluid Dynamics 2002, pp. 77–81. Springer, Heidelberg (2003)
6. Oliker, L.: PLUM parallel load balancing for unstructured adaptive meshes. Technical report RIACS-TR-98-01, RIACS, NASA Ames Research Center (1998)
7. Shirgaonkar, A.A., MacIver, M.A., Patankar, N.A.: A new mathematical formulation and fast algorithm for fully resolved simulation of self-propulsion. J. Comput. Phys. **228**(7), 2366–2390 (2009)