The Potential of Diffusive Load Balancing at Large Scale

Matthias Lieber
TU Dresden
01062 Dresden, Germany
matthias.lieber@tu-dresden.de

Kerstin Gößner
TU Dresden
01062 Dresden, Germany
kerstin.goessner@tu-dresden.de

Wolfgang E. Nagel
TU Dresden
01062 Dresden, Germany
wolfgang.nagel@tu-dresden.de

ABSTRACT

Dynamic load balancing with diffusive methods is known to provide minimal load transfer and requires communication between neighbor nodes only. These are very attractive properties for highly parallel systems. We compare diffusive methods with state-of-the-art geometrical and graph-based partitioning methods on thousands of nodes. When load balancing overheads, i. e. repartitioning computation time and migration, have to be minimized, diffusive methods provide substantial benefits.

Keywords

HPC; Dynamic Load Balancing; Workload Diffusion

1. INTRODUCTION

Load balance is one important challenge for HPC applications that use tens of thousands or even – in future Exascale systems – millions of computing nodes [8, 13]. Many scientific applications exhibit dynamic workload variations, e. g. due to adaptivity in time and/or space. Additionally, hardware variability is expected to increase [13]. Thus, dynamic load balancing is required that reacts to the variations and repartitions the application to minimize idling at synchronization points while keeping the edge-cut low. Ideally, a load balancing scheme would also cause little overhead such that it can be called frequently. Overheads are caused by the computing time of the method itself and by the amount of resulting task migration.

Motivation: Various different methods for repartitioning have been developed [17]. They can be categorized into geometrical methods that use coordinates of computational tasks to partition them and graph-based methods that use topological information of the tasks. Centralized repartitioners will clearly fail for highly parallel applications. However, even most of the parallel methods require at least some global communication, which hinders their scalability. In previous work [12] we developed a hierarchical geometrical method based on space-filling curves (SFCs) that performs only little global communication and demonstrated a very good scalability. However, the amount of migration is very high, which is typical for SFC-based partitioning. A fully distributed algorithm based on gossip is presented by Menon and Kalé [14]. It is fast and scales very well, but it does not take the edge-cut into account.

Contribution: In this paper we practically evaluate the potential of diffusive load balancing algorithms, which have been mainly developed in the 1990s, for large-scale dynamic load balancing. To the best of our knowledge, publications about the application of diffusive load balancing in HPC are very rare [7, 18], especially in the recent years.[1] In this work we compare the relevant performance metrics (load balance, run time, migration, edge-cut) of five different diffusive schemes with four other load balancing methods (SFC, recursive bisection, ParMetis, and hierarchical SFC) and demonstrate that there are no reasons not to use diffusive schemes. Especially when load balancing overhead matters, they provide substantial benefits.

EuroMPI '16, September 25 - 28, 2016, Edinburgh, United Kingdom

ISBN 978-1-4503-4234-6/16/09…$15.00

DOI: http://dx.doi.org/10.1145/2966884.2966887

2. DIFFUSIVE LOAD BALANCING

Diffusive load balancing operates on an undirected, connected graph $G = (V, E)$ of computing nodes, usually defined by the network topology. In several iterations, nodes balance a load value with their neighbors N_v and eventually the algorithm converges to global balance. In this way each node creates a load transfer vector for all its neighbors. In a second step, called task selection, local tasks are selected for migration to satisfy the load transfers.

Original diffusion (OD): In the original algorithm [6], each node v updates its load l_v per iteration i according to:

$$l_v^{i+1} = l_v^i + \sum_{w \in N_v} \alpha_{vw}(l_w^i - l_v^i)$$

where α_{vw} is the diffusion parameter. The optimal value depends on the topology of G.

Second-order diffusion (SO): The second-order algorithm [15] extends OD such that the previous iteration's transfer influences the current. The parameter $\beta \in (0, 2)$ controls the influence. Optimal values are derived in [9].

Improved diffusion (ID): In this algorithm, also called CHEBY [10], the update rule depends on the iteration i and its parameters are derived from the smallest and largest positive eigenvalues of the weighted Laplacian matrix of G. Their computation is required only once if G is fixed.

[1] One exception is their use in popular multilevel graph partitioners such as ParMetis [16], but here they are embedded into a much more complex method.

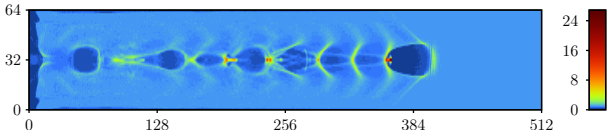


Figure 1: Visualization of the LWFA scenario's load distribution on a slice through the center of the 3D mesh. The load is shown relative to the average.

Chemotaxis-inspired diffusion (CT): The biologically inspired algorithm [5] extends OD by additionally exchanging a so-called signal. It is the difference of l_v and the capacity (target node load) and guides the load diffusion. CT requires an initial collective communication to determine the capacity and twice the number of messages per iteration compared to the previous methods.

Dimension exchange (DE): The dimension exchange algorithm [19] does not build on OD and, in a strict sense, is not a diffusion method. The main difference is that the current load is updated immediately before exchanging with the next neighbor, which allows load to flow faster. This requires proper scheduling of communication, which is straightforward on regular graphs such as meshes. However, due to more frequent synchronization, a single iteration may take longer compared to OD, SO, and ID.

Task selection: After load transfers have been computed, migrating tasks need to be selected to create a balanced partitioning. Typically tasks are communicating and thus form a graph that should be respected to reduce the amount of edges crossing partition borders. Like diffusion, task selection is usually implemented such that only neighbors communicate, if necessary, in several passes [7, 18].

3. BENCHMARK PROGRAM

We implemented the diffusive load balancing methods introduced in Sec. 2 in an MPI-based benchmark program. All three test scenarios consist of a 3D mesh of weighted tasks to be mapped onto a 3D mesh of nodes of size $k_1 \times k_2 \times k_3$.

Artificial scenarios: The POINT and BOX scenarios define a region of overloaded nodes that lead to a load balance of $avg(l_v)/max(l_v) = 0.9$. In POINT, only a single node at $k_i/2$ is overloaded, whereas in BOX a rectangular subset of nodes of size $k_i/2+1$ around $k_i/2$ for each dimension $i \in \{1,2,3\}$ is equally overloaded. Initially each node owns $8 \times 8 \times 8 = 512$ tasks of equal load.

Real application scenario: We use recorded workload data of a laser wakefield acceleration (LWFA) simulation with the open source particle-in-cell code PIConGPU [3, 4], shown in Fig. 1. We partitioned the grid of $64 \times 512 \times 32$ tasks into $16 \times 16 \times 8$ equally sized rectangular partitions leading to a load balance of 0.82 for the selected time step.

Diffusion implementation: Except for DE, the communication is implemented with persistent messages. Independent of the actual number of neighbors, three MPI functions are called per iteration: MPI_Startall, MPI_Waitall (recv), MPI_Waitall (send). Each message is 8 byte. We use optimal diffusion parameters for 3D meshes [19] in OD, SO, and CT ($\alpha = 1/6$) and also in DE ($\alpha = (1+sin(\pi/k_{max}))^{-1}$). For SO we choose $\beta = 1.8$, which turned out to be robust for all our configurations (we observed that the optimal β depends on the termination criterion, scenario, node count, and topology). For the signal diffusion in CT we use $\alpha_{sig} = 1/max(deg^s, deg^r)$ where deg^s and deg^r are the node degrees of sender and receiver, respectively.

Task selection: We implemented a simple task selection scheme that, for each neighbor separately, iteratively selects tasks that lead to the highest edge-cut gain (or lowest loss) until the load transfer is satisfied. Only one pass is executed, i. e. tasks do not hop across nodes. The only communication involved is sending the result to the neighbors. The complexity depends on the number of neighbors and local tasks, but not on the total number of nodes.

Non-diffusive methods: We included the geometrical methods recursive bisection (RCB) and Hilbert space-filling curves (HSFC) as well as graph partitioning with ParMetis' AdaptiveRepart [16] routine using the Zoltan load balancing library [1, 2] into the benchmark. Additionally, we ran our hierarchical SFC-based method (Hier SFC) that we implemented in the FD4 library [11, 12]. Note that these methods do not take the 3D mesh topology of the nodes into account.

HPC systems: We performed measurements on two Petaflop-class HPC systems: The IBM Blue Gene/Q system Juqueen (16 cores/node, PowerPC A2, 5D torus network, BGQ driver V1R2M4) and the Bull HPC cluster Taurus (24 cores/node, Intel Xeon E5 2860v3, Infiniband FDR fat tree, Intel MPI 5.1.3). On Juqueen, we embedded the benchmark's 3D node mesh into the 5D torus by two times folding one dimension into two (or two dimensions into three), which leads to a mapping where all neighbors in the 3D mesh are also neighbors in the hardware network. Except for the last experiment, we use one process per Juqueen node. Note that we do not take the hardware topology on Taurus into account and always use all 24 cores per hardware node.

Performance metrics: We report the max. **run time** of the load balancing algorithm among all processes. For diffusive methods, we additionally report the **iteration** count until our termination criterion $avg(l_v)/max(l_v) = 0.999$ for the virtual load is reached. After an initial run to determine the number of iterations, we run (without checking the criterion) 61 repetitions on Taurus (19 on Juqueen, where we observed nearly no variation) and report the median run time. For ID, we do not include the eigenvalue calculation in the run times. With **TransferMax** we indicate the maximum load transfer between any two nodes relative to the average load $avg(l_v)$, whereas **TransferTot** denotes the sum of all load transfers relative to the total load Σl_v.

Performance metrics that include task selection: We report the remaining **load imbalance** after task selection as $max(l_i)/avg(l_i) - 1$, where 0 implies perfect balance. With **MigrationMax** we indicate the maximum number of tasks a node sends and receives. **MigrationTot** denotes the total number of migrated tasks relative to the total number of tasks in the mesh. The edge-cut, i. e. the number of edges in the task mesh cut by partition borders, is measured as the maximum among all nodes **EdgeCutMax** and the total amount **EdgeCutTot** relative to all edges.

4. RESULTS

Influence of scenario: Fig. 2 shows results for 2048 nodes arranged in a $16 \times 16 \times 8$ mesh. In all cases, the task mesh consists of 1 048 576 tasks. Looking at the diffusion methods first, we can generally summarize w. r. t. iterations, run time, migration, and edge-cut that the POINT scenario induces the lowest costs and the LWFA scenario the

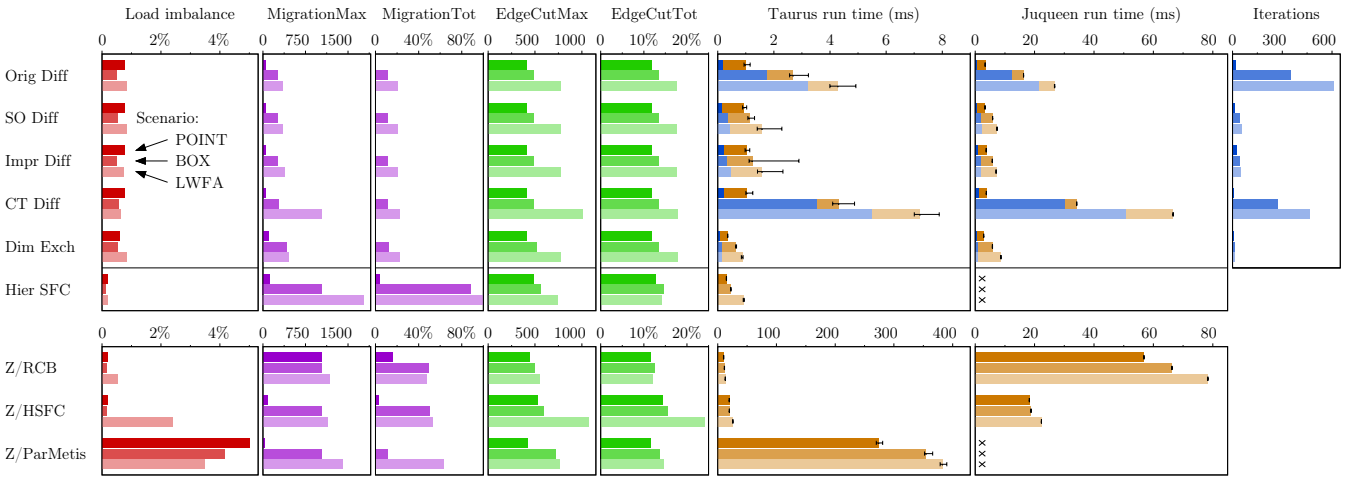


Figure 2: Performance of diffusive algorithms and the hierarchical SFC-based method (upper part) compared to Zoltan methods (lower part) on a mesh of 16×16×8 = 2048 nodes. The three bars per method show metrics for the POINT (top), BOX (middle), and LWFA (bottom) scenarios. Notes: Scales for the Zoltan and diffusion metrics are the same except for the Taurus run times. Run time for diffusive methods is split up in diffusion (blue/left part) and task selection (brown/right part). Error bars denote 25/75 percentiles of total run time. Skipped runs on Juqueen are marked with a cross.

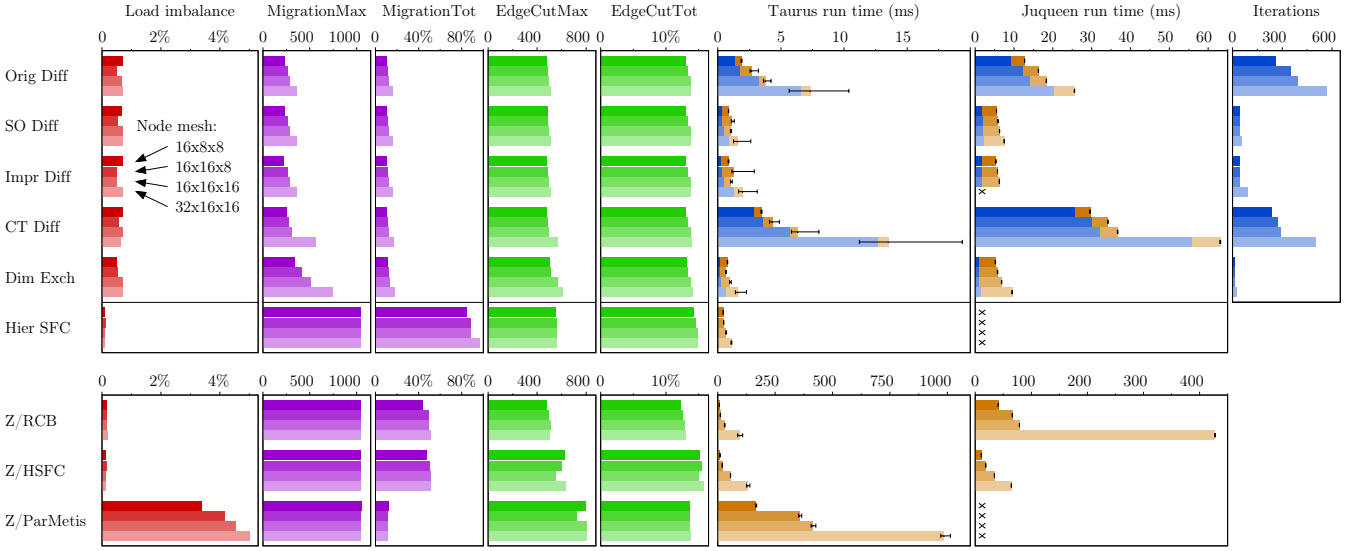
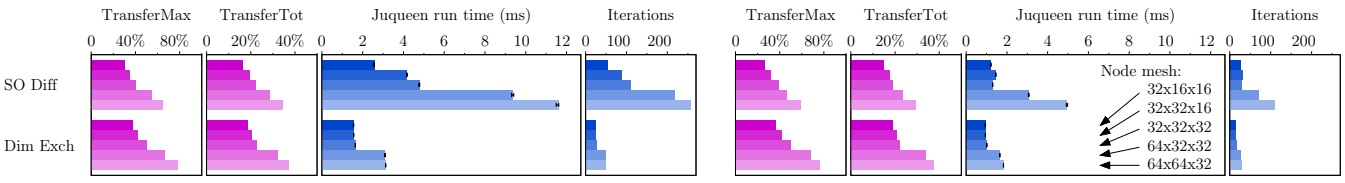


Figure 3: Scalability of diffusive algorithms and the hierarchical SFC-based method (upper part) compared to Zoltan methods (lower part) for the BOX scenario. The four bars per method show metrics for 1024 (top), 2048, 4096, and 8192 (bottom) nodes. Notes: Scales for the Zoltan and diffusion metrics are the same except for the run times. Run time for diffusive methods is split up in diffusion (blue/left part) and task selection (brown/right part). Error bars denote 25/75 percentiles of total run time. Skipped runs on Juqueen are marked with a cross.



(a) Target balance = 0.999

(b) Target balance = 0.99

Figure 4: Scalability of second order diffusion and dimension exchange (without task selection) for the BOX scenario. The five bars per method show metrics for 8192 (top), 16 384, 32 768, 65 536, and 131 072 (bottom) nodes, which are mapped to hardware cores in this case.

highest. Regarding run time, SO, ID and DE are the clear leaders. However, DE computes larger transfers, which induces slightly higher migration costs. SO and ID, on the other side, always result in approx. the same minimal task migration among the diffusive methods.

Hier SFC is the fastest method in two cases, but it generates clearly more migration, especially for the LWFA scenario. The Zoltan methods take much longer then the tree fastest diffusive algorithms. But on Juqueen, due to fast MPI collectives, HSFC is able to outperform at least OD and CT in some cases. RCB appears to be the best Zoltan method w. r. t. balance and edge-cut and improves over diffusion in some cases, while ParMetis is not able to provide a high load balance. However, none of the Zoltan methods are able to achieve the low task migration the diffusive methods provide for all three scenarios.

Scalability up to 8192: With the same set of methods we show a scalability comparison for the BOX scenario in Fig. 3. Regarding the diffusive methods and Hier SFC, we can make the same observations as before: SO, ID, and DE are the best with DE resulting in higher task migration and Hier SFC is approx. 1.5 times faster but generates by far the largest migration among all. The run time advantage of the three best diffusive methods over the Zoltan methods gets even greater at larger scales. At 8192 nodes, SO is 8.6 times faster than HSFC on Juqueen and even 60 times faster than RCB on Taurus. Also the migration costs are clearly higher with Zoltan, except for the total migration with ParMetis at the expense of load balance.

On Juqueen we can also confirm the expectation that the time per iteration does not change with the number of nodes. It is 38 μs for SO, 54 μs for DE, and 110 μs for CT.

Scalability up to 128 Ki: To study the behavior at larger scales, we selected the two best diffusive algorithms SO and DE (favoring SO over ID, since it is much simpler and achieves approx. the same results) and performed measurements with up to 128 Ki nodes without task selection, whose run time does not depend on the node count. Here, we treated the 16 cores per Juqueen node as a sixth network dimension to emulate a system with a larger node count. Fig. 4 shows the results for two different target load balances. We can see that DE increases its run time with the size of the largest dimension, leading to a better scalability behavior than SO. Reducing the target load balance to 0.99, which might be sufficient for most applications since load is usually estimated, decreases the run time by factor 2.1–3.7 with SO and by factor 1.7–1.9 with DE. Even if we would add the task selection time, diffusion allows to stay within the low millisecond range for load balancing of 128 Ki nodes.

5. CONCLUSIONS AND FUTURE WORK

We compared five diffusive load balancing algorithms with other load balancing methods on thousands of nodes. The results show that diffusive schemes are attractive when load balancing overhead, i. e. computation time and task migration, has to be very low, e. g. in case of frequent rebalancing.

However, there are some remaining research questions to be solved for the efficient application of diffusive schemes: (a) What is a fast and high-quality method for task selection allowing to trade-off balance, migration and edge-cut? (b) How do we scalably implement the termination criterion for diffusion when no fast collectives are available? E. g. would an estimation or auto-tuned value for the iteration count be sufficient? And finally, (c) how can we apply diffusion on today's common hardware topologies like fat trees?

Acknowledgments: We thank the Jülich Supercomputing Centre, Germany, for access to JUQUEEN. This work is supported by the 'Center for Advancing Electronics Dresden' (cfaed) and the German priority program 1648 'Software for Exascale Computing' via the project FFMK.

6. REFERENCES

[1] Zoltan home page. http://www.cs.sandia.gov/Zoltan.

[2] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia Parallel Toolkits for Combinatorial Scientific Computing: Partitioning, Ordering, and Coloring. *Scientific Programming*, 20(2):129–150, 2012.

[3] PIConGPU home page. http://picongpu.hzdr.de.

[4] M. Bussmann et al. Radiative Signatures of the Relativistic Kelvin-Helmholtz Instability. In *SC '13*, 2013.

[5] G. Canright, A. Deutsch, and T. Urnes. Chemotaxis-inspired load balancing. *ComPlexUs*, 3:8–23, 2006.

[6] G. Cybenko. Dynamic Load Balancing for Distributed Memory Multiprocessors. *J. Parallel Distr. Com.*, 7(2):279–301, 1989.

[7] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM. *Parallel Computing*, 26(12):1555–1581, 2000.

[8] J. Dongarra et al. The International Exascale Software Project Roadmap. *Int. J. High Perform. C.*, 25(1):3–60, 2011.

[9] R. Elsässer, B. Monien, and R. Preis. Diffusion Schemes for Load Balancing on Heterogeneous Networks. *Theory Comput. Sys.*, 35(3):305–320, 2002.

[10] Y. F. Hu and R. J. Blake. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing*, 25(4):417–444, 1999.

[11] FD4. http://www.tu-dresden.de/zih/clouds.

[12] M. Lieber and W. E. Nagel. Scalable high-quality 1D partitioning. In *HPCS 2014*, pages 112–119, 2014.

[13] R. Lucas et al. Top Ten Exascale Research Challenges. DOE ASCAC subcommittee report, 2014.

[14] H. Menon and L. Kalé. A Distributed Dynamic Load Balancer for Iterative Applications. In *SC '13*, 2013.

[15] S. Muthukrishnan, B. Ghosh, and M. H. Schultz. First and Second Order Diffusive Methods for Rapid, Coarse, Distributed Load Balancing. *Theory Comput. Sys.*, 31:331–354, 1998.

[16] K. Schloegel, G. Karypis, and V. Kumar. A Unified Algorithm for Load-balancing Adaptive Scientific Simulations. In *SC 2000*, 2000.

[17] J. D. Teresco, K. D. Devine, and J. E. Flaherty. Partitioning and Dynamic Load Balancing for the Numerical Solution of Partial Differential Equations. volume 51 of *LNCSE*, pages 55–88. Springer, 2006.

[18] J. Watts and S. Taylor. A Practical Approach to Dynamic Load Balancing. *IEEE T. Parall. Distr.*, 9:235–248, 1998.

[19] C. Z. Xu, B. Monien, R. Lüling, and F. C. M. Lau. Nearest-neighbor algorithms for load-balancing in parallel computers. *Conc. Pract. E.*, 7:707–736, 1995.