2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

Dynamic Load Balancing for Unstructured Meshes on Space-Filling Curves

Daniel F. Harlacher, Harald Klimach, Sabine Roller
Applied Supercomputing in Engineering
German Research School for Simulation Sciences GmbH
and RWTH University
Aachen, Germany
e-mail: d.harlacher@grs-sim.de, h.klimach@grs-sim.de, s.roller@grs-sim.de

Christian Siebert, Felix Wolf
Parallel Programming
German Research School for Simulation Sciences GmbH
and RWTH University
Aachen, Germany
e-mail: c.siebert@grs-sim.de, f.wolf@grs-sim.de

***Abstract*—Load imbalance is an important impediment on the path towards higher degrees of parallelism—especially for engineering codes with their highly unstructured problem domains. In particular, when load conditions change dynamically, efficient mesh partitioning becomes an indispensable ingredient of scalable design. However, popular graph-based methods such as those used by ParMetis require global knowledge, which effectively limits the problem size on distributed-memory machines. On such architectures, space-filling curves (SFCs) offer a memory-efficient alternative and many sophisticated schemes have already been proposed. In this paper, we present a simple strategy based on SFCs that is custom-tailored to the needs of static meshes with dynamically changing computational load. Exploiting the properties of this class of problems, it is not only easy to implement but also reduces memory requirements substantially. Moreover, exclusively relying on MPI collective operations, our load-balancing scheme also offers portable performance across a broad range of HPC systems. Experimental evaluation shows excellent scaling behavior for up to 16,384 cores on a Nehalem-Infiniband system and up to 294,912 processes on a Blue Gene/P system.**

***Keywords*-partitioning; space-filling curve; load balancing; scalability**

I. INTRODUCTION

Fluid dynamic applications in the field of industrial engineering require high degrees of parallelism to achieve an acceptable time to solution for large problem sizes. Typical mesh-based approaches therefore rely on suitable partitioning strategies to distribute the computational load across the set of processes. This partitioning is an optimization task with two goals. The work load has to be distributed evenly and at the same time the interfacing boundaries between partitions should be as small as possible. The first requirement is derived from the need to avoid waiting times of processing units, while the second is imposed by the need to minimize the time spent in communication. Although this optimization problem is NP-hard in general, several successful heuristics exist. A popular option for the partitioning of meshes is provided by the ParMetis [10] library, which uses a graph-based algorithm. This library solved the partitioning problem for unstructured meshes very well for a long time. However, with an increasing number of processes, graph-based partitioning algorithms seem to reach their scalability limits. In particular, one scalability problem arises due to the required memory. As the sophisticated graph-partitioning algorithms need information on the complete graph of the unstructured mesh, their memory consumption grows linearly with the graph size. Satisfying this requirement becomes infeasible at large scale, raising the need for alternatives which avoid this memory problem.

Such an alternative is offered by the second class of partitioning methods, which are based on space-filling curves (SFCs). The basic concept of performing such a partitioning of unstructured meshes using space-filling curves has been described before [1]. SFCs map the one-dimensional unit interval onto a higher dimensional space such that neighboring points on the unit interval are also neighboring points in the target space. Thus, SFCs also preserve some locality in the inverse mapping from the higher dimensional space onto the unit interval. This method ignores the edges of the full graph information. Instead, it relies on the spatial properties of the curve to ensure a reasonable partition shape. An upper limit of expected remote accesses in SFC partitioned domains has been shown by Tirthapura et al. for arbitrary curves [11]. Their analysis demonstrates that the approach behaves well and results in an acceptable communication overhead, even for sparse meshes with complex embedded geometries. Usually, the communication surface can be improved by taking advantage of the full graph information. However, this improvement is not guaranteed—the limited communication surface of the SFC is usually good enough for the class of applications considered here. For these reasons, a partitioning based on SFCs appears attractive and opens a path to low-memory partitioning strategies.

This work targets a dynamic solver for compressible flows with p and t adaptivity. That is, the numerical approximation of the flow field can be adapted by the chosen polynomial degree p within each element, and the local time step of the explicit time marching scheme is chosen ideally for each element to fulfill the numerical stability criteria [5]. With this numerical scheme, it is possible to avoid expensive re-meshing (h-adaptivity), while the computation can still adapt dynamically to the simulated field. This adaptivity on a rigid spatial mesh, which tries to allocate the computational resources only to those spots where they are actually needed, introduces computational imbalances. Different elements can drastically vary in their individual computation times, due to different time step sizes and the flow field seen by them. This load imbalance can also be very volatile during the overall running time of the simulation.

Commonly available partitioning implementations are often concerned with the more general case where re-meshing is important, and therefore introduce additional computational complexity, which is not needed for the application at hand. This work presents a specific and efficient balancing solution for the problem of dynamically changing work on a fixed mesh. Balancing of one-dimensional workload distributions is also known as chains-on-chains partitioning (CCP) in the literature [7]. With the help of a SFC, the partitioning of three-dimensional unstructured meshes can be reduced to this type of problem. The introduction of a one-dimensional sorting for the unstructured mesh has also some further advantages with respect to parallel distributed simulations. With a known order of elements, neighbor identification becomes locally computable, and files can be read and written completely in parallel when following the SFC-induced layout. We implemented such a balancing solution for distributed systems using the Message Passing Interface (MPI) and the Morton curve ordering. Our evaluation shows that this implementation is highly scalable with respect to the required time and memory. As a reference, we compare the SFC-based partitioning to ParMetis with the main focus on the memory consumption for large scale simulations. Finally the dynamic deployment of the algorithm in the adaptive flow solver is described and the observed impact on the application is analyzed.

II. Partitioning Strategy

With the advent of highly distributed parallel systems in high performance computing, a major issue for algorithms is the memory consumption per process when deployed on many processes. Unfortunately, graph-based partitioning as used in ParMetis, does not cope well in this respect and does not work for applications on larger numbers of processes. In contrast to that, partitioning schemes based on a SFC ordering of elements yield the possibility to reduce the memory consumption to a constant size, independent of the number of processes. When a SFC is used to partition an unstructured mesh, the order of the distributed elements must be kept, to maintain the locality property of the resulting partitions, leading to a CCP problem for the load balancing. Common to all CCP algorithms is the need to compute prefix sums of some weights [7]. In case of our application, these weights represent the workload of single elements in the unstructured mesh. Fortunately, the MPI standard already provides the functionality to compute these prefix sums in parallel. Although being quite powerful, these prefix sums can be implemented efficiently within MPI libraries as has been shown for example by Sanders and Träff in [9]. Though algorithms to find optimal solutions for the CCP are known, they are limited in their scalability. A very promising approximate solution to the CCP problem was suggested by Miguet and Pierson in [6]. It has the advantage, that splits between partitions can be determined completely local. The only global information required for this operation are the prefix sums and the total amount of work. These are attractive properties for a highly scalable parallel algorithm. Therefore, this heuristic is used in the implementations presented in this work. Various implementations of this SFC partitioning algorithm (SPartA) are described in the following section, afterwards the scalability of these implementations is investigated. Furthermore, we compare this method to ParMetis with respect to three criteria: i) the balancing quality, ii) the required memory, and iii) the running time.

A. Implementations

In the following subsection, the investigated implementations of SPartA are presented in detail and an example of the general algorithm is provided.

We consider an arbitrary mesh serialized into an one dimensional vector using a SFC. The vector has the length N which corresponds to the number of mesh cells. Weights are given as w_i for each element, where i corresponds to the global index of the element. These weights approximate the computational effort of each element, and can be derived from an on-line time measurement or a performance model. To achieve a balanced workload, the elements need to be moved between the partitions. It is the task of the balancing step to find which elements have to be moved to which process. In this paper, we use p to denote the total number processes, and assume that every such process can be uniquely identified by a process number called *rank* in the range $0 \leq \mathit{rank} < p$. For the proposed chains-on-chains approach on the serialized mesh, the partitions can be determined with the help of a prefix sum. In particular, we will use the exclusive prefix sum, which is defined as follows:

$$\mathit{prefix}(I) = \sum_{i=0}^{N-1} w_i \tag{1}$$

for $0 < I \leq N$ and with $\mathit{prefix}(0) = 0$. In order to calculate the distributed prefix sum over all processes, local prefix sums are computed, and the global offsets are adjusted afterwards using the MPI_Exscan() collective with MPI_SUM as reduction operation. After this step, each process has the global prefix sum for each of its local elements.

Under the assumption that the chosen weights correctly represent the workload of each element, the ideal work load per partition is given by $w_{opt} = \frac{w_{globsum}}{p}$, where $w_{globsum}$ is the global sum of all weights. Similar to the prefix sum, this global sum can be obtained by using the MPI_Allreduce() collective again with MPI_SUM as reduction operation. As the last process inferred the information on the overall sum already through the computed prefix sum, an alternative uses MPI_Bcast() with the last process acting as root. Both options result in a similar running time and memory complexity. Afterwards, the splitting positions between balanced partitions can be computed locally for all processes on each process. That is no further communication is required for the decision on which elements should be moved to which processes. Thus the complete information necessary for the partitioning algorithm can be obtained with only two collective operations in MPI. Both collectives can be implemented efficiently using an asymptotic running time and memory complexity of $\mathcal{O}(\log p)$ (cf. [8], [9]).

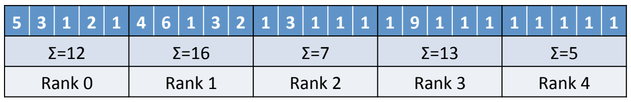
The splitting positions for the new balanced partitions in the local elements can be found efficiently using binary search in the ordered list of prefix values. Assuming homogeneous processors, ideal splitters are multiples of w_{opt}, i.e., $r \cdot w_{opt}$ for all integers r with $1 \leq r < p$. The closest splitting positions between the actual elements to these ideal splitters can be found by comparison with the global prefix sums computed for all elements.

If the $prefix(I)$ of a local element is larger than $r \cdot w_{opt}$ and smaller than $(r+1) \cdot w_{opt}$ then this element needs to be sent to the process with rank r. To obtain a partitioning closer to the optimal balancing, the final splitting position is decided by the minimal distance of the elements enclosing the optimal splitter: $min(|\mathit{prefix}(I) - r \cdot w_{opt}|, |\mathit{prefix}(I-1) - r \cdot w_{opt}|)$. Using this heuristic the load imbalance is limited by w_{max}/w_{opt}, where w_{max} is the maximum weight of a single element in the complete domain [6]. In general, the efficiency E of the distributed work load is limited by the slowest process, and thus cannot be better than:

$$E = \frac{w_{opt}}{max_{r=0}^{p-1}(w_{sum}(r))} \tag{2}$$

Where $w_{sum}(r)$ is the sum of all weights in partition r. This efficiency metric is used as a quality criterion for the resulting partitions.

Example: To illustrate the algorithm, a small domain with $N = 25$ elements distributed across $p = 5$ processors is used. The initial distribution corresponds to equally-sized parts and is shown in Figure 1. The individual weights attached to each of the elements result in load imbalance. Such an initial load imbalance might for example arise from the input data, without any a-priori information on the computational costs for the individual elements. As the overall work with the amount of 53 should be distributed equally over 5 processes, the optimal work load for each partition in this example is $w_{opt} = 10.6$. The resulting ideal splitters $10.6, 21.2, 32.8$, and 43.4 are depicted in Figure 2. The thick red lines show the final splitting positions that are closest to these optimal splitting positions. The work-



5	3	1	2	1	4	6	1	3	2	1	3	1	1	1	1	9	1	1	1	1	1	1	1	1
Σ=12					Σ=16					Σ=7					Σ=13					Σ=5				
Rank 0					Rank 1					Rank 2					Rank 3					Rank 4				

Figure 1. Decomposed domain with different workloads per process.

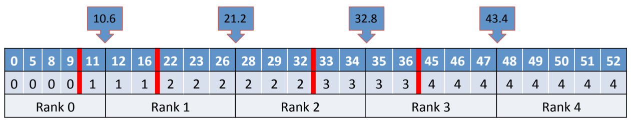
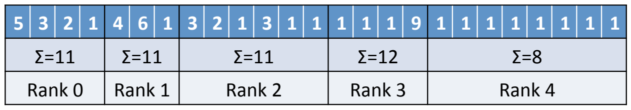


Figure 2. Prefix sums of the weights and optimum prefix values between the processes marked above. The resulting splitters are marked in red and the destination process of each element is shown in the second row.

balanced distribution after the re-partitioning is shown in Figure 3. The efficiency E (cf. Equation 2) in this example improved from 66% in the initial distribution to 88% for the final distribution. As such, it resembles the optimal partitioning for the chains-on-chains problem in this case, besides using a heuristic to find the partition splitters. The missing gap to 100% efficiency results from two facts: i) the unfavorable ratio of the maximum individual weight and the optimal partition work load of $9/10.6$, and ii) its unfortunate positioning. This is intentionally chosen to exhibit the major potential problem of this approach. However, such unlucky work load distributions are not expected in continuously load balanced flow simulations, where the optimal work load per process is normally much higher than the maximal weight of a single element. More realistic examples are analyzed with respect to their quality in Section IV.

Exchange of Elements: After the information for a better balanced partitioning is known, elements actually need to be relocated. This relocation, or exchange of elements, is done via communication between processes. Unfortunately, so far only the senders know which elements need to be sent to which processes. The receivers do not know that they will eventually receive elements. When using message passing, the receivers need to be informed prior to the actual exchange of elements. Three different options to realize this exchange are pointed out here. A first method uses a regular all-to-all collective operation to inform all processes about their communication partners before doing the actual exchange of the elements with an irregular all-to-all collective operation (e.g., using MPI_Alltoallv). This method is straightforward to implement and also the method-of-choice used after a ParMetis partitioning. Since both all-to-all variants are an essential part of many applications, they



5	3	2	1	4	6	1	3	2	1	3	1	1	1	1	1	9	1	1	1	1	1	1	1	1
Σ=11				Σ=11			Σ=11						Σ=12				Σ=8							
Rank 0				Rank 1			Rank 2						Rank 3				Rank 4							

Figure 3. Final distribution of the elements onto the processes.

have been optimized extensively for at least one of our target architectures [4]. As such, they can be expected to perform efficiently, especially in a dense case such as the initial partitioning (i.e., many exchanges between many processes). Alternatively, elements can be propagated only between neighboring processes in an iterative fashion. The elements are flagged with the destination process and forwarded in a virtual ring topology until they reach their destination. This approach can be benign when the re-partitioning modifies an existing distribution of elements only slightly. This could be expected if weights are changing slowly and re-partitionings are done frequently. In a beneficial case, only few exchanges, mainly between neighboring or at least "close" processes, are required. Unfortunately, worse cases can lead to $\mathcal{O}(p)$ forwarded messages, which becomes highly inefficient for larger number of processes. This scheme avoids the usage of $\mathcal{O}(p)$ memory needed for the input data of the all-to-all operation at the expense of a serialized communication pattern. It also reduces the required interconnect links to two for the direct neighbors in the linear list of partitions. Therefore, the iterative method offers a safe fallback if memory consumption is so important that it would otherwise inhibit the execution of the application. A third option fills the gap between these two extremes and involves a more sophisticated protocol involving a non-blocking barrier [2]. This approach for the dynamic sparse data exchange uses the fact that each sender has all the required information to start the communication. Therefore all processes begin to send their data to the appropriate processes, whereas the receiving parts just listen for messages from any source. However, this procedure results in a termination problem, as the receiving processes have no information, when to stop listening for new messages. With the help of a non-blocking barrier acting as a distributed marker, the authors solved this problem. This enables a very efficient implementation, where each process just sends its information to the appropriate target processes, and thus minimizes memory requirements. It would therefore combine the strengths of the previously described options at the expense of implementation complexity.

Although the third option is most promising, the necessary non-blocking barrier is only proposed for the upcoming MPI version 3 and therefore not yet a standardized operation to rely on. The first option is selected for the implementation in this work, as the required memory for the all-to-all communication is not yet a limiting factor. This choice also allows a fair comparison with ParMetis, where this all-to-all operation has to be done. Even when using all 294,912 available processes on the largest Blue Gene/P installation "Jugene" at the Jülich Supercomputing Center (JSC), the memory consumption is still well below 10 MB.

III. Scaling Analysis

Two generic cases of load balancing are investigated. The first one is an extremely imbalanced mesh, for which a strong scaling analysis is performed. This mesh builds a torus consisting of 30 million elements, where the small weights are scattered across many small elements at the inner ring, whereas to the outer side fewer larger elements are found with large computational weights attached to them. It has been intentionally designed to be especially unsuited for space-filling curve approaches, though in real application cases heavy loads are normally confined in smaller local volumes. Therefore, the worst case scenario for actual applications should be covered by this example. The second test case is initialized with uniformly distributed random numbers as weights for the elements. This is used in a weak scaling analysis with $10,000$ random weights per process, and represents a more realistic simulation with smaller load imbalances. The algorithm is investigated on two very similar Intel Nehalem based cluster systems: one located at the High Performance Computing Center in Stuttgart (HLRS) and the other at the JSC. To evaluate the behavior of the presented methods on larger number of processes, the Blue Gene/P system Jugene at JSC is used. The proposed partitioning schemes are compared to ParMetis. Two different graphs are fed into ParMetis, one with the full graph of the real mesh (ParMetis on Graph), and one with a pseudo-graph resembling the linear space-filling curve only with links between immediate neighbors on the curve (ParMetis on SFC).

A. Memory Usage

A major concern, especially on distributed systems with limited main memory per core, is the memory required for the algorithms used. This section presents an analysis of the virtual memory usage per process. The necessary information is gathered from the status information in the pseudo file system *proc*, provided by Linux. A sleek memory footprint for the proposed algorithm is considered a key feature for the usage on future architectures on which the shrinking memory per core will become an increasing bottleneck for most applications. As the memory consumption for the partitioning algorithm is independent from the memory consumption of the application, the results can be directly used to judge the impact on the memory-footprint for any application. Thus the amount of memory measured in this analysis can be understood as an overhead cost attached to the chosen partitioning strategy. Figure 4 shows the memory usage of ParMetis, when it needs to handle the full graph for the mesh with 30 million elements, compared to the case where it has to handle only the simplified linear graph. The third series in this graph shows the memory consumption per core for the presented SPartA algorithm. This measurement was done on the HLRS Nehalem cluster Laki with OpenMPI 1.4.3 and an executable, compiled with the Intel Fortran 11.1 compiler. As can be seen, the usage of ParMetis with the simplified graph needs less memory than the partitioning with the full graph. Figure 5 compares

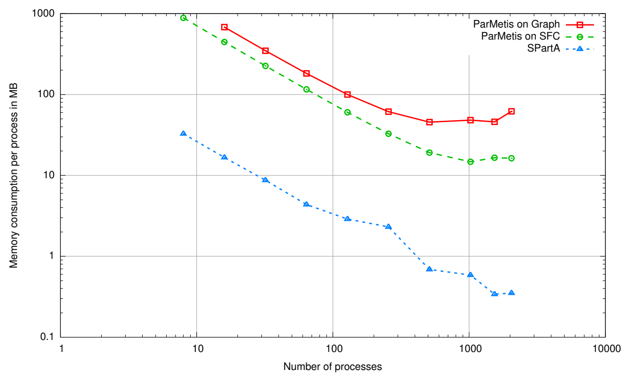


Figure 4. Memory consumption under strong scaling on Laki (OpenMPI) and 30 million elements.

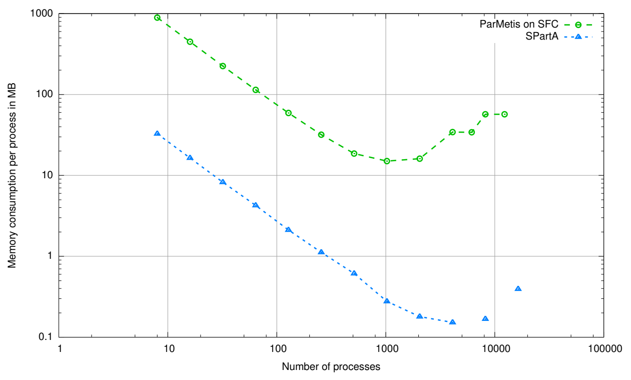


Figure 5. Memory consumption under strong scaling on Juropa.

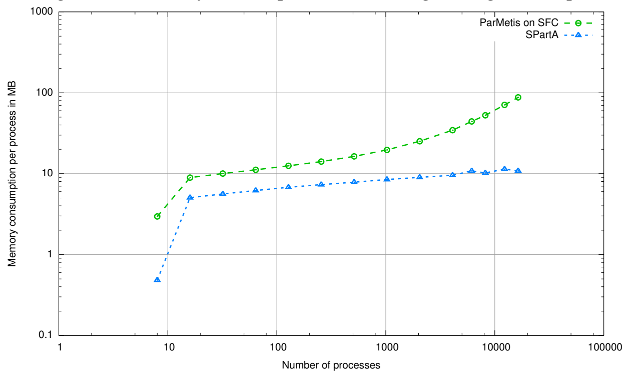


Figure 6. Memory consumption under weak scaling on Juropa.

the memory behavior of ParMetis to SPartA partitioning for the fixed mesh size of 30 million elements on the Nehalem cluster Juropa at JSC with ParaStation MPI 5.0. Unfortunately, the test with a complete graph of the mesh in ParMetis required a pre-allocation of all possible communication buffers beforehand on this machine, rendering any useful memory measurement for this case impossible. However, as already shown, the ParMetis partitioning with the simplified graph provides a lower bound for the full graph partitioning of ParMetis. It therefore can be used as an approximation for the comparison with SPartA. Figure 6 shows the memory consumption in the weak scaling experiment with $10,000$ elements per process. The strong scaling behavior shows a linear decline in memory consumption for both ParMetis and SPartA for smaller number of processes but a significant memory overhead is needed by ParMetis (up to a factor of 25). SPartA scales very well up to 4,096 cores before a memory increment gets visible. These memory requirements arise from the necessary buffers for all processes in the subsequent all-to-all operation. Contrary, ParMetis scales only up to 1024 cores in memory, and then starts to demand significantly more memory. It can also be seen that the memory consumption of ParMetis depends on the next higher power of two in terms of the number of processes and therefore 6,144 cores already need as much memory as 8,192. The same behavior is observed for 12,288 cores which requires the same amount of memory as 16,384. The weak scaling of SPartA behaves very well with only a small slope and stagnation at around 10 MB per core for large numbers of cores. In contrast, a more than linear growth of memory consumption per core can be observed with ParMetis. Both ParMetis and SPartA show a jump in memory consumption from 8 to 16 cores where communication between computing nodes has to be done across the network. Both scaling results indicate excellent scaling of SPartA even on highly parallel systems with a small amount of memory per process.

B. Execution Time

Frequently repeated load balancing for highly dynamic simulations demand a short execution time of the balancing algorithm, especially on large numbers of cores. Figure 7 shows the execution times on Jugene for ParMetis both using the full graph and the simplified one, and SPartA for the fixed mesh of 30 million elements. Due to the limitation of 500 MB of main memory per core on this Blue Gene/P system, ParMetis simply fails at a certain number of cores as it requires too much memory. With the full graph it does not work with more than $16,384$ processes. Reducing the problem to the simplified graph, ParMetis succeeds up to $65,536$ cores. It should be noted, that these experiments are done without any real application data, which would reduce the memory available to the balancing algorithm even further. These memory issues are completely avoided by the simpler algorithm based on the space-filling curve. It can be seen that the execution time of SPartA is dominated by a behavior according to $\mathcal{O}(p)$ starting from $16,384$ processes. This is due to the allocation and initialization of arrays of the size p for the subsequent all-to-all operations. The weak scaling using $10,000$ elements on each process is shown in Figure 8, and confirms the already described trends.

Overall, it can be observed in these measurements on

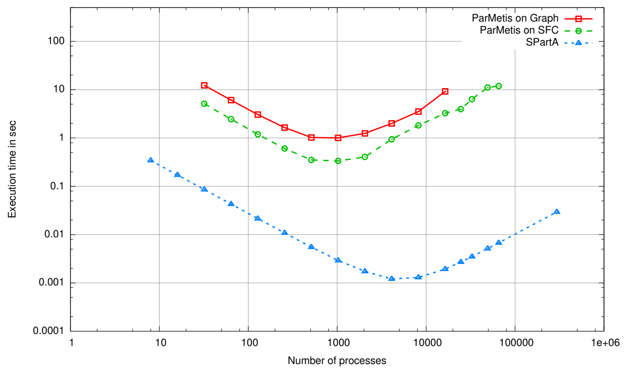


Figure 7. Execution time under strong scaling on Jugene.

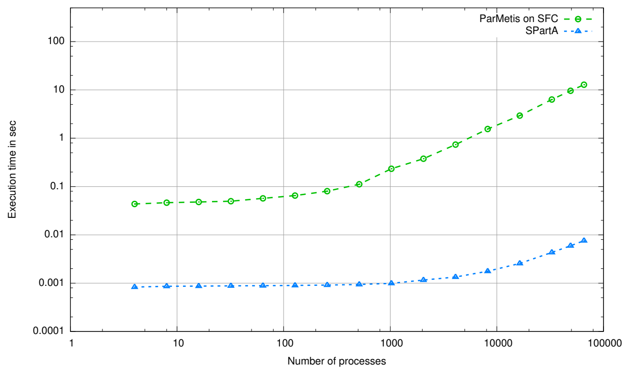


Figure 8. Execution time under weak scaling on Jugene.

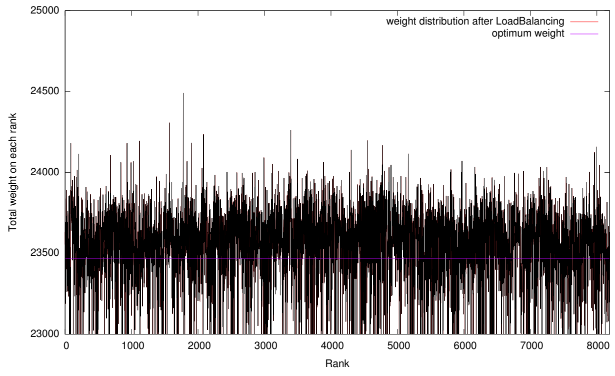


Figure 9. Workload distribution on 8,192 processes after ParMetis.

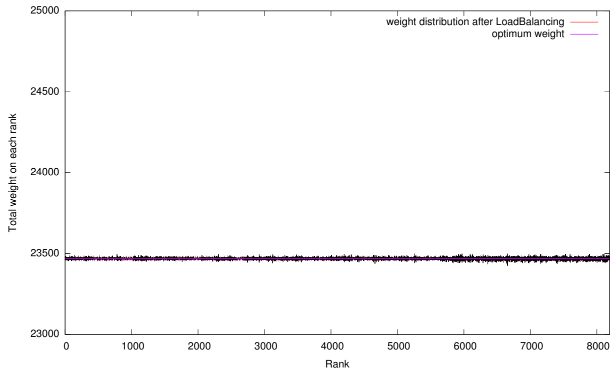


Figure 10. Workload distribution on 8,192 processes after SPartA.

the different machines, that the simpler SPartA method compared to the more complicated graph-based alternative scales much better with respect to memory and time. For the extreme scaling beyond 10 thousands of cores on supercomputing machines in the foreseeable future, this is an important property. The memory restriction is a hard constraint that decides if an simulation can be done at all or not. The time consumption of the balancing influences the overall application efficiency, especially when it needs to be executed repeatedly such as in increasingly important dynamic numerical simulations.

IV. Load-Balancing Quality

Figure 9 and 10 show the resulting sum of workloads $w_{sum}(r)$ for each process r after balancing the mesh with 30 million elements on $8,192$ processes using ParMetis and SPartA, respectively. As this comparison focuses on quality and neither on memory consumption nor on running time, ParMetis was given the full graph information. Nevertheless, the remaining load imbalance after re-partitioning induces an efficiency E of only 95.8% with ParMetis, compared to 99.9% for SPartA. As can be seen in the two figures, the resulting workload distribution is much more regular when SPartA is used instead of ParMetis. That is true for overloading as well as underloading, while ParMetis has some processes, which are significantly underloaded. This is due to the fact, that the important factor for the running time is the bottleneck, that is the relation of largest load share to the average. However, it is important to note, that with such a strategy the achieved balancing is worse then it could be. Also this difference in the achievable overall efficiency has to be recovered in the graph-based approach by an accordingly reduced communication effort. However, with a highly local application as the fluid dynamic solver considered in the next section, the computational load is usually higher than the communication. Thus, potential running time advantages by the graph-based approach are diminished, and can be neglected in most scenarios.

V. Deployment in Application

SPartA and ParMetis are deployed within a compressible Navier-Stokes solver to balance the load dynamically during the simulation of a supersonic turbulent free stream. Both methods are accessible from the same interface within the application. The simulated problem is highly volatile and propagates shocks through the computational domain, resulting in drastic changes of the computational effort between time steps. We will analyze the dynamic behavior of this specific application and apply the two different partitioning algorithms to it. As already mentioned, the application is capable of adapting the time-step in each cell such, that it is optimal in the sense of the stability criterium for explicit time integration. This results in individual time-steps for each element rather than a single global time step for all. These time-steps are strongly dependent on the size and form of the cell as well as the physical phenomena within the cell. Besides this factor, there are several other influences on the computational cost of a single cell, like the adaptable order of the polynomial representation, usage of geometrical conversions for curved walls and special treatments of shocks within an element. To cover all the various aspects accurately, a time measurement is performed for the individual elements. These timings are then used as the weights in the balancing algorithm.

MPI persistent communication is used during the simulation to exchange data between adjacent elements on different processes. The mesh handling is based on GEUM [3] which employs a morton-curve to linearize the mesh and provides the solver with an initial mesh distribution. This initial distribution divides the number of elements equally on each process, regardless of their respective loads.

The investigated flow simulation is a highly turbulent super-sonic free-stream configuration using a hybrid mesh with 4 million elements. This mesh has extreme differences in the spatial resolution, and the ratio of the largest to the smallest volume is around 100. Using just a second order representation of the solution in the entire domain the simulation will contain 16 million degrees of freedom.

As already pointed out, the best possible estimation of the workload is important to obtain a balanced computation. However, this load depends on many different factors, which can also change at run time. Therefore, additional instrumentations were introduced into the code to measure the amount of time required to actually compute each element. These measurements also enable an evaluation of the current partitioning efficiency during the simulation. To avoid potentially needless re-partitioning at every step, this efficiency indicator can be used to decide if a re-partitioning is worthwhile or not. The measurement is done locally for several iterations of each element, and does not require any communication. However, there are some user-defined intervals, at which all processes have to synchronize. At these points output can be written to disk and additional administrative tasks might be executed. These points in time are natural choices to determine the current load distribution across the complete domain, as well as to perform the actual re-balancing if needed. The decision to perform a new partitioning of the mesh is based on the bottleneck factor, limiting the maximum parallel efficiency. A re-balancing is only done if the ratio of maximal load to average load falls below a user-defined threshold. While ParMetis gets the full graph information to find a better partitioning, SPartA only uses the measured weights. A fair comparison of the following simulations is achieved by using identical input parameters and the same test case on the Nehalem cluster at JSC Jülich using 1024 cores. Non-deterministic behavior of time-based re-partioning and scheduler-dependent mapping of the processes onto the actual network layout leads to different distributions of elements onto the processes. Therefore, all comparisons show some unavoidable inaccuracies. However, this error is negligible compared to the differences between the two partitioning approaches.

A. Scaling of the Memory

The application itself is nicely scaling in memory, as the required memory is directly attached to the elements in the mesh. Thus, a reduced partition size leads to a reduced memory footprint. Therefore, the main effects that can be observed are those that are related to the balancing mechanism. However, the application requires some more memory for the data to be transferred for the moved elements. In total, the quality of the already measured results for the balancing method still holds for the overall application. For a simulation with 1024 processes, we observed a peak memory consumption of 1.79 GB with SPartA, while ParMetis leads to a peak of 2.85 GB.

B. Communication overhead

Given the simple nature of the proposed partitioning algorithm, the communication surface of each computational domain is not actively optimized. It is rather based on the inherent locality given by the SFC. Therefore, an additional overhead in the communication time during the simulation is to be expected. A comparison of the measured overall times needed to simulate a given Δt between two load balancing steps shows however, that the overhead has little impact on the consumed running time, when compared to the graph-based partitioning from ParMetis. For the first five intervals between load balancings, the ParMetis partitioning yields a total running time of 2664 seconds without the balancing itself. In contrast, the much simpler SPartA method yields 2700 seconds. Thus, the difference in running time between the two approaches is less than 2%.

C. Dynamic behavior of the application

Due to the dynamic nature of the flow phenomena which occurs in this particular simulation, dynamic load balancing is a key feature to ensure a high parallel efficiency during the complete simulation. Figure 11 illustrates the achieved parallel efficiency over the simulation time induced by the different load balancing methods. The same simulation is done three times: once without any load balancing, and two times with the different partitioning approaches. Partitioning is applied dynamically when the efficiency falls below a threshold of 80%. In the case of applied load balancing, an a-priori estimation of the workload per element is done, and

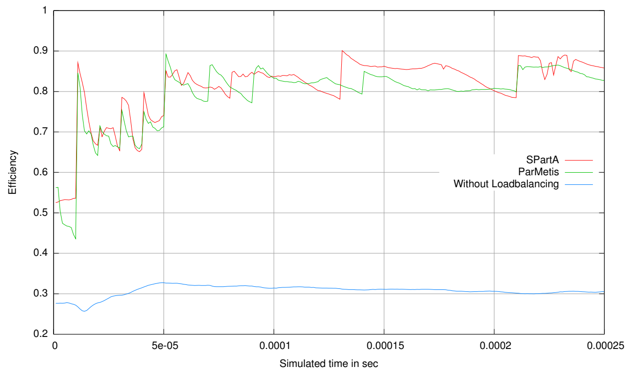


Figure 11. Efficiency of the simulation over the simulated time.

the mesh is distributed accordingly before the simulation starts. Figure 11 does not depict the theoretical efficiency E directly after the redistribution of the elements as the weights w_i on which the distribution is based on may no longer be valid for the new distribution. The change of characteristics of elements, e.g. communication cells can become inner cells or vice versa, re-introduces new load imbalances. Therefore, the resulting efficiency E for this application scenario is lower than in the previous benchmark scenarios. Without any load balancing, the initial equal distribution of elements to all processes is used during the complete simulation. As can be seen, the efficiency of the simulation is quite similar for both partitioning methods, while only one third of the optimum is achieved without any balancing.

VI. Conclusion

In this paper, we studied different re-partitioning methods to balance dynamically changing workload on static but unstructured meshes. We presented a simple but both efficient and effective re-partitioning approach based on space-filling curves and compared it to the popular graph-based method used in ParMetis. An adaptable flow solver was chosen as a benchmark.

Memory consumption was identified as the most critical advantage of our method over ParMetis. While the graph-based approach in ParMetis was not capable to run our test case on more than $65,536$ processors, even with a drastically reduced graph, our own approach was shown to scale well up to $294,912$ processors. Potential for further memory reductions was outlined, which would satisfy the needs of future large-scale systems with lower memory-per-core ratio. Disadvantages in comparison to ParMetis in terms of suboptimal communication surface and locality during the actual simulation were found to be rather small in the order of 1-2%. On the other hand, the curve-based partitioning operation itself is orders of magnitude faster, which alleviates the above penalty to some degree. We expect to benefit from this compensatory effect especially in highly dynamic simulations where re-partitioning occurs frequently. Finally, our implementation based on MPI collective operations is supposed to deliver portable performance across a broad range of architectures.

Acknowledgment

The authors would like to thank the Gauss Centre for Supercomputing (GCS) which provided us with the necessary resources on different HPC systems. This work has been funded by the Federal Ministry for Education and Research (BMBF) in the framework of the HPC software initiative in the project STEDG.

References

[1] S. Aluru and F. Sevilgen. Parallel domain decomposition and load balancing using space-filling curves. In *High-Performance Computing, 1997. Proceedings. Fourth International Conference on*, pages 230–235, 1997.

[2] T. Hoefler, C. Siebert, and A. Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 159–168, New York, NY, USA, 2010. ACM.

[3] H. Klimach and S. Roller. Distributed coupling for multiscale simulations. In P. Ivanyi and B. Topping, editors, *Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*. Civil-Comp Ltd., 2011.

[4] S. Kumar, Y. Sabharwal, R. Garg, and P. Heidelberger. Optimization of All-to-All Communication on the Blue Gene/L Supercomputer. In *Proc. of the 37th International Conference on Parallel Processing*, pages 320–329, Washington, DC, USA, 2008. IEEE Computer Society.

[5] F. Lörcher, G. Gassner, and C. Munz. A discontinuous galerkin scheme based on a Space–Time expansion. I. inviscid compressible flow in one space dimension. *Journal of Scientific Computing*, 32(2):175–199, 2007.

[6] S. Miguet and J.-M. Pierson. Heuristics for 1d rectilinear partitioning as a low cost and high quality answer to dynamic load balancing. In B. Hertzberger and P. Sloot, editors, *High-Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 550–564. Springer Berlin / Heidelberg, 1997.

[7] A. Pinar and C. Aykanat. Fast optimal load balancing algorithms for 1d partitioning. *Journal of Parallel and Distributed Computing*, 64(8):974 – 996, 2004.

[8] R. Rabenseifner. Optimization of Collective Reduction Operations. In M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *International Conference on Computational Science*, volume 3036 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2004.

[9] P. Sanders and J. Träff. Parallel Prefix (Scan) Algorithms for MPI. In B. Mohr, J. Träff, J. Worringen, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science*, pages 49–57. Springer Berlin / Heidelberg, 2006.

[10] K. Schloegel, G. Karypis, and V. Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[11] S. Tirthapura, S. Seal, and S. Aluru. A formal analysis of space filling curves for parallel domain decomposition. In *International Conference on Parallel Processing, 2006. ICPP 2006*, pages 505–512. IEEE, Aug. 2006.