Running Jet Engine Models on Thousands of Processors with LS-DYNA Implicit

Cleve Ashcraft[2], Jef Dawson[1], Roger Grimes[2], Erman Guleryuz[3], Seid Koric[3], Robert Lucas[2], James Ong[4], Francois-Henry Rouet[2], Todd Simons[4], and Ting-Ting Zhu[1]

Cray[1],
Livermore Software Technology Corporation[2],
National Center for Supercomputing Applications[3] at the University of Illinois
Rolls-Royce[4]

1 Introduction

Only time and resource constraints limit the size and complexity of the implicit analyses that LS-DYNA users would like to perform. Rolls-Royce is an example thereof, challenging its suppliers of computers and mechanical computer aided engineering (MCAE) software to run ever larger models, with more physics, in shorter periods of time. This will allow CAE to have a greater impact on the design cycle for new engines, and is a step towards the long-term vision of digital twins. Towards this end, Rolls-Royce created a family of representative engine models, with as many as 66 million finite elements. Figure 1 depicts a cross-section of the representative engine model.

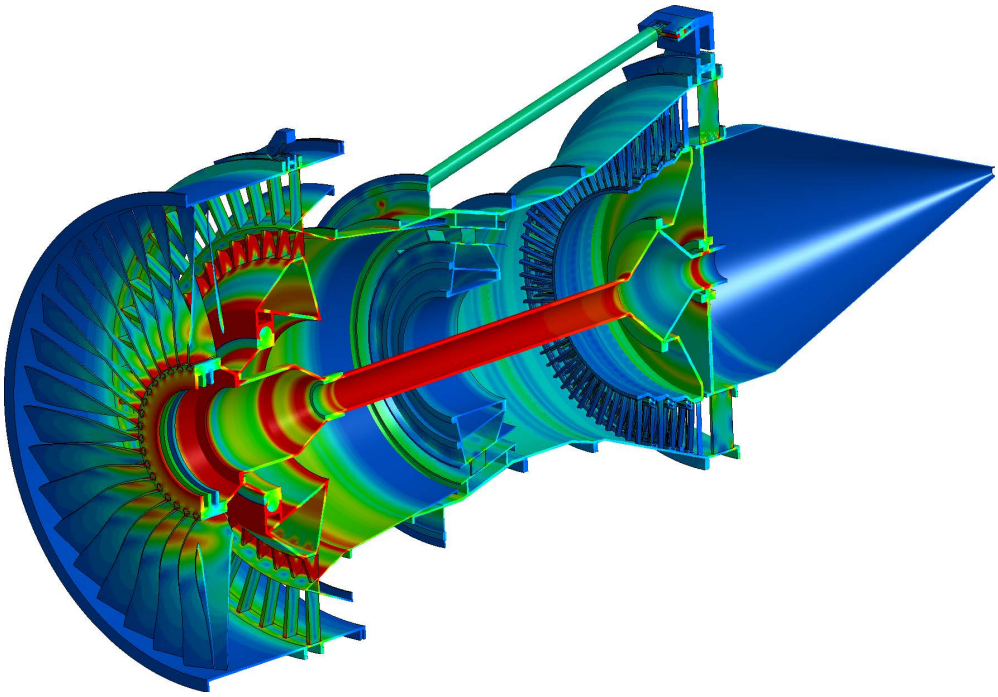

Fig.1: Cross-section of the Rolls-Royce Representative Engine Model

A steady state load analysis was first performed explicitly, and on the largest of the representative engine models (LREM), took over seven weeks. Switching to implicit finite element analysis (FEA) brought this down to one week, still too long for use as a pratical engineering tool. Therefore, Rolls-Royce formed a consortium with Cray, the University of Illinois' National Center for Supercomputing Applications (NCSA), and Livermore Software Technology Corporation (LSTC) to examine the performance of LS-DYNA running the representative engine models on increasing numbers of processors of the Blue Waters Cray XE/XK supercomputer.

We recall that the computational complexity of implicit FEA is dominated by the analysis and solution of a sparse linear system of equations. The main steps of this analysis are the following:

1. Assemble the stiffness matrix *K* and the right-hand side *f* coming from the forces. Assemble the constraint matrix *C* and the corresponding constraint vector *g*. The goal is to minimize $\left\| u^T K u - u^T f \right\|$ subject to $Cu = g$, where *u* is the displacement vector. This is equivalent to the *augmented system* (or *KK^T system*):

$$\begin{bmatrix} K & C^T \\ C & 0 \end{bmatrix} \begin{Bmatrix} u \\ v \end{Bmatrix} = \begin{Bmatrix} f \\ g \end{Bmatrix}$$

2. The constraint matrix *C* is analyzed and the set of degrees of freedom is partitioned into two subsets: the *dependent* and *independent* degrees of freedom. We eliminate the dependent degrees of freedom and form a Schur complement called the *reduced system* that we then pass to the sparse linear solver. This technique is called *direct elimination of constraints*, or the *null-space method*.

3. The reduced system is permuted with a *fill-reducing ordering* tool.

4. A *symbolic factorization* of the reduced matrix is performed.

5. The reduced matrix is factored numerically $\left[LDL^T \right]$. LS-DYNA normally uses its own version of the multifrontal method [1], and a similar multi-frontal solver has been shown to scale to tens of thousands of cores [2].

6. The reduced system is solved with two triangular solves.

7. The reduced solution is expanded from the set of independent degrees of freedom to the whole set of unknowns.

For nonlinear problems with contact solved in this work, within each quasi-static time increment, a system of nonlinear equations is linearized and solved with a Newton-Raphson (NR) iteration scheme, in this case BFGS [3]. For every such NR iteration, a sparse linear system is solved that requires repeating most of the above steps.

Historically, step 5 (numerical factorization) has been the most time-consuming step of implicit FEA. A lot of attention has been devoted over the past decades to make this step perform well on large numbers of cores. Unfortunately, steps 2, 3, and 4 became major performance and storage bottlenecks for large processor counts. The following sections describe these steps and the improvements that were recently integrated to LS-DYNA. This is followed with a discussion of the impact on overall performance when running the engine models on many thousands of cores, and our plans for further improvements.

2 Constraint analysis

The analysis of the constraint matrix, performed by our constraint processing package LCPACK, is currently a sequential bottleneck. The constraint matrix is initially distributed but is then gathered to processor 0 to perform the analysis. Processor 0 identifies the set of independent degrees of freedom, inverts a subset of the constraint matrix, and scatters the results to all the other processors. By analyzing traces of experiments performed on NCSA's Blue Waters and Oak Ridge National Laboratory's Titan, we identified parts of the code where run time would *increase* linearly with the number of processors, due to a suboptimal communication pattern in the gathers and scatters of the constraint information. This was not noticeable for low processor counts, but became a bottleneck for 512 MPI ranks, and more. This is illustrated in Figure 2, which depicts the memory available during the

course of the execution of Rolls-Royce's 35 million element small representative engine model (SREM). The run time for 2048 MPI ranks, Figure 2(d), is dominated by the constraint processing, which appears as long plateaus in the available storage plot.

We rewrote some of the constraint processing code and obtained speed-ups up to 37 between the old and the new version. As a result, constraint analysis no longer dominates the total run time for the jet engine models. Nevertheless, constraint analysis is still a sequential bottleneck, and designing a completely parallel constraint analysis code is currently work in progress.

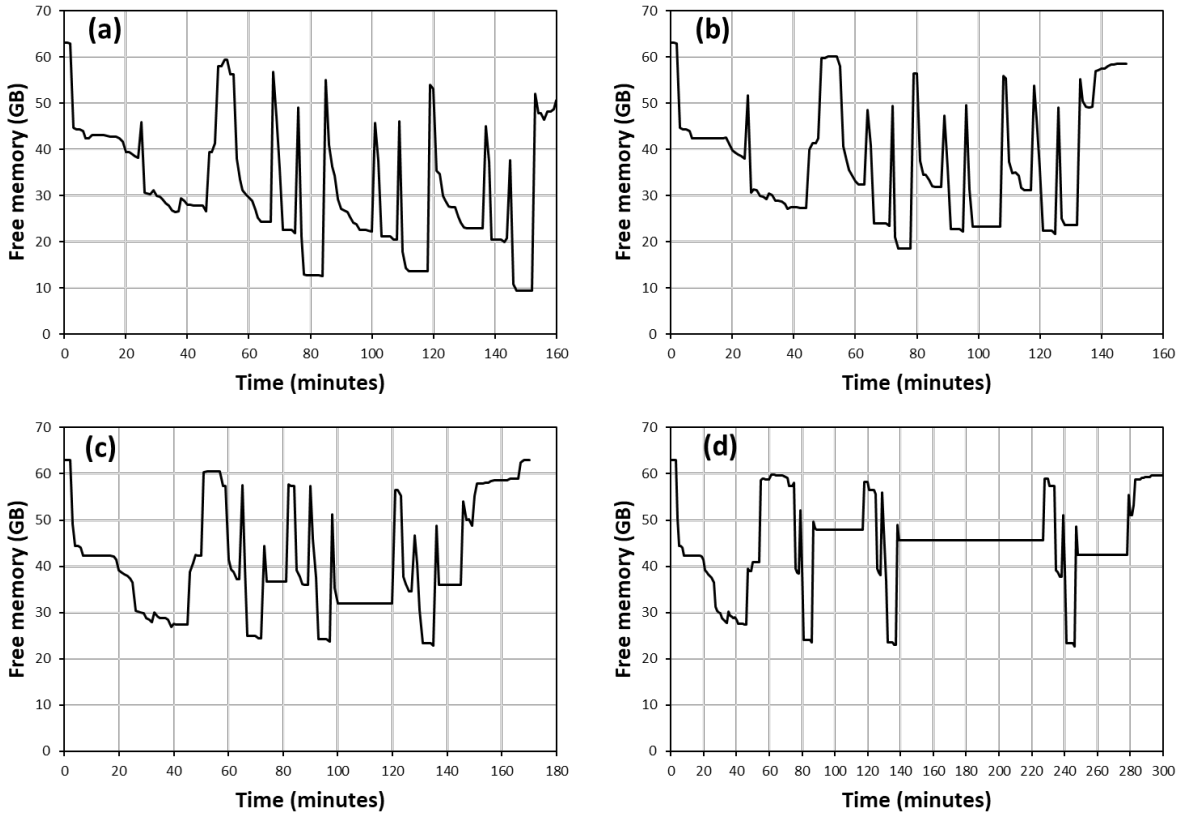

Fig.2: Available memory versus time when running the SREM on Blue Waters, with MPI ranks of 256 (a), 512 (b), 1024 (c), and 2048 (d).

3 Reordering

After factoring a sparse matrix, the factors of the matrix typically have lots more non-zeros than the original matrix. This *fill-in* is defined as the ratio of non-zero entries in the factored matrix divided by the non-zero entries in the original stiffness matrix. The amount of fill-in depends on the *order* that one follows to eliminate the unknowns during the factorization. Finding an order that minimizes fill-in is an NP-complete problem [4]. Nested dissection [5], the most popular fill-reducing reordering heuristic, is based on recursively partitioning a graph that represents the non-zero structure of the sparse matrix. Every row and column of the matrix is a vertex in the graph, and every off-diagonal nonzero is an edge.

The most well-known graph partitioning tools used for reordering are METIS [6] and Scotch [7] and their distributed-memory counterparts ParMETIS [8] and PT-Scotch [9]. METIS has been used in LS-Dyna for many years, but concerns about scalability led to the creation of our own tool, LS-GPart [10]. LS-GPart is a parallel graph partitioning tool like ParMETIS and PT-Scotch, but uses different algorithms to perform the partitioning. LS-GPart was made available as of R11 and recent experiments on Blue Waters and Titan have allowed us to improve its performance significantly.

The results in the figure below show that LS-GPart is typically slower than ParMETIS and PT-Scotch for low a processor count, but catches up for large processor counts (1,024 and more). The *quality* of the ordering, i.e., the amount of fill-in, is similar to that of ParMETIS, and better than that of PT-Scotch.

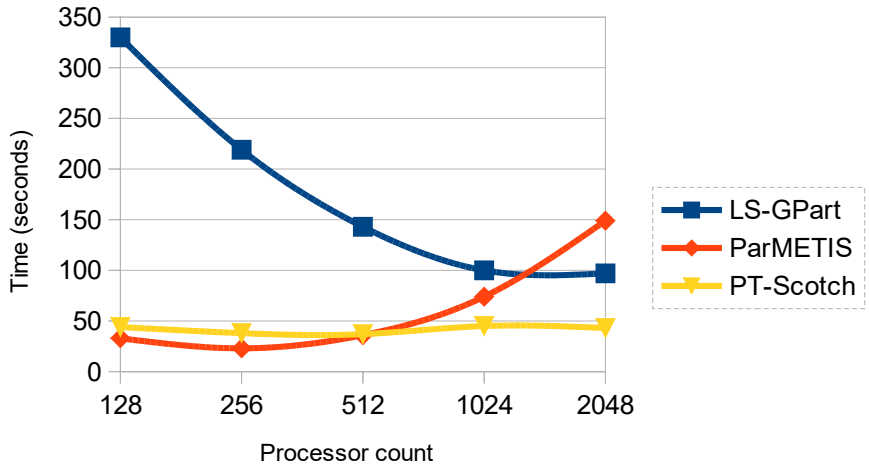

Fig.3: Reordering time on Titan for the small representative engine model using LS-GPart, ParMETIS, and PT-Scotch.

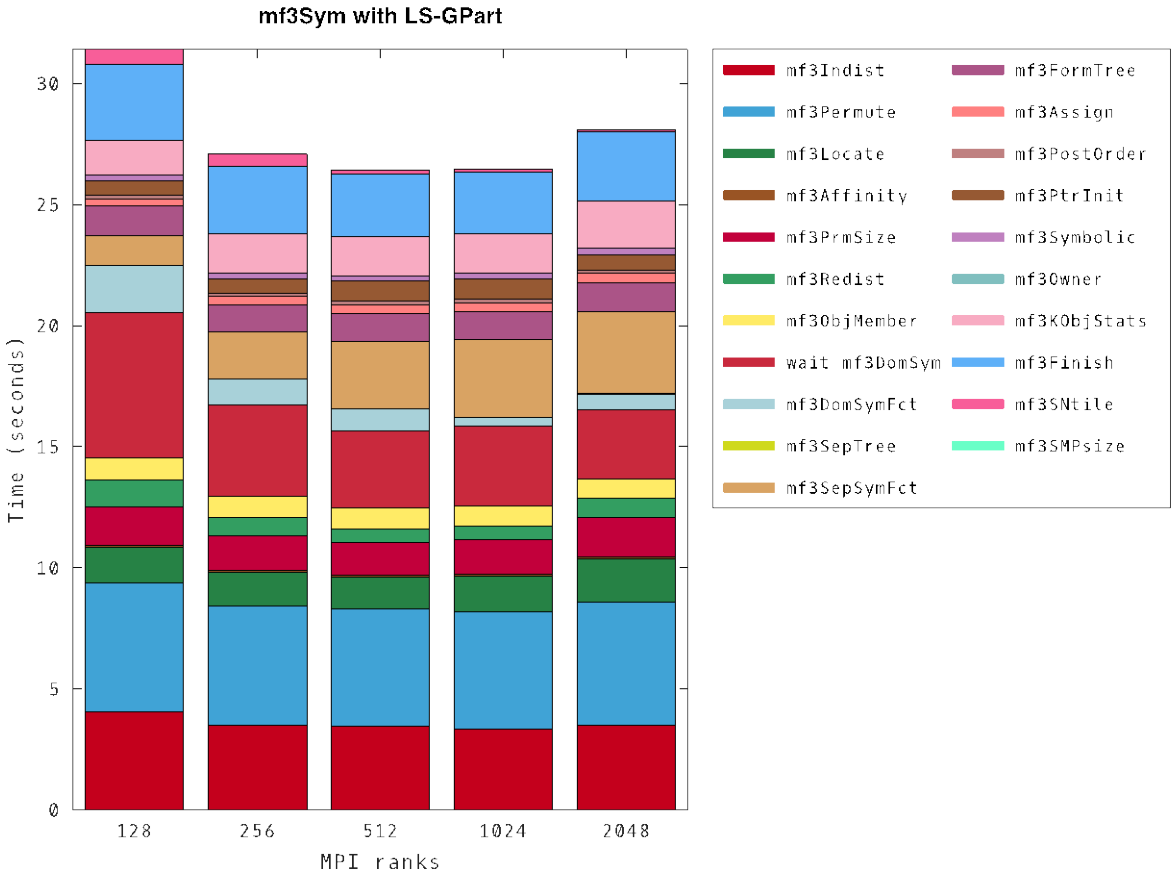

Fig.4: Breakdown of time spent in symbolic factorization of the SREM.

4 Symbolic factorization

Once reordering has been completed, it is desirable to predetermine the non-zero structure of a factored sparse matrix, before attempting to actually factor it. This is called symbolic factorization, and the output is an elimination tree that guides the numerical factorization, as well as the non-zero structure of the resulting matrix factors, assuming no numerical pivoting. Whereas reordering is NP-complete, symbolic factorization is relatively straightforward. Historically, reordering took longer than symbolic factorization on one processor, and hence we were not motivated to parallelize this

computation. With the emergence of LS-Gpart (that replaced the serial code METIS) sequential symbolic factorization became a bottleneck, both in terms of the time and storage consumed.

To address this problem, we have developed a new parallel symbolic factorization. A nested dissection reordering, like LS-GPart, divides the graph of the matrix into separators, which partition the graph, and a remaining set of disjoint domains. We perform symbolic factorization of the domains independently, on separate processors. We then climb the tree of separators, communicating any new adjacencies found in their children, and then complete their own symbolic factorization.

Figure 4 above shows the time being spent in the different phases of the new parallel symbolic factorization code, while processing the SREM as the number of MPI ranks grows from 128 to 2048. The first three phases parse the input, recovering the indistinguishable vertices, the ordering, and the location (domains and separators) of the vertices of the graph. The next four phases permute and redistribute the graph, so that its ready for symbolic factorization. The next three phases are the parallel symbolic factorization itself, first any domains assigned to this processor, then a brief wait due to load imbalance as the size and complexity of the domains varies, and finally the separators. Note, the time spent in the separators increases with processors, as the nested dissection ordering creates additional levels of separators. The remaining time is spent preparing the data structures that will drive the subsequent numeric factorization and triangular solves. As with constraint analysis, we have addressed the sequential bottleneck that symbolic factorization used to present. In the future, we will have to revisit the interface to this function, so that parsing its input doesn't take almost half of its runtime, and it continues to scale with processor count.

5 Overall performance

Figure 5 shows the improvement of LS-Dyna's scalability during the course of this project. The blue curve (square markers) was the first test campaign on Blue Waters (mid 2017). It used a hybrid parallel version of LS-DYNA r119196. Every run uses 8 threads per MPI rank. The figure shows that time would turn up past 512 MPI ranks; this was mostly due to the constraint processing issue mentioned in Section 2.

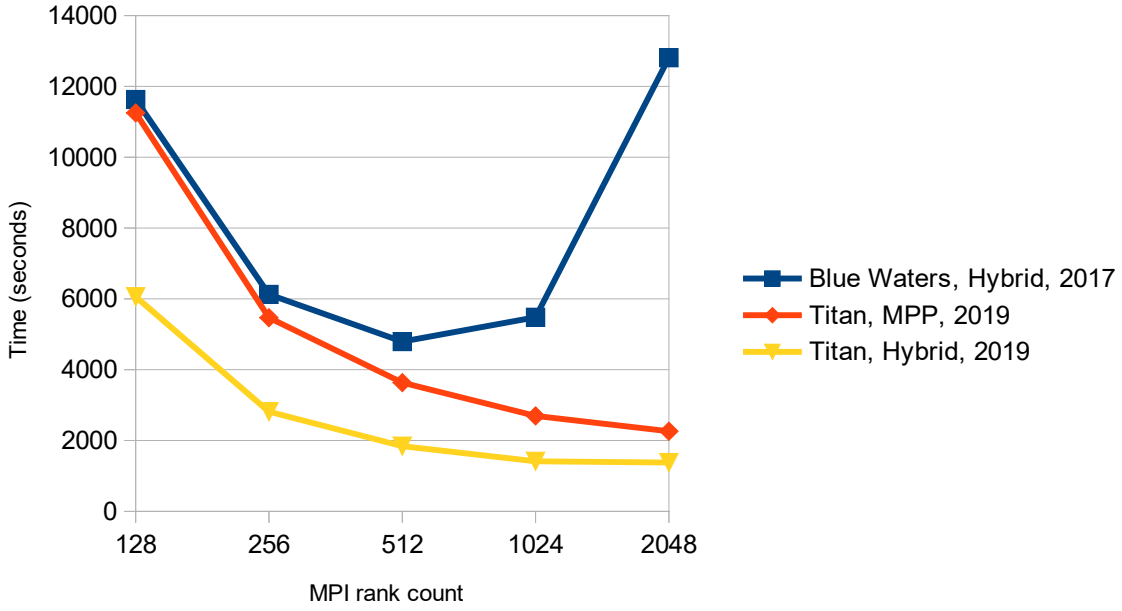

Fig. 5: Strong scaling experiment, 3 load steps of the SREM

The orange (diamond markers) and yellow (triangle markers) curves show recent results (early 2019) obtained on the Titan system at Oak Ridge National Laboratory. Blue Waters and Titan are very similar machines and use exactly the same processors (AMD Opteron 6274), therefore the comparison is fair. The orange curve is pure MPI parallelism, while the yellow curve uses hybrid parallelism with 8 threads per MPI rank. Both use LS-DYNA r134158, which incorporates all the improvements described in this paper: improved constraint processing, ordering with LS-GPart instead

of METIS, and parallel symbolic factorization. Both curves show that LS-DYNA now scales to 2,048 MPI ranks (16,384 cores in the Hybrid case).

The last observation from the figure is that for this problem, Hybrid parallelism (yellow curve) with 8 threads gives an acceleration of about 2x over pure MPI parallelism (orange curve). This is due to the fact that only part of the code is multithreaded: element calculation routines, numerical factorization and solution are multithreaded, but input processing, constraint processing (LCPACK), ordering (LS-GPart) and symbolic factorization only use pure MPI parallelism. Adding multithreading to these parts of the code is also work in progress.

6 Summary

Rolls-Royce challenged Cray, LSTC, and NCSA to demonstrate that implicit FEA of large-scale models could be performed in a timely manner using large-scale computing systems. The engine models created for this study are, to our knowledge, the largest implicit models ever run with LS-DYNA. Similarly, the Blue Waters and Titan computers are the largest systems LS-DYNA has ever run on. This increase in the scale of both the models and the computing systems has led to the need to address parallel processing challeneges that had previously received relatively little attention, in LS-DYNA, or elsewhere in the literature. Prominant amongst these were constraint processing, reordering, and symbolic factorization. LS-DYNA has now been demonstrated to perform implicit analysis using thousands of MPI ranks, each augmented with eight threads.

Of course, there is much more work to be done, so that the number of processors that LS-DYNA can effectively be use continues to increase. On thousands of processors, parsing the input, decomposing it, and initializing all of the processors is increasingly time consuming, as is output of the results. The interfaces between the steps of FEA described above need to be revisited, lest they become Amdahl fractions. And of course, the numerical factorizaiton and triangular solves also would surely benefit from additonal performance optimization. Therefore, this remains on on-going effort.

7 Acknowledgements

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. The authors would like to thank the NCSA Industry Program and the Blue Waters sustained-petascale computing project at the National Center for Supercomputing Applications (NCSA). Blue Waters is supported by the National Science Foundation (award numbers OCI 07-25070 and ACI-1238993) and the state of Illinois.

8 Literature

[1] Duff, I. S., Reid, J. K.: The multifrontal solution of indefinite sparse symmetric linear equations. ACM Transactions on Mathematical Software, 9(3), pp. 302-325 (1983).
[2] Koric, S. and A. Gupta: Sparse Matrix Factorization in the Implicit Finite Element Method on Petascale Architecture, Computer Methods in Applied Mechanics and Eng., v.32, 281-292, 2016.
[3] http://ftp.lstc.com/anonymous/outgoing/jday/manuals/DRAFT_Theory.pdf
[4] Yannakakis, N: Computing the minimum fill-in is NP-complete. SIAM Journal on Algebraic Discrete Methods, 2(1), pp. 77-79 (1981).
[5] George, A.: Nested dissection of a regular finite element mesh. SIAM Journal on Numerical Analysis, 10(2), pp. 345-363 (1973).
[6] Karypis, G., Kumar, V.: A fast and high-quality multilevel scheme for partitioning irregular graphs. SIAM Journal on scientific Computing pp. 359-392 (1998), DOI: 10.1137/S1064827595287997
[7] Pellegrini, F., Roman, J.: Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In: International Conference on High Performance Computing and Networking. pp. 493-498. Springer (1996), DOI: 10.1007/3-540-61142-8588
[8] Karypis, G., Kumar, V.: A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. Journal of Parallel and Distributed Computing pp. 71-95 (1998), DOI: 10.1006/jpdc.1997.1403
[9] Chevalier, C., Pellegrini, F.: PT-scotch: A tool for efficient parallel graph ordering. Parallel Computing pp. 318-331 (2008), DOI: 10.1016/j.parco.2007.12.001
[10] Ashcraft, C., Rouet, F-H.: A global, distributed ordering library. Presentation at the SIAM Workshop on Combinatorial Scientific Computing CSC16 (2016)