Computers & Fluids 80 (2013) 372–380

Contents lists available at SciVerse ScienceDirect

Computers & Fluids

journal homepage: www.elsevier.com/locate/compfluid

Rank reordering for MPI communication optimization

B. Brandfass [a,*], T. Alrutz [b], T. Gerhold [a]

[a] *DLR German Aerospace Center, Institute of Aerodynamics and Flow Technology, $C^2A^2S^2E$ Center for Computer Applications in AeroSpace Science and Engineering, Bunsenstr. 10, 37073 Göttingen, Germany*
[b] *T-Systems Solutions for Research GmbH, Scientific Solutions, Bunsenstr. 10, 37073 Göttingen, Germany*

ARTICLE INFO

Article history:
Received 15 September 2011
Received in revised form 13 January 2012
Accepted 22 January 2012
Available online 31 January 2012

Keywords:
Parallelization
Domain decomposition
Computational Fluid Dynamics
MPI rank reordering

ABSTRACT

In this paper we describe a procedure for optimizing the MPI communication of an unstructured CFD code in a parallel multi-core environment. By reordering the MPI ranks, a mapping of MPI processes to CPU cores is established, such that the main communication takes place within the compute nodes. The motivation of this approach is based on the observation that the communication between CPU cores on the same compute node is usually much faster than the communication between CPU cores on different nodes.

The generic nature of our approach provides an out-of-the-box optimization tool, which can be easily used with other CFD codes due to the external MPI rank reordering procedure. The optimization tool was successfully tested with the DLR TAU code and the results of the optimization are demonstrated by benchmark computations for different geometries of aircraft configurations.

1. Introduction

Computational Fluid Dynamics (CFD) is a key technology for the aircraft industry. Growing use of CFD in the aircraft design and the aerodynamic loads process demands increased simulation capabilities and simulation capacity. Thus, optimization of runtime performance and parallel scalability is one important aspect to be faced among others like algorithmic acceleration. This is especially true for the current multi-core computer systems, because more and more parallel processes become available for single simulations. The challenge will be to keep and even increase the parallel efficiency for massive parallel simulations on future many-core computers.

This paper describes a method for improving the MPI communication performance of an unstructured CFD code in a parallel multi-core environment. By reordering the MPI ranks, a mapping of MPI processes to CPU cores is established, such that the main communication takes place within the compute nodes. Since intra-node communication is usually faster than inter-node communication, by this means the overall communication and execution time can be reduced. The developed optimization tool for the MPI communication is generic in the sense that it does not rely on a specific network architecture and that any MPI application – as far as the communication pattern of the application is known – can benefit from the MPI rank reordering. The optimization tool is tested with the DLR TAU code and three aerodynamic configurations to demonstrate the results.

* Corresponding author.
E-mail address: barbara.brandfass@dlr.de (B. Brandfass).

2. Related work

The optimal mapping of processes to processors has been addressed in several scientific articles. Two common approaches can be distinguished: One is to solve the mapping problem directly during the partitioning of the computational grid. The other is to partition the grid focusing on optimal load balance and to solve the mapping problem for the given partitioning afterwards.

One of the first available software packages for partitioning unstructured grids – Chaco [1] from Sandia National Laboratories – allows to perform a partitioning optimized for single-core architectures with 1D, 2D, 3D mesh or hypercube network topologies. Heavily communicating processes are mapped to processors considered nearby in the respective context. Although not explicitly supported, applications running on multi-core architectures with arbitrary network topologies can still benefit to some extend from using the 1D mesh metric, since nearby processors in this context include processors on the same compute node.

Walshaw and Cross describe in [2] a multilevel graph partitioning algorithm which takes account of a user supplied network communication model. The mapping problem is solved for the initial partition on the coarsest graph level (using a cost function very similar to the one we use). The refinement of the graph is done also with respect to the network model. The method has been implemented into the JOSTLE mesh partitioning software tool [3].

In [4] the authors used the topology information *inside* the allocated compute nodes in order to optimize the communication with regard to Non-Uniform Memory Access (NUMA) effects. Communication using the network is assumed to be the most expensive but otherwise the topology of the network is not taken into account. The communication pattern of the application is extracted by tracing the size of MPI user data exchanged between processes. The resulting graph problem is then solved with the SCOTCH software package [5] to get an optimized mapping.

Bhatele et al. performed in [6] a case study for communication optimization on 3D mesh interconnects with the OpenAtom software package [7]. The article describes how to derive an optimal mapping for the used application on a 3D torus or mesh network. The solution presented is based on OpenAtoms specific structured communication pattern and thus is not suitable for generic communication patterns.

A different approach was proposed by Berti and Träff in [8]. In this article the usage of the MPI topology functionality [9, chap. 7] was discussed for the solution of the process re-mapping problem. The authors show that specific communication patterns can significantly benefit from a non-trivial implementation of the MPI topology functionality but also state that it does not provide enough information for an optimal mapping of unstructured communication since the MPI specification does not allow for weighted communication graphs.

3. The DLR TAU code

The DLR TAU code is a modern software system for the prediction of turbulent, viscous or inviscid flows about complex geometries from the low subsonic to the hypersonic flow regime, employing hybrid unstructured grids. TAU is composed of a number of modules and libraries, which can both be used as standalone tools with corresponding file I/O or within a Python scripting framework without file I/O, i.e. using common memory allocation. TAU provides a flow solver and functionality for grid handling or manipulation like grid partitioning, deformation, re- and de-refinement as well as post processing like extraction of arbitrary sub-regions of the grid and provides several export filters for different formats [10].

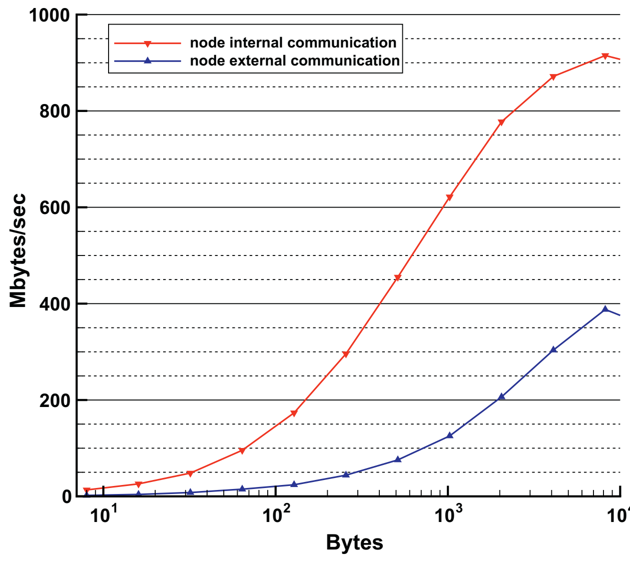
All modules of TAU are parallelized based on domain decomposition and the message passing concept using MPI. Thus, a complete process chain for a simulation can use a single set of processors.

For multi-disciplinary simulations data interfaces and extensions are available for coupling TAU with codes for structure and flight mechanics as well as for simulations of aircrafts in trimmed flight. In order to allow for in-memory transfer of data to other software in between a simulation, TAU can also be used within the FlowSimulator environment [11].

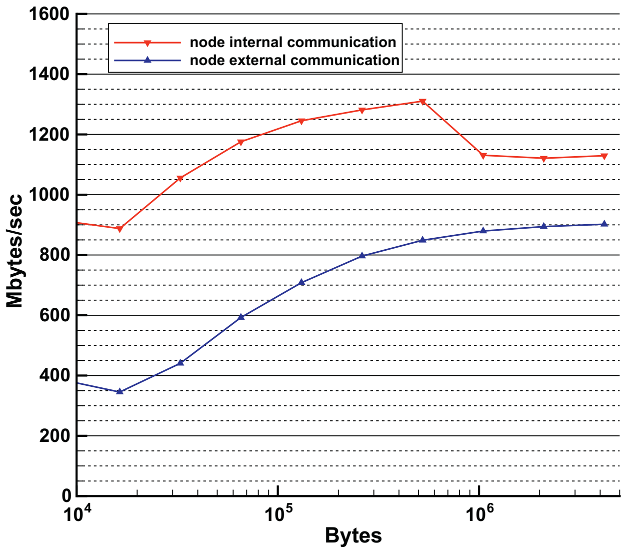
TAU is heavily used by the European Aerospace Industry, by DLR, other research centres and many universities. Many users apply TAU for their daily work keeping several thousand cores busy 24 h every day. Thus, optimization of runtime and parallel efficiency is one of the important topics in its development roadmap.

4. Motivation

The motivation of the described work is derived from an observation made in previous work. In [12] various optimization techniques with respect to an overall runtime reduction of the TAU code flow solver were proposed and the effects were tested on two different clusters. The main difference between those two clusters is that the first has 4 sockets and 16 CPU cores and the second has 2 sockets and 8 CPU cores per compute node. An interesting outcome of the scalability assessments of the flow solver was that on the first cluster the code scaled up to 2048 cores quite well and on the second the scalability ended at around 1024 cores. A closer look at the communication pattern of the tested case showed a clear link between node-internal communication and better speedup. On the cluster with 16 cores per node more of the MPI messages reside within the compute nodes whereas on the cluster with only 8 cores per node more of the MPI messages have to be send via the Infiniband network. Thus, in order to get a better speedup and an overall runtime reduction one has to ensure that most of the MPI communication will take place within the compute nodes. This becomes even more essential, when the number of CPU cores per compute node increases.



(a) Small messages up to 10 KB



(b) Larger messages up to 4 MB

Fig. 1. MPI PingPong benchmark on an Infiniband SDR cluster.

In Fig. 1 one can see the difference between node-internal and node-external MPI bandwidth for an AMD Barcelona cluster connected with Infiniband Single Data Rate (SDR).

5. MPI communication optimization

The time needed for a communication between two MPI processes depends on the size of the message as well as the location of the sending and receiving process within the compute cluster.

In order to get a correct model of the communication cost for an application, its communication pattern as well as the topology of the computing platform on which it is executed has to be taken into account.

For an application running with n processes on n CPU cores, let $A \in \mathbb{R}^{n\times n}$ be the *communication matrix*, whose entries a_{ij} denote the amount of communication from process i to process j and let $B \in \mathbb{R}^{n\times n}$ be the *topology* or *distance matrix*, whose entries b_{ij} denote the distance between CPU core i and CPU core j. Now the optimal mapping of MPI processes to CPU cores can be formulated as a quadratic assignment problem (QAP) as follows.

Find a one-to-one mapping π of objects to locations, which minimizes the overall communication cost

$$Z(A,B,\pi) = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{\pi(i)\pi(j)} \cdot b_{ij} \tag{1}$$

A solution for the QAP is given as a permutation π of the set $\{1,2,\ldots,n\}$, meaning that object $\pi(i)$ is assigned to location i.

The MPI communication optimization procedure now consists of three steps: modeling of communication and distance matrix, solving the quadratic assignment problem and reordering the MPI ranks according to the solution.

5.1. Modeling of communication and distance matrix

The communication matrix has to represent the communication requirements of the parallel application. The parallelization of the DLR TAU code is based on a domain decomposition of the computational grid. In each solver iteration, data has to be exchanged on the domain boundaries (halo). The communication requirement for the TAU flow solver can be modeled, by setting the entries a_{ij} of communication matrix $A = (a_{ij})$ to the number of halo-points of domain i belonging to domain j, since this number is directly proportional to the size of the message which has to be send from process i to process j during each communication call.

The distance matrix on the other hand has to represent the time needed to send a message from a process located on CPU core i to a process located on CPU core j. In particular it has to reflect the relation between the communication speed over the attached network and within the compute nodes. This is accomplished by setting the entries b_{ij} of the distance matrix $B = (b_{ij})$ to a small value if CPU core i and CPU core j are located on the same compute node and to a larger value otherwise. If the exact topology of the underlying network is known, a refined distance model can be achieved by multiplying the entries b_{ij} with the number of hops between CPU cores i and j in the network.

Fig. 2 shows an example of a distance matrix for 12 CPU cores with 4 cores per compute node. Here the distance value for CPU cores located on the same node is set to 1 and for cores located on different nodes to 10.

$$\begin{pmatrix}
0 & 1 & 1 & 1 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
1 & 0 & 1 & 1 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
1 & 1 & 0 & 1 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
1 & 1 & 1 & 0 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
10 & 10 & 10 & 10 & 0 & 1 & 1 & 1 & 10 & 10 & 10 & 10 \\
10 & 10 & 10 & 10 & 1 & 0 & 1 & 1 & 10 & 10 & 10 & 10 \\
10 & 10 & 10 & 10 & 1 & 1 & 0 & 1 & 10 & 10 & 10 & 10 \\
10 & 10 & 10 & 10 & 1 & 1 & 1 & 0 & 10 & 10 & 10 & 10 \\
10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 0 & 1 & 1 & 1 \\
10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 1 & 0 & 1 & 1 \\
10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 1 & 1 & 0 & 1 \\
10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 1 & 1 & 1 & 0
\end{pmatrix}$$

Fig. 2. Distance matrix for 12 CPU cores with 4 cores per node.

Remark: A relocation of two processes within the same compute node will not change the communication cost. This is due to the characteristic block structure of the distance matrices [13, chap. 4.2].

5.2. Solving the quadratic assignment problem

The QAP is a strongly NP-hard combinatorial optimization problem [14]. By exact solution methods, only small instances of up to $n = 20$ can be solved in reasonable time [15]. Due to the relative large number of CPU cores involved in a parallel computation, these methods are not suitable for our MPI communication optimization. Therefore we have to apply a heuristic to get a near optimal solution to the problem. Several heuristic approaches for approximately solving QAPs can be found in the literature (for an overview see e.g. [15]). Unfortunately most of these algorithms still require huge computation time for large problems.

Two heuristics have been found, which provide acceptable solutions in comparatively short time: A construction method introduced by Müller-Merbach [16] and an improvement method based on an algorithm proposed by Heider [17]. They have been tested on several platforms with different aircraft configurations.

5.2.1. Construction method

The algorithm starts with an empty permutation and iteratively assigns processes to CPU cores until a full one-to-one mapping is constructed.

In the first step, for every process p the total communication load

$$A_p = \sum_{i=1}^{n}(a_{ip} + a_{pi})$$

and for every CPU core q the total distance

$$B_q = \sum_{i=1}^{n}(b_{iq} + b_{qi})$$

is computed and the process with the largest communication load is assigned to the CPU core with the smallest total distance.

After this first assignment, only communication requirements and distances between already assigned and still unassigned processes and CPU cores are considered. Thus, in step $k = 2,\ldots,n$ let $P^{(k)}$ and $Q^{(k)}$ denote the index sets of the processes and CPU cores, which have already been assigned during steps $1,\ldots,k-1$. Now, for every still unassigned process $p \notin P^{(k)}$ the communication sum

$$A_p^{(k)} = \sum_{i\in P^{(k)}}(a_{ip} + a_{pi})$$

and for every unassigned CPU core $q \notin Q^{(k)}$ the distance sum

$$B_q^{(k)} = \sum_{i\in Q^{(k)}}(b_{iq} + b_{qi})$$

is computed and the process with the largest communication sum is assigned to the CPU core with the smallest distance sum.

The benefit of this method is that the overall execution time is quite fast (for up to $n = 1024$ lower than 3 s [13, chap. 6.2]). But the main drawback of the procedure is that a better solution, which will lead to a minimized target function value, cannot be guaranteed.

5.2.2. Improvement method

The second heuristic belongs to the class of local search algorithms. The algorithms starts with an initial feasible solution $\pi \in \mathscr{S}_n$ and tries to improve it by substituting it with a better solution from its *pair-exchange* neighborhood $N(\pi) = \{\pi^{i,j} | \; i, j = 1,2,\ldots,n\}$, where

$$\pi^{i,j}(k) := \begin{cases} \pi(k) & k \notin \{i,j\} \\ \pi(i) & k = j \\ \pi(j) & k = i \end{cases}$$

$N(\pi)$ consists of all permutations which can be obtained from π by swapping two elements, that is by reassigning process $\pi(i)$ to CPU core j and process $\pi(j)$ to CPU core i.

Basic Algorithm. For every pair (i,j) the neighbor $\pi^{i,j}$ of the current solution π is examined. The neighborhood is searched in a cyclic manner, beginning with $i = 1$ and $j = 2$. In each step the pair (i,j) is updated to

$$(i,j) = \begin{cases} (i,j+1) & j < n \\ (i+1,i+2) & j = n,\ i < n-1 \\ (1,2) & j = n,\ i = n-1 \end{cases} \tag{2}$$

As soon as a permutation is found which yields lower communication cost, it replaces the current solution and the search is continued in the neighborhood of the new solution with the next pair (i,j) in the predefined order. This procedure is repeated until no further improvement is found.

The runtime of this algorithm increases quite fast with increasing problem size n. Although the calculation of the communication cost $Z(A,B,\pi^{i,j})$ can be done in $\mathcal{O}(n)$ time, once the value $Z(A,B,\pi)$ is known, scanning the whole neighborhood of a solution takes $\mathcal{O}(n^3)$ time, as the size of the neighborhood itself is $\binom{n}{2}$.

To reduce the runtime of the algorithm we introduced the following modifications (for a more detailed description see [13, chap. 5.2]):

(a) We use symmetric matrices as input for the algorithm, since for these only about half the number of arithmetic operations is needed to calculate $Z(A,B,\pi^{i,j})$. In our case the distance matrix B is already symmetric. The communication matrix A, which is not symmetric, is substituted by matrix $A' := \frac{1}{2}(A + A^T)$. This does not change the outcome of the algorithm, if at least one of the matrices A or B is symmetric, the problem of solving QAP(A,B) is equivalent to solving QAP$\left(\frac{1}{2}(A + A^T), \frac{1}{2}(B + B^T)\right)$.

(b) A further reduction of the runtime is achieved by neglecting those pairs (i,j) for which the target function will not change due to the structure of the distance matrix (see remark in Section 5.1).

(c) We partition the search space and start the search successively on each part. The size and the number of these subsets of neighborhood $N(\pi)$ are defined by a parameter $s < n$. For $k = 1, \ldots, \lceil \frac{n}{s} \rceil$ the kth subset is given by

$$N^{(k)}(\pi) = \{\pi^{i,j} | i,j = 1 + (k-1) \cdot s, \ldots, \min(k \cdot s, n)\}.$$

The calculation of the target function still depends on the problem size n, but reducing the number of possible pairs (i,j) from $\mathcal{O}(n^2)$ to $\mathcal{O}(s^2)$ leads to a significant reduction of runtime if $s \ll n$. Of course this also means that solutions outside the subsets $N^{(k)}(\pi)$ cannot be found. For parameter s we found the value 64 to be a good tradeoff between quality of the solution and runtime of the method.

The result of the improvement method depends strongly on the initial solution. Several tests of the method were performed in [13] and the main conclusion of these tests was that it is recommended to use the solution of the construction method described above as an input rather than the identity permutation. This workflow leads to much better results not only for the solution but also for the overall runtime of the method (see also [13, chap. 6.1]).

5.3. MPI rank reordering

The optimized mapping of MPI processes to CPU cores is established by reordering the MPI ranks according to the solution π. This is done by permuting the entries of the machinefile used for the start-up of the parallel application. Here the inverse permutation π^{-1} has to be used for the reordering of the entries in the machinefile. It is assumed that the corresponding MPI will always start up the processes in a fill-up manner (e.g. process 0 to first entry, process 1 to second entry, etc.) and not in a round-robin fashion. Furthermore it is expected that the entries in the machinefile also appear in a fill-up manner (e.g. for 8 cores per compute node: entries 1–8 on the first allocated node, entries 9–16 on the second allocated node, etc.).

6. Results

In this section we will discuss the results obtained with the MPI communication optimization technique.

6.1. Setup of the benchmarks

In order to cover a wide range of up-to-date HPC clusters we have used three different systems for our tests. All the systems are connected by Infiniband and have a reasonable number of compute nodes (see Table 1 for more details).

For the benchmarks we employed the flow solver of the DLR TAU code and three aerodynamic configurations with varying grid sizes (see Table 2) to investigate the potential of the optimization method.

For all configurations unstructured grids are used. They are partitioned according to the number of used MPI processes employing one of the following methods:

1. Recursive coordinate bisection
2. Graph partitioning

After the domain decomposition of the computational grids, we run the flow solver with the settings described in Table 3 several times on the same set of compute nodes.

Table 1
Details of used HPC clusters.

	CPU – cores per node	Nodes	Cores	IB[a]
A	Nehalem 2.53 GHz – 8	200	1600	DDR
B	Westmere 2.93 GHz – 12	120	1440	QDR
C	Westmere 2.93 GHz – 12	648	7776	QDR[b]

[a] Infiniband data rates: DDR 20 GBit and QDR 40 GBit.
[b] 50 % blocking factor for more than 24 compute nodes.

Table 2
Details of used aerodynamic configurations.

	Configuration	Grid points	Vol. elements
1	Cruise DLR-F6	2,000,000	5,000,000
2	Cruise DLR-F6 DPWII	15,800,000	39,000,000
3	High-lift	7,600,000	19,500,000

Table 3
Main flow solver settings for the benchmarks.

Turbulence model	Spalart Allmaras
Discretization	Central scheme
Runge–Kutta steps	3
Iterations	200
Multigrid cycle	no, 3v and 4w

The first run was performed with the initial allocated node set without optimization of the process placement. The second run was performed with the optimized MPI process placement (permutation of the machinefile) computed by the construction method (CM). The third run of the solver was performed with a process placement computed with both methods (IM), first the construction method to produce an initial solution and afterwards the improvement method to get a better solution.

Table 4
Initial and optimized communication cost for DLR-F6 on cluster A.

Cores	32	64	128
Initial	2,275,577	3,720,654	5,526,111
CM	849,077	1,352,493	2,599,455
IM	734,309	1,351,899	2,159,661

Table 5
Initial and optimized communication cost for DLR-F6 on cluster B.

Cores	48	96	240
Initial	2,603,424	4,832,475	7,495,225
CM	969,132	1,784,859	3,118,930
IM	826,842	1,624,974	2,793,904

6.2. Results of the heuristics

In Tables 4 and 5 selected results for the DLR-F6 test case with both heuristics are listed. From the results in both tables one can see that the construction method (CM) is able to minimize the target function value for the tested case by a significant amount (down to 36% of the initial value for 64 domains). The improvement method (IM) further improves the solution with respect to the target function value (see IM in Tables 4 and 5).

The effect of the optimization can be visualized by permuting the communication matrix with the solution of the heuristic. An example for the DLR-F6 test case with the TAU build-in recursive coordinate bisection partitioner for 128 domains can be found in Fig. 3. It can be seen, that by employing the permutations obtained by the heuristics, the communication partners are shifted towards the blocks on the main diagonal which can be associated with being *inside* a compute node. Comparison of Fig. 3b and c reveals no big difference between the effect of the construction method (CM) and the improvement method (IM). It can also be seen that it is not possible to map all communication inside the compute nodes since each domain has to many communication partners.

Graph partitioners usually produce better partitions in terms of the number of neighbors. For the communication matrices for the DLR-F6 for 360 domains shown in Fig. 4 the Zoltan graph partitioner from Sandia National Laboratories [18,19] has been used. When using Zoltan for an initial partitioning with non-distributed input data, the resulting communication pattern already looks

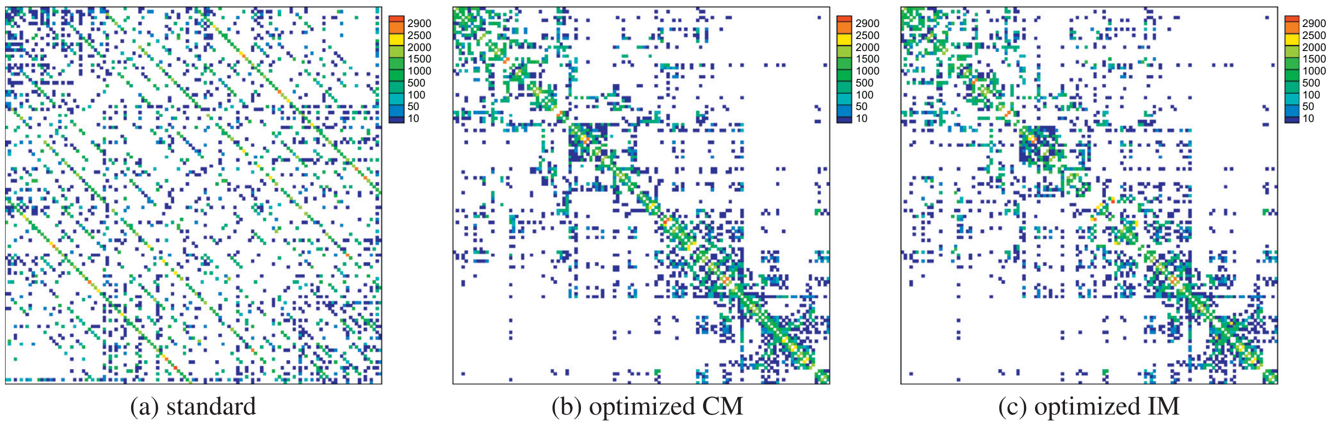


Fig. 3. Communication matrix for DLR-F6 on 128 domains and the recursive bisection partitioner standard (a) and optimized for a 8 cores/node architecture with the construction method (b) and with the improvement method (c).

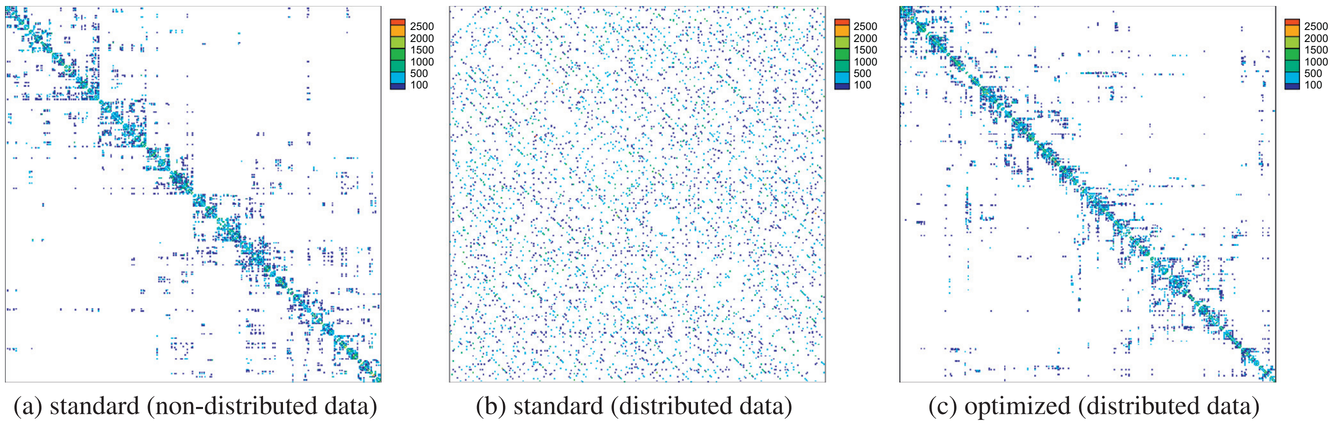


Fig. 4. Communication matrix for DLR-F6 on 360 domains and the Zoltan graph partitioner (a) standard with non-distributed input data, and with distributed input data (b) standard and (c) optimized for a 12 cores/node architecture with the construction method.

pretty good (Fig. 4a). But this approach of course has the disadvantage of not scaling with respect to memory and cannot be used for very large computational grids. Unfortunately the communication matrix looks quite different when Zoltan is used with distributed input data: Fig. 4b shows that the communication partners are scattered all over the available cores. In Fig. 4c it can be seen that employing the permutation obtained by the construction method again results in a communication pattern similar to the one achieved with non-distributed input data.

The expected result for the flow solver runs with the optimized process placement is a reduction in runtime. By a closer inspection of the corresponding benchmarks for the DLR-F6 test case, which are listed in Tables 6 and 7, one can see that there is in fact a significant reduction in the flow solver runtime due to the optimized process placement.

But there is no significant difference in the flow solver runtimes between the construction method (CM) and the improvement method (IM). For some benchmarks with the optimized process placement obtained by the improvement method the runtime of the flow solver was even slower compared to the ones where only the construction method was used (see 128 cores in Table 6 and 48 cores in Table 7). Furthermore the time needed to run the improvement method is quite large compared to the construction method (e.g. 33.5 s (IM) vs. 0.16 s (CM) for $n = 256$).

Due to this observation we have put our focus in the following discussion on the results obtained with the construction method.

6.3. Discussion

The achieved performance gain with the optimized MPI process placement for the selected test cases and clusters is shown in the following figures. For each benchmark run, the runtime in seconds for 100 iterations of the flow solver is plotted together with the relative speedup compared to the number of cores on one compute node, i.e. 8 for cluster A or 12 for clusters B and C (for the DLR-F6 DPWII case on cluster A 16 cores are used instead of 8).

In Figs. 5, 6, 7 and 9, 10, 11, 12 the results for the TAU build-in recursive coordinate bisection partitioner (RCB) are shown. In Figs. 8 and 13 the Zoltan graph partitioner (with distributed input data) was used for the benchmarks.

DLR-F6. At a first glance all timings with optimization are better compared to the ones without rank reordering. By a closer view on the results for the DLR-F6 test case one can see that the reduction in solver runtime without multigrid acceleration is only visible in the better speedup.

For cluster A we see a slightly enhanced speedup for 64 cores and beyond (see Fig. 5). For cluster B a similar improvement of

Table 6
Runtime of the flow solver for DLR-F6 on cluster A before and after optimization.

Cores	32		64		128	
Multigrid	no	3v	no	3v	no	3v
Standard	70.1	119.1	38.3	68.2	21.3	42.3
CM	69.5	115.5	36.7	62.4	19.7	37.9
IM	68.8	114.1	36.7	62.5	20.3	39.4

Table 7
Runtime of the flow solver for DLR-F6 on cluster B before and after optimization.

Cores	48		96		240	
Multigrid	3v	4w	3v	4w	3v	4w
Standard	88.5	110.3	55.2	81.4	37.7	78.6
CM	85.6	103.8	49.9	71.5	34.0	73.3
IM	86.4	104.5	50.0	71.4	33.8	72.5

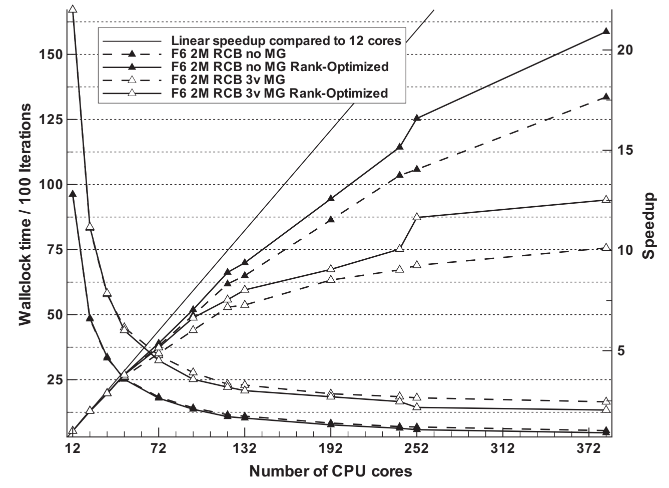


Fig. 6. Results of the DLR-F6 on cluster B with the RCB partitioner for a 3v multigrid cycle and without multigrid acceleration.

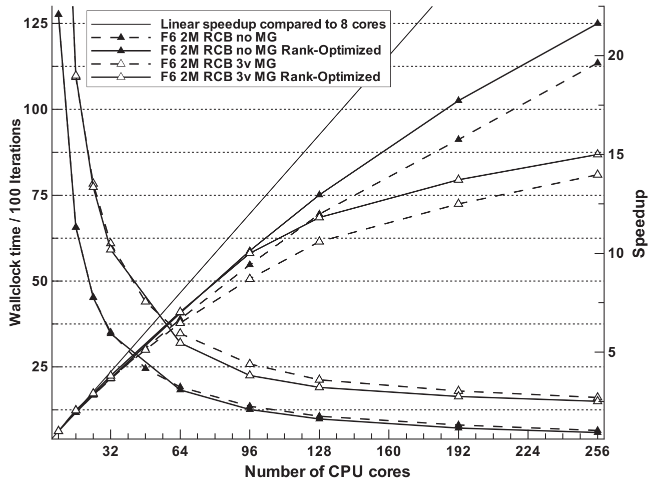


Fig. 5. Results of the DLR-F6 on cluster A with the RCB partitioner for a 3v multigrid cycle and without multigrid acceleration.

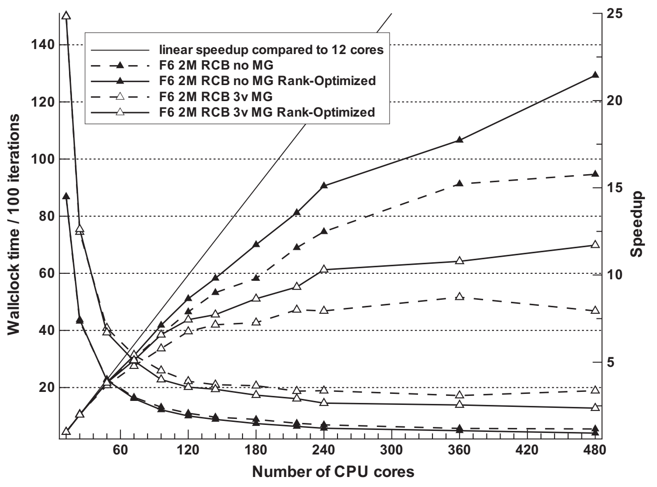


Fig. 7. Results of the DLR-F6 on cluster C with the RCB partitioner for a 3v multigrid cycle and without multigrid acceleration.

speedup is observed between 72 and 240 cores but for 252 and 384 cores we can see a significant offset to the initial MPI process placement (see Fig. 6).

For cluster C (see Fig. 7), where we have a 50% blocking factor in the Infiniband network if more than 24 compute nodes are used, the speedup is improved starting at around 72 cores up to 360 cores with a considerable offset at 480 cores (speedup: optimized 23 compared to initial 17). When the Zoltan graph partitioner is used for the domain decomposition the situation changes only slightly. Graph partitioners usually produce less neighbors during partitioning than geometric partitioners and the effect on the runtime of the flow solver and the corresponding speedup curve can be seen by a comparison of the runs without optimization in Figs. 7 and 8. The performance gain of the optimized process placement with Zoltan on cluster C (see Fig. 8) is up to 240 cores only small, but for more than 20 compute nodes there is a clear offset to the runs with initial placement (speedup: optimized 24 to initial 17 for 480 cores). The main reason for this can be seen in Fig. 4b and c. In the initial placement the communication partners are scattered all over the available cores (or nodes) and the rank reordering method shifts most of the communication partners towards the blocks on the main diagonal, which can be associated with

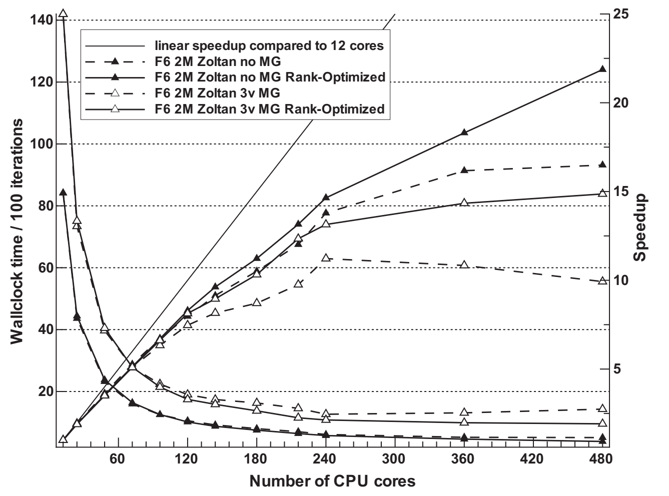


Fig. 8. Results of the DLR-F6 on cluster C with the Zoltan partitioner for a 3v multigrid cycle and without multigrid acceleration.

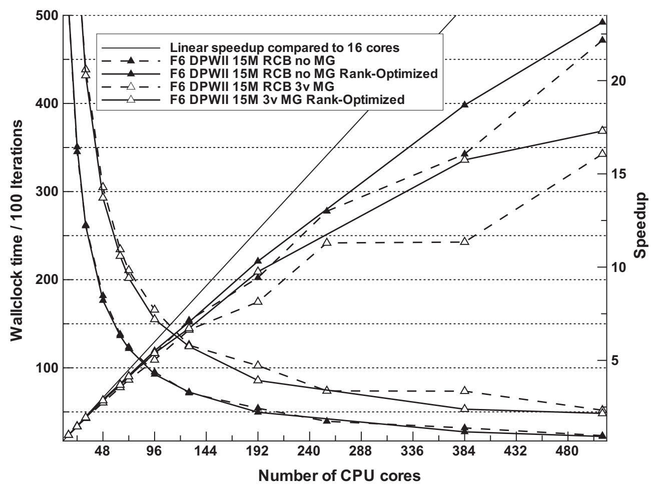


Fig. 9. Results of the DLR-F6 DPWII on cluster A with the RCB partitioner for a 3v multigrid cycle and without multigrid acceleration.

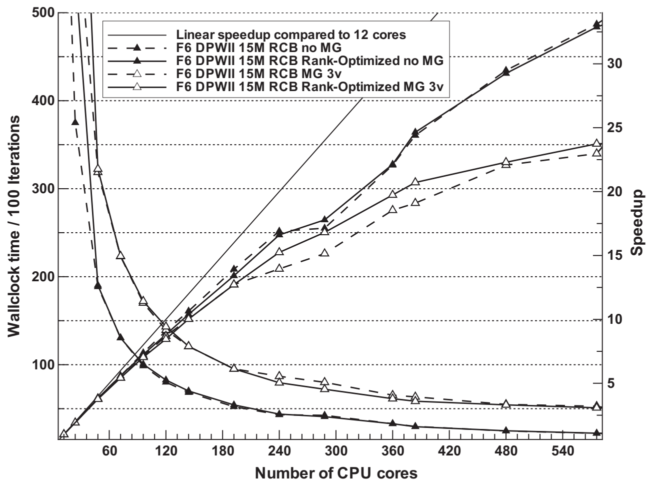


Fig. 10. Results of the DLR-F6 DPWII on cluster B with the RCB partitioner for a 3v multigrid cycle and without multigrid acceleration.

being *inside* a compute node. With the optimized placement there is less communication over the network and the blocking factor of the network[1] has a lesser impact on the performance and scalability.

When we take a closer look on the results for the DLR-F6 with multigrid acceleration, we can see that the improvement on all tested clusters becomes more visible. There is a clear offset between the runtimes of the benchmarks with optimized process placement and the ones with initial placement (starting at 32 cores for cluster A – Fig. 5, 48 cores on cluster B – Fig. 6 and 72/96 cores on cluster C – Figs. 7 and 8).

DLR-F6 DPWII. For the DLR-F6 DPWII test case no effects are visible for the runs without multigrid acceleration on small numbers of cores (up to 128 cores no differences in runtime on cluster A see Fig. 9). For larger numbers of cores the improved MPI process placement leads to slightly better scalability on cluster A (speedup: optimized 23 instead of initial 22.2). On cluster B there is no improvement at all except for 288 and 384 cores (see Fig. 10).

When using multigrid acceleration the optimized MPI process placement results on all clusters in a significant runtime reduction and better scalability. There is only one exception, which is the run on cluster C with 240 cores (see Fig. 11). The effect of the optimized process placement becomes visible for cluster A at 96 cores (see Fig. 9), for cluster B at 240 cores (see Fig. 10) and for cluster C at 144 cores (see Fig. 11).

High-lift. For the last test case (high-lift), which was only benchmarked on cluster C, the optimized process placement leads to a better overall scalability. For the benchmarks with the recursive bisection partitioner the reduction of the solver runtime becomes visible at 240 cores (with and without multigrid acceleration). However the run with 360 cores is a little bit out of place for the optimized process placement, because no reduction in runtime could be observed for the 3v multigrid cycle and even a slower runtime has been accounted for the run without multigrid acceleration (see Fig. 12).

It seems that the 360 core run for this test case with the recursive bisection partitioner could not be improved by the rank reordering. When the Zoltan graph partitioner is used for the same test case (see Fig. 13) there is no such strange behavior for the 360 core run. For both benchmarked multigrid cycles the optimized process placement reduces the runtime of the flow slover by a significant amount (starting around 240 cores) and the speedup is considerably improved compared to the initial placement.

[1] 50% for more than 24 compute nodes.

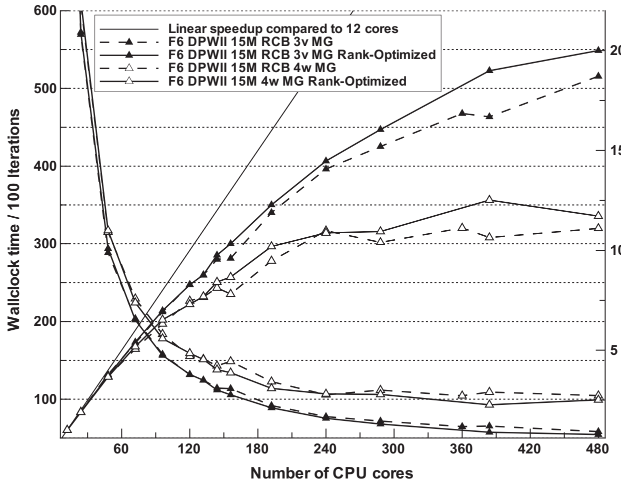


Fig. 11. Results of the DRL-F6 DPWII on cluster C with the RCB partitioner for a 3v and 4w multigrid cycle.

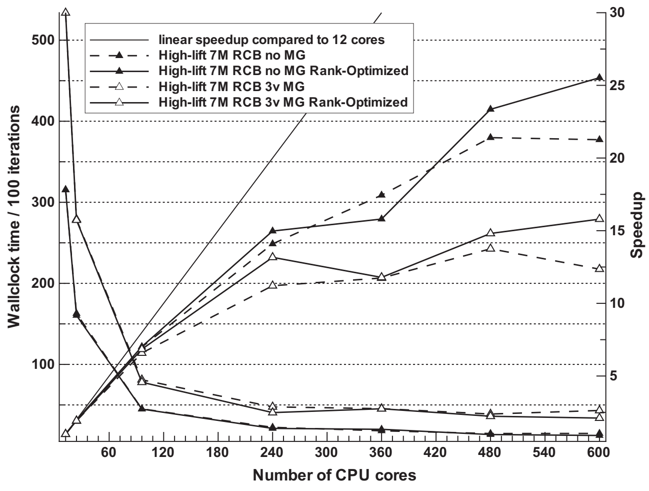


Fig. 12. Results of the high-lift case on cluster C with the RCB partitioner for a 3v multigrid cycle and without multigrid acceleration.

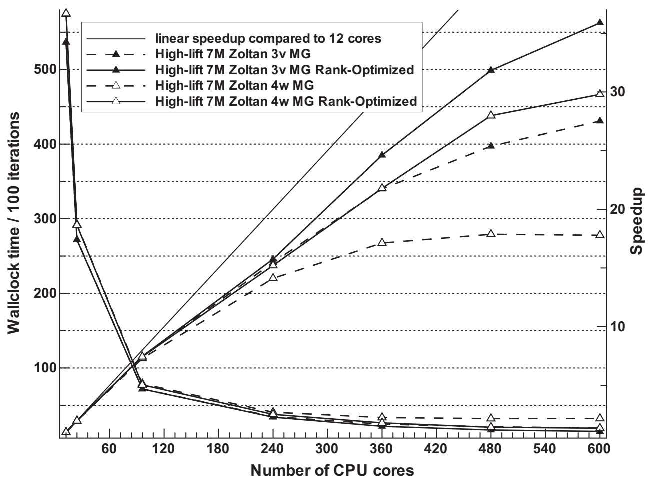


Fig. 13. Results of the high-lift case on cluster C with the Zoltan partitioner for a 3v and 4w multigrid cycle.

7. Conclusion

In this paper we have described an easy to use method for communication optimization due to a MPI rank reordering. We have successfully demonstrated the performance benefit of this generic MPI optimization tool with the DLR TAU code. The obtained results for all tested configurations and clusters are quite promising and it is very likely that other software packages also can benefit from the presented rank reordering method.

Despite the fact that there are partitioning software packages, which are capable to produce quite good process mappings like Zoltan [19] or Chaco [1], the presented method in this paper is of a more general nature. Instead of solving the problem during the partitioning, our method is completely independent from the used partitioner and can be employed on any partitioned grids afterwards. Furthermore it does not rely on non-distributed data like Chaco (no parallel mode) or Zoltan (see Fig. 4a and b) to produce *good* mappings. Another advantage of the method is, that it is not necessary to run the partitioner again, when changing from e.g. 8 cores/node to 12 cores/node on a heterogenous cluster. On the other hand our method could also be used with a more fine-grained network topology modeling. In Section 5.1 we have used only a quite simple model to distinguish between the communication cost inside and outside a compute node. This basic model can easily be refined in order to reflect a more complex node and network topology (e.g. 4 socket nodes, fat-tree networks with a blocking factor, 3D tori or even mixed tori and switch topologies).

Due to the external approach this tool can be applied to any MPI application without the need to perform verification or validation on the optimized application afterwards. The last point is of great interest for legacy CFD codes, due to the high amount of work, which has to be invested for verification and validation when changing the code for optimization purposes. The described rank reordering method does not need such a review process because it only affects the MPI machinefile. Furthermore the method can easily be integrated in most of the workflows for modern CFD software packages.

References

[1] Hendrickson, B., Leland, R. Chaco: Software for Partitioning Graphs. <http://www.sandia.gov/~bahendr/chaco.html>.
[2] Walshaw C, Cross M. Multilevel mesh partitioning for heterogeneous communication networks. Future Gen Comput Syst 2001;17:601–23.
[3] JOSTLE – graph partitioning software. <http://staffweb.cms.gre.ac.uk/wc06/jostle/>.
[4] Mercier G, Clet-Ortega J. Towards an efficient process placement policy for MPI applications in multicore environments. In: EuroPVM/MPI. Lecture notes in computer science, vol. 5759. Espoo, Finland: Springer; 2009. p. 104–15.
[5] Scotch and PT-Scotch: software package and libraries for sequential and parallel graph partitioning, static mapping, and sparse matrix block ordering, and sequential mesh and hypergraph partitioning. <http://www.labri.fr/perso/pelegrin/scotch/>.
[6] Bhatelé A, Bohm E, Kalé LV. A case study of communication optimizations on 3D mesh interconnects. In: Euro-Par, vol. 5704. LNCS; 2009. p. 1015–28.
[7] OpenAtom <http://charm.cs.uiuc.edu/OpenAtom/>.
[8] Berti G, Träff JL. What MPI could (and cannot) do for mesh-partitioning on non-homogeneous networks. In: Mohr B, Träff JL, Worringen J, Dongarra J, editors. Proceedings of EuroPVM/MPI 2006. LNCS, vol. 4192. Bonn, Germany: Springer; 2006. p. 293–302.
[9] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, Version 2.2, High Performance Computing Center Stuttgart (HLRS); 2009.
[10] Schwamborn D, Gerhold T, Heinrich R. The DLR TAU-Code: recent applications int research and industry. In: Wesseling P, Onate E, Periaux J (Eds.), Proceedings, ECCOMAS CFD 2006 CONFERENCE; 2006.
[11] Meinel M, Einarsson GO. The flow simulator framework for massively parallel CFD applications. In: PARA 2010 conference: state of the art in scientific and parallel computing; 2010.
[12] Alrutz T, Simmendinger C, Gerhold T. Efficiency enhancement of an unstructured CFD-code on distributed computing systems. In: Biswas R, editor. Parallel computational fluid dynamics, NASA AMES Research Center, NASA Advanced Supercomputing Division; 2009. p. 561–8.
[13] Brandfass B. Optimierung der MPI-Prozess-Kommunikation bei Multicore-Rechnerarchitekturen mit Hilfe des quadratischen Zuordnungsproblems.

Diplomarbeit, Universität Göttingen, DLR-IB 224-2011 A 60; 2010. <http://elib.dlr.de/71972/>.
[14] Sahni S, Gonzales T. P-complete approximation problems. J ACM 1976;23:555–65.
[15] Çela E. The quadratic assignment problem: theory and algorithms. Combinatorial optimization, vol. 1. Dordrecht: Kluwer Academic Publishers; 1998.
[16] Müller-Merbach H. Optimale Reihenfolgen. Berlin: Springer; 1970.
[17] Heider CH. A computationally simplified pair-exchange algorithm for the quadratic assignment problem. Professional paper 101. Arlington (Va): Center for Naval Analysis; 1972.
[18] Devine K, Boman E, Heaphy R, Hendrickson B, Vaughan C. Zoltan data management services for parallel dynamic applications. Comput Sci Eng 2002;4(2):90–7.
[19] Boman E, Devine K, Fisk LA, Heaphy R, Hendrickson B, Leung V, et al. Zoltan home page; 1999. <http://www.cs.sandia.gov/Zoltan>.