CHALMERS

Research report 2013:05

MPI-Parallelization of a Structured Grid CFD Solver including an Integrated Octree Grid Generator

BJÖRN ANDERSSON
ANDERS ÅLUND
ANDREAS MARK
FREDRIK EDELVIK

Department of Applied Mechanics
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2013

Research report 2013:05

MPI-Parallelization of a Structured Grid CFD Solver including an Integrated Octree Grid Generator

by

BJÖRN ANDERSSON
ANDERS ÅLUND
ANDREAS MARK
FREDRIK EDELVIK

Department of Applied Mechanics
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, 2013

MPI-Parallelization of a Structured Grid CFD Solver including an Integrated Octree Grid Generator

BJÖRN ANDERSSON,ANDERS ÅLUND, ANDREAS MARK, FREDRIK EDELVIK

Research report 2013:05
ISSN 1652-8549

Department of Applied Mechanics
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31 772 1000

Abstract

An existing Computational Fluid Dynamics (CFD) solver is parallelized by means of MPI. The solver includes a dynamic and adaptive grid generator for Cartesian Quadtree and Octree grids, which therefore also have to be parallelized. The grid generator generates grids fulfilling a specific set of rules, that have to be enforced also in parallel. The assembly of the large sparse matrices resulting from the implicit discretization of Navier-Stokes equations is done in parallel, as is the solving process. The parallel performance of both of these processes depends heavily on a good load balancing in order to reach satisfactory speedup. Two versions of load balancing are demonstrated, one based on block swapping, and the other by utilizing the Metis or Parmetis software packages for load balancing of graphs. Results are presented for load balancing and for the parallel speedup of solving the linear algebra system of equations.

1 Introduction

An existing serial code for generating structured computational grids based on Quadtree and Octree structures developed at Fraunhofer-Chalmer Centre (FCC) have been parallelized by means of MPI. The grid generator is only a small part of a larger solver framework primarily used for fluid dynamics simulations where it is tightly integrated and both builds the initial grid and adaptively refines it over time. As the overall goal for the software package is to minimize the total solution time: grid generation, matrix assembly, and solution of the equation system it does not make sense to look at scalings with problem size and cluster size of only the grid generator isolated. The main focus in this regard will therefore be on the load balancing of the computational grid, and not the running time of the parallel grid generator.

2 Load balancing

The total solution time can be divided in two parts: computation and communication. The computation time may be assumed to be proportional to the number of computational cells, raised to some power $\gamma \geq 1$. Communication time is assumed to be proportional to the number of cells at the boundary of a process' block of cells.

Assume a homogeneous cubic grid with N^3 cells parallelized on P number of processes, where P is the cube of some natural number. In this setting each process would be assigned a sub-cube of the computational domain. The solution time can then be written,

$$T_s = \alpha \frac{\left(N^3\right)^\gamma}{P} + \beta \left(\frac{N^3}{P}\right)^{2/3} = \alpha \frac{N^{3\gamma}}{P} + \beta \frac{N^2}{P^{2/3}}, \tag{1}$$

where the first term corresponds to the computation time and the second term to the communication time, and α and β are two constants. We see that both terms decrease with increasing P, and this is because each process only communicates with its six closest neighbors, and there is no need to gather the result at the end.

This partition is a very idealized one that is not very realistic. In general, the communication cost when dealing with sparse problems is proportional to the number of cells having a neighbor on another process. This becomes a topological problem, where the assignment of blocks to processes has to take the position of the blocks into account.

When dealing with octrees the most natural building block is a cube, which corresponds to a node at some level of the tree. For simplicity, assume that the computational grid is being divided into M equal size cubes, where $M \geq P$. This corresponds to taking all nodes at a given level of the octree. The task is then to assign these cubes to processes in such a way that the total solution time is minimized. In our case, synchronization among the processes will take place frequently

so we can focus on the process with largest solution time for one iteration. That is,

$$T_I = \max_{i \in [1,\, P]} \left[\alpha \left(N_i^s \right)^\gamma + \beta N_i^c \right], \tag{2}$$

where N_i^s is the number of computational cells in process i, and N_i^c is the corresponding number of cell faces that have a neighbor on another process. In the above idealized example $N^s = N^3/P$ and $N^c = N^2/P^{2/3}$.

Given a partitioning in equal sized cubic blocks as mentioned above N_i^s and N_i^c are easy to evaluate, but the number of possible partitions soon becomes very large as M and P grows. Given that the number of computational cells in the cubic blocks may vary considerably, M in general needs to be much larger than P. A rule of thumb could be that the largest block of cells, where a block is the smallest unit assigned to a process, should contain less than approximately N^3/P cells. The first term in eq. (2) may become prohibitively large otherwise, unless the communication cost is very large compared to computation cost, *i.e.* $\beta \gg \alpha$.

2.1 Mathematical formulation

The problem of finding a partition $\mathcal{P}$ of the blocks minimizing eq. (2) can be stated as the minimax problem

$$\underset{\mathcal{P}}{\operatorname{argmin}} \left[\max_{i \in [1,\, P]} \left[\alpha \left(N_i^s \right)^\gamma + \beta N_i^c \right] \right]. \tag{3}$$

In order to evaluate N_i^s and N_i^c we need to store some information about the blocks. Let the vector n^s of length M store the number of computational cells in each block, and the matrix n^c of size $M \times M$ store the number of faces at the boundary between two blocks if they are neighbors, and 0 otherwise. Let the vector x of length M be the solution vector that in each position stores a number in the interval $[1,\, P]$ that states which process a block is assigned. With these definitions we can evaluate N_i^s and N_i^c as

$$N_i^s = \sum_{k=1}^{M} n_k^s \, \delta_{i,x(k)}, \tag{4}$$

$$N_i^c = \sum_{k=1}^{M} n_{ik}^c \left(1 - \delta_{i,x(k)} \right), \tag{5}$$

where $\delta_{k,l}$ is the Kronecker delta that is equal to 1 if $k = l$ and 0 otherwise.

This type of problem is what the Metis [1, 2] package is designed to solve. By thinking of the blocks and their connections as a graph with weights associated with both vertices and edges the problem can be formulated in a format suitable for the Metis package. With the above definitions of n^s and n^c the vertex cost is $\alpha \left(n_k^s \right)^\gamma$ and the edge cost is βn_{kl}^c. There is another cost that needs to be taken into account, however, when the grid is updated, and that is the cost of transferring blocks from one process to another. There is no way of telling the Metis package which process

a vertex initially belongs to, so the solution found may very well require that each and every block has to be transferred to a new process. The Parmetis [3] packages has a solution to this problem, where the cost of transferring a block can be weighed against the extra cost of having a slightly unbalanced partition. Parmetis is in addition parallel, which may reduce the solution time slightly compared to solving it on the master process as the Metis package does.

2.2 Simplified load balancing scheme

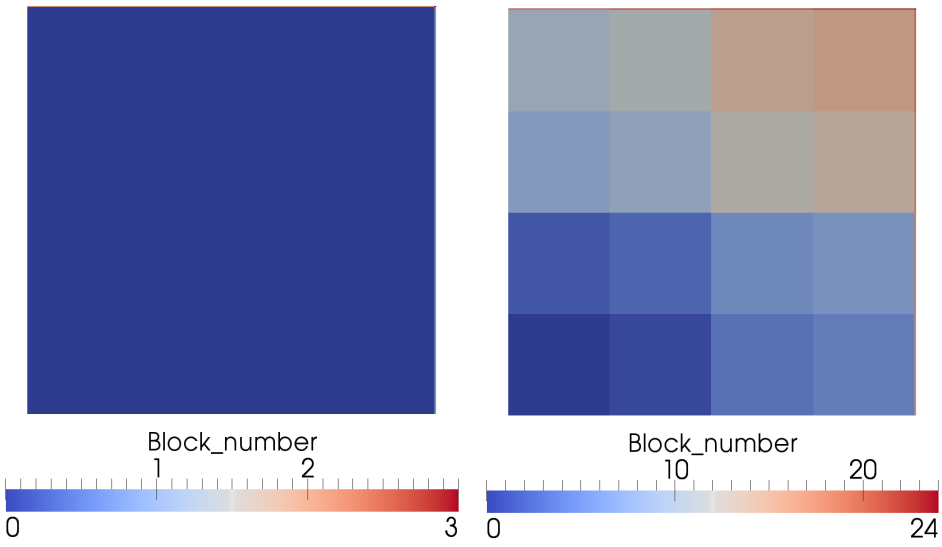
In addition to the Metis/Parmetis solution described above, a simple load balancing scheme have been implemented where blocks of cells are assigned different processes with the aim of minimizing T_I in the case of $\beta = 0$, *i.e.* communication time is negligible. This assumption reduces the problem to finding a partition $\mathcal{P}$ fulfilling

$$\underset{\mathcal{P}}{\operatorname{argmin}} \left[\max_{i \in [1,\, P]} \left(N_i^s - \frac{N^3}{P} \right) \right]. \tag{6}$$

As an example of the load balancing an homogeneous grid in two dimensions with side length $N = 514$ was created on $P = 2$ processes. The minimum amount of equal sized blocks larger than P is $M = 4$. The largest block is of size 512×512, there are two blocks of size 512×2 and one 2×2 block, see fig. 1(a). This results in a very bad load balancing, as the minimax partitioning of eq. (6) gives $512^2/514^2 \approx 99.2\%$ cells on one processor. By splitting the blocks into smaller pieces a better partitioning of blocks is possible. See fig. 1(b) where 25 blocks have been created, sixteen 128×128, eight 128×2 and one 2×2 block. By assigning these blocks to two different processes the most well balanced partitioning has two more cells on one process than the optimal value, $514^2/2$. See fig. 2 for an example of such a partitioning. Note that this partition does not minimize the full time consumption formula in eq. (2) as the rectangular blocks in the lower right corner are assigned to process 1 and therefore creating unnecessary communication. Exchanging these two red blocks with two equal size blocks from the upper row would reduce the communication cost a little bit.

2.2.1 Implementation

The strategy implemented that attempts to solve eq. (6) is a simple block transfer procedure, where in each round all processes evaluates how many computational cells they are currently assigned. Initially all processes are assigned equally many blocks sequentially. As blocks may contain different number of cells this may or may not be a good solution. In order to balance the partition the process with most elements and the one with least are then assigned the task to transfer a block of cells so that the maximum cells on any single process is smaller than before the exchange took place, and that the local balance between these two processes is as even as possible. If a transferable block is found the loop continues until no more block transfer gives a better solution to eq. (6). The end result is not guaranteed



(a) 4 blocks. Note the three very small blocks on the upper and right sides.

(b) 25 blocks.

Figure 1: Different block divisions of a 514×514 grid.

to be a true solution, but is is a local minimum with respect to the block transfer operator, and in practice it seems to work reasonably well.

As described above the algorithm halts when no block can be transferred from the largest process to the smallest one without increasing the unevenness of the load balancing. The maximum number of iterations in the algorithm is determined by the initial state and on how large the blocks are. Assuming the smallest block is of size $\alpha N^3/M$, for some $\alpha < 1$, and the initial partition is the worst possible, *i.e.* all blocks on one process, it would take at most $\left(N^3 - N^3/P\right) / \left(\alpha N^3/M\right) = M\left(1 - 1/P\right)/\alpha = \mathcal{O}(M)$ transfers, as each transfer improves the quality by at least $\alpha N^3/M$ units.

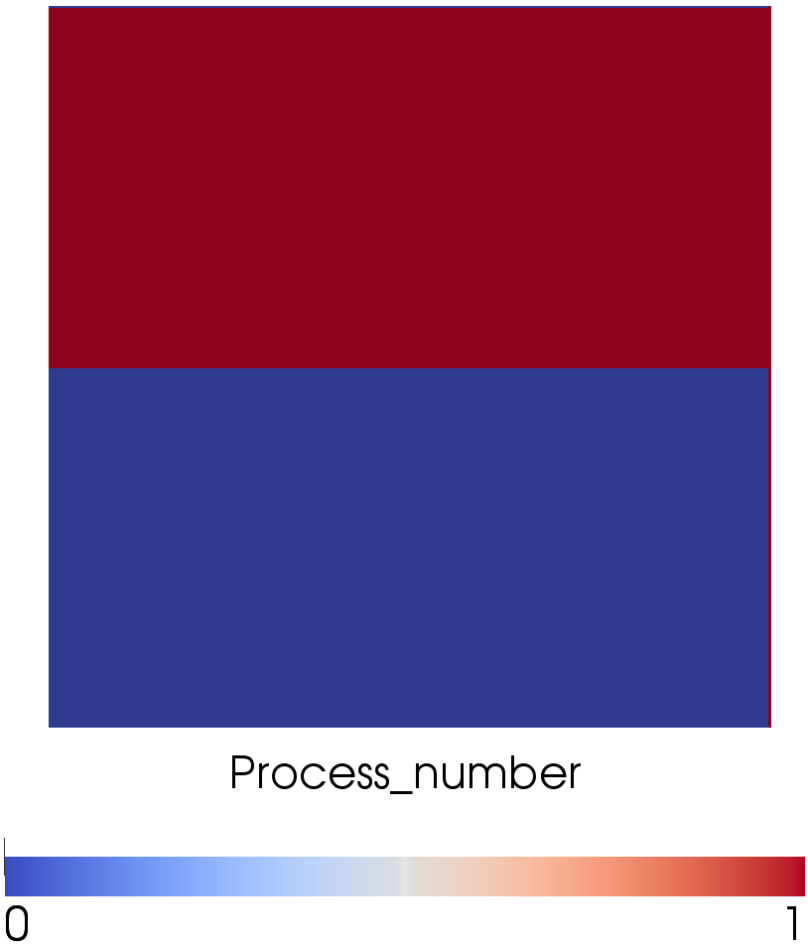


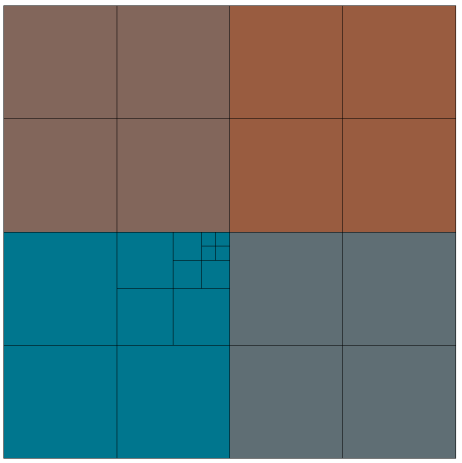
Figure 2: Partitioning of the 25 blocks in fig. 1(b) on two processes.

3 Grid balancing

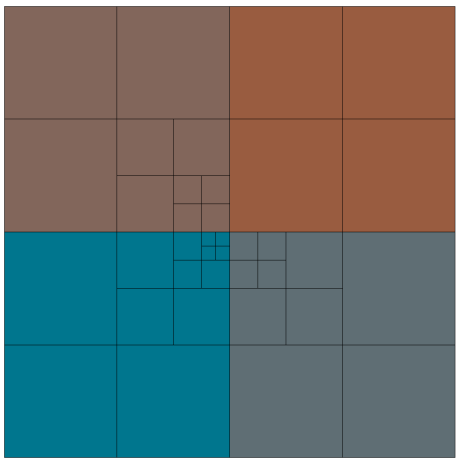
A requirement on the meshes generated is the so-called factor-of-2 rule, stating that neighboring cells may differ in size by at most a factor of 2. Neighbors in this context can be interpreted as either von Neumann or Moore neighbors, *i.e.* either Manhattan or Chebyshev distance of 1, depending on the discretization scheme applied. For cell centered Finite Volume discretization it is enough with the von Neumann neighbors, but Finite Element nodal basis functions require the Moore neighborhood to fulfill the factor-of-2 rule.

This constraint implies that once a subgrid on one process has been refined, the neighboring block has to be notified and refined accordingly. This becomes an iterative process that stops when no part of the grid is updated.

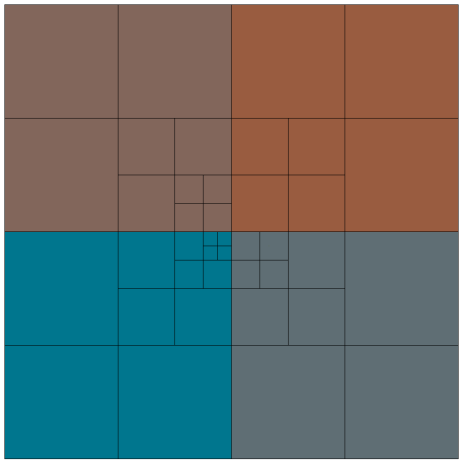
Figure 3(a) shows a situation where the lower left subgrid has been refined in the corner three times. This information is propagated to the lower right and upper left block which are refined two times to fulfill the factor-of-2 rule, see fig. 3(b). In the next iteration of the algorithm the upper right block is refined one time and the balancing is complete for the von Neumann neighborhood, see fig. 3(c). If a Moore neighborhood fulfilling the factor-of-2 rule is desired, the algorithm continues one more step and refines the lower left cell in the upper right block one more time, see fig. 3(d).



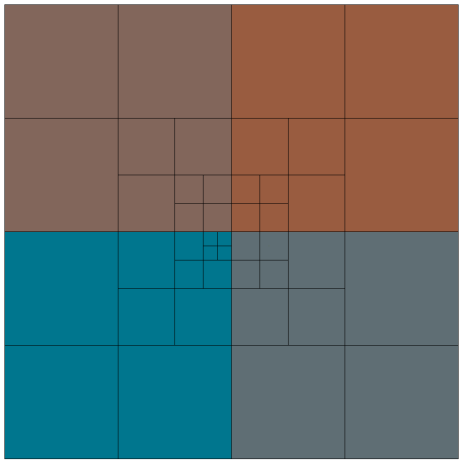
(a) The upper right corner of the lower left block has been refined three times.



(b) The upper left and lower right block has been refined as a consequence of the balancing rule.



(c) The upper right block has been refined and the balancing is complete for a von Neumann neighborhood.



(d) The upper right block has been refined twice and the balancing is complete for a Moore neighborhood.

Figure 3: The different stages of grid balancing.

4 Results

4.1 Load balancing

Figure 4 shows the distribution of computational cells in a homogeneous square grid in two dimensions on two processes as a function of mesh size. Two partitionings are shown, first a naive one with a minimal number of blocks $M \geq P$ where the number of cells in each block is not taken in consideration when assigning blocks to processes. The alternative partitioning is optimized as described above in section 2.2.1 and the number of blocks available were at least 5 per process, *i.e.* $M \geq 5P$. We see that the former partitioning often assigns almost all cells to one process and close to none to the other one. The pattern resembles a sawtooth when plotted against the total number of cells, and this is because the process with the largest number of cells is assigned the same number of cells in each interval, until perfect balance is reached and after that it gets all of them again.

The optimized partitioning has a worst case of about 10% more cells than theoretical optimum in this case with 5 blocks per cell. Splitting the blocks further reduces this number, but has the drawback that the partitioning may be very fragmented, in effect introducing large communication costs if that is not taken into account.

Figure 5 shows the same scaling, but for $P = 7$ processes.

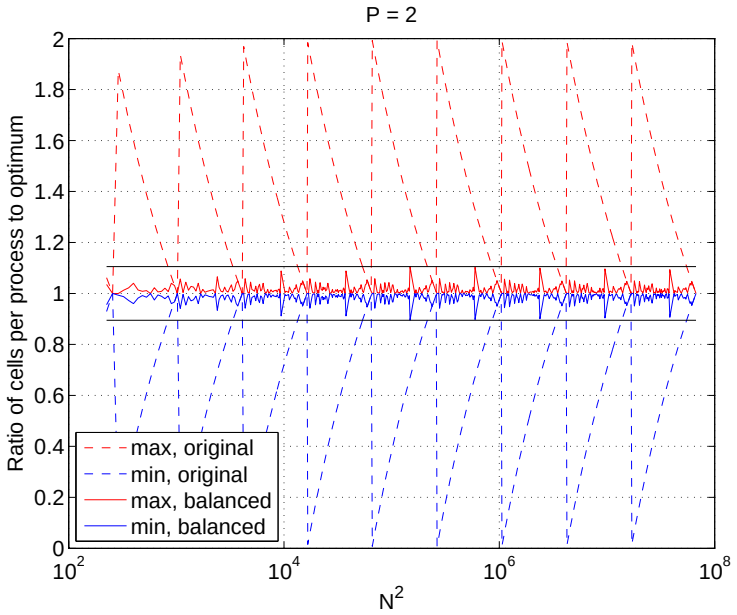


Figure 4: Least and largest amount of computational cells on a single process as a function of grid size. Dashed line corresponds to a naive partitioning and full lines optimized partitionings as described in section 2.2.1. Horizontal full black lines shows worst case for the optimized version.

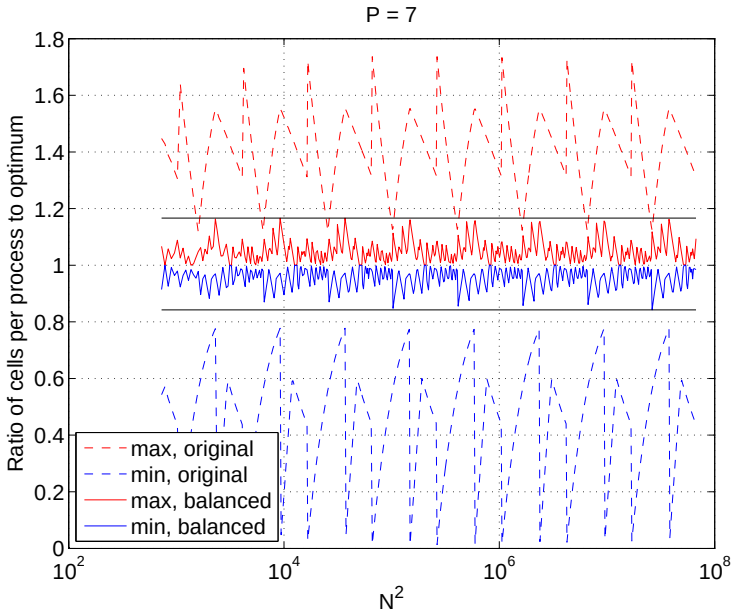


Figure 5: Same as fig. 4 but for 7 processes.

Figures 6 and 7 show the load balancing for the same two cases, but for the balancing scheme described in section 2.2.1 and that obtained by the Metis package. We see that the worst case is worse for the Metis package when it comes to number of computational cells, but this is a simplified view as the communication cost is not plotted.

Figure 8 shows the distribution of computational cells in a homogeneous cubic grid in three dimensions as a function of mesh size. Two partitionings are shown, first one obtained by the Metis package, and the other one is optimized as described above in section 2.2.1. The number of blocks available were at least 5 per process, *i.e.* $M \geq 5P$. We see that, as in the two dimensional example described above, the worst case is worse for the Metis package when it comes to number of computational cells assigned to a single process.

4.2 Paraver analysis

The program has been analyzed with Paraver which is a graphical program that displays traces of the communication pattern. Figure 9 shows a trace of a 1500×1500 grid parallelized on $P = 7$ processes, with load balancing as described in section 2.2.1. From left to right the program builds the grid, balances it according to the factor-of-2 rule, saves the grid to disk, performs the load balancing, and saves it to disk once more. The first three bursts of communication are part of the grid balancing, and the last communication is part of the load balancing. Figures 10 and 11 show the two balancing operations in some more detail. It is clear from the traces that the first and last processes have less load than the five middle ones.

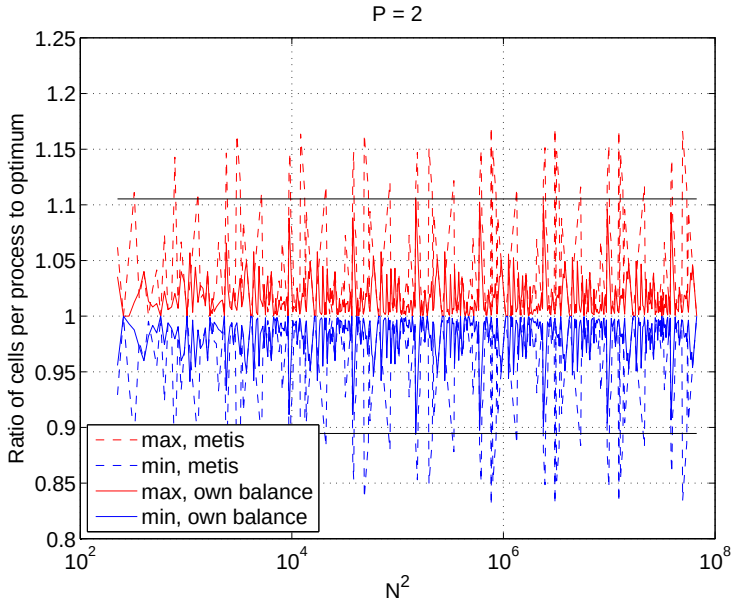


Figure 6: Least and largest amount of computational cells on a single process as a function of grid size. Dashed line corresponds to a partitioning obtained by Metis and full lines optimized partitionings as described in section 2.2.1. Horizontal full black lines shows worst case for the optimized version.

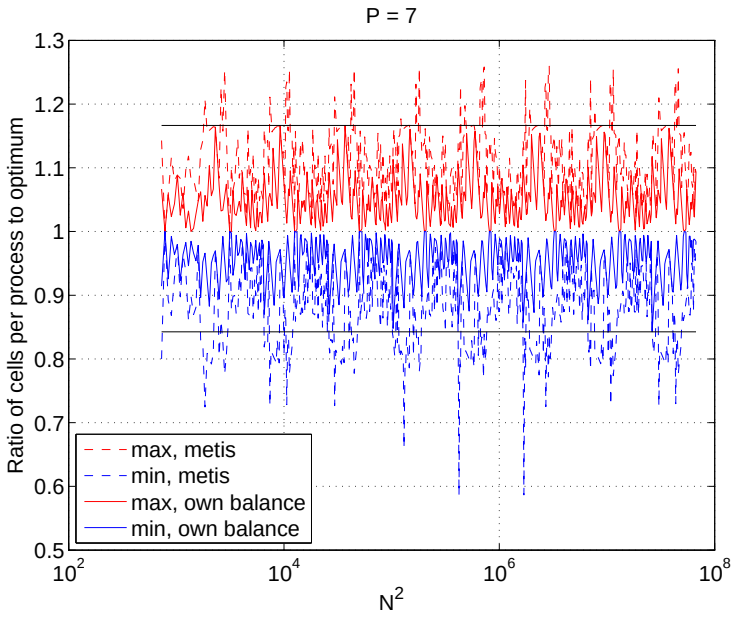
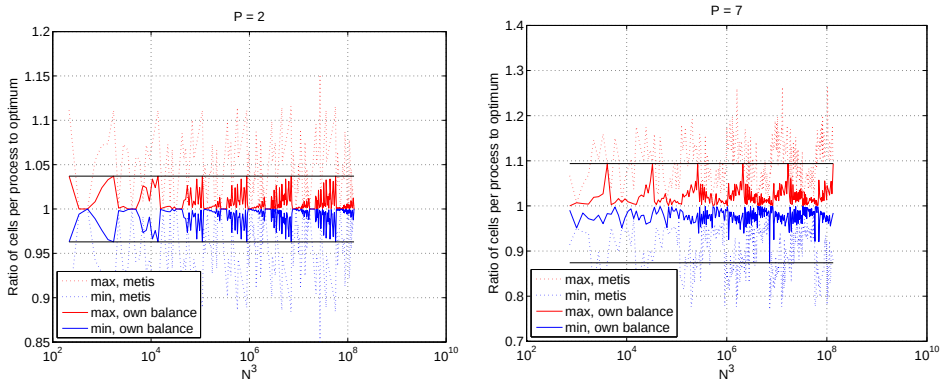


Figure 7: Same as fig. 6 but for 7 processes.



(a) $P = 2$ (b) $P = 7$

Figure 8: Least and largest amount of computational cells on a single process as a function of grid size on a three dimensional homogeneous grid. Dotted lines correspond to a partitioning obtained by Metis and full lines optimized partitionings as described in section 2.2.1. Horizontal full black lines shows worst case for the optimized version.

This can be seen in many ways, but perhaps most apparent during the disk output part where the first process completes about five times faster than the middle ones. By looking at the load balancing part in fig. 11 we see that the first process is sent four blocks from processes 2-5. The result is a much more even balance as apparent during the final store that completes the program. In this case the smallest process had 20.5% of the optimally load balanced number of cells before the balance operation and 96.2% after. The largest process had 122.3% before and 105.5% after.

Figure 12 shows the communication pattern when the load balance is performed by the Metis package. The grid balancing part is the same as above but if we look closer to the load balancing part, see fig. 13, we see that it is drastically different from the communication in fig. 11. The Metis load balancing results in massive amounts of communication, but it can be done in parallel as it is known in advance what the partition is. On the other hand, the load balancing described in section 2.2.1 creates a more even load balancing in terms of number of computational cells in just 4 block transfers. By comparing figs. 9 and 12 we see that the final store to disk is less balanced for the Metis package. The smallest process is allotted 81.6% of the optimally load balanced number of cells and the largest process had 116.6%.

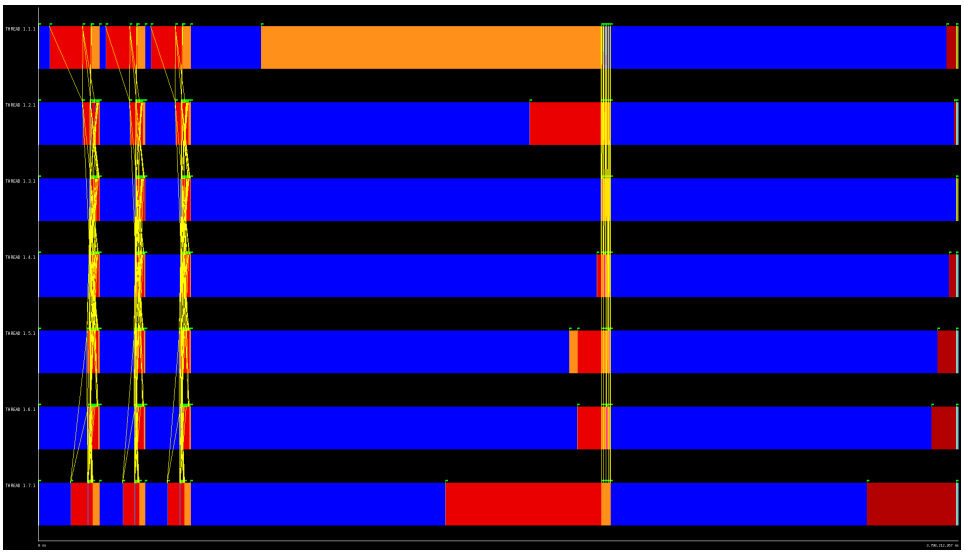


Figure 9: Trace of the program generating a 1500×1500 grid run on 7 processes. Blue indicates non-MPI work, red is `MPI_Wait()` of some kind, and orange is various collective operations. Load balancing as described in section 2.2.1.

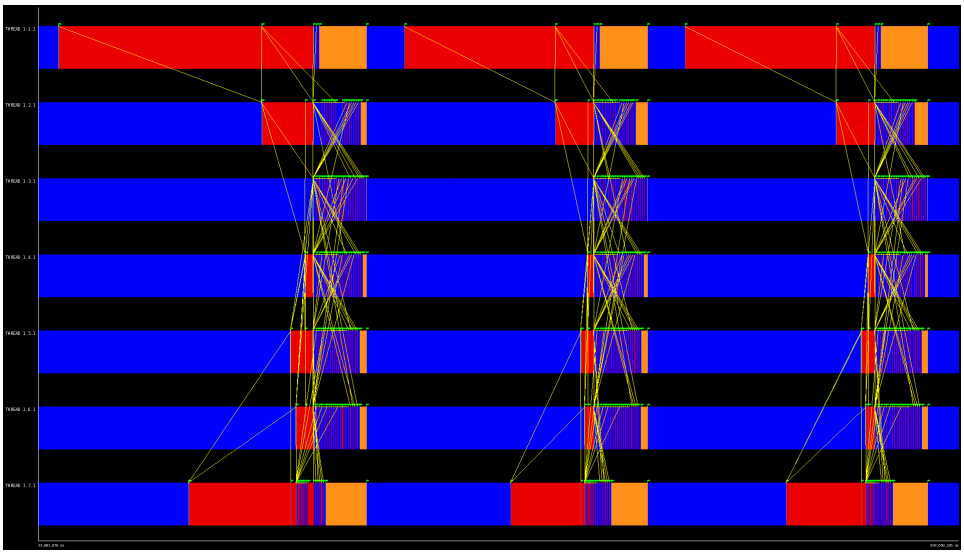


Figure 10: Zoom on the grid balancing part of fig. 9.

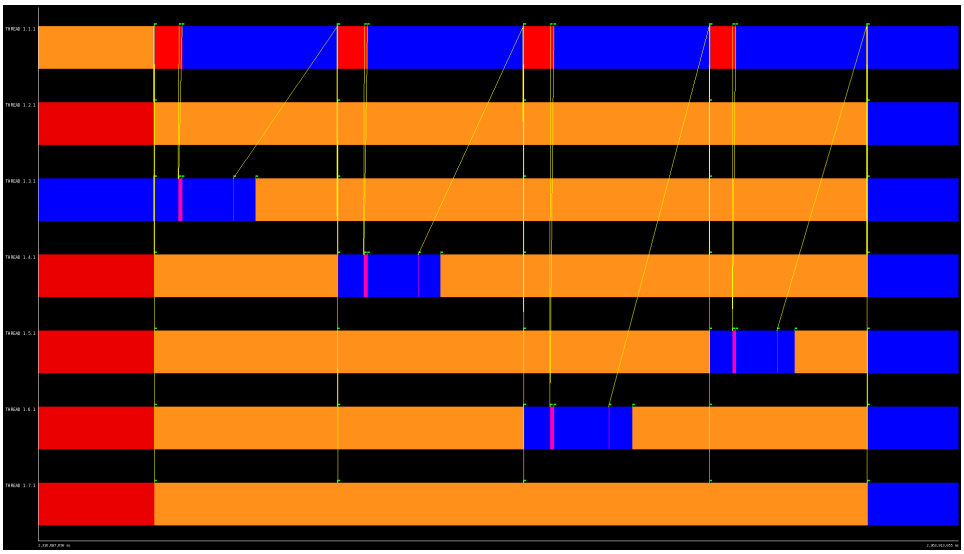


Figure 11: Zoom on the load balancing part of fig. 9.

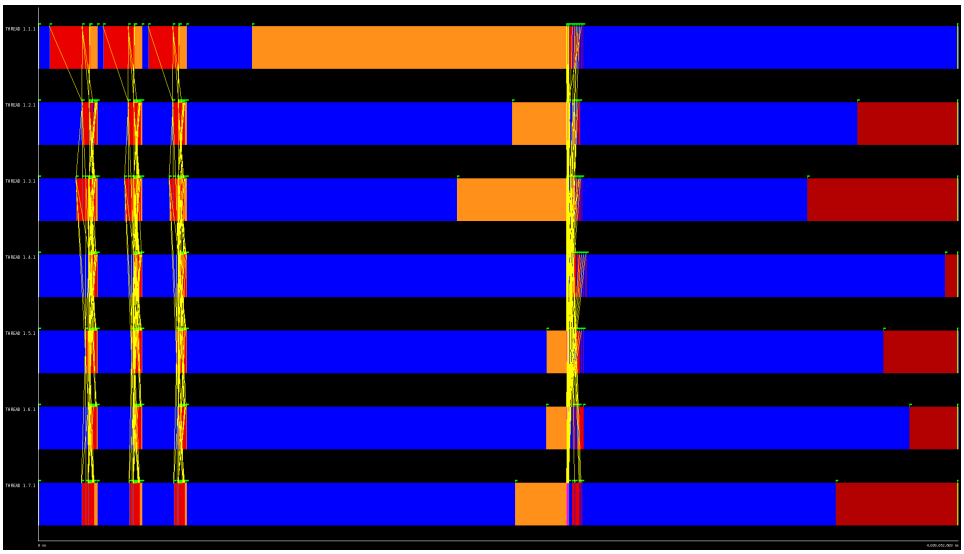


Figure 12: Trace of the program generating a 1500×1500 grid run on 7 processes. Blue indicates non-MPI work, red is `MPI_Wait()` of some kind, and orange is various collective operations. Load balancing by the Metis package.

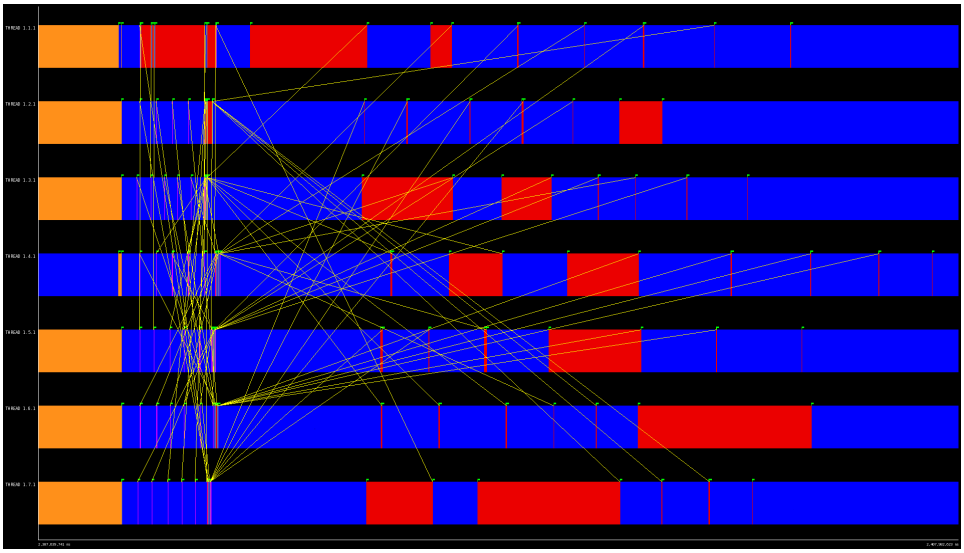


Figure 13: Zoom on the load balancing part of fig. 12.

Figure 14 shows the amount of parallelism as a function of time for the load balancing described in section 2.2.1, and fig. 15 shows parts where the program performs useful work and where it waits for communication.

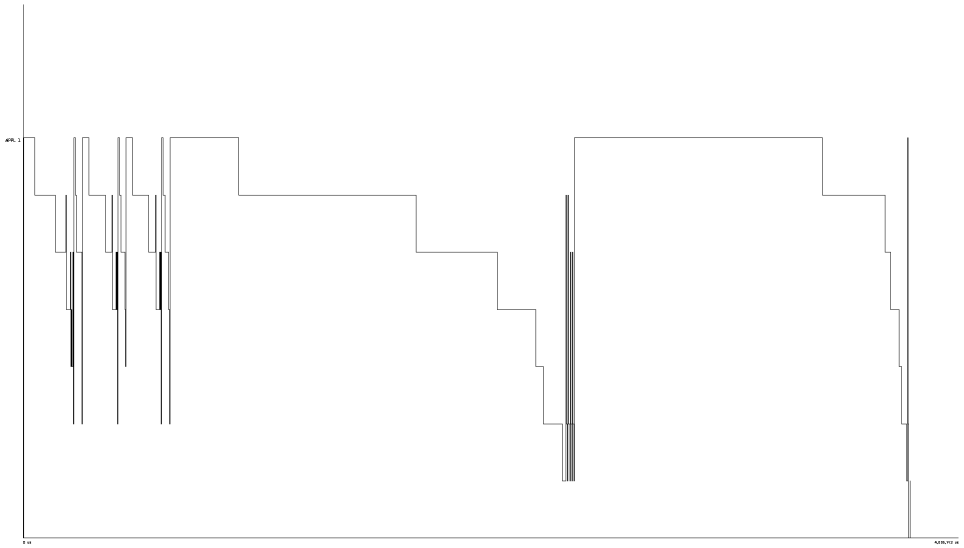


Figure 14: Number of running processes as a function of time. Load balancing as described in section 2.2.1.

4.3 Linear solvers

A very large fraction of the total simulation time is spent solving large sparse linear algebra systems. It is therefore important to have a solver that scales good, but it of course also has to be fast. Figure 16 shows the scaling for the GMRes solver of the Hypre [4] package and the algebraic multigrid solver AGMG [5, 6] for the momentum equation of a standard channel flow case. Load balancing is performed by the Metis package. We see that for both solvers the scaling is good. The Hypre solver even scales slightly super-linearly for 5 million computational cells, which can happen due to cache effects. However, if we look at the solution time in fig. 17, we see that Hypre performs really bad in terms of total solution time. Thus, regardless of how good it scales it will anyway be too slow for practical use.

The momentum equation is a parabolic one, that in this case is dominated by the convection term. This makes it relatively easy to solve in terms of how many iterations are required to reach the desired accuracy. The pressure equation on the other hand is elliptic and the solution is truly globally dependent on the right hand side vector, even for moderate solution accuracies. In other words more iterations are needed for convergence with a standard Krylov subspace solver without preconditioning. In all cases presented here the stopping criteria is a norm of the relative residual less than 10^{-6}.

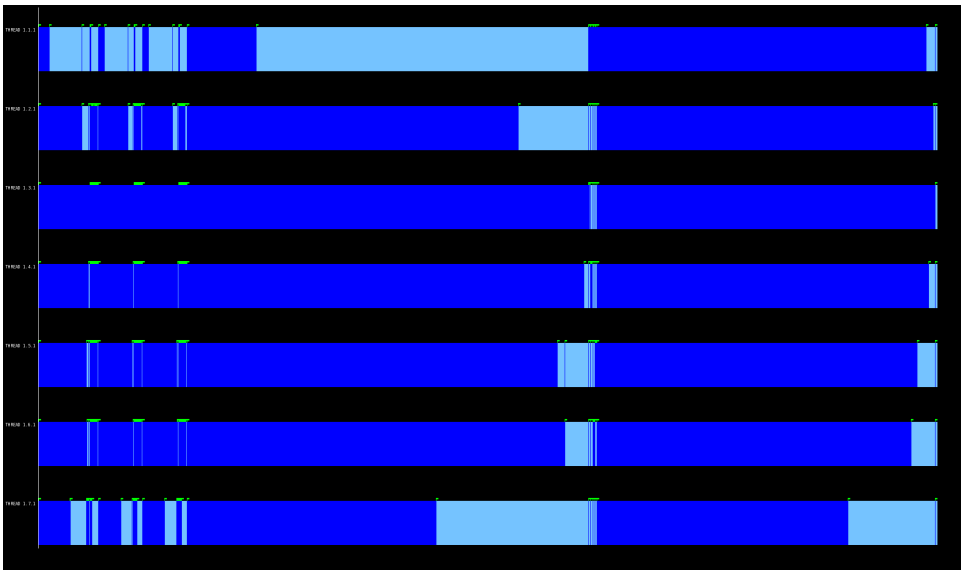
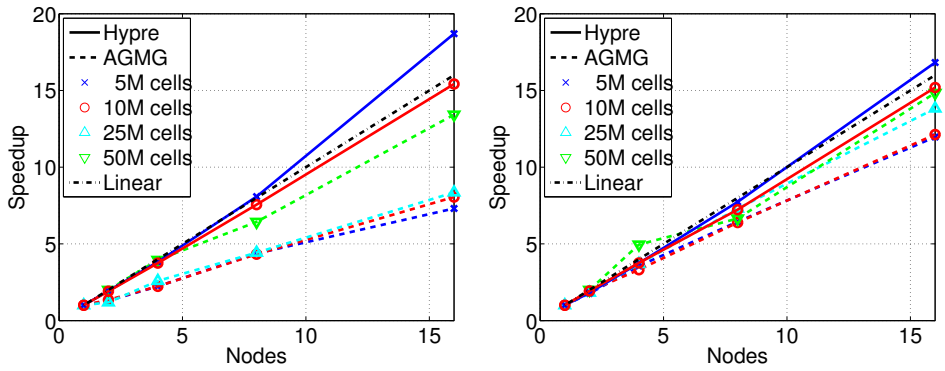
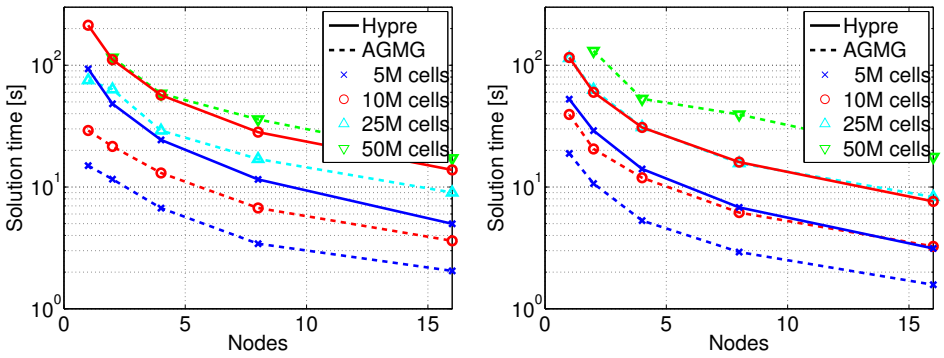


Figure 15: Useful (dark blue) and wait (light blue) time for each process. Load balancing as described in section 2.2.1.



(a) 2D (b) 3D

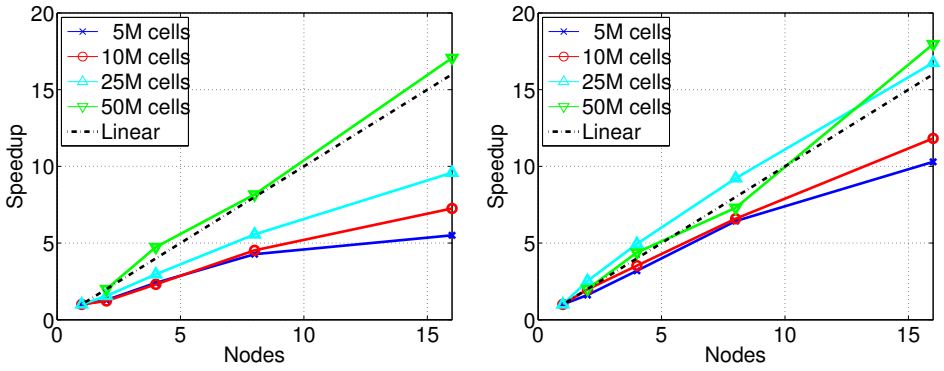
Figure 16: Solution time scaling with the number of computational nodes for the momentum equation. Four different sizes of the system of equations are shown with different marker types. Two solvers are shown, Hypre with full lines and AGMG with dashed lines. The linear scaling is shown as a dash-dotted black line without markers.



(a) 2D (b) 3D

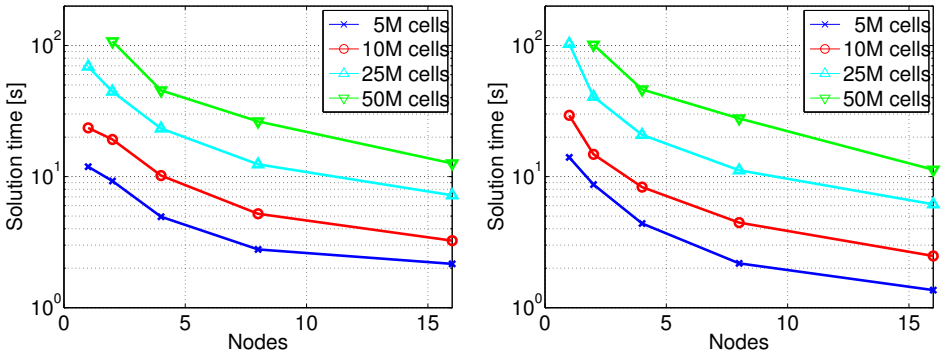
Figure 17: Total solution time as a function of the number of computational nodes for the momentum equation. Four different sizes of the system of equations are shown with different marker types. Two solvers are shown, Hypre with full lines and AGMG with dashed lines.

Figure 18 shows the scaling for the solution of the pressure equation with the AGMG solver. The Hypre GMRes solver is not presented as it is not a realistic example due to excessive solution time. Figure 19 shows the actual solution time. We see that the scaling is good in this case as well, but that, especially in 2D, a few million cells per node is required for optimal scaling. In 3D this is not an as strong requirement, even though large systems certainly scales better.



(a) 2D (b) 3D

Figure 18: Solution time scaling with the number of computational nodes for the pressure equation. Four different sizes of the system of equations are shown with different marker types. Results for the AGMG solver. The linear scaling is shown as a dash-dotted black line without markers.



(a) 2D (b) 3D

Figure 19: Total solution time as a function of the number of computational nodes for the pressure equation. Four different sizes of the system of equations are shown with different marker types. Results for the AGMG solver.

5 Conclusions

An existing CFD solver including its grid generator for Cartesian Quadtree and Octree meshes have been parallelized by means of MPI. Grid balancing as well as two versions of load balancing have been implemented. A cost function for the load balancing problem has been derived and the two solutions to it have been compared. The simplified version is shown to perform very well in terms of assigned number of computational cells. When solving the systems of linear equations it is however also important to minimize the cost of communication. The communication volume is proportional to the surface area of a process' assigned block of cells, and it is therefore important to assign neighboring cells to the same process as much as possible.

The Metis [1] and Parmetis [3] takes this into account and produces cell partitions suitable for solving the linear equations. The solvers AGMG [5] and Hypre [4] are demonstrated to scale very well with the number of processors, even though Hypre is a bit too slow for practical use. The AGMG solver is very fast on the other hand, giving a total solution time for both the momentum and pressure equations of less than 30 s for 50 million computational cells on 16 nodes on the C3SE cluster Beda.

Acknowledgment Part of the computations were performed on resources at Chalmers Centre for Computational Science and Engineering (C3SE) provided by the Swedish National Infrastructure for Computing (SNIC).

References

[1] `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`.

[2] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 1998.

[3] `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`.

[4] R. Falgout. Hypre linear algebra solver package. http://computation.llnl.gov/casc/hypre/software.html.

[5] Y. Notay. An aggregation-based algebraic multigrid method. Technical Report GANMN 08-02, Universite Libre de Bruxelles, Brussels, Belgium, 2008 (revised 2009). Available online at http://homepages.ulb.ac.be/∼ynotay/.

[6] Y. Notay. Agmg code and documentation. Available online at http://homepages.ulb.ac.be/∼ynotay/.