Comparison of Different Computer Platforms for Running the Versatile Advection Code

Gábor Tóth[1] and Rony Keppens[2]

[1] Department of Atomic Physics, Puskin u. 5-7, Budapest 1088, Hungary
E-mail: gtoth@hercules.elte.hu
[2] FOM Institute for Plasma-Physics Rijnhuizen, Postbus 1207, 3430 BE Nieuwegein, The Netherlands
E-mail: keppens@rijnh.nl

Presenting author: Gábor Tóth

E-mail: toth@hercules.elte.hu
Tel.: +36 1 2667024
Fax: +36 1 2660206

Comparison of Different Computer Platforms for Running the Versatile Advection Code

Gábor Tóth[1] and Rony Keppens[2]

[1] Department of Atomic Physics, Puskin u. 5-7, Budapest 1088, Hungary
E-mail: gtoth@hercules.elte.hu
[2] FOM Institute for Plasma-Physics Rijnhuizen, Postbus 1207, 3430 BE Nieuwegein, The Netherlands
E-mail: keppens@rijnh.nl

Abstract. The Versatile Advection Code is a general tool for solving hydrodynamical and magnetohydrodynamical problems arising in astrophysics. We compare the performance of the code on different computer platforms, including work stations and vector and parallel supercomputers. Good parallel scaling can be achieved with the data parallelism expressed in High Performance Fortran. With the aid of the automatic preprocessors, the same source code can be compiled with Fortran 77, Fortran 90, and High Performance Fortran compilers and the code can be run efficiently on computers ranging from a Linux PC to the Cray C90 and Cray T3E. Optimization for different machines and compilers is discussed.

1 Introduction

There is a strong need in the astrophysical community to use numerical simulations as a tool to investigate hydrodynamical and magnetohydrodynamical problems. The aim of the Versatile Advection Code (VAC) [4, 5] is to provide the community with a modern, versatile, and user-friendly software, which can be adapted to the various applications, and which runs efficiently on work stations, vector super computers, and parallel architectures as well.

In this paper, we compare the performance of VAC on different computer architectures, such as personal computers, work stations, vector super computers (namely Cray C90 and J90), and parallel super computers (namely IBM SP2, Cray T3D, and Cray T3E). In fact, the code can be run on any platform where a Fortran 77 (F77), Fortran 90 (F90), or, on parallel architectures, a High Performance Fortran (HPF) compiler is available.

In Sect. 2 the software design, the choice of the source language and the preprocessors are discussed. Section 3 briefly describes the equations that VAC can solve, the numerical methods used, and presents the test problem. The performance results are presented in Sect. 4 with discussion of optimization strategies. Conclusions are summarized in Sect. 5.

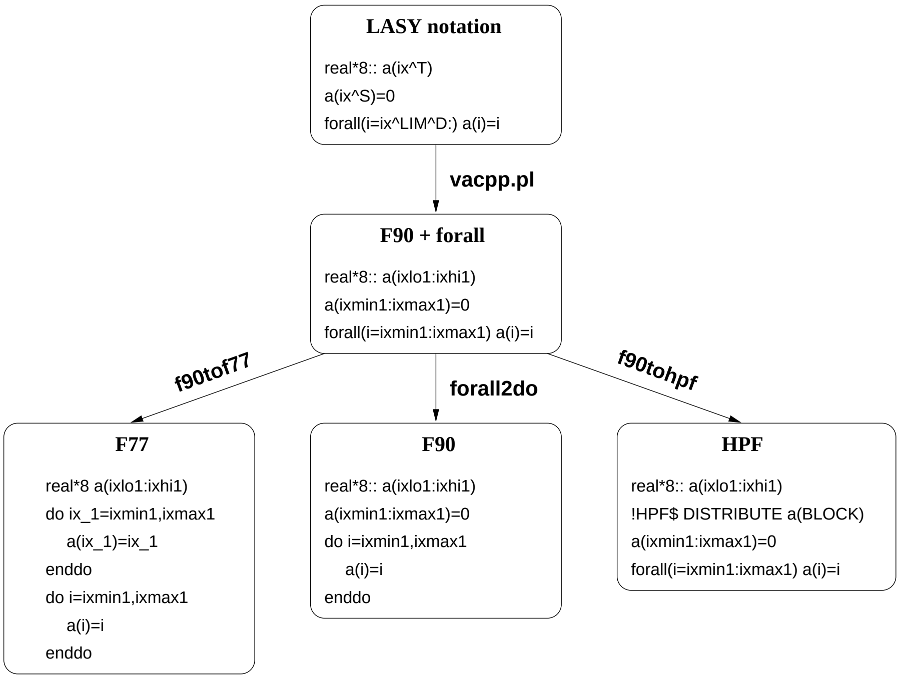

Fig. 1. Translation steps of the VAC source code. The names of the translator programs are written next to the arrows, the boxes contain simple examples translated for one spatial dimension.

2 Software Design

VAC has been developed since 1994 as a general purpose tool for hydrodynamic and magnetohydrodynamic astrophysical applications. VAC uses various shock capturing numerical methods [8], explicit, semi-implicit, or fully implicit time stepping [1, 3, 7] on 1, 2, or 3 dimensional finite volume grids. The various numerical schemes and equations are implemented in modules. The original Loop Annotation Syntax (LASY) [6] provides a dimension independent notation, thus the same source code can be configured and preprocessed for different geometries with the VAC preprocessor `vacpp.pl`. Further translation to F77, F90, and HPF is done by the Perl scripts `f90tof77`, `forall2do`, and `f90tohpf`, respectively, as it is schematically shown in Fig. 1. The software package is complete with manual pages written in hypertext, a user interface based on web browsers, and visualization macros for the most popular visualization softwares. The ever growing number of users and applications proves that the concept of a single well designed general purpose scientific software package is a good alternative to the typical specialized scientific codes.

From the beginning, VAC was designed to be able to run on parallel architectures. Although Fortran 90 and High Performance Fortran compilers were hardly existing when the project started, it was decided that data parallelism expressed directly in the source language would be used instead of the more usual domain decomposition with message passing approach. The concept was based on limited experience with Connection Machine Fortran (CMF), an early predecessor of HPF, and the availability of the ADAPTOR software [2] which could translate CMF to F77 with message passing, thus the code could run on machines without a CMF/HPF/F90 compiler. In practice, ADAPTOR was only used to translate the source code to F77 on a single node machine, which is now done by a much simpler `f90tof77` Perl script, while on parallel machines the code is compiled with the ever maturing HPF compilers.

After the dimension independent LASY notation is translated, VAC uses the following language elements of F90/HPF: data parallelism expressed by the array syntax, the `where` construct, the `forall` statement, and the scalar valued `sum, maxval, minval`, etc. functions. The programming style is improved with the `case` construct, F90 style variable declaration syntax, and free source format.

This choice of language elements was mainly dictated by the ADAPTOR software, but it also turns out to be a good compromise. The selected data parallel constructs are sufficient for the algorithms used in a fluid dynamics code, while other language elements, like dynamic allocation, array valued functions, pointers, structures, etc. are not used, since they would pose difficulties for translation to F77, for the users accustomed to F77, and for many of the not-so-mature HPF compilers. Simplicity helps the users to understand the source code, the compilers to produce an efficient executable code, and the program developers to easily spot and avoid possible compiler bugs.

Besides expressing data parallelism clearly, the array syntax also implies very simple loops for vectorized execution. Inside a `do` loop of a traditional F77 program there are lots of temporary scalars, function calls, conditional statements, etc., which do not vectorize well in general. In contrast, the array assignments and the `forall` statements are equivalent with a loop or nested loops with a single assignment inside, while the `where` construct is equivalent with a single `if-then-else` construct and several independent assignments inside the `do` loop(s). Writing such a well vectorizing code directly in Fortran 77 is possible, but neither intuitive, nor convenient.

When VAC is compiled with an HPF compiler, the distribution directives are inserted into the source code by the `f90tohpf` Perl script. The arrays to be distributed over the processors are recognized from their declaration in the source code. Automatic insertion has several advantages over writing down each directive manually. It is easy to forget about an array that should be distributed and to leave it replicated by accident. The resulting code will compile and run correctly, but the excessive communication will destroy the parallel scaling. Another advantage of the automatic insertion is that the distribution directives can easily be changed, e.g. from `(BLOCK,*)` to `(*,BLOCK)`, which is useful, since the optimal distribution is not the same for different grid geometries and/or computer platforms.

Currently, the iterative methods and the preconditioner used for implicit time stepping are implemented in F77, thus only the explicit part of the code can be compiled for parallel computers. We are currently rewriting these parts of the implicit module to the data parallel source language.

3 Equations and Test Problem

VAC contains general shock-capturing methods to solve conservation laws with additional source terms. The implemented equation modules can model hydrodynamics, adiabatic hydrodynamics which includes the shallow water equations, and isothermal and full magnetohydrodynamics (MHD). Source terms may describe, e.g. external gravitational field, thermal conduction, or viscosity.

The test problem used for performance comparison describes the motion of the water surface in a square shaped pool surrounded with reflective walls. We solve the shallow water equations

$$\frac{\partial h}{\partial t} + \nabla \cdot h\mathbf{v} = 0 \tag{1}$$

$$\frac{\partial h\mathbf{v}}{\partial t} + \nabla \cdot \mathbf{v}h\mathbf{v} + \nabla \frac{g}{2}h^2 = 0 \tag{2}$$

where $h, \mathbf{v}$, and g are the surface height, the velocity vector, and the vertical gravitational acceleration, respectively. The equations are discretized on a two-dimensional Cartesian grid with a Total Variation Diminishing method using an approximate Riemann solver. This numerical scheme is one of the most complex spatial discretizations implemented in VAC, so it represents the methods used in real applications. We use dimensional splitting. The one-dimensional stencil includes 5-points, thus the boundary conditions can be conveniently represented with 2 layers of ghost cells surrounding the computational domain.

In the test problem, a fairly modest 100×100 resolution is used, thus the total size of the computational grid is 104×104. There are 3 flow variables per grid cell: surface height and the two components of the momentum. The initial condition contains a peak with $h = 1.3$ in the middle four grid cells and $h = 1$ at other places. The velocity is initially zero everywhere. The initial condition and the results after 100 time steps are shown in Fig. 2.

In real applications the results are periodically dumped into data files, and this takes up some CPU time. We do not compare the I/O performance of different architectures in this paper, thus reading the input file, initialization, and saving results are excluded from the timing. We note, however that unformatted (binary) read and write operations are usually much faster than formatted (ASCII) input/output. In our experience, this difference is especially significant on parallel computers. Binary data format, on the other hand, may cause difficulties when the data files should be transferred between different machines for visualization purposes, for example. Even binary I/O becomes slow if a large number of processors are used on a parallel machine.

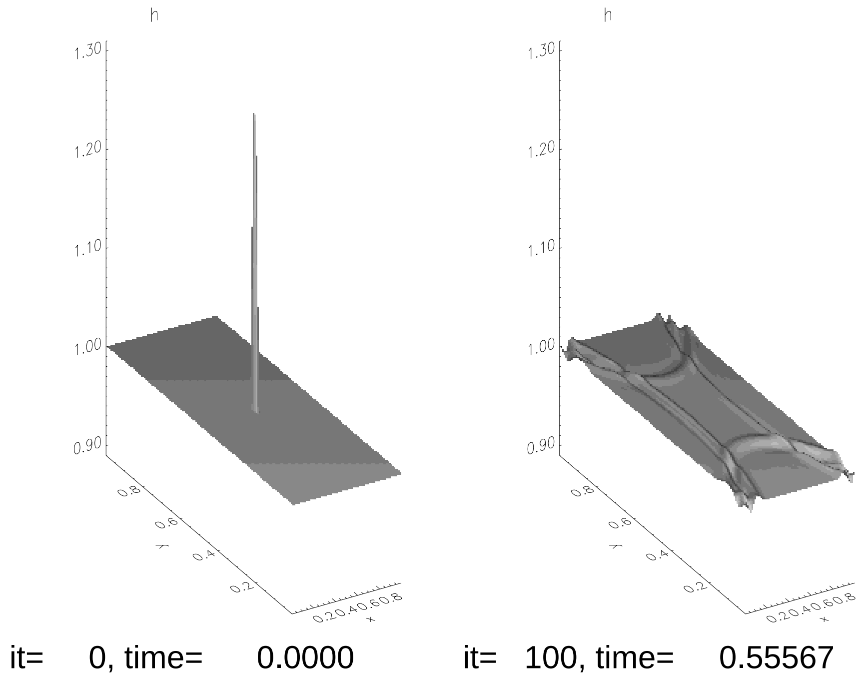

Fig. 2. Initial and evolved state of the water surface in the test problem. By time 0.55 the circular wave has already reflected from the walls.

4 Performance Results and Optimization Strategies

It is informative to compare the execution time of the same code and the same problem on different platforms. On each machine, VAC evolves the test problem for 50 time steps. We use the available optimization of the compilers, but we do not rewrite the code for the sake of a single architecture.

4.1 PC and work station

We start with a 166MHz Intel Pentium PC with a Linux operating system and a Gnu Fortran compiler `g77` 5.19.1 using `gcc` 2.7.2.1. The source code is translated with the `f90tof77` Perl script and compiled with the `-O2` flag. It takes 30.5 seconds to do the 50 time steps, while without optimization it takes 71 seconds. For this particular case at least, other optimization flags, like `-funroll-loops -m486`, seem to make little difference.

As a typical work station, we take a DEC Alpha/400 with the Digital `f90` 4.0-1 compiler. With the `-fast -transform_loops` and the default `-O4` compiler flags the test problem finishes in 31.5 seconds, while without optimization (`-O0`) in 118 seconds. It is surprising how well the Pentium PC performs in comparison with a workstation that was one of the fastest just a few years ago.

4.2 Vector Super Computers

We investigate two vector machines, a Cray C90 and a Cray J90. On the C90, we compile VAC with the Cray `f90` 2.0.3.4 compiler using `-O3` optimization. On a single node it takes a mere 1.35 seconds to finish the 50 time steps, and 3.4 seconds with no optimization. We checked that 100 steps take twice the CPU time. We also have the choice of using the `cf77` compiler after translation by `f90tof77`. With the `-O1` optimization we get essentially the same timings as for the optimized Fortran 90 compiled code, which shows that the two compilers are quite comparable despite the fact that the F90 compiler is much newer. We can also conclude that the automatic translation to F77 produces an efficient code.

On the Cray J90, we use the same `-O3` optimization for the `f90` compiler, and we measure 7.5 seconds for the execution time. This is much slower than the Cray C90, but much faster than typical work stations.

4.3 Parallel Computers

Fast inter-node communication is vital for parallel computers. For the parallel architectures examined here, there is a big difference between the speed of global communication between two arbitrary processors and local communication between neighboring processors. The HPF compilers usually recognize that assignments like `a(1:n)=b(0:n-1)+b(2:n+1)` require local communication only. Since the Fortran 90 source code of VAC contains these expressions in a more complicated form, like `a(imin:imax)=b(hmin:hmax)+b(jmin:jmax)`, it was necessary to have the option of translating the latter general index shifts to the previous simpler form. This is also achieved by an automatic translation and some comments helping the VAC preprocessor.

Another HPF related optimization is that only those arrays are distributed in which elements change their value during the time integration of the equations, like the flow variables. Static arrays, e.g. those describing the grid geometry, should be replicated (if memory permits), since they are initialized at the beginning of the calculation, and having them available on all the processors reduces the communication overhead during the rest of the calculation. In the source code, we simply used uppercase letters in the declaration of the arrays that should not be distributed by `f90tohpf`.

We also examine which dimensions of the arrays should be distributed. In two dimensions there are three options: slicing in the x direction, slicing in the y direction, or cutting the computational domain in both. We note that the number of grid cells along the x or y directions is 104, which is 8*13, thus the load balance is fairly good for 1, 2, 4, 8, and 13 processors. It is not perfect though, since the two layers of ghost cells at the edges of the grid require fewer operations than physical cells during the time step, and they require extra operations for boundary updates. Using 26 or more processors for such a small test problem is not very realistic, since I/O time becomes a large fraction of execution time.

While good scaling of execution time with the number of processors indicates good load balance and efficient communication, the single node performance is

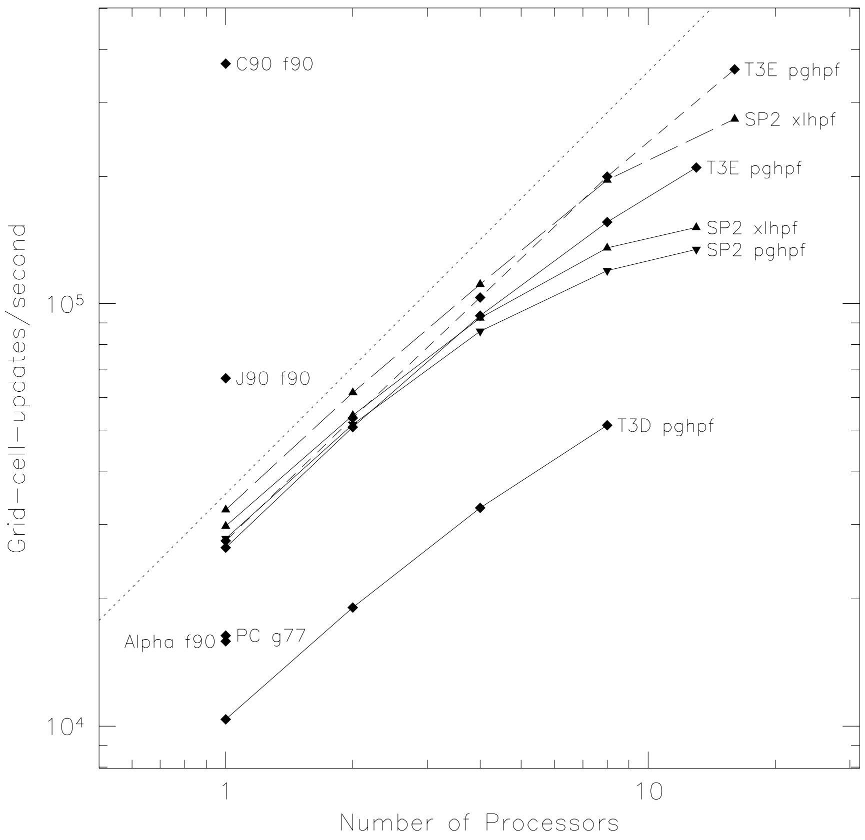

Fig. 3. Timing results for the 104 × 104 test problem. The number of physical grid cell updates per second is plotted against the number of processors for the different computer platforms and compilers as indicated by the labels. Execution times are measured with the best optimization on 1, 2, 4, 8, and 13 processors. The dotted line indicates the slope corresponding to the ideal linear speedup on the log-log plot. The dashed lines show measurements for a bigger 208 × 208 grid on 1, 2, 4, 8, and 16 processors. Note the improved scaling.

also vital. Comparison of the execution times of the F90 and HPF compiled codes on a single node, and comparison with a workstation having a similar CPU as the parallel machine are useful tests.

The first parallel machine examined is an **IBM SP2** with 76 nodes. Each node has an RS/6000 processor, which is comparable to a DEC Alpha/400. Indeed, compiled with the AIX F90 compiler `xlf90` with the -O2 flag, the test problem takes 25 seconds to finish. We tested two HPF compilers on the SP2, IBM's AIX `xlhpf90` and Portland Group's `pghpf` 2.2-2. The very first finding is that both HPF compilers produce a better single node performance than the F90 compiler. This is rather surprising, since the AIX compilers `xlf90` and `xlhpf90` presumably use the same kernels, while `pghpf` works as a preprocessor and it also uses the AIX Fortran kernel for the final compilation. The fact remains: VAC compiled with `xlhpf -O2` and `pghpf -O2` completes the test problem in 17 and 18 seconds, respectively. A possible explanation could be the difference in the default settings for the HPF and F90 compilers. Optimizing VAC for multiple-node execution, we find that distributing the 2nd dimension is slightly faster than the other two options. Parallel scaling results up to 13 processors are presented in Fig. 3.

We used the same version 2.2-2 of the `pghpf` compiler on a **Cray T3D** with 256 nodes as on the IBM SP2. Single node performance is rather poor for the T3D, with `-O3` optimization the execution time is 48 seconds, more than 50% worse than on the DEC Alpha/400, which has the same type of CPU. Probably it is the cache conflicts that are responsible for the bad single node optimization. Parallel scaling, as can be seen in Fig. 3, is quite good: it is closer to linear than for the IBM SP2. Note however that poor local optimization makes parallel speedup look good, since communication times become *relatively* shorter.

The last platform tested is a **Cray T3E** with 80 nodes. Each node has a DEC Alpha 21164 RISC processor. The single node performance is tested first with the Cray `f90` 3.0.0.1 compiler using `-O3` optimization. The execution time is about the same as for the IBM SP2, the T3E needs 18 seconds to do the test problem on one processor. Still, this is big improvement relative to the Cray T3D. For parallel execution, the `pghpf` 2.3-1 compiler with `-O3` optimization was used. The single node execution time is 19 seconds, which is about the same as for the `f90` compiler. The scaling results shown in Fig. 3 are the best among the parallel machines, probably due to the efficient communication. We find that distributing the second dimension is strongly preferred to distributing the first or both, especially as the number of processors increase. This preference may be due to better local optimization of loops over a fully local first dimension than over a distributed first dimension, and/or it may be due to faster transfer of array rows than of columns. Note that Fortran memory allocation puts array row elements, differing in their first index, next to each other.

5 Conclusions

In summary, the Versatile Advection Code fulfills its original design specifications by being versatile, portable, and efficient on modern computer platforms. Our experience with the VAC code shows that modern PC-s are competitive with work stations, and parallel machines are beginning to be competitive with traditional vector super computers. Fortran 90 and High Performance Fortran compilers have improved tremendously in recent years and now it is feasible to do large scientific simulations on parallel machines with HPF programs. We expect the situation to improve further in the future, especially in the following areas: the reliability of the HPF compilers, parallel I/O, and HPF libraries.

The test problem we selected is much smaller than a typical application that would be done on a super computer. We re-did the test problem with a 4 times bigger 208×208 grid to demonstrate the improved parallel scaling properties, which is shown by the dashed lines on Fig. 3.

More detailed and up-to-date information on VAC and the MPR project is available at http://www.fys.ruu.nl/~toth and http://www.fys.ruu.nl/~mpr.

Acknowledgments

The Versatile Advection Code was developed as part of the project on 'Parallel Computational Magneto-Fluid Dynamics', funded by the Dutch Science Foundation (NWO) Priority Program on Massively Parallel Computing, and coordinated by Prof. Dr. J.P. Goedbloed. Computer time on the Cray C90, Cray T3E, and IBM SP2 machines was sponsored by the Dutch Stichting Nationale Computerfaciliteiten (NCF). R.K. performed the simulations on the Cray T3D and J90 at the Edinburgh Parallel Computing Centre with support from the TRACS programme as part of his research program at FOM. G.T. thanks the Astronomical Institute at Utrecht for its hospitality during his work there, and acknowledges continued support from the 'Parallel Computational Magneto-Fluid Dynamics' project of NWO. G.T. is currently supported by a Hungarian Science Foundation (OTKA) postdoctoral fellowship and receives partial support from the OTKA grant F 017313. We thank Doug Miles of Portland Group for his helpful attitude to answer our questions regarding the `pghpf` compilers.

References

1. van der Ploeg, A., Keppens, R., Tóth, G.: Block Incomplete LU-preconditioners for Implicit Solution of Advection Dominated Problems, in *Proceedings of High Performance Computing and Networking Europe 1997*, Lecture Notes in Computer Science, **1225**, edited by B. Hertzberger and P. Sloot (Springer-Verlag, 1997), p. 421–430
2. Brandes, T.: ADAPTOR, Automatic DAta Parallelism translatOR, available via anonymous ftp from e.g. `ftp.gmd.de:gmd/adaptor`, (1994)
3. Keppens, R., Tóth, G., Botchev, M. A., van der Ploeg, A.: Implicit and Semi-Implicit Schemes in the Versatile Advection Code: algorithms, submitted for publication to the Int. J. Num. Meth. in Fluids (1997)
4. Tóth G.: Versatile Advection Code, in *Proceedings of High Performance Computing and Networking Europe 1997*, Lecture Notes in Computer Science, **1225**, edited by B. Hertzberger and P. Sloot (Springer-Verlag, 1997), p. 253–262
5. Tóth, G.: A general code for modeling MHD flows on parallel computers: Versatile Advection Code, Astrophys. Lett. & Comm. **34** (1996) 245–250
6. Tóth, G.: The LASY Preprocessor and its Application to General Multi-Dimensional Codes, J. Comput. Phys. **138** (1997)
7. Tóth, G., Keppens, R., Botchev, M. A.: Implicit and semi-implicit schemes in the Versatile Advection Code: numerical tests, accepted for publication in Astron. & Astroph. (1997)
8. Tóth, G., Odstrčil, D.: Comparison of some Flux Corrected Transport and Total Variation Diminishing Numerical Schemes for Hydrodynamic and Magnetohydrodynamic Problems. J. Comput. Phys. **128** (1996) 82–100