COVER SHEET

NOTE: This conversheet is intended for you to list your article title and author(s) name only—this page will not appear on the CD-ROM.

Title: Report on the development of a generic discontinuous Galerkin framework in .NET

Authors: Florian Kummer, Nehzat Emamy, Roozbeh Mousavi Belfeh Teymouri, Martin Oberlack

PAPER DEADLINE: ***SEPTEMBER 30, 2009****

PAPER LENGTH: ****16 PAGES (Maximum)****

ABSTRACT

In the past $1\frac{1}{2}$ years, the authors have been working on an object-oriented framework for the discontinuous Galerkin (spectral element) method, with a strong aim on CFD applications. This library was programmed in C# for Microsoft .NET and Mono framework. Up to our knowledge, it's the first ambitious CFD code which was implemented using the .NET framework.

This paper covers various topics:

A brief overview of features, library layout and current implementation status. Advantages of managed programming languages (in comparison to classical languages) and possible performance pitfalls in managed platforms are discussed. Benchmarks supporting these arguments are given and discussed.

Finally, a mathematical framework for parallel IO is presented.

DESIGN GOALS OF THE LIBRARY

Within the last 10 years, the discontinuous Galerkin method (DG) [3, 9, 8], sometimes also referred to as spectral element method, has been established a a very powerful method for solving hyperbolic conservation laws, especially in computational fluid dynamics (CFD) [5, 10], and is recently strongly emerging into incompressible CFD [4, 6].

As our institution is particulary active in combustion and turbulence modeling, modularity, expandability and scalability are central design goals for the library. Hence, the library is designed to model arbitrary systems of the form

$$c_\gamma \cdot \frac{\partial}{\partial t} u_\gamma + \operatorname{div}\left(\mathbf{f}_\gamma(\mathbf{x},U) + \mathbf{L}_\gamma(\mathbf{x})U + \nu(\mathbf{x}) \cdot \nabla G_\gamma U\right) + q_\gamma(\mathbf{x},U) + W_\gamma(\mathbf{x})U = 0, \quad (1)$$

with $\gamma = 1,\ldots,\Gamma$, $c_\gamma \in \{0,1\}$, where $U = (u_1,\ldots,u_\Gamma)^T$ denotes the vector of unknowns ("dependent variables"), $t \in \mathbb{R}_{>0}$ denotes time and $\mathbf{x} \in \Omega \subset \mathbb{R}^D$ is the spatial coordinate. Actually, the library supports spatial dimensions $D \in \{1,2,3\}$ with both, structured and unstructured grids. Each equation of the system can contain nonlinear fluxes $\mathbf{f}_\gamma$ and sources q_γ, as well as linear fluxes $\mathbf{L}_\gamma(\mathbf{x}) \in \mathbb{R}^{D\times\Gamma}$ and sources

F. Kummer, N. Emamy, R. Mousavi B.T., M. Oberlack, TU Darmstadt, Chair of Fluid Dynamics, Hochschulstraße 1, 64289 Darmstadt, Germany; kummer@fdy.tu-darmstadt.de

$W_\gamma(\mathbf{x}) \in \mathbb{R}^{1\times\Gamma}$. The term $v(\mathbf{x})\cdot\nabla G_\gamma U$, where $G_\gamma \in \mathbb{R}^{1\times\Gamma}$, is intended to discretize linear equations with derivatives of second order (e.g. the Poisson equation) directly, i.e. without breaking them up into a system of first order equations.

The user provides the components of the equation above as well as Riemann solvers for the flux components $\mathbf{f}_\gamma$, $\mathbf{L}_\gamma$ and G_γ by implementing interfaces defined by the library. For the linear components, the code is able to construct sparse matrices automatically from the analytical functions which are provided by the user.

In order to handle large scale problems on parallel machines, the code is MPI parallelized, and is capable of handling arbitrary repartitioning of the grid (load balancing). In addition, BoSSS features a (parallel) database system for storing, loading and *organizing* the data which is produced by the numerical methods. The design of the database allows to balance the IO load onto multiple file-servers, e.g. normal Network-Attached-Storage devices (NAS) and hence does not rely on parallel file systems. If IO is no bottleneck, all MPI processes can access one file server.

Preliminary Results

Navier-Stokes equations. Within the framework presented in equation 1, the momentum equations for an incompressible fluid flow in 2D can be written as

$$\begin{aligned}
\frac{\partial}{\partial t}u \quad &+\operatorname{div}\left(u\,\mathbf{u}+\mathbf{e}_1\,p+\frac{-1}{Re}\,\sigma_{1,-}\right)=0\\
&\operatorname{div}(\mathbf{e}_1\,u)-\sigma_{1,1}=0\\
&\operatorname{div}(\mathbf{e}_2\,u)-\sigma_{1,2}=0\\
\frac{\partial}{\partial t}v \quad &+\operatorname{div}\left(v\,\mathbf{u}+\mathbf{e}_2\,p+\frac{-1}{Re}\,\sigma_{2,-}\right)=0\\
&\operatorname{div}(\mathbf{e}_1\,v)-\sigma_{2,1}=0\\
&\operatorname{div}(\mathbf{e}_2\,v)-\sigma_{2,2}=0. \qquad (2)
\end{aligned}$$

Here, $\mathbf{u}=(u,v)^T$ denotes the velocity vector, p denotes pressure, $\mathbf{e}_i$ is the $i-$ th standard basis vector, *Re* the Reynolds number and $\sigma=\begin{pmatrix}\sigma_{1,-}\\ \sigma_{2,-}\end{pmatrix}=\begin{pmatrix}\sigma_{1,1} & \sigma_{1,2}\\ \sigma_{2,1} & \sigma_{2,2}\end{pmatrix}$ the viscous stress tensor.

Temporal discretization is performed employing the so-called projection method [18], which may be implemented as a Lie-splitting method [19]. For a timestep Δt, three steps are performed to compute an approximation to the solution $\mathbf{u}(\Delta t,-)$ of system 2 for an initial value $\mathbf{u}(0,-)$:

$$\mathbf{u}(0,-)\xrightarrow{\text{non. lin.}}\mathbf{u}^*\xrightarrow{\text{viscous}}\mathbf{u}^{**}\xrightarrow{\text{proj.}}\mathbf{u}^{***}\approx\mathbf{u}(\Delta t,-).$$

The spatial discretization by the discontinuous Galerkin method is standard and beyond the scope of this article.

Nonlinear terms. A 4^{th} - order explicit Runge-Kutta DG - method (see [9]) is used to compute a solution $\mathbf{u}(\Delta t,-) =: \mathbf{u}^* =: (u^*,v^*)^T$ to the system

$$\begin{aligned} \frac{\partial}{\partial t}u + \operatorname{div}(u\,\mathbf{u}) &= 0 \\ \frac{\partial}{\partial t}v + \operatorname{div}(v\,\mathbf{u}) &= 0 \end{aligned} \tag{3}$$

at time Δt for an initial value $\mathbf{u}(0,-)$. For the spatial discretization, local Lax-Friedrichs fluxes were used. Details can be found e.g. in [4].

Viscous terms. An Implicit Euler method is used to compute a solution $u(\Delta t,-) =: u^{**}$ to the system

$$\begin{aligned} \frac{\partial}{\partial t}u \quad + \quad \operatorname{div}\left(\tfrac{-1}{Re}\,\sigma_{1,-}\right) &= 0 \\ \operatorname{div}(\mathbf{e}_1\,u) - \sigma_{1,1} &= 0 \\ \operatorname{div}(\mathbf{e}_2\,u) - \sigma_{1,2} &= 0 \end{aligned} \tag{4}$$

at time Δt for an initial value $u(0,-) = u^*$. The spatial discretization is done by the so-called Local DG method, details can be found in [8]. In the same way, the viscous terms for velocity in y-direction are treated to get v^{**}.

Projection. Finally, $\mathbf{u}^{**}$ is projected onto the space of divergence-free vector fields. Therefore, the poisson equation

$$\operatorname{div}(\nabla\phi) = \operatorname{div}(\mathbf{u}^{**}) \tag{5}$$

is solved by means of the Interior Penalty method [3]. Knowing ϕ, the divergence-free part of $\mathbf{u}^{**}$, $\mathbf{u}^{***}$ can be computed by

$$\mathbf{u}^{***} = \mathbf{u}^{**} - \nabla\phi. \tag{6}$$

Benchmark results. Three classical 2D-benchmark problems were investigated. The results are given in figures 1, 2 and 3.

THE PARALLEL IO SYSTEM

Introduction: A notation for distributed storage of grid-related vectors

To describe the principles of the parallel IO system, some notation needs to be introduced; The numerical grid on the spatial domain $\Omega \subset \mathbb{R}^D$, can be written as a set of (pair-wise disjoint) cells $\mathfrak{C} \subset \{K;\ K \subseteq \Omega\}$, for which

$$\Omega = \bigcup_{K\in\mathfrak{C}} K, \tag{7}$$

holds. (The full definition of a "sane" numerical grid is beyond the scope of this article.) Furthermore, a bijective mapping

$$id : \{1,\cdots,J\} \to \mathfrak{C} \tag{8}$$

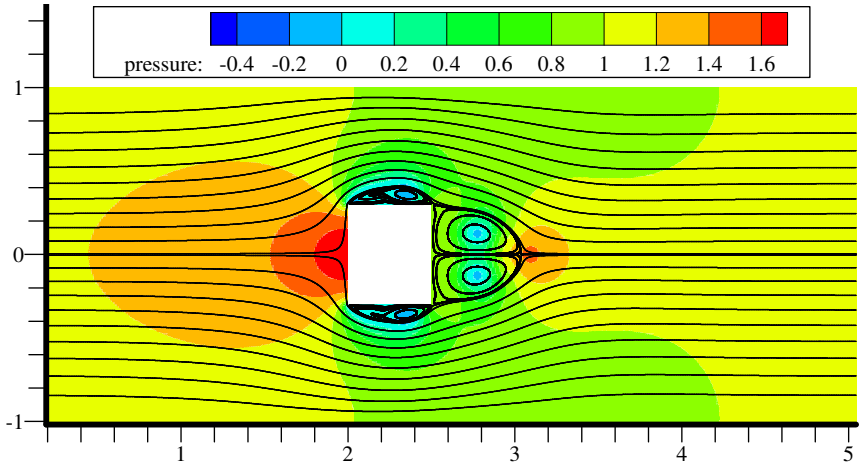

Figure 1. *Incompressible, unsteady Navier-Stokes, channel flow around obstacle; Parabolic profile at inlet:* $\boldsymbol{u}(x,y) = (1-y^2,0)^T$*; DG polynomial degree is 2; Grid of* 874×279 *cells, calculation domain is* $(x,y) \in (0,10) \times (-1,1)$*, Reynolds number 1000; Pressure distribution and streamtraces; Timestep No. 2331, Physical Time* $t = 1.289$*;*

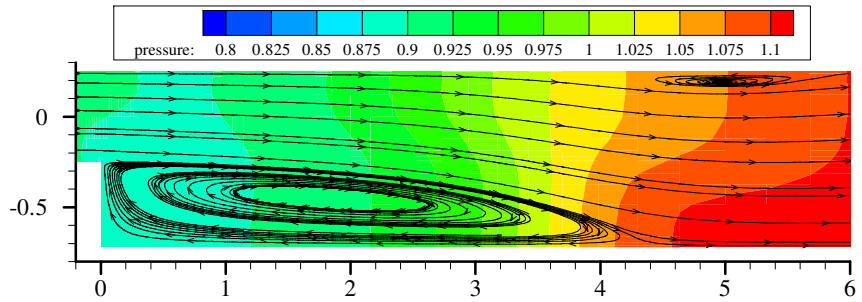

Figure 2. *Incompressible, unsteady Navier-Stokes, backward facing step; Parabolic profile at inlet:* $\boldsymbol{u}(x,y) = (1-(4\,y)^2,0)^T$*; DG polynomial degree is 2; Grid of* 2250 *cells, calculation domain is* $(x,y) \in (-2.5,0) \times (-0.25,0.25) \cup (0,100) \times (-0.75,0.25)$*, Reynolds number 500; Pressure distribution and streamtraces; Quasi steady-state – solution after* 100000 *timesteps;*

is assumed. The Number $id(K)$ is referred to as the identification, or simply "Id" of cell K. Independent form the numbering or indexing of the numerical grid during the numerical computation, $id(K)$ is a unique identifier for some cell K.

The set of s MPI processes is identified – or bijectively mapped – to the set $\{1,\cdots,s\}$. In contrast to the MPI convention, we denote the first process by number 1. From this, a grid partition,

$$P : \mathfrak{C} \to \{1,\cdots,s\}, \tag{9}$$

which assigns each cell to one MPI process, or simply process in below can be chosen. Knowing P, two helper variables,

$$N_p = \#\{K \in \mathfrak{C}; P(K) = p\}, \quad \text{and} \tag{10}$$

$$M_p = \sum_{q=1}^{p} N_q \tag{11}$$

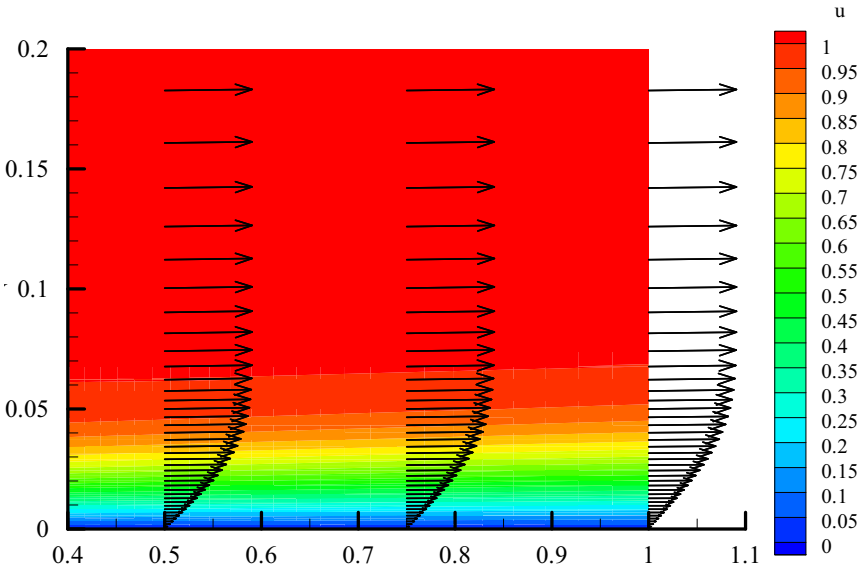

Figure 3. *Incompressible, unsteady Navier-Stokes, laminar boundary layer on a flat plate; Inlet: Blasius profile for velocity in x-direction computed at a distance of 1 meter from the leading edge; Grid of* 5000 *cells, calculation domain is* $(x,y) \in (0,1)^2$*, Reynolds number 10000; Velocity in x-direction;*

can be defined.

Beside the gird identification *id*, an additional grid numbering/indexing needs to be defined, which realizes an "proper arrangement" of the grid in memory. This, of course, depends on the numerical algorithm, the partition *P*, memory locality considerations,...)

Because $\mathfrak{C}$ is bijectively mapped to $\{1,\cdots,J\}$, this numbering can be written as some permutation $\tau \in S^J$, where S^J denotes the *symmetric group*, i.e. the group of all permutations of the set $\{1,\cdots,J\}$.

With objects and notation defined above, for any kind of data vector $\mathbf{v}$ which "lives" on the grid,

$$\begin{aligned} \mathbf{v}: \quad \mathfrak{C} \quad &\to M \\ id^{-1}(j) \quad &\mapsto v_j, \end{aligned} \tag{12}$$

a data distribution over the set of processes is given:

$$\begin{aligned}
\text{Process \#1}: \quad & (\tau(1),\cdots,\tau(N_1)) && =: \tau_{\text{part}}(1,P) \\
& \left(v_{\tau(1)},\cdots,v_{\tau(N_1)}\right) && =: \mathbf{v}_{\text{part}}(1,P,\tau) \\
\text{Process \#2}: \quad & (\tau(M_1+1),\cdots,\tau(N_2+M_1)) && =: \tau_{\text{part}}(2,P) \\
& \left(v_{\tau(M_1+1)},\cdots,v_{\tau(N_2+M_1)}\right) && =: \mathbf{v}_{\text{part}}(2,P,\tau) \\
& \vdots \\
\text{Process \#}s: \quad & (\tau(M_{s-1}+1),\cdots,\tau(N_s+M_{s-1})) && =: \tau_{\text{part}}(s,P) \\
& \left(v_{\tau(M_{s-1}+1)},\cdots,v_{\tau(N_s+M_{s-1})}\right) && =: \mathbf{v}_{\text{part}}(s,P,\tau).
\end{aligned}$$

M denotes an arbitrary set of data objects that are not further defined, e.g. sparse matrices stored in the popular MSR (modified sparse row, see [17]) format, where

each entry of $\mathbf{v}$ represents one row of a matrix. Note that τ itself is stored distributed. For further reference, we define

$$\mathbf{v} \circ \tau := \left(v_{\tau(1)}, \cdots, v_{\tau(J)}\right) = \left(\mathbf{v}_{\text{part}}(1, P, \tau) \mid \cdots \mid \mathbf{v}_{\text{part}}(s, P, \tau)\right). \tag{13}$$

Saving data

From the software-engineering point of view, it is necessary that the objects $\mathfrak{o} \in M$ which should be saved are serializeable, i.e. there type either has got the *Serializesable* - attribute or implements the *ISerializeable* interface. Additionally, it is assumed that two different objects v_j and v_k ($k \neq j$) do not reference each other in the sense of .NET object references.

In order to implement IO parallel, basically each process saves "it's own" data, so the IO load can be balanced among a number of fileservers. The problem is, that the data is permuted by τ, which may change, e.g. in case of a restart.

So, for storing $\mathbf{v}$ – in a way that makes it usable later on – there are two options: The first is to restore $\mathbf{v}$, which is not present in memory from $\mathbf{v} \circ \tau$ and then write $\mathbf{v}$ to the storage device. The disadvantage of this method is, that each saving may produce network traffic, which can be significant, depending on τ.

The second option, which is implemented in BoSSS, is to store $\mathbf{v} \circ \tau$ and τ itself so that the reconstruction of $\mathbf{v}$ is postponed, in some sense. Here, independent of the choice of τ, no interprocess - communication occurs, but the amount of data which must be saved is higher. Considering a typical solver run, in which multiple objects like $\mathbf{v}$ are saved, usually hundreds or thousands, τ remains constant or is only changed a few times. In this scenario, the data overhead is rather small, if τ is stored only once and than referenced, which is easy to implement.

So process p writes one file containing $\tau_{\text{part}}(1, P)$ and a second file which will be referred to as `data.`p`-of-`s, containing $\mathbf{v}_{\text{part}}(1, P, \tau)$ and some additional info to refer to the first file.

A container format

To load data vectors from the file `data.`p`-of-`s, it may be necessary to extract individual elements or sub-vectors from $\mathbf{v}_{\text{part}}(1, P, \tau)$. Therefore, if $\mathbf{v}_{\text{part}}(1, P, \tau)$ would be serialized as one big block, it would be necessary to de-serialize the entire vector $\mathbf{v}_{\text{part}}(1, P, \tau)$, what is IO- and compute - intensive.

Therefore, the data container frame shown in figure 4 is used. The elements of

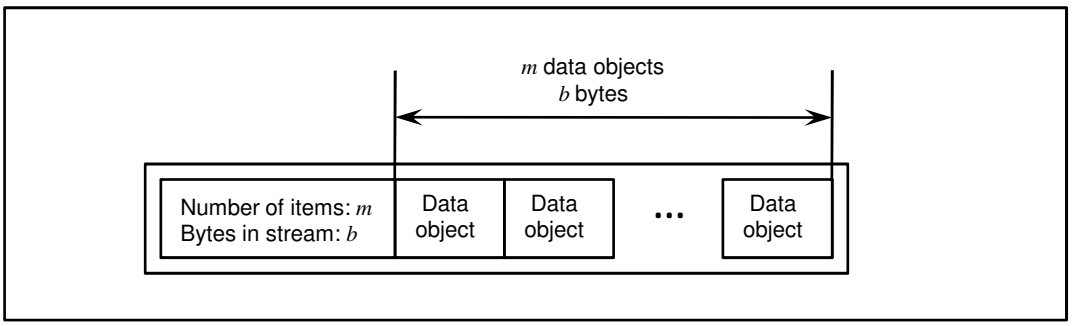

Figure 4. *The data container frame which is used in BoSSS to serialize vectors*

$\mathbf{v}_{\text{part}}(1,P,\tau)$ are serialized to a buffer in system memory until this buffer exceeds a certain threshold (typically a few kilobyte). Then, this buffer, together with a header (see figure 4) is streamed to the storage device.

This approach is a compromise between serializing every single vector entry and serializing the vector as one big block of binary data : The first one would result in a big overhead if the vector entries contain only few bytes, which makes the size of the header information not negligible in comparison to the size of the data items. The second one makes it expensive in terms of IO traffic and CPU load to extract one item).

Loading data

Assume that the vector, stored like described above, should be read from the storage device, with respect to a changed set of processes, $\{1,\cdots,\tilde{s}\}$ and therefore, a changed grid partition $\tilde{P}$ and cell numbering $\sigma \in S^J$. In other words, $\mathbf{v}\circ\sigma$ should be loaded, or, on each process $\tilde{p} \in \{1,\cdots,\tilde{s}\}$, the permuted sub-vector $\mathbf{v}_{\text{part}}(\tilde{p},\tilde{P},\sigma)$.

Again, there are two options: The first is to load $\mathbf{v}_{\text{part}}(\tilde{p},\tilde{P},\sigma)$ directly, which would require – in the worst case – that each process $\tilde{p}$ needs to access all files `data.`p`-of-`s (with $1 \le p \le s$)), which is, obviously, a bad choice.

The second option is to first de-serialize the vector in the original permutation τ, i.e. to load $\mathbf{v}_{\text{part}}(\tilde{p},\tilde{P},\tau)$ and then redistribute it to get $\mathbf{v}\circ\sigma$. Hence, this option also requires to load τ – additionally to the allready present σ – hence, on each process, the following data is present:

$$
\begin{aligned}
\text{Process \#1 :} \quad & \big(\tau(1),\cdots,\tau(\tilde{N}_1)\big) && = \tau_{\text{part}}(1,\tilde{P}) \\
& \big(\sigma(1),\cdots,\sigma(\tilde{N}_1)\big) && = \sigma_{\text{part}}(1,\tilde{P}) \\
& \Big(v_{\tau(1)},\cdots,v_{\tau(\tilde{N}_1)}\Big) && = \mathbf{v}_{\text{part}}(1,\tilde{P},\tau) \\
\text{Process \#2 :} \quad & \big(\tau(\tilde{M}_1+1),\cdots,\tau(\tilde{N}_2+\tilde{M}_1)\big) && = \tau_{\text{part}}(2,\tilde{P}) \\
& \big(\sigma(\tilde{M}_1+1),\cdots,\sigma(\tilde{N}_2+\tilde{M}_1)\big) && = \sigma_{\text{part}}(2,\tilde{P}) \\
& \Big(v_{\tau(\tilde{M}_1+1)},\cdots,v_{\tau(\tilde{N}_2+\tilde{M}_1)}\Big) && = \mathbf{v}_{\text{part}}(2,\tilde{P},\tau) \\
& \vdots \\
\text{Process \#}\tilde{s}\text{ :} \quad & \big(\tau(\tilde{M}_{\tilde{s}-1}+1),\cdots,\tau(\tilde{N}_{\tilde{s}}+\tilde{M}_{\tilde{s}-1})\big) && = \tau_{\text{part}}(\tilde{s},\tilde{P}) \\
& \big(\sigma(\tilde{M}_{\tilde{s}-1}+1),\cdots,\sigma(\tilde{N}_{\tilde{s}}+\tilde{M}_{\tilde{s}-1})\big) && = \sigma_{\text{part}}(\tilde{s},\tilde{P}) \\
& \Big(v_{\tau(\tilde{M}_{\tilde{s}-1}+1)},\cdots,v_{\tau(\tilde{N}_{\tilde{s}}+\tilde{M}_{\tilde{s}-1})}\Big) && = \mathbf{v}_{\text{part}}(\tilde{s},\tilde{P},\tau)
\end{aligned}
$$

The desired redistribution (to get $\mathbf{v}\circ\sigma$) is given by the permutation

$$\varepsilon := \sigma^{-1}\circ\tau, \tag{14}$$

i.e.

$$\mathbf{v}\circ\sigma = (\mathbf{v}\circ\tau)\circ\varepsilon. \tag{15}$$

This is obvious to verify: By using the notation $\mathbf{v}\circ\tau =: (V_1,\cdots,V_J)$ and $\mathbf{v}\circ\sigma =: (W_1,\cdots,W_J)$ it follows that $W_{\varepsilon(i)} = V_i$, because $W_j = v_{\sigma(j)} = v_{\tau(\tau^{-1}(\sigma(j)))} = v_{\tau(\varepsilon^{-1}(j))} = V_{\varepsilon^{-1}(j)}$.

In other words, this means, that, on process $\#\tilde{p}$, we have to copy the $(i - \tilde{M}_{\tilde{p}-1})$ – th entry of $\mathbf{v}_{\text{part}}(\tilde{p}, \tau, \tilde{P})$ to the $\varepsilon(i)$ – th entry of the target vector to get the desired permutation $\mathbf{v} \circ \sigma$.

The coping can be done locally, i.e. without MPI communication, if $\tilde{M}_{\tilde{p}-1} < \varepsilon(i) \leq \tilde{M}_{\tilde{p}}$, otherwise interprocess-communication is involved.

So, two "core" algorithms of the IO system are the parallel inversion and composition of permutations that are distributed among several MPI processes. These algorithms can be programmed in parallel, the amount of communication depends on the choice of the permutation.

The same theory also applies when the grid should be re-partitioned at runtime.

ADVANTAGES OF MANAGED PLATFORMS

The major difference between so-called *managed languages* like Java or C# and "classical" programming languages is that the first ones are compiled to use a runtime Environment (the .NET framework for C# or the Java runtime) which does not only deliver an extensive runtime library but also runtime services like *garbage collection*, *runtime-type-information* and *serialization*. This is in clear contrast to classical languages like FORTRAN or C, where such services are – at least – very hard to implement (if possible at all).

A second advantage gained from using managed languages plays a role in software development: Software projects which are developed at Universities or similar institutions usually suffer from the fact that manpower is the most limited resource. It is difficult to argue about that, but we think that the "functionality - per - line of code" measure of BoSSS is considerably high (see table I). During the development of BoSSS we have found that the usage of a managed language together with a comprehensive toolchain (an Integrated Development Environment, or IDE) is considerably improving our workflow compared to the "classical" development environment (text editor, shell, makefile): IDE's free their user from the necessity of writing and debugging makefiles, ship with a powerful integrated debugger, and generally offer a broad variety of "little helpers" (like code complementation, tools for refactoring, ...) for every-day coding .

TABLE I. CODE STATISTICS

Layer	Description	No. of files	Lines of code	Lines of comment
L4 (Application)	Navier-Stokes – solver	10	740	465
L3 (Solution)	time discretization, application hints	29	3468	394
L2 (Foundation)	DG discretization, grid handling, quadrature, IO	69	10140	5627
L1 (platform)	wrappers, utilities	12	1118	1081
L0 (native)	3rd party: MPI, BLAS, LAPACK, Hypre, ParMETIS			

A third issue which we found very helpful with BoSSS is the fact that .NET - programs are platform independent on a binary level. This means that once the .NET runtime (on Unix systems the compatible Mono runtime [14]) and some native libraries, e.g. BLAS, MPI, LAPACK, HYPRE [13] are installed on a cluster or

supercomputer, the binaries from the development system, usually a local workstation, can be directly copied and executed on the supercomputer, without the need of recompiling the code.

Further advantages are gained by the use of some essential technologies in .NET, which cannot be found in classical languages: Serialization is the process of packing an arbitrary graph of objects into an continuous block of memory, which can be streamed either to hard disk or via MPI to another process. As with multi-purpose numerical codes the necessary data structures may become complex, serialization proofed rather helpful for File IO, parallelization and general interprocess communication.

A wide range of programming errors, e.g. buffer overflow, which usually occur during FORTRAN of C development, are prevented by the design of the .NET runtime. Memory leaks like in C, where allocated heap memory is forgotten to be de-allocated, cannot occur in .NET. A so-called *garbage collector* keeps track of all heap objects, and de-allocates them if they are not referenced by the application anymore and if memory is needed. Furthermore, the garbage collector is also capable of compacting the heap (see also figure 6).

At this point, portability of .NET Programs to an arbitrary computer is a somewhat controversial issue on non- MS-Windows – platforms. There the portability depends on the availability of the Mono framework. In this context, we consider the Portable.NET - project [15], in terms of speed, stability, completeness of implementation and compatibility to other .NET - implementations being not ready for production yet. In the present situation, Mono is generally available for Linux on x86, EM64T, AMD64 and IA-64 systems, which currently covers more than 88% of the Top 500 systems (June 2009 edition, [16]). Here, we assume that most of these systems run Linux, although [16] gives no information on that. Beside a low number (below 1%) of other architectures, a rather big share (11 %) of the Top500 – systems are based on the IBM Power architecture. Again we have no detailed information on that, but we assume that the majority of these systems are either BlueGene- or AIX - systems. For both of them, the principal answer on the question wether Mono compiles should be yes, but as long as this is not officially supported, we would expect a considerable high workload to do the port.

MANAGED PLATFORMS AND PERFORMANCE

It is a popular fallacy that C# - or Java - is not suitable for High Performance Computing, because it is an interpreted language. This is not the case and, in fact, not the experience with BoSSS. The output of the C#-compiler is so-called *Common Intermediate Language* (CLI), rather similar in form to an assembler program plus metadata to represent object oriented structures. At load time, this CLI code is transformed to native-machine code.

Since the first JIT-versions of Java have been available, various authors have made performance comparisons between C and Java, e.g. [7] and [2], just to mention two of them. These investigations concluded that Java, when properly used, is suitable for high performance computing. When .NET was released, it's performance was and is continuously compared to Java (e.g. [1]). C#, in contrast to Java, further has the option to declare specific regions of the code as *unsafe*. Within these sections, runtime security features, such as array bounds - checking, are turned of, and pointer arithmetics as in C is supported.

Up to our knowledge, the most comprehensive C# benchmarks on scientific algorithms, like LINPACK and SciMark, were done in [11]. In order to verify this, some benchmarks on basic algorithms (naive DGEMM and the Sieve of Eratosthenes) were performed and the results are shown in table II.

TABLE II. RUNTIME OF (NAIVLY CODED) DGEMM AND THE "SIEVE OF ERATOSTHENES" IN MANAGED AND CLASSICAL LANGUAGES (SYSTEM: PENTIUM 4 (PRESCOTT), 3GHZ, WINDOWS XP SERVICE PACK 3).

	DGEMM, $N = 1000$	Sieve of Eratosthenes, $p < 2 \cdot 10^7$
C#, Debug, .NET 3.5	29.8 sec	39.6 sec
C#, Release, .NET 3.5	16.7 sec	37.6 sec
C#, Release, Mono 2.2	34.9 sec	38.9 sec
C#, unsafe code, .NET 3.5	9.8 sec	37.7 sec
C#, unsafe code, Mono 2.2	12.4 sec	39.5 sec
C, MinGW-gcc 3.4.5	16.8 sec	30.3 sec
C, MinGW-gcc 3.4.5 -O3	9.8 sec	20.9 sec
C, MS-cl 15.00.21022.08	15.5 sec	26.8 sec
C, MS-cl 15.00.21022.08 /O2	9.7 sec	37.6 sec
ACML-BLAS 4.1.0	1.28 sec	n.a.

From our own test and benchmarks done by others we reason that...

- Optimized binary libraries from hardware vendors are usually superior to naive implementations build by any compiler.
- LINPACK and SciMark show that classical languages are still faster, but not by much - especially for the LINPACK case (see [11]).
- The performance loss of using a managed language in the worst cases is about 50%. Whether this is really observed in "real-world applications" is uncertain.
- Overall, Java and .NET provide equal performances. However, on specific algorithms, there may be a considerable gap.
- It is difficult to find proper benchmarks, because more complex programs cannot be translated one-by-one from C# to C or vice versa.
- Unsafe code performs very often head-to-head with state-of-the-art C compilers.
- Especially Mono seems to profit a lot from using unsafe code (see figure 5).
- Although, by using .NET we may expect a performance loss in the range of 20 to 40%, in comparison to classical programming techniques, the advantages in development outbalance the performance loss.
- Data, on which performance-critical algorithms work on, should be organized in arrays of value types. Complex graphs of heap objects should be avoided.
- Performance-critical sections should be "vectorized", i.e. implemented as multiple instruction-multiple data. Loops should be preferred to multiple function calls, i.e. the stack should be kept shallow. This opens the possibility to optimize the code with unsafe sections.

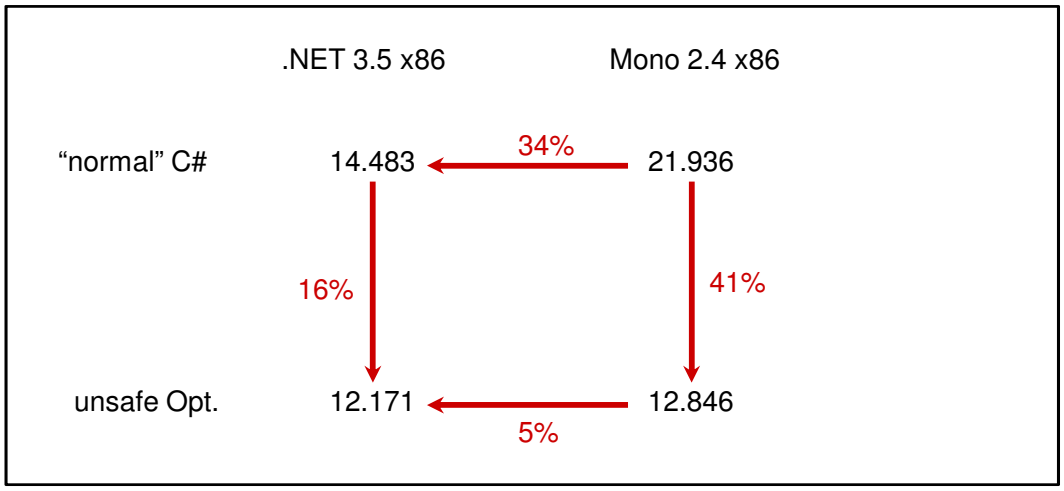

Figure 5. *Performance relation between .NET, Mono, "normal" and "unsafe" code for (naively coded) DGEMM (runtime in seconds).*

ON THE USAGE OF EXTERNAL LIBRARIES

From the experience during the development of BoSSS, there are three reasons for using native code in .NET applications for supercomputers: First, the need for near-system libraries like MPI, which are not available as native .NET libraries. Second, the use of complex mathematical libraries like sparse solvers ("legacy code"), which would produce a large overhead to re-implement them in C#. And third, the need for performance. As shown above (see table II), a DGEMM operation from an optimized library is superior by nearly a magnitude of 10 to non-optimized ("naive") implementations in *any* programming language.

P/Invoke

The standard way of using native code within C# is called *P/Invoke*. Native code is compiled/linked into a shared library (.dll - files in Windows, .so - file on Unix) and within the C#-code, a function prototype together with an attribute that points to the shared library, is defined. P/Invoke *marshalles* the managed objects to the unmanaged code. The most important thing is to prevent the garbage collector (which may run at any time in it's own thread) from moving the base address (as illustrated in figure 6) of some object while an unmanaged function accesses it. This "looking" of objects for the garbage collector is referred to as *pinning*. For standard function calls, which take some input data and process some output, pinning is performed automatically by P/Invoke.

Besides that, there are certain function calls which require that pinning continues for some time after they return - e.g. nonblocking MPI calls like MPI_Irecv. In .NET, it is relatively easy to control pinning manually (which is not possible in Java), by means within the *System.Runtime.InteropServices* – namespace. It is illustrated in figure 7, that the *GCHandle.Alloc(...)* – method can be used to acquire a lock on an object, while methods of the *Marshall* – class can be used to get the base address of the pinned array in memory. The *Marshall* – class can further be used to deal with string conversion (ANSI to Unicode and vice-versa), to allocate unmanaged memory,

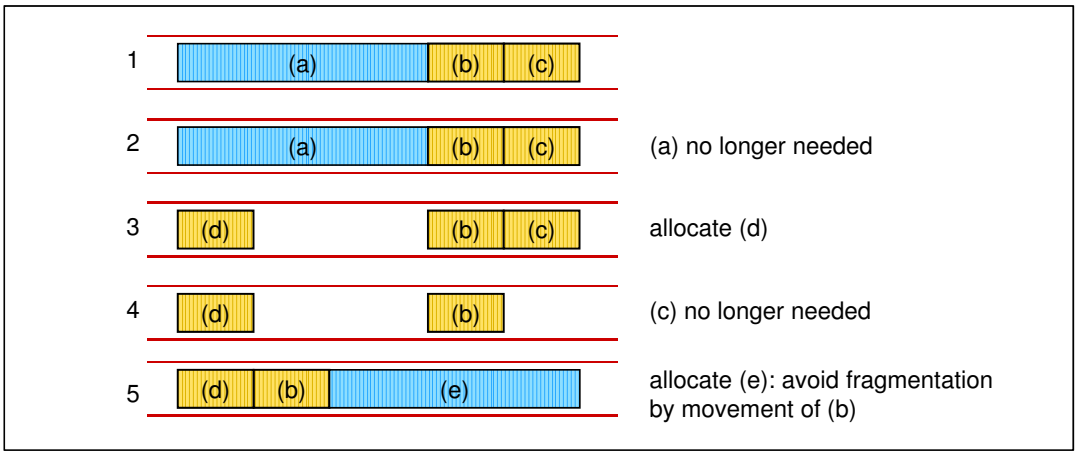

Figure 6. *memory reallocation by garbage collector (source: [14]).*

just to mention a few features.

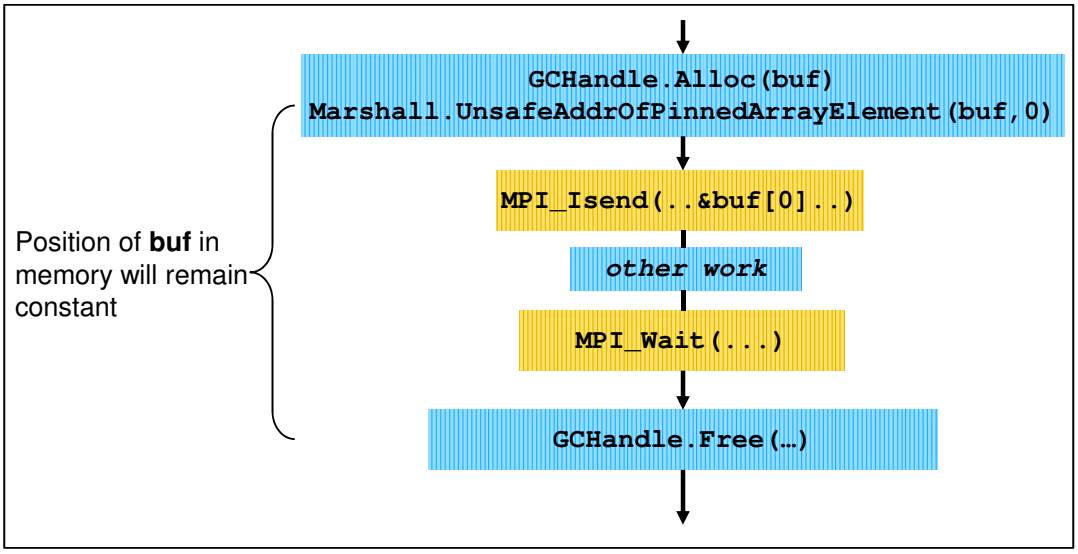

Figure 7. *manual pinning by GCHandle.Alloc(...) and GCHandle.Free(...).*

PARALLEL PERFORMANCE AND SCALING

A simple hyperbolic scalar convection,

$$\frac{\partial}{\partial t}c + \mathrm{div}(c\,\mathbf{u}) = 0 \tag{16}$$

in a 3D domain, $\mathbf{x} \in (0,10) \times (-1,1) \times (-1,1)$ with $\mathbf{u} = (1,0,0)^T$ was considered as a benchmark problem. This initial-value problem was solved using a fully explicit 4^{th} - order explicit Runge-Kutta DG - method (see [9]) on a equidistant cartesian grid with $N_x \times N_y \times N_z$ cells.

The domain was decomposed along the x-Axis, so that the sub-grid on every MPI process p consisted of $\frac{N_x}{p} \times N_y \times N_z$ cells. Therefore, N_x was chosen to be a common multiple of all investigated number of processors. In this configuration, all processes except for the "first" and "last" one have to communicate with two neighbors. So, for tests with more than 3 processors, the amount of data exchange per process, in terms of bytes, stays constant. Therefore, the runtime of the 4-processor-test was taken as a reference run to compare speedup to.

All tests were carried out on a 64-processor cluster, organized in 8 nodes with two quad core Opteron processors at 2 Ghz a Gigabit Ethernet interconnect, running Windows Server 2008 HPC edition.

Strong scaling. The test was carried out on a grid of $960 \times 9 \times 19$ cells. The definition of speedup used is

$$a(s) = \frac{s_{\text{ref}}\, t_{\text{ref}}}{t(s)}, \tag{17}$$

where $t(s)$ is the measured runtime on s processors and $s_{\text{ref}} = 4$ and $t_{\text{ref}} = t(4)$ is the runtime of the reference run on 4 processors. There we define parallel efficiency as

$$eff(s) = \frac{s(n)}{n}. \tag{18}$$

Results of the strong-scaling – test can be found in table III and on the left of figure 8. As expected, the speedup of a test with constant problem size is limited, because the ratio between communication and calculation gets worse with increasing processors. The number of cells to exchange stays constant (342 cells to send and receive for each process) and e.g. for the 4-processor test, there are 41040 cells on each processor to calculate. For 48 processors, the number of cells per processor drops down to 3420. The efficiency measure would get better if a larger grid would be considered, but due to the memory limits (the native libraries used by BoSSS currently compile only in 32 bit, so also the managed parts need to run in 32-bit - mode) the grid size is restricted on the 2-processor test.

TABLE III. PARALLEL SPEEDUP OF A BENCHMARK WITH CONSTANT PROBLEM SIZE. GRID IS $960 \times 9 \times 19$ CELLS.

# of Procs s	Runtime [sec] $t(s)$	# of Nodes	Speedup $a(s)$	Parallel Efficiency $eff(s)$
2	36371	1	2.03	1.01
4	18422	1	4	1
8	9877	1	7.46	0.93
12	6606	2	11.15	0.93
16	4897	3	15.05	0.94
24	3318	4	22.21	0.93
32	2559	5	28.8	0.9
48	1765	7	41.75	0.87

Weak scaling. The test was carried out on a grid of $20 \cdot s \times 9 \times 19$ cells. The definition of speedup used is

$$a(s) = \frac{s\, t_{\text{ref}}}{t(n)}, \tag{19}$$

where $t(s)$ is the measured runtime on s processors and and $t_{\text{ref}} = t(4)$ is the runtime of the reference run on 4 processors. Parallel efficiency, $eff(s)$ is defined as above.

Results of the weak-scaling – test can be found in table IV and on the right of figure 8. Within this test, a significant performance drop between 8 and 24 processors was noticed, while beyond that point the parallel efficiency seems to remain constant. Whether this is a result of hardware "saturation" (e.g. the network switch) or results from the algorithm has to be determined by test on clusters with better interconnect and more processors. It is clear, that for 1 processor, where communication is turned off, the speedup in comparison to 4 processors is greater than 1.0.

TABLE IV. PARALLEL SPEEDUP OF A BENCHMARK WITH GROWING PROBLEM SIZE. GRID IS $20 \cdot s \times 9 \times 19$ CELLS. (SPEEDUP SHOULD BE UNDERSTOOD AS RUNTIME IN COMPARISON TO THE EXTRAPOLATED RUNTIME FORM THE 4-PROCESSOR TEST.)

# of Procs s	Runtime [sec] $t(s)$	# of Nodes	Speedup $a(s)$	Parallel Efficiency $eff(s)$
1	1496	1	1.15	1.149
2	1693	1	2.03	1.015
4	1719	1	4.00	1
8	1759	2	7.82	0.977
12	1725	2	11.96	0.997
16	1742	3	15.79	0.987
24	1787	4	3.09	0.962
32	1788	5	30.77	0.961
48	1786	7	46.20	0.962
56	1776	8	54.20	0.968

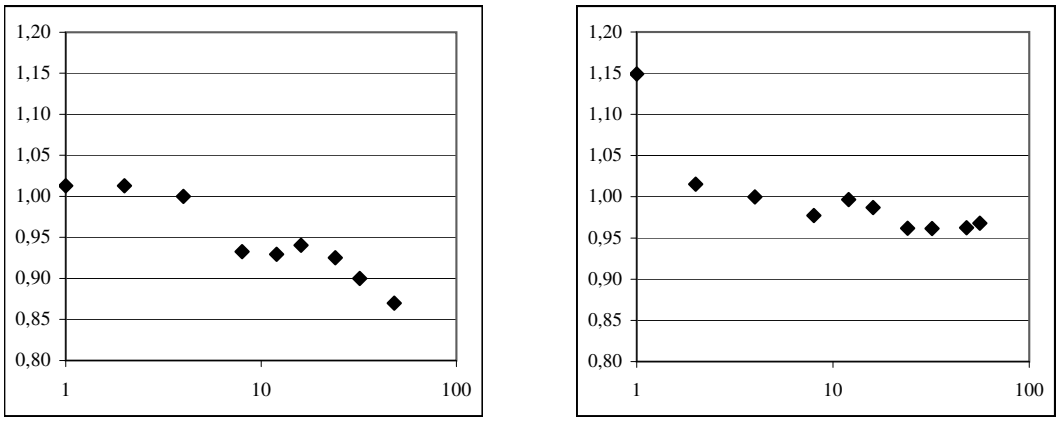

Figure 8. *Parallel efficiency versus number of processors. (left side: constant size problem (strong scaling), right side: growing size problem (weak scaling))*

ACKNOWLEDGEMENTS

This work is funded by the German Research Foundation (“Deutsche Forschungsgemeinschaft”, DFG) in Collaborative Research Center 568 (“Sonderforschungsbereich”, SFB 568).

REFERENCES

1. Vogels W. 2003. “Benchmarking the CLI for high performance computing,” *IEE Proc.-Softw.*, 150(5).
2. Bull J.M., Smith L.A., Pottage L., Freeman R. 2001. “Benchmarking Java against C and Fortran for scientific applications,” *Proc. ACM Conf. on Java Grande/(ISCOPE)*, Stanford University, CA, 2-4 June 2001,pp. 97-105.
3. Arnold D.N., Brezzi F., Cockburn B., Marini L.D. 2002. “Unified analysis of dincontinuous Galerkin methods for elliptic problems,” *SIAM J. Numer. Anal.*, 39(5):1749-1779.
4. Shahbazi K., Fischer P.F., Ethier C.R. 2007. “A high-order discontinuous Galerkin method for the unsteady incompressible Navier-Stokes equations,” *J. Comp. Phys.* 222:391-407.
5. Bassi F., Ghidoni A., Rebay S., Tesini P. 2008. “High-order accurate p-multigrid discontinuous Galerkin solution of the Euler equations,” *Int. J. Numer. Meth. Fluids*, Early View.
6. Griault V., Riviere B., Wheeler M.F. 2005. “A splitting method using discontinuous galerkin for the transient incompressible Navier-Stokes equations,” *ESIAM: M2AN* 39(6):1115-1147.
7. Java Grande Forum Panel “Java Grande Forum Report: Making Java work for high-end computing,” Presented at Supercomputing 1998, 13 November 1998, http://www.javagrande.org/reports.htm.
8. Hesthaven J.S., Warburton T. 2008. “ Nodal Discontinuous Galerkin Methods - Algorithms, Analysis and Applications,” *Springer-Verlag* ISBN 978-0-387-72065-4.
9. Cockburn B., Karniadakis G.E., Shu C.W. 2000. “Discontinuous Galerkin Methods, Theory, Computation and Applications,” *Springer-Verlag* ISBN 3-540-66787-3.
10. Dolejsi V., Feistauer M. 2004. “A semi-implicit discontinuous Galerkin finite element method for the numerical solution of inviscid compressible flow,” *J. Comp. Phys.* 198:727-746.
11. Shudo K. 2005. “Performance Comparison of Java/.NET Runtimes,” *http://www.shudo.net/jit/perf*.
12. Karypis G., Kumar V. 1996. “Parallel multilevel *k*-way partitioning scheme for irregular graphs,” Technical Report TR 96-036, Department of Computer Science, University of Minnesota.
13. HYPRE - high performance preconditioners. *http://acts.nersc.gov/hypre*
14. Novell; “Mono” *http://www.mono-project.com*
15. The Free Software Foundation; “DotGNU Portable.NET,” *http://www.gnu.org/software/dotgnu/pnet.html*.
16. TOP500 Team, “TOP500 Report for June 2009,” *http://www.top500.org*.
17. Sahadid J.N., Tuminaro R.S., “Sparse iterative algorithm software for large-scale MIMD machines: An initial discussion and implementation,” *Concurrency: Practice and Experience*, 4(6):481-497, September 1992.
18. Weinan E., Jian-Guo L., “Projection Method I: Convergence and Numerical Boundary Layers” *SIAM Journal on Numerical Analysis*, 32(4):1017-1091, 1995.
19. McLachlan R.I., Quispel G.R.W., “Splitting methods” *Acta Numerica*, 2002:341-434.