IPP-Report

IPP Max-Planck-Institut für Plasmaphysik

R. Preuss, R. Fischer, M. Rampp, K. Hallatschek,
U. von Toussaint, L. Giannone, P. McCarthy

Parallel equilibrium algorithm for real-time control of tokamak plasmas

IPP R/47
Februar, 2012

Parallel equilibrium algorithm for real-time control of tokamak plasmas

R. Preuss, R. Fischer, M. Rampp, K. Hallatschek,
U. von Toussaint, L. Giannone, P. McCarthy
Rechenzentrum der Max-Planck-Gesellschaft,
Max-Planck-Institut für Plasmaphysik,
EURATOM Association,
Boltzmannstr. 2, 85748 Garching, Germany

February 9, 2012

Abstract

In order to achieve real-time control of fusion plasmas the flux distribution and derived quantities have to be calculated within the time of the machine control cycle which is typically of the order of 1 ms. This requires a fast solver of the Grad-Shafranov equation together with optimized procedures for the utilization of the result. An algorithm for a fast solver is presented which allows exploitation of the parallel capabilities of modern multi-core processors. Our implementation termed GPEC (Garching parallel equilibrium code) is based entirely on open source software components. For a numerical grid of size 32×64 our new code requires only 0.04 ms (0.11 ms for 64×128) for a single iteration of the Grad-Shafranov solver using a standard Intel Xeon quadcore CPU (3.2 GHz).

1 Introduction

Plasma equilibrium in a tokamak is a key discharge property determining the evolution of a variety of instabilities, such as those related to the bootstrap currents – among them the neoclassical tearing modes (NTM) to name only the most prominent ones. The possibility of getting rid of NTMs with electron cyclotron current drive (ECCD) [1] whenever their exact position is known motivates the development of numerical methods for the real-time computation of the plasma flux distribution. Moreover, with limitations on the number of (expensive) discharges future fusion machines like ITER or DEMO rely on real-time capabilities for equilibrium calculation in order to adjust plasma control online.

So far only approximative approaches and function parametrization are capable of delivering flux distributions updated within the time scale of one or a few control cycles. Their obvious drawback is that they suffer from flux distribution uncertainties and usually fail to cover new or unusual discharges. Solvers of the MHD equilibrium described by the Grad-Shafranov equation, like EFIT [2] or GEC (Garching equilibrium code) [3] are fast enough only if the size of the grid or the number of basis functions (representing the plasma current distribution) is down-sized, sometimes to a large extent so that the results are rendered questionable.

In this work a numerically optimized equilibrium solver called GPEC (Garching parallel equilibrium code) shall be provided which allows to employ as many basis functions as regarded meaningful in an as large as possible grid to calculate a plasma flux distribution within one control cycle of ASDEX Upgrade, i.e. 1.5 ms. The main idea is to exploit parallel capabilities of common multicore computer processors by a well known ansatz which was already developed in the 1970s for the solution of Poisson's differential equation (e.g. [4]). It combines Fast Fourier transformation and a solution of a tridiagonal system of equations, which essentially decouples the discretization stencil in one of the two grid dimensions. In order to remain independent from licensing issues and to avoid any kind of "vendor lock-in" the new algorithm is implemented in open source code and can be distributed entirely under an open source licensing model. This meets, in addition, the software policy of ITER.

The paper is organized as follows: Sections 2 and 3 describe the theoretical foundations. In Section 4 we describe the algorithm we employ for the solution of the Grad-Shafranov equation and its discretization. In Section 5 we outline the main concepts for the parallelization. Section 6 provides details of the actual numerical implementation and computational performance of the new GPEC solver. Sections 7 and 8 finally summarize the

work and provide an outlook on possible applications at ASDEX Upgrade, respectively.

2 Calculation of the flux distribution

For the toroidal symmetry of a tokamak, ideal magnetohydrodynamic equilibrium is described by the second-order partial differential Grad-Shafranov equation [5, 6] for the poloidal flux function ψ

$$\Delta^* \psi = -4\pi^2 \mu_0 r j_\phi \quad , \tag{1}$$

with the differential operator in cylindrical coordinates $(r,\, z)$

$$\Delta^* = r \frac{\partial}{\partial r} \frac{1}{r} \frac{\partial}{\partial r} + \frac{\partial^2}{\partial z^2} \quad . \tag{2}$$

The toroidal current density profile j_ϕ consists of two terms

$$j_\phi = r \frac{\partial p(r,\psi)}{\partial \psi} + \mu_0 \frac{F(\psi)}{r} \frac{\mathrm{d}F(\psi)}{\mathrm{d}\psi} \quad , \tag{3}$$

where p is the plasma pressure and F denotes the flux function of the poloidal plasma currents. We consider the isotropic case, i.e. $p(r,\psi) = p(\psi)$, so both terms are functions of ψ only and, in the framework of MHD, can be specified arbitrarily [3], i.e. they will be determined by experiment in our case. This is usually done by linear superposition of a number N_p and N_F of so-called basis functions $\pi(\psi)$ and $\varphi(\psi)$, respectively [7],

$$p'(\psi) = \sum_{h=1}^{N_p} c_h \pi_h(\psi) \quad , \tag{4}$$

$$FF'(\psi) = \sum_{k=1}^{N_F} d_k \varphi_k(\psi) \quad . \tag{5}$$

π and φ are either chosen as spline functions, as in CLISTE [7], or special polynomial functions, like in EFIT [2]. In equilibrium codes Eq. (1) is repeatedly solved by inserting on the right-hand side (rhs) for j_ϕ each basis function π_h and φ_k alone. The latter are calculated with an "old" flux distribution taken from either initial settings or a previous iteration cycle (see next chapter):

$$\Delta^* \psi_h = -4\pi^2 \mu_0 r^2 \pi_h(\psi^{\mathrm{old}}) \quad , \tag{6}$$

$$\Delta^* \psi_{N_p+k} = -4\pi^2 \mu_0^2 \varphi_k(\psi^{\mathrm{old}}) \quad . \tag{7}$$

With the resulting $N_p + N_F$ flux distributions $\boldsymbol{\psi}$ a so-called 'response matrix' $\boldsymbol{B}$ consisting of predictions $b_v(\psi_w)$ for a certain set of N_m diagnostic signals $\vec{m} = (m_1, m_2, \ldots, m_{N_m})^T$ is calculated. The prediction $b_v(\psi_w)$ is some function employing the flux distribution ψ_w to produce the ideal measurement signal, i.e. without measurement noise. An example of such a function may be the response of the flux distribution to the poloidal magnetic probes. Simple linear regression of the equation

$$\begin{pmatrix} m_1 \\ m_2 \\ \vdots \\ m_{N_m} \end{pmatrix} = \begin{pmatrix} b_1(\psi_1) & b_1(\psi_2) & \ldots & b_1(\psi_{N_p+N_F}) \\ b_2(\psi_1) & b_2(\psi_2) & \ldots & b_2(\psi_{N_p+N_F}) \\ \vdots & \vdots & \vdots & \vdots \\ b_{N_m}(\psi_1) & b_{N_m}(\psi_2) & \ldots & b_{N_m}(\psi_{N_p+N_F}) \end{pmatrix} \cdot \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_{N_p} \\ d_1 \\ d_2 \\ \vdots \\ d_{N_F} \end{pmatrix} \tag{8}$$

provides the coefficients $\vec{c}$ and $\vec{d}$ of Eqs. (4) and (5). The final result for the flux distribution ψ^{tot} from one such calculation scheme for the Grad-Shafranov equation is

$$\psi^{tot} = \sum_{i=1}^{N_p} c_i \psi_i + \sum_{j=1}^{N_F} d_j \psi_{N_p+j} \quad . \tag{9}$$

In the following the rhs of Eq. (1) shall be denoted by a general function $g(r, z)$ (which will be either one of π_i or φ_j according to Eqn. (6) and (7)). This states our target problem of solving

$$\Delta^* \psi = -g(r, z) \quad , \tag{10}$$

which is, in mathematical terms, an elliptic differential equation or Poisson equation. The explicit dependence on (r, z), instead of ψ, refers to the fact that the flux function ψ is generated in the above way from measurements mapped to the cylindrical coordinate system and is supposed to resemble Eq. (7) of Ref. [3] which is the starting point for the next section.

3 Lackner's Algorithm

The main idea to tackle Eq. (10) is to use a modified Picard iteration scheme where the new ψ for the next iteration step is adjusted by comparison of derived quantities with diagnostic signals as described above. In each cycle

Eq. (10) is solved individually for the $N_p + N_F$ different right-hand sides to calculate ψ^{tot} according to Eq. (9). With the latter one obtains an update for $g(r, z)$. This procedure is iterated until convergence criteria are fulfilled, e.g. that the largest flux element of ψ does not change within a desired accuracy of usually 0.01 percent of its value. One such iteration cycle is called an "outer" iteration. It turns out that the number of outer iterations is usually on the order of 10, but, depending on the initial flux distribution, can be as small as two or three.

The possibility to reach time ranges on the order of the control cycle is intimately linked with the existence of a fast solver for Eq. (10). First steps in this direction were performed by Bunemann in 1967 by presenting a fast Poisson solver for a rectangular grid relying on a cyclic reduction method [8]. Buzbee showed in 1971 how to embed an arbitrarily shaped boundary for which Dirichlet conditions exist in a rectangular grid by first invoking the Poisson solver with artificial zero boundary and correcting for the proper boundary conditions with a second call of the solver [9]. Based on the above findings Hagenow and Lackner in 1975 [10] described an algorithm solving Eq. (10) with the following three steps:

1. Calculate ψ^0 as a solution of Eq. (10) with $\psi^0 = 0$ on the (rectangular) boundary.

2. Computation of ψ at the boundary employing a Greens function [3]

$$\psi(r) = -\oint_{\partial R} \frac{1}{r^*} G(r, r^*) \left(\frac{\partial \psi^0(r^*)}{\partial n} \right) \mathrm{d}s^* \quad . \tag{11}$$

3. Finally, the boundary conditions computed with Eq. (11) are used to calculate ψ in a second application of the fast solver.

Invoking the fast solver two times (in steps 1 and 3) constitutes the so-called "inner" iteration.

Though for the past 30 years efforts were made to optimize Buneman's fast solver on the fastest serial computers, e.g. by employing assembler code, it was not possible to enter the control cycle time range. More recent efforts to implement the algorithm on parallel systems proved futile since the cyclic reduction method contains dependent loops with loop lengths too short to profit from parallelization.

Our new approach presented here employs the Fourier transform method, a classical tool for solving partial differential equations since the early 1970s [4]. Fourier decomposition leads to a system of linear equations represented by a set of symmetric tridiagonal matrices which can be solved independently.

Fourier back transformation concludes the solver step. The advantages of these well known methods are evident – especially when implementing for multicore processors. For the Fourier transformation as well as for the solution of the tridiagonal matrix highly optimized numerical library routines are available. Moreover, both methods split the numerical task into independent portions which allow a highly parallel work flow (see Section 5).

4 Solution of the discretized Grad-Shafranov equation

The first step is the discretization of the Grad-Shafranov equation in Eq. (10). In cylindrical coordinates with radius r and height z we consider a uniformly spaced two dimensional grid of size $M \times N$ with (see Fig. 1)

$$r_i = r_0 + i \cdot \Delta_r \,, \quad i = 0, ..., M \,, \tag{12}$$

$$z_j = z_0 + j \cdot \Delta_z \,, \quad j = 0, ..., N \,, \tag{13}$$

where $r_0 = r_{min}$, $r_M = r_{max}$, $\Delta_r = (r_M - r_0)/M$ and $z_0 = z_{min}$, $z_N = z_{max}$, $\Delta_z = (z_N - z_0)/N$.

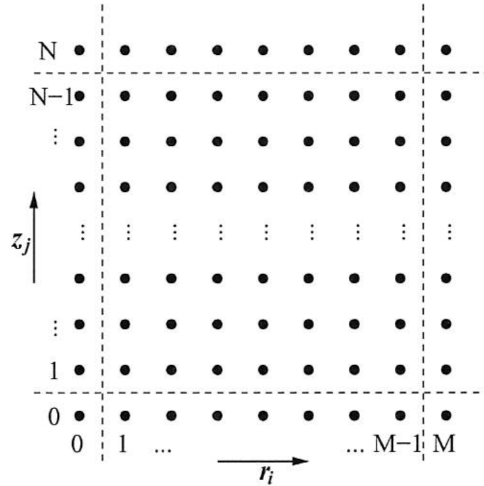

Figure 1: Topography for a $M \times N$ grid. The dashed lines clarify the index values for the boundary grid points. Since the calculation of the flux distribution affects only the inner grid points (and the nearest neighbours at the boundary), the flux values of the four corners have been defined to be the mean of the two nearest neighbours, respectively.

With the usual five-point discretization stencil the differential operator Δ^* from Eq. (2) reads

$$r\frac{\partial}{\partial r}\frac{1}{r}\frac{\partial}{\partial r}\psi \quad \rightarrow \quad \frac{r_i}{\Delta_r^2}\left[\left(\frac{\psi_{i+1,j}}{r_{i+1/2}} - \frac{\psi_{i,j}}{r_{i-1/2}}\right) - \left(\frac{\psi_{i,j}}{r_{i+1/2}} - \frac{\psi_{i-1,j}}{r_{i-1/2}}\right)\right] \quad , \tag{14}$$

$$\frac{\partial^2}{\partial z^2}\psi \quad \rightarrow \quad \frac{1}{\Delta_z^2}\left[\psi_{i,j+1} - 2\psi_{i,j} + \psi_{i,j-1}\right] \quad . \tag{15}$$

We define

$$s = \frac{\Delta_z}{\Delta_r} \; , \; s_i^+ = \frac{s^2}{r_{i+1/2}} \; , \; s_i^- = \frac{s^2}{r_{i-1/2}} \; , \; \delta_i = -s_i^+ - s_i^- - \frac{2}{r_i} \; , \; g_{i,j} = g(r_i, z_j) \; . \tag{16}$$

Employing the above definitions we get the discretized version of Eq. (10),

$$s_i^+\psi_{i+1,j} + s_i^-\psi_{i-1,j} + \frac{1}{r_i}\psi_{i,j+1} + \frac{1}{r_i}\psi_{i,j-1} + \delta_i\psi_{i,j} = -\frac{g_{i,j}}{r_i}\Delta_z^2 \quad . \tag{17}$$

We shift $\psi_{i,0}$ and $\psi_{i,N}$ to the right-hand side, because they are either zero (as in step 1 of Lackner's algorithm) or constant (as in step 3 of Lackner's algorithm). Let us represent the right-hand side by a single variable ρ:

$$\rho_{i,j} = \begin{cases} -g_{i,1}\Delta_z^2/r_i - \psi_{i,0}/r_i & \text{for} \quad j = 1 \quad , \\ -g_{i,j}\Delta_z^2/r_i & \text{for} \quad j = 2, ..., N-2 \; , \\ -g_{i,N-1}\Delta_z^2/r_i - \psi_{i,N}/r_i & \text{for} \quad j = N-1 \quad . \end{cases} \tag{18}$$

At this point it is possible to decouple the two-dimensional dependency on adjacent grid points by employing the real sine Fourier transformation, which is commonly termed "discrete sine transform" (DST):

$$\psi_{i,j} \quad = \quad \frac{2}{N}\sum_{l=1}^{N-1}\hat{\psi}_{i,l}\sin(\pi j l/N) \quad , \tag{19}$$

$$\rho_{i,j} \quad = \quad \frac{2}{N}\sum_{l=1}^{N-1}\hat{\rho}_{i,l}\sin(\pi j l/N) \quad . \tag{20}$$

Regarding the z-direction Eq. (19) reads

$$\psi_{i,j\pm1} = \frac{2}{N}\sum_{l=1}^{N-1}\hat{\psi}_{i,l}\sin(\pi j l/N \pm \pi l/N)) \quad . \tag{21}$$

We make use of

$$\sin(\alpha \pm \beta) = \sin(\alpha)\cos(\beta) \pm \cos(\alpha)\sin(\beta) \quad , \tag{22}$$

and insert the modified Eq. (21) together with Eqs. (19) and (20) in Eq. (17):

$$
\begin{aligned}
s_i^+ \frac{2}{N} \sum_{l=1}^{N-1} \hat{\psi}_{i+1,l} \sin(\pi jl/N) &+ s_i^- \frac{2}{N} \sum_{l=1}^{N-1} \hat{\psi}_{i-1,l} \sin(\pi jl/N) \\
+\frac{1}{r_i}\frac{2}{N} \sum_{l=1}^{N-1} \hat{\psi}_{i,l} \sin(\frac{\pi jl}{N}) \cos(\frac{\pi l}{N}) &+ \frac{1}{r_i}\frac{2}{N} \sum_{l=1}^{N-1} \hat{\psi}_{i,l} \cos(\frac{\pi jl}{N}) \sin(\frac{\pi l}{N}) \\
+\frac{1}{r_i}\frac{2}{N} \sum_{l=1}^{N-1} \hat{\psi}_{i,l} \sin(\frac{\pi jl}{N}) \cos(\frac{\pi l}{N}) &- \frac{1}{r_i}\frac{2}{N} \sum_{l=1}^{N-1} \hat{\psi}_{i,l} \cos(\frac{\pi jl}{N}) \sin(\frac{\pi l}{N}) \\
+\delta_i \frac{2}{N} \sum_{l=1}^{N-1} \hat{\psi}_{i,l} \sin(\pi jl/N) &= \frac{2}{N} \sum_{l=1}^{N-1} \hat{\rho}_{i,l} \sin(\pi jl/N) \quad . \qquad (23)
\end{aligned}
$$

Postulating that Eq. (23) has a solution if it can be solved for every value of the index l individually, we get

$$
s_i^+ \hat{\psi}_{i+1,l} + s_i^- \hat{\psi}_{i-1,l} + \frac{2}{r_i} \hat{\psi}_{i,l} \cos(\pi l/N) + \delta_i \hat{\psi}_{i,l} = \hat{\rho}_{i,l} \quad , \qquad (24)
$$

for all $l = 1, ..., N-1$ and $i = 1, ..., M-1$. With the help of $s_i^- = s_{i-1}^+$ and defining $\hat{\delta}_{i,l} = [2\cos(\pi l/N)/r_i + \delta_i]$ we finally obtain

$$
s_i^+ \hat{\psi}_{i+1,l} + \hat{\delta}_{i,l} \hat{\psi}_{i,l} + s_{i-1}^+ \hat{\psi}_{i-1,l} = \hat{\rho}_{i,l} \quad . \qquad (25)
$$

This establishes $(N-1)$ independent linear systems of size $(M-1)$ which can be written as

$$
\boldsymbol{D}_l \cdot \hat{\boldsymbol{\psi}}_l = \hat{\boldsymbol{\rho}}_l \quad , \quad l = 1, ..., N-1 \quad . \qquad (26)
$$

On the left-hand side Eq. (26) has the $(M-1)\times(M-1)$ symmetric tridiagonal matrix $\boldsymbol{D}_l$ and the target vector $\hat{\boldsymbol{\psi}}_l$

$$
\boldsymbol{D}_l = \begin{pmatrix}
\hat{\delta}_{1,l} & s_1^+ & & & & 0 \\
s_1^+ & \hat{\delta}_{2,l} & s_2^+ & & & \\
& s_2^+ & \hat{\delta}_{3,l} & s_3^+ & & \\
& & \ddots & \ddots & \ddots & \\
& & & s_{M-3}^+ & \hat{\delta}_{M-2,l} & s_{M-2}^+ \\
0 & & & & s_{M-2}^+ & \hat{\delta}_{M-1,l}
\end{pmatrix} , \quad \hat{\boldsymbol{\psi}}_l = \begin{pmatrix}
\hat{\psi}_{1,l} \\ \hat{\psi}_{2,l} \\ \hat{\psi}_{3,l} \\ \vdots \\ \hat{\psi}_{M-2,l} \\ \hat{\psi}_{M-1,l}
\end{pmatrix} , \qquad (27)
$$

and on the right-hand side the vector $\hat{\boldsymbol{\rho}}_l$ with the aforementioned subtractions of the boundary values of $\hat{\psi}$

$$
\hat{\boldsymbol{\rho}}_l = \begin{pmatrix}
\hat{\rho}_{1,l} - s_0^+ \hat{\psi}_{0,l} \\ \hat{\rho}_{2,l} \\ \hat{\rho}_{3,l} \\ \vdots \\ \hat{\rho}_{M-2,l} \\ \hat{\rho}_{M-1,l} - s_{M-1}^+ \hat{\psi}_{M,l}
\end{pmatrix} \quad . \qquad (28)
$$

The $\hat{\rho}_{i,l}$ are the backward sine Fourier transforms (or "inverse discrete sine transform", iDST) of $\rho_{i,l}$ of Eq. (18)

$$\hat{\rho}_{i,l} = \sum_{j=1}^{N-1} \rho_{i,j} \sin(\pi j l / N) \quad , \tag{29}$$

with $i = 1, ..., M-1$. Eventually, the $\hat{\psi}$-values at the boundary are obtained from backward sine Fourier transformation for $i = 0$ and $i = M$ only:

$$\hat{\psi}_{i,l} = \sum_{j=1}^{N-1} \psi_{i,j} \sin(\pi j l / N) \quad . \tag{30}$$

According to the algorithm the boundary values of ψ are set to zero for the first solver step and the backward Fourier transformation can be skipped. However, for the second call of the solver the boundary-ψ are nonzero and Eq. (30) has to be evaluated.

5 Parallelizable algorithm

With the above equations Lackner's algorithm presented in Section 3 can be implemented such that parallel capabilities of common processors can be exploited. Start by obtaining $\hat{\boldsymbol{\rho}}_l$ in Eq. (28) from the backward Fourier transformation of an initial flux distribution (either from initial conditions or from a previous outer iteration step). Since the boundary conditions in the first step of the solver are set to $\psi^0 = 0$, only Eq. (29) is needed. The computation can be performed in $(M-1)$ independent tasks. Then the linear system of Eq. (26) is solved to obtain the target vector $\hat{\boldsymbol{\psi}}_l$ which allows to compute the proper boundary for ψ. Since the linear system decouples into $(N-1)$ independent tridiagonal systems, this step can be performed in $(N-1)$ parallel tasks.

As already stated in Ref. [11] it is possible to spare some work in the subsequent process. In our realization the resulting $\hat{\boldsymbol{\psi}}_l$ are saved for later use at all grid positions except at the boundary. Instead of performing a Fourier transformation of the complete vector field $\hat{\boldsymbol{\psi}}$ only the $\psi_{i,j}$ are computed according to Eq. (19) at grid points adjacent to the boundary, since they are all what is needed for the normal derivative $\partial\psi^0/\partial n$ in Eq. (11). This results in the independent and therefore parallelizable calculation of $2 \times (N-1)$ Fourier coefficients for all $1 \leq j \leq N-1$ with $i = 1$ and $i = M-1$, respectively, and $2 \times (M-3)$ Fourier coefficients for all $2 \leq i \leq M-2$ with $j = 1$ and $j = N-1$, respectively. The correct boundary of ψ can then

be calculated with Eq. (11), where the integral is done by a matrix vector multiplication with a precomputed discretized Greens function.

Following Lackner's algorithm the aforementioned steps are performed a second time, but this time with proper boundary conditions. Again some work can be spared in obtaining $\hat{\boldsymbol{\rho}}_l$ from Eq. (28). To begin with, remember that for nonzero boundary conditions ρ is modified according to Eq. (18). This changes every array element of the Fourier transformed $\hat{\rho}$ by

$$\hat{\rho}_{il}^{\text{boundary}\neq 0} = \hat{\rho}_{il}^{\text{boundary}=0} - \frac{\hat{\psi}_{i,0}}{r_i} \sin(\pi l/N) - \frac{\hat{\psi}_{i,N}}{r_i} \sin(\pi(N-1)l/N) \quad .(31)$$

Second, only the first and last entry in the vector $\hat{\boldsymbol{\rho}}_l$ is affected by non-zero boundary values $\hat{\psi}_{0,l}$ and $\hat{\psi}_{M,l}$, respectively. Therefore, only the latter have to be calculated with Eq. (30) and are simply subtracted from the modified $\hat{\rho}$ in Eq. (31). Finally, the tridiagonal system of Eq. (26) is solved to get the target vector $\hat{\boldsymbol{\psi}}_l$, which is Fourier transformed according to Eq. (19) to give the resulting flux distribution ψ.

To refer both to the origins of Lackner's algorithm and to the new parallel capabilities we baptized the implemented program code as *Garching parallel equilibrium code* (GPEC).

6 Implementation and computational performance of GPEC

In this section we shall present our implementation and parallelization approach for the algorithm described in the previous Sections (4, 5) and document its computational performance. In the general context of the Grad-Shafranov equation our solver computes a fast numerical solution of Eq. (10), i.e. for a *single basis function* (cf. Sect.1). Here, our basic parallelization strategy is based on shared-memory threads which are mapped to the cores of a present-day multicore CPU.

For *multiple basis functions* (Eq. 9) each of the required $N_p + N_F$ individual calls of the solver for Eq. (10) can be performed on a separate multicore CPU which may be mounted in a multi-socket (shared memory) or multi-server (distributed memory) system. The distribution of the individual solver calls onto the CPUs is conceptually straightforward and requires only minor communication. It will be implemented using the Message Passing Interface (MPI) or the new coarray features of the FORTRAN 2008 standard [12]. Such a hybrid parallelization strategy is expected to provide sufficient flexibility to cope with processor and server developments in the foreseeable future.

The last subsection of this chapter briefly reports on our attempts to implement the algorithm as described in Sect. 4 on a general-purpose Graphics Processing Unit (GPU).

6.1 Overview

The solver for Eq. (10) is implemented in FORTRAN 90 (using double-precision arithmetics), and employs the industry-standard OpenMP shared-memory programming model for multi-thread parallelization[1]. For reasons of portability and long-term sustainability we did not consider low-level optimizations like, for example, developing assembler code for the most relevant computational routines. The program is based entirely on Open-Source software components, i.e. the complete source code of the solver algorithm, including all numerical libraries required is available to us and can be released under a suitable Open-Source license like GPL, if desired.

In the following we will report results obtained with a standalone program which we had rewritten based on the techniques realized in the CLISTE package for the ease of development and testing. Our implementation basically comprises the rewrite of a subroutine called `EQUIL(psi(:,:))`. The new equilibrium code with GPEC is already being used for offline analysis of ASDEX Upgrade data.

6.1.1 Basic structure of the algorithm

As described in detail in Sects. 4, 5 the overall structure of the algorithm for an outer iteration step can be characterized by the following sequence of computations:

- $\texttt{dst}_1$: $(M-1)$ inverse discrete sine transforms (iDSTs) of size $(N-1)$, cf. Eq. (28)
- $\texttt{trid}_1$: solution of $(N-1)$ tridiagonal linear systems of size $(M-1)$, cf. Eq. (26)
- $\texttt{dst}_{\text{spare1}}$: 2 DSTs of size $(N-1)$ plus $2 \times (M-3)$ scalar products of size $(N-1)$ for computing $2(N-1+M-3)$ Fourier coefficients, cf. Eqs. (11,19)
- $\texttt{matvec}$: 1 matrix-vector multiplication of rank $2(N+M-2) \times 2(N+M-2)$, cf. Eq. (11)
- $\texttt{dst}_{\text{spare2}}$: 2 iDSTs of size $(N-1)$ plus a corresponding rank-1 update of a $N \times M$ matrix, cf. Eq. (31)
- $\texttt{trid}_2$: solution of $(N-1)$ tridiagonal linear systems of size $(M-1)$, cf. Eq. (26)
- $\texttt{dst}_2$: $(M-1)$ DSTs of size $(N-1)$, cf. Eq. (19)

[1] see www.openmp.org

Note that in the following the notation DST refers to both, the discrete sine transform and its inverse (iDST) because both variants are equivalent concerning implementation and computational performance.

6.1.2 Parallelization strategy

The calculations of $(M-1)$ or 2 DSTs required in steps $\texttt{dst}_1$, $\texttt{dst}_2$ or steps $\texttt{dst}_{\text{spare1}}$, $\texttt{dst}_{\text{spare2}}$, respectively, are mutually independent and can obviously be computed in parallel. The same applies for computing the $(N-1)$ solutions of the tridiagonal systems corresponding to the steps labelled $\texttt{trid}_1$ and $\texttt{trid}_2$. Since each of the consecutive steps $\texttt{dst}_1$, $\texttt{trid}_1$, ..., $\texttt{dst}_2$ requires the complete update of the work arrays from the previous step, an implicit synchronization barrier is implied after each of the individual steps.

Our implementation generally tries to maximize the computational work for each thread in order to reduce the impact of OpenMP parallelization overhead. This is of critical importance since in our application a parallel loop typically contains only a few microseconds of computational work for a single loop iteration. Typical OpenMP synchronization overheads can be on the order of up to a few microseconds, according to the EPCC OpenMP micro-benchmark suite [13].

6.1.3 Data layout

In our FORTRAN 90 implementation we chose a data layout that is transposed wrt. the conventional notation used for the $M \times N$ numerical grid (Sect. 4), i.e. the main two-dimensional work array `x` which represents $\psi, \rho, \ldots$ is dimensioned as `x(N,M)`. Thus, each individual DST can work efficiently on a contiguous memory section, corresponding to FORTRAN's so-called "column-major" array ordering. As a consequence, an orthogonal access pattern along rows applies for the solution of the set of tridiagonal linear systems which in general is very unfavorable for the performance due to strided memory accesses. However, we shall present a highly efficient algorithm for treating the set of tridiagonal linear systems which avoids strided memory access without having to perform expensive transpositions of the work arrays.

The following three subsections provide implementation details for the steps labelled `dst` (Sect. 6.2), `trid` (Sect. 6.3), and `matvec` (Sect. 6.4), respectively.

6.2 Discrete sine transform (DST)

For computing the discrete sine transforms required in steps $\mathtt{dst}_1$, $\mathtt{dst}_2$, $\mathtt{dst}_{\text{spare1}}$, $\mathtt{dst}_{\text{spare2}}$ (cf. Sect. 6.1.1) we employ the free software package FFTW (www.fftw.org, currently in version 3.2.2). FFTW is Open-Source software (released under the GNU General Public License) and is probably *the* most popular and widely used FFT library in scientific high-performance computing.

For the DSTs in steps $\mathtt{dst}_1$, $\mathtt{dst}_2$ a number of n so called FFTW-'plans' are precomputed in an initialization step, where n is the number of parallel OpenMP threads to be specified at program startup. Note that in our application these initializations do not contribute to the relevant computing-time budget. Each FFTW-plan corresponds to the computation of a chunk of $\lceil (M-1)/n \rceil$ independent, one-dimensional discrete sine transforms, where $\lceil x \rceil$ denotes the smallest integer larger than or equal x. Compared with a naive implementation using $(M-1)$ plans this improves serial performance (via its "advanced interface" FFTW provides routines for efficiently computing many independent one-dimensional transforms within a single plan) and diminishes parallel overhead (by reducing the number of plans resp. parallel loop iterations from $M-1$ to n). Given the fact that the amount of computational work is the same for each individual DST our grouping of individual DSTs into chunks of largest reasonable size requires that n should divide $(M-1)$ with a zero or large remainder. Otherwise, the load imbalance introduced by one thread which has to compute less than $\lceil (M-1)/n \rceil$ DSTs would obviously degrade parallel efficiency. For example, with $M = 33$ and $n = 6$ this load imbalance would impose an upper limit of $32/\lceil 32/6 \rceil = 5.2$ on the achievable parallel speedup.

For performing the DSTs required in steps $\mathtt{dst}_{\text{spare1}}$, $\mathtt{dst}_{\text{spare2}}$ (Sect. 6.1.1) we precompute two additional FFTW-plans and execute them in parallel. In step $\mathtt{dst}_{\text{spare1}}$ the remaining Fourier coefficients at grid points adjacent to the boundary, $\psi_{i,1}$ and $\psi_{i,N-1}$ $(2 \le i \le M-2)$, are computed by evaluating the two scalar products which are defined by Eq. (19) with $j = 1$, and $j = N-1$, respectively. We use precomputed values for the vectors $\sin(\pi l/N)$ and $\sin(\pi l(N-1)/N), 1 \le l \le N-1$. Computing only this smaller set of $2(N-1+M-3)$ Fourier coefficients is obviously computationally much more efficient than employing a full set of $(M-1)$ DSTs of size $(N-1)$ and discarding $\psi_{i,j}$ for all $1 \le i \le N, 1 \le j \le M$. Similarly, as described in Sect. (5), the operations required for step $\mathtt{dst}_{\text{spare2}}$ can be reduced to the computation of only 2 DSTs (producing the two M-element vectors $\hat{\psi}_{i,0}$ and $\hat{\psi}_{i,N}$) which enter the rank-1 update of the $M \times N$ matrix $\hat{\rho}_{i,j}$ (cf. Eq. 31).

As an alternative and for numerical reference our program optionally supports the discrete sine transform from Intel's proprietary Math Kernel Library (MKL), which is closed-source software, as well as the routine `sinft` from the popular Open-Source library "Numerical Recipes" [14]. Performance comparisons between the libraries will be given below. Note that we did not make any attempts to optimize Numerical Recipes code.

6.3 Tridiagonal solver

We have implemented an efficient solver for sets of mutually independent tridiagonal linear systems (steps $\mathtt{trid}_1$, $\mathtt{trid}_2$ in Sect. 6.1.1). It is based on the classic Thomas algorithm, which essentially is a standard Gaussian elimination method (without pivoting) specialized for a tridiagonal system. Since in our case the matrix elements depend only on the numerical grid (cf. Eq. 25), we may precalculate and store a LU decomposition, the computation of which does not contribute to the computing-time budget that is relevant for our application (cf. [11]).

The time-critical backsubstitution step is implemented such that we can take advantage of thread-level parallelization and at the same time exploit SIMD-vectorization for boosting single-thread performance on modern processors which support the Streaming SIMD Extensions (SSE) to the standard x86 instruction set or similar schemes like AVX. Instead of performing a call to a standard tridiagonal solver individually for each of the $N-1$ systems, our solver works on chunks of $\lceil (N-1)/n \rceil$ systems per call, where n is the number of parallel OpenMP threads. Much similar to the aforementioned case of grouping together individual DSTs (Sect. 6.2) n has to divide $(N-1)$ with a zero or a large remainder in order not to introduce significant load-imbalances. Within a thread or such a solver call, respectively, SIMD-vectorization can be achieved by performing a simple loop interchange which makes the loop over the $\lceil (N-1)/n \rceil$ independent systems an inner loop and hence amenable to auto-vectorization by the compiler[2]. The basic implementation of this solver was taken from the astrophysical radiation-hydrodynamics code VERTEX [15]. The basic technique is illustrated by the following code fragment which is a "vectorized" rewrite of the Numerical Recipes routine `tridag` [14].

```
! n: system size
! a: lower codiagonal, b: main diagonal, c: upper codiagonal
! r: right-hand side, u: solution vector
```

[2]M.R. acknowledges Dr. Rudolph Fischer (NEC inc.) for originally pointing out the idea in this context to him.

```
... ! handle j=1 here
do j=2,n       ! forward elimination step
  do i=1,k     ! loop over k tridiagonal systems, SIMD vectorizable
      gam(i,j)=c(i,j-1)/bet(i)
      bet(i)=b(i,j)-a(i,j-1)*gam(i,j)
      u(i,j)=(r(i,j)-a(i,j-1)*u(i,j-1))/bet(i)
  end do
end do

do j=n-1,1,-1 ! back substitution step
  do i=1,k     ! loop over k tridiagonal systems, SIMD vectorizable
      u(i,j)=u(i,j)-gam(i,j+1)*u(i,j+1)
  end do
end do
```

For typical numerical grids used in our applications the coefficients of the tridiagonal matrix (Eq. 25) satisfy the diagonal-dominance criterion $|\hat{\delta}_{i,l}| \geq |s_i^+|+|s_{i-1}^+|$ $(\forall i, l)$. This allows us to omit a pivot search in the solver (cf. [14]) and hence to avoid introducing indirect indexing of the arrays which usually impedes SIMD-vectorization.

As a reference for the numerical accuracy and for assessing the performance of our implementation, we also support the option to employ the unmodified Numerical Recipes routine `tridag` [14] or the MKL routines `DDTTRFB`, `DDTTRSB`. The latter are pivot-less variants of the standard LAPACK routines `DDTTRF`, `DDTTRS` for factorization and solution of tridiagonal linear systems. Performance comparisons will be given below.

6.4 Matrix-vector multiplication

For the matrix-vector multiplication (step `matvec` in Sect. 6.1.1) we use the Level-2 BLAS routine `DGEMV` from the freely available GotoBLAS2 library, the source code of which is distributed by the Texas Advanced Computing Centre, (http://www.tacc.utexas.edu/).

When compiled with OpenMP support, the GotoBLAS2 implementation of `DGEMV` shows superior parallel performance for our application when compared with either of the FORTRAN 90 intrinsic procedure `matmul`, with a (naively) parallelized implementation as a straightforward double loop, or even with optimized `DGEMV` implementations taken from latest versions of the commercial numerical libraries MKL or NAG.

We note that the GotoBLAS2 library is no longer under active development but we expect the algorithms or at least the essential ideas to be adopted by relevant Open-Source, high-performance BLAS-implementation

projects like ATLAS (http://math-atlas.sourceforge.net/) or OpenBLAS[3] (https://github.com/xianyi/OpenBLAS/wiki).

6.5 Computational performance

6.5.1 Benchmark platform

Our primary hardware platform for performing the benchmarks we report in the following is a standard two-socket compute server (HP ProLiant DL370 G6) hosted by RZG. The server is equipped with two Intel Xeon W5580 quad-core CPUs with a clock frequency of 3.2 GHz. We use a standard Intel software stack (Intel C and Fortran compiler suite, version 12.1 and — optionally — the Math Kernel Library, MKL 10.3) on top of the Novell/SUSE SLES11 Linux operating system. All required software is part of the standard software stack available on RZG Linux systems. We note that our code and the libraries FFTW and GotoBLAS can also be compiled using the GNU compiler collection (gcc) which points out the option to base the entire application on a completely free and Open-Source software stack. However, we have not yet made any attempts to optimize our application for gcc. When compiled with gcc (4.6.0), the performance currently falls behind the numbers obtained using Intel's compilers by roughly a factor of two.

At runtime we pin OpenMP threads to the available physical processor cores using environment settings of the Intel runtime. We achieved best performance by setting `KMP_AFFINITY="granularity=compact"`, which, for example pins 4 OpenMP threads to a *single* quad-core CPU on our two-socket server. In order to minimize OpenMP overhead we specify static loop scheduling by setting `OMP_SCHEDULE=static`. Furthermore, in order to prohibit that temporarily idling threads are recurrently "freed" and have to be reclaimed from the operating system again we set `OMP_WAIT_POLICY=active`. The latter two environment settings are part of the OpenMP standard. When working with a non-Intel software stack (e.g. GNU gcc) the proprietary `KMP_AFFINITY` setting could be replaced, e.g. by employing standard GNU tools like `numactl` for controlling the policy for pinning processes and shared memory.

[3]The OpenBLAS project has taken over the GotoBLAS2 source code. In fact, the current version 0.1alpha of OpenBLAS shows the same `DGEMV` performance in our application as GotoBLAS2.

6.5.2 Overview of benchmark setup and results

We consider a numerical grid with $M \times N = 64 \times 128$ zones as our standard case. The corresponding resolution is generally regarded as sufficient for real-time applications in the present context. Using all four cores of the 3.2 GHz CPU of our benchmark platform we achieve a floating point performance of almost 7 Gflops, resulting in a total runtime of 0.114 ms for a single call of the solver (i.e. a single "outer" iteration step). Computing the DSTs with Intel's proprietary MKL library instead of FFTW would account for a 20% speedup, i.e. a total runtime of 0.093 ms.

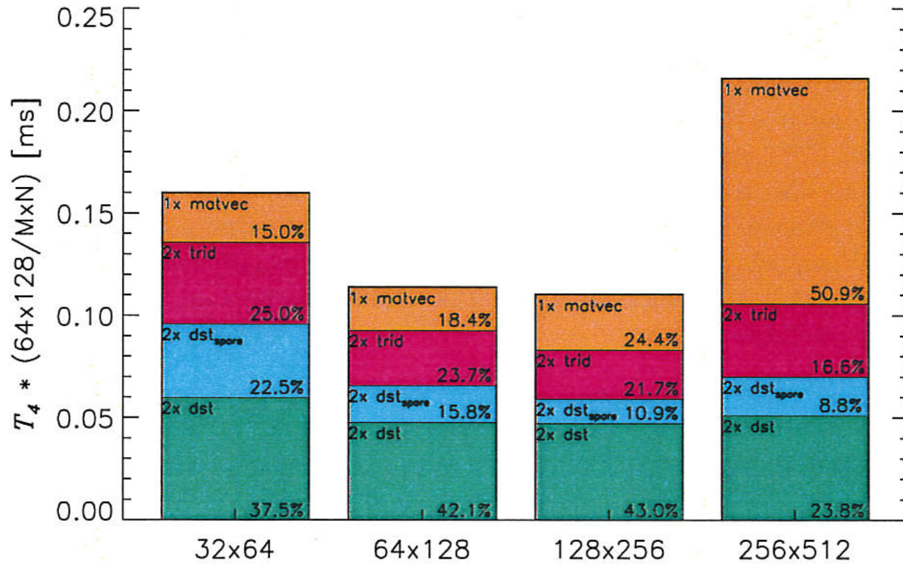

Figure 2: Parallel runtimes T_4 and subroutine breakdown using all four cores of the 3.2 GHz CPU Xeon W5580. All runtimes were normalized to the total number of grid points. The "standard" grid with $M \times N = 64 \times 128$ was chosen as a reference value. Hence, for the other grid dimensions 32×64, 128×256, or 256×512 the actual runtimes can be obtained by multiplication with factors 1/4, 4, or 16, respectively.

In the following we shall present more detailed timings for the standard 64×128 grid. For comparison selected timings are shown also for smaller (32×64), and larger (128×256, 256×512) grid sizes. Unless explicitly stated otherwise all benchmark numbers reported refer to a *single* so-called "outer" iteration step of Lackner's Algorithm (Sect. 3). All timings exclude

any "initialization", i.e. operations which can be precalculated and stored in memory before the sequence of "outer" iterations is computed in a real-time application. This comprises, for example, the setup of the numerical grid and derived quantities like s^+, s^-, δ (Sect. 4), computing the LU decomposition for the tridiagonal solver (Sect. 6.3) or the FFTW-plans (Sect. 6.2). For the runtimes we noticed variations on the order of 10% when executing the program several times in a row. This issue has not yet been analyzed and needs to be revisited when the application is finally going to be prepared for implementation, e.g. at the ASDEX experiment. We expect that specific tuning of the operating system parameters will help alleviating the problem on a particular target compute server.

Figure 2 and Table 1 provide the basic overview of the total runtimes of the solver for different sizes of the numerical grid. Figure 2, in addition, shows the contribution of the individual subroutines to the total runtime. Table 1 also compares the total runtimes obtained on our standard `Xeon W5580` system with CPU models with lower clock frequency and different number of cores (and also memory bandwidth). In all cases we chose the number of OpenMP threads equal to the number n of physical cores a single CPU provides. When using more cores the parallel efficiency usually becomes poor unless comparably large problem sizes (128×256 or larger) are considered (cf. Fig. 3 and Table 3).

grid			32×64	64×128	128×256	256×512
		n	T_n [ms]	T_n [ms]	T_n [ms]	T_n [ms]
`Xeon W5580`	3.2 GHz	4	0.040	0.114	0.432	3.550
`Xeon E5540`	2.7 GHz	4	0.043	0.130	0.489	3.163
`Xeon X7542`	2.7 GHz	6	0.048	0.113	0.343	3.379

Table 1: *Parallel* execution times, T_n using a number n of cores for different sizes, $M \times N$, of the numerical grid. The table also compares the 3.2 GHz CPU of our standard benchmark platform with the more moderately clocked Intel Xeon E5540 (quadcore) and X7542 (hexacore) CPUs, both running at 2.7 GHz (the Xeon E5540 CPU, which has a nominal clock frequency of 2.53 GHz is operated in "turbo mode").

Overall, the measured runtimes roughly reflect the theoretical compute performance (clock frequency times number of cores) of the different CPU models. We observe a few exceptions from this general trend which, in principle can be explained by taking into account differences in the memory bandwidth delivered by the CPUs[4], and also load imbalances for different

[4]According to the STREAM benchmark the memory bandwidth of the *faster* Xeon

combinations of n and $M \times N$ (cf. Sects. 6.2, 6.3). In the following we will mostly confine our analysis to the standard `Xeon W5580` benchmark platform.

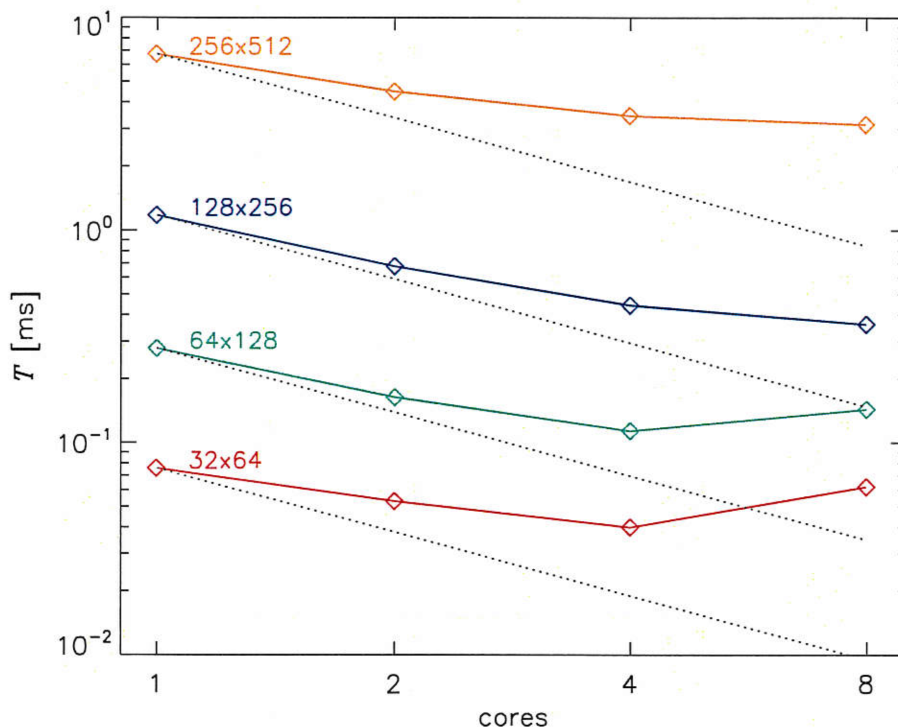

Figure 3: Overview of the parallel execution times, T_n as a function of the number n of cores for different sizes, $M \times N$, of the numerical grid. Runtimes were measured on the 3.2 GHz CPU Xeon W5580. The dotted straight lines indicate ideal parallel scaling (corresponding to a parallel efficiency of $\eta = 100\%$).

Table 2 and Figure 2 summarize the contribution of the major individual components of the solver in absolute terms and relative to the solver's runtime. Table 2 shows serial runtimes, Figure 2 shows parallel runtimes obtained on four cores. Note that here and also in Table 3 multiple contributions of the same algorithmic steps defined in Sect. 6.1.1 were added up, i.e. $T(2\times\mathrm{dst}) \equiv T(\mathrm{dst}_1)+T(\mathrm{dst}_2)$, $T(2\times\mathrm{dst}_{\mathrm{spare}}) \equiv T(\mathrm{dst}_{\mathrm{spare1}})+T(\mathrm{dst}_{\mathrm{spare2}})$, and $T(2 \times \mathrm{trid}) \equiv T(\mathrm{trid}_1) + T(\mathrm{trid}_2)$. Notably, already the serial algorithm performs very well. Runtimes for moderate grid sizes (64×128 or smaller)

W5580 CPU is actually 10% *smaller* when compared with the Xeon E5540 model.

grid	32 × 64		64 × 128		128 × 256		256 × 512	
	T [ms]	%	T [ms]	%	T [ms]	%	T [ms]	%
2×`dst`	0.036	48.4	0.149	53.4	0.653	55.2	2.999	44.4
2×$\texttt{dst}_{\text{spare}}$	0.012	16.3	0.032	11.5	0.117	9.9	0.577	8.5
2×`trid`	0.013	16.8	0.048	17.2	0.191	16.2	1.112	16.4
1×`matvec`	0.014	17.9	0.049	17.6	0.220	18.6	2.074	30.7
solver	0.076	100.0	0.279	100.0	1.182	100.0	6.764	100.0

Table 2: *Single-core* runtimes T and subroutine breakdown for different sizes $M \times N$ of the numerical grid.

are well below the millisecond range, i.e. already with a single core it would be possible to run a couple of outer iteration steps within a typical control cycle of the experiment. Table 2 and Figure 2 also shows that the DSTs consume the largest fraction of the computing time which justifies our particular choice of the data layout (Sect. 6.1.3). For very large grids the matrix vector multiplication becomes the dominant component.

The transformations which we employ for steps $\texttt{dst}_{\text{spare1}}$ and $\texttt{dst}_{\text{spare2}}$ (Sect. 6.2) are a factor of 3 to 5 faster when compared with a naive implementation which would require another two complete sets of $(M-1)$ DSTs of size $(N-1)$. This can be seen by comparing rows labelled 2×$\texttt{dst}_{\text{spare}}$ and 2×`dst` in Table 2 (see also Table 3).

Within individual rows of Table 2 we find the runtimes to roughly quadruple with doubling both values, M and N. This is in accordance with the computational complexity of the underlying algorithms, which are $\mathcal{O}(N \cdot M)$ for the tridiagonal systems, $\mathcal{O}(M \cdot N \log N)$ for the DSTs, and $\mathcal{O}((N+M)^2)$ for the matrix-vector multiplication, respectively. For the largest grid-size one notes deviations from this trend, which is due to the computations becoming memory bound.

Focusing on a grid size of 64×128, Table 3 shows the parallel scaling behaviour of the individual components of the algorithm. For all components a rather good parallel efficiency is achieved up to the maximum of 4 cores provided by a single CPU (see also Fig. 3). Although there are no relevant serial parts remaining in the algorithm or our implementation, Figure 3 indicates some deviation from the ideal scaling curve (which corresponds to a parallel efficiency $\eta = 100\%$). This behaviour is rather explained by the overhead introduced by OpenMP parallelization (mostly setup and synchronization of threads in the parallel loops) and by the fact that at most 2 threads can be utilized in steps $\texttt{dst}_{\text{spare1}}$ and $\texttt{dst}_{\text{spare2}}$.
The significant decrease of the parallel efficiency from 4 to 8 cores (see also

cores	1		2		4		8	
	T_1 [ms]	η	T_2 [ms]	η	T_4 [ms]	η	T_8 [ms]	η
2×`dst`	0.149	1.00	0.081	0.93	0.048	0.78	0.062	0.30
2×`dst`$_{\text{spare}}$	0.032	1.00	0.021	0.76	0.017	0.47	0.026	0.15
2×`trid`	0.048	1.00	0.033	0.73	0.027	0.44	0.038	0.16
1×`matvec`	0.049	1.00	0.028	0.87	0.021	0.58	0.018	0.34
solver	0.279	1.00	0.164	0.85	0.114	0.61	0.144	0.24

Table 3: Parallel runtimes, T_n for different numbers n of cores, and parallel efficiency, η, conventionally defined as $\eta \equiv T_1/(n \cdot T_n)$ for grid size 64×128.

Fig. 3) is caused by a combination of two effects. First, the OpenMP overhead gets larger relative to the total computational work performed by a thread. Second, NUMA effects (i.e. slower access to memory and cache of the "remote" CPU) become relevant when both CPUs of the two-socket server are utilized by 8 threads[5]. Both effects, OpenMP overhead and memory bottlenecks could probably be further reduced by performing elaborate profiling and tuning of the program. For the time being we refrain from spending considerable efforts on such further tuning, noting that in the final application scenario multiple CPUs can presumably be utilized very efficiently by running multiple independent copies of the solver, where each instance corresponds to a different basis function. Assuming perfect parallelization, which is not unreasonable since only minimal inter-CPU communication is required, a server with four quadcore CPUs of present-day technology would, for example, be able to compute $1\,\text{ms}/(8 \cdot 0.114\,\text{ms}/4) \approx 4$ outer iteration steps within a millisecond, provided that $N_p + N_F = 8$ basis functions and a numerical grid of size 64×128 ($T_4 = 0.114\,\text{ms}$) are chosen, or 12 outer iterations for a grid with 32×64 zones ($T_4 = 0.04\,\text{ms}$).

6.5.3 Performance of the DST

Our measurements show that for the grid sizes considered (N and M correspond to the length of an individual DST, and to their total number, respectively) the DST routines provided by Intel's proprietary Math Kernel Library (MKL 10.3) are faster by approximately a factor of 1.5 compared to FFTW. Both, MKL and FFTW routines show roughly the same parallel scaling behavior with increasing number of cores. Overall, this translates to a 20% performance improvement of the complete solver when using the DSTs from MKL (Table 4).
As expected, the routine `sinft` from Numerical Recipes is not competitive

[5]Test runs on an Intel 10-core CPU Xeon E7 8870 indeed showed that in the absence

grid ($M \times N$)			32×64	64×128	128×256	256×512
		n	T_n [ms]	T_n [ms]	T_n [ms]	T_n [ms]
`Xeon W5580`	3.2 GHz	4	0.034	0.093	0.372	2.861
`Xeon E5540`	2.7 GHz	4	0.033	0.092	0.346	3.295
`Xeon X7542`	2.7 GHz	6	0.043	0.091	0.283	3.188

Table 4: Same as Table 1, but showing the performance of the solver using DSTs from the Intel Math Kernel Library (MKL 10.3) instead of FFTW.

performancewise. It was implemented rather for comparing the numerical accuracy delivered by different DST libraries and can also serve as fallback for building the solver, as the source code for `sinft` and all dependencies from Numerical Recipes are packaged with our code.

We intentionally refrain here from providing more systematic benchmark numbers for comparing DSTs from Intel MKL and FFTW, because our experiments have shown that the single-core performance and in particular the parallel performance is highly sensitive to the size of the individual transforms and to subtleties of the OpenMP parallelization, respectively. As an example, we note that we were successful to achieve decent parallel efficiency with the DSTs from MKL only if the work arrays were declared as "threadprivate" (in the OpenMP sense) module variables. Using instead a local variable with private scope in the OpenMP parallel region or loop, which is programmatically apparently equivalent we noticed very poor scaling of MKL's DSTs in our application.

Finally, we note that even for the smallest grid size considered here (32×64) a direct implementation of Eqs. (19) and (20) as a matrix multiplication using a multi-threaded level-3 BLAS routine `DGEMM` with a precomputed coefficient matrix, $\sin(\pi j l/N)$ does not outperform FFT-based DSTs taken from platform-optimized libraries like Intel MKL or FFTW.

6.5.4 Performance of the tridiagonal solver

Unlike in the case of DST (see previous subsection) a meaningful and unbiased comparison of the performance of our new tridiagonal solver with alternative routines from Intel MKL, Numerical Recipes or alike is not possible due to the particular data layout used in our program. Specifically, our choice of dimensioning arrays as `x(N,M)` entails memory access with large strides when employing a generic tridiagonal solver routine like `DDTTRSB` (Intel MKL) or `tridag` (Numerical Recipes). For example, the call sequence for

of NUMA positive parallel speedups are obtained up to the maximum of 10 threads.

the DDTTRSB routine,

```
do l = 1,N
  call DDTTRSB('N',M,1,dl(l,:),d(l,:),du(l,:),b(l,:),1,info)
enddo
```

where dl, d, du and b are the $N \times M$ arrays corresponding to the LU decomposition (computed by DDTTRFB) and the right hand side/solution vector, respectively, would obviously be a very inefficient one. Hence, this setting does not allow a fair comparison with our own solver, which in turn is tailored exactly towards such an "unfavourable" data layout. In short, DDTTRSB is not competitive in our application, although the routine as such is highly optimized for a standard application scenario[6].

	$T_{(1\times128)}$ [ms]	$T_{(16\times128)}$ [ms]	gain	$T_{(64\times128)}$ [ms]	gain
VTRS	0.0017	0.0063	4.3	0.021	5.2
DDTTRSB (MKL)	0.0008	0.0131	–	0.053	–
tridag (NR)	0.0022	0.0339	–	0.148	–
vtridag	0.0025	0.0097	4.1	0.037	4.3

Table 5: Serial execution times for different tridiagonal solvers. The "gain" measure is defined as $k \cdot T_{(1\times128)}/T_{(k\times128)}$, where k is the number of independent tridiagonal systems to be solved.

In order to still demonstrate the potential of our solver in an unbiased as possible way we provide here benchmark numbers in an artificial setting, which grants each routine the type of data layout which is natural and presumably optimal.

In this setting, Table 5 compares serial execution times of our vectorized solver for multiple systems (named VTRS) with those of Intel's MKL (DDTTRSB). Both routines rely on a precomputed LU decomposition and do not perform row or column interchanges due to pivoting. For reference we also provide numbers for the "standalone" solvers (i.e. no precomputed LU decomposition available) from Numerical Recipes (tridag) and its "vectorized" variant for multiple systems (named vtridag) which was also implemented by us (cf. Sect. 6.3).

The three columns of Table 5 show serial execution times for solving different numbers (k) of independent tridiagonal systems of size 128 each.

[6]Routines like DDTTRSB are commonly designed for computing the backsubstitution of multiple right hand sides, given a single LU decomposition. In our application, by contrast, each right hand side comes with a different LU decomposition.

For a single system ($k = 1$), the platform-optimized MKL routine `DDTTRSB` outperforms our solver `VTRS` by a factor of two. However, unlike `DDTTRSB` the serial execution time of `VTRS` does *not* increase proportional to the number of systems. For example, when solving $k = 16$ tridiagonal systems, `VTRS` is more than twice as fast as the MKL routine owing to gains of more than a factor of 4 due to SIMD-vectorization and better cache locality.

As documented by the lower two lines of Table 5 the Numerical Recipes routine `tridag` is not competitive, performancewise. Our "vectorized" variant for multiple systems (`vtridag`), however, outperforms the MKL routine (`DDTTRSB`) already for $k = 16$ systems, even though the latter saves on the order of 50% of the operations by making use of a precomputed LU decomposition. In analogy to the DSTs, support for the original Numerical Recipes routine, as well as for our variant `vtridag` was implemented mainly for reasons of numerical reference and as a fallback.

6.6 Code validation

Throughout its development the solver was embedded in a program environment which resembles the actual call to the equilibrium calculation realized as a subroutine in the CLISTE program (subroutine `EQUIL`). This was achieved by storing the contents of all variables used by CLISTE when calling this subroutine. The final state of all variables after processing the CLISTE subroutine was stored as well and used for the code validation. The differences in the results of the old CLISTE subroutine and the new solver were within computational accuracy, i.e. maximum relative deviations are on the order of 10^{-12}.

6.7 Experiments with general-purpose graphics processing units (GPU)

In the context of an earlier master's thesis project [16] co-supervised by three of us (U. v.T., M. R., R. F.), the suitability of the parallel algorithms described above for implementation on general-purpose graphics processing units (GPU) was assessed. Such devices could in principle serve as a more powerful and also cost-efficient alternative to a standard CPU server.

Using NVIDIA's popular CUDA programming model, different algorithmic variants of the main components, namely the discrete sine transform (DST) and the tridiagonal linear solver, were implemented and tested on a NVIDIA C1060 GPU and also on its successor, the C2050 model, called "Fermi". Here, we quote results only for the latter platform, which is sig-

nificantly more powerful, in particular for performing double precision numerics, which is the arithmetic precision that is relevant for our application. Using parallel cyclic reduction [17] for the tridiagonal systems and NVIDIA's CUFFT library (available at www.nvidia.com) for the DSTs, runtimes for a single solver step on the order of a millisecond were achieved for grid sizes ranging between 32×64 and 128×256. This is a factor of 10 slower compared to the performance of our solver on a quad-core CPU (see above), not even counting the time which is spent for the data transfer between the host CPU to the GPU. Given the nominal peak performance (double precision) of 500 Gflops of the C2050 GPU (vs. 50 Gflops for the CPU), this is clearly a disappointing result. It can be explained by the fact that in our application the size of the problem resp. the compute intensity is simply too small to overcome various latencies, e.g. for starting and synchronizing individual compute kernels or memory accesses on the GPU. In fact, only when we artificially increased the problem size (e.g. computing thousands of DSTs of size 128) the high thread-concurrency of the GPU could be fully exploited and decent performance values on the order of hundreds of Gflops were obtained. Such problem sizes could in principle arise in practical applications if very fine numerical resolution and a large number of basis functions were demanded for high accuracy. While in such cases the GPU might be able to significantly outperform the CPU in terms of operations per second (Gflops), absolute runtimes clearly are prohibitively large for real-time applications.

7 Summary

We presented GPEC, a numerically optimized solver for the two-dimensional Poisson problem posed by the Grad-Shafranov equation describing magnetohydrodynamic equilibrium in tokamaks like ASDEX Upgrade. GPEC is implemented in FORTRAN 90 and relies solely on open source software components. It is therefore considered sustainable and meets the software policy of ITER. The new code has been validated with results from the currently used equilibrium code CLISTE.

The parallel capabilities of common multicore processors motivated to employ a solution method developed nearly fifty years ago which consists of the combination of Fourier transformation and solutions of tridiagonal linear systems. Both methods in their optimized form, i.e. fast Fourier transformation taken from an optimized numerical library and a SIMD-vectorized solver for a set of tridiagonal system of equations which was developed by us, together with OpenMP parallelization of the numerical tasks enabled to achieve execution times of about 0.04 ms on quad-core x86 processors (Intel

Nehalem, 3.2 GHz) for a grid size of 32×64 which is considered to be sufficiently large to derive quantities for plasma control. This runtime is way below of the control cycle for fusion machines which is typically in the millisecond time range and hence allows for a significant enhancement of the operational regime for real-time plasma control. With doubling both grid dimensions the runtimes quadruple, which is in accordance with the computational complexity of the algorithm. GPEC shows good parallel efficiency up to the total number of cores of a single CPU socket. NUMA effects and – to a lesser degree – parallel overhead deteriorate the efficiency of parallel scaling beyond a single socket. However, this drawback can be compensated by the utilization of the solver in equilibrium codes (which is the primary application of the new solver) where a number of solutions of the Grad-Shafranov equation are computed independently for a whole set of basis functions. This allows to efficiently utilize multiple CPU sockets in parallel as shown by a first prototype implementation (to be described elsewhere) of GPEC into the equilibrium code "IDE" [18] using the message-passing interface (MPI) standard. For a grid size of 32×64 the computation of an outer iteration takes approx. 0.08 ms per basis function on a single CPU. In the MPI-parallel equilibrium code this runtime is expected to scale with the number of basis functions divided by the number of CPU sockets which, for example, readily enables real-time applications using eight basis functions and four CPUs. Moreover, this strategy of parallelizing the Grad-Shafranov solver for the multicore architecture of a single socket and distributing the basis functions across multiple sockets opens up the possibility to calculate highly resolving grids (say 128×256) and a large number of basis functions (say up to the order of 30) within reasonable time, using an adequately equipped, but still mainstream computer system. This will enable to address scientific problems relying on high spatial resolution with moderate hardware costs.

8 Outlook to the application of GPEC at ASDEX Upgrade

There are two options for real-time realization of plasma control in an environment such as ASDEX Upgrade: First, as a real-time diagnostic and, second, fully incorporated into the control system. The first option is to run the evaluation of the equilibrium as a stand-alone diagnostic with data transfer from and to the control system. This is – based on a Labview system [11] – currently implemented and further developed at ASDEX Upgrade for the control of NTM instabilities. The equilibrium data, e.g. the position

of q-surfaces, are transferred in real-time, which means here within a few control cycles, to the control system for further evaluation of the steering parameters of the electron cyclotron heating mirrors.

The second option is to fully include the real-time evaluation of the equilibrium into the control system running the plasma discharge. This has the advantage that the data transfer over the heavily loaded network is reduced and that much more information can be provided and used for plasma control. This additional information can be detailed equilibrium information which is otherwise limited by the network capabilities, reliability measures of the equilibrium, and measures of failure of gradually degrading measured signals for automatically driven tools for robust diagnostic data analysis.

The real-time equilibrium needed for plasma control is calculated currently at ASDEX Upgrade with a function parametrization (FP) algorithm [19]. This FP algorithm is known to be fast and robust as long as the magnetic data are reliable and no degradation or failure of individual magnetic data occur. In the case of failure of magnetic data the FP coefficients have to be re-analysed which has to be done off-line. Moreover, the FP equilibrium is known to deviate from the equilibrium computed with the Grad-Shafranov solver. Though this can usually be tolerated for plasma position control, the deviances might no longer be acceptable for more sophisticated applications depending on, e.g. well known plasma core and edge parameters. This emphasizes the need for a real-time Grad-Shafranov solver to support or replace the FP equilibrium in advanced applications of plasma control.

Recently, there is a lot of progress in the development of real-time diagnostics, e.g. for profile estimation. On the one hand, these real-time diagnostics can provide input data, such as kinetic profiles and fast particle pressure, for the real-time Grad-Shafranov solver. On the other hand, the real-time interpretation of these diagnostics data is expected to benefit from an improved real-time equilibrium.

The reliability of the real-time equilibrium will depend on the settings of the grid size, the number of basis functions, and the convergence criteria chosen. The plasma position, such as the position of the magnetic centre and the separatrix contour, can be well determined for most purposes with a reduced grid size of 32×64 and a reduced set of four to ten basis functions. The number of outer iterations required for convergence usually is small in cases of stationary or slowly changing plasmas and a good initial guess of the current distribution. But for the start-up, ramp-down, and transient plasma phases on the order of 10-50 outer iterations might be necessary. For high-resolution edge pressure or edge current distributions the number of basis functions is typically in the order of 25 and the typical grid size needed is 128×256. Due to the parallelization concept introduced in this report,

such an increase in the number of basis functions can be compensated by an increase in the number of CPUs employed. Challenging settings can be realized with a sufficient number of CPU-sockets. Todays standard Linux-clusters do provide such an environment.

9 Acknowledgement

We are grateful for discussions with C. Fuchs, R. Hatzky, J. Hobirk, K. Lackner and W. Suttrop. Thanks to P. Martin for providing a subversion of the CLISTE code with which this study started and to S. Gori for the latest FORTRAN 90-version. We would also like to thank R. Tisma for his help on integrating DST routines. We thank I. Zhukov for his master thesis work on the GPU implementation and K. Reuter and J. Hacker (TU München) for supporting the thesis project.

References

[1] M. Maraschek, G. Gantenbein, T. P. Goodman, S. Günter, D. F. Howell, F. Leuterer, A. Mück, O. Sauter, and H. Zohm. Active control of MHD instabilities by ECCD in ASDEX upgrade. *Nucl. Fusion*, 45:1369, 2005.

[2] L. L. Lao, J. R. Ferron, R. J. Groebner, W. Howl, H. St. John, E. J. Strait, and T. S. Taylor. *Nucl. Fusion*, 26:1035, 1990.

[3] K. Lackner. Computation of ideal MHD equilibria. *Comp. Phys. Comm.*, 12:33, 1976.

[4] J. Hugill and J. Sheffield. *Nucl. Fusion*, 18:15, 1978.

[5] V. D. Shafranov. *Zh. Eksp. Teor. Fiz.*, 33:710, 1957.

[6] R. Lüst and A. Schlüter. *Z. Naturforsch.*, 129:850, 1957.

[7] P. J. McCarthy, P. Martin, and W. Schneider. The CLISTE interpretive equilibrium code. Technical Report IPP 5/85, Max-Planck-Institut für Plasmaphysik, 1999.

[8] O. Buneman. Technical Report SUIPR 294, Standford, 1967.

[9] B. L. Buzbee, F. W. Dorr, J. A. George, and G. H. Golub. *SIAM J. Num. Analysis*, 8:722, 1971.

[10] K. v. Hagenow and K. Lackner. In *Proc. 7th Conf. Num. Sim. of Plasmas*, page 140, 1975.

[11] L. Giannone, R. Fischer, K. Lackner, Q. Ruan, A. Veeramani, M. Cerna, J. Nagle, M. Ravindran, D. Schmidt, A. Vrancic, and L. Wenzel. Computational strategies in optimizing a real-time grad-shafranov pde solver using high-level graphical programming and cots technology. 2010.

[12] John Reid. Coarrays in the next fortran standard. *SIGPLAN Fortran Forum*, 29:10–27, July 2010.

[13] J. M. Bull. Measuring synchronisation and scheduling overheads in OpenMP. In *In Proceedings of First European Workshop on OpenMP*, pages 99–105, 1999.

[14] W.H. Press et al. *Numerical Recipes.* Cambridge University Press, 2007.

[15] M. Rampp and H.-Th. Janka. Radiation hydrodynamics with neutrinos: Variable eddington factor method for core-collapse supernova simulations. *Astronomy & Astrophysics*, 396:361–392, 2002.

[16] Ilya Zhukov. Development of a grad-shafranov equation solver for GPGPUs. Master's thesis, TU München, 2010.

[17] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the GPU. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 127–136, New York, NY, USA, 2010. ACM.

[18] R. Fischer. Integrated data analysis with equilibrium. In *Proceedings of the 7th Workshop on Fusion Data Processing Validation and Analysis*, 2012. In preparation.

[19] P. J. McCarthy. An integrated data interpretation system for tokamak discharges. Technical report, University College Cork, 1992. PhD Thesis.