F. Merz, J. Geiger, M. Rampp

Optimization strategy for the VMEC stellarator equilibrium code

IPP R/48
October 2013

Optimization strategy for the VMEC stellarator equilibrium code

F. Merz[1], J. Geiger[2], M. Rampp[3]

[1] IBM Germany
[2] Max-Planck-Institut für Plasmaphysik,
EURATOM Association,
Boltzmannstr. 2, 85748 Garching, Germany
[3] Rechenzentrum (RZG) der Max-Planck-Gesellschaft
und des Max-Planck-Institut für Plasmaphysik

October 2, 2013

Abstract

VMEC (Variational Moments Equilibrium Code) [1, 2] is the main workhorse for computing three-dimensional MHD equilibria in stellarator experiments such as Wendelstein 7-X (W7-X). There is a great interest in the community for significantly reducing the runtimes of individual VMEC simulations which, for typical setups, can range up to hours of computing time on modern processors. In particular, the ability to enter the regime of "real-time" diagnostics during the operation of the W7-X machine is considered highly desirable. Here, we present the results from the assessment and prototypical optimization of the computational performance of VMEC and propose a strategy for adapting the serial code to modern multicore-processor architectures. Starting off from the most recent VMEC version 8.49 we shall demonstrate that up to threefold speedups can be readily obtained for the most time-consuming routines by simply eliminating legacy program structures which presumably were dictated by the prevalence of vector supercomputers back in the 1980's when VMEC was originally written. As a side effect, the code regains readability which had to be sacrificed for achieving high performance on traditional vector processors.

Once the structure has been updated, the code is amenable to parallelization using threads (OpenMP) and message pass-

ing (MPI). With an OpenMP parallelization of the relevant subroutines we achieve speedups by a factor of 10 on a modern Intel Xeon processor with eight cores, when comparing with the original code executed on the same hardware and using "real-world" parameter sets. On top of these optimizations we shall outline a second level of restructuring which is based on a transposed data layout for the three-dimensional physical domain. This will open up the possibility to implement a hybrid MPI/OpenMP parallelization which allows to distribute a VMEC run across multiple nodes of a compute cluster and thus to gain another order of magnitude in computational performance.

Although conceptually straightforward, the implementation of these concepts throughout the entire VMEC code will clearly be a tedious and time-consuming task and requires careful planning, code management and thorough validation strategies.

1 Introduction

The development of the **V**ariational **M**oments **E**quilibrium **C**ode VMEC [1, 2] goes back to the early 1980's. The code employs the MHD energy principle and utilizes a variational method to derive the equilibrium equations in a conservative form. These equations are solved using a steepest descent algorithm based on the assumption of toroidally nested flux surfaces. Thus, the code does not treat islands or stochastic field regions in the magnetic configuration. The necessary physics input for the code consists of two profiles to be provided as functions of the flux surface label, i.e. the pressure profile and either the profile of the toroidal current or that of the rotational transform. Additionally, a boundary surface must be supplied which VMEC takes as last flux surface of the computational domain as well as a value for the total toroidal flux enclosed by the boundary. The code can run in "fixed-boundary" mode which means that the geometry of the boundary does not change during the iteration steps. The other possibility is to also allow the boundary to adjust to an externally given magnetic field. This so-called free-boundary mode needs the additional specification of the external magnetic field.

During the past 30 years a lot of work was invested into the code for improving performance and robustness and to broaden its applicability with respect to 3D-MHD equilibrium problems. In particular due to its robustness and reliability VMEC has become rather popular for stellarator applications. It has also been a centerpiece of the stellarator optimization efforts which started in the 1980's. Earlier efforts to optimize the computational performance, however, were targeted at computer architectures of the pre-multicore era, i.e. vector computers of the traditional kind [3, 4]. Today, VMEC is used at virtually all stellarator experiments around the world, although new 3D-MHD equilibrium codes have also been developed, mainly to avoid the assumption of nested flux surfaces and to be able to treat islands and stochastic regions in the magnetic field configuration [5, 6, 7, 8]. However, these codes are computationally very demanding, and hence their ability to supply large numbers of equilibrium solutions as required for evaluating many experiments is limited.

To date, VMEC has been employed as a post-processing tool to evaluate MHD equilibria for experiments. Runtime was not a major issue as parameter studies, i.e. the computation of sets of equilibria with different input parameters, could exploit available computing resources simply by distributing many individual (and mutually independent) VMEC runs to different processors. Such "embarrasingly parallel" computations, however, are no longer relevant if quasi-steady conditions in the operation of stellarators apply and equilibrium calculations need to be performed during long discharges — for the Wendelstein 7-X (W7-X) machine the aim is 30 minutes. In such an application scenario, the runtime of an individual VMEC simulation needs to be significantly reduced. Currently, typical runtimes for computing W7-X equilibria with the serial VMEC code range from 30 minutes to a few hours, depending on the parameter settings and the employed hardware.

This report provides an assessment of the computational performance of the VMEC code and presents results from a number of prototypical optimizations. Specifically, we shall report on our efforts to optimize VMEC with respect to its serial (”single-core”) performance, demonstrate the potential for parallelization on contemporary multicore processors and outline a strategy for distributed parallelization. So far the assessment and optimization is confined to the current implementation of the code (VMEC 8.49), focussing on the most time consuming subroutines and using relevant test setups that are used in production for W7-X analysis at IPP. Possible algorithmic optimizations are beyond the scope of this study but should certainly not be disregarded as an option for further speeding up the VMEC code.

2 Assessment and optimization of computational performance

2.1 Preliminaries

VMEC in its present version has been implemented with an apparent focus on achieving maximum performance on traditional vector computers. The data is stored in three dimensional arrays corresponding to (s, m, n) or (s, θ, ζ) coordinates, where s is the flux surface label and (m, n) and (θ, ζ) are the two periodic dimensions (poloidal, toroidal) on a given flux surface in Fourier space and direct space representations, respectively [9]. VMEC is a Fortran code, which means that the first index of an array is the ’fastest varying’ index, i.e. it labels contiguous memory locations. In most subroutines, the three-dimensional arrays ($3\mathrm{D}_\mathrm{p}$), corresponding to the three-dimensional physical domain are mapped (”serialized”) to one-dimensional arrays ($1\mathrm{D}_\mathrm{n}$) and these arrays are manipulated (added, multiplied, . . .) as a whole[1]. This

[1]The subscripts p and n distinguish between the original, ”physical” dimension of an array and its dimension used in the ”numerical” implementation, respectively.

increases the so-called vector length which was crucial for achieving good performance on traditional vector processors, but — as we shall show below — turns out to be detrimental for the performance on modern, cache-based, multicore processors.

In a first step we will demonstrate that already by abandoning the serialization of the $3\mathrm{D}_\mathrm{p}$ arrays and introducing loops over the new dimensions, appreciable performance improvements can be achieved. In a second step we will assess the parallelizability of the VMEC code. We note that the code employs a steepest-descent algorithm for computing the equilibrium solution of the given boundary value problem. Numerically, this leads to a "pseudo-time" evolution similar to solving an initial-value problem. As a consequence, parallelization in the code is performed within a single iteration step.

2.2 Test cases and hardware platform

To evaluate the performance and the potential for optimization, we use two different test setups which are representative for applications at W7-X.

Case 1 represents the currently used standard grid resolution for free-boundary calculations for W7-X, namely, $ns = 99, nu = 30, nv = 36, mpol = 12, ntor = 12$. Here, ns is the number of flux surfaces, nu and nv are the numbers of poloidal and toroidal grid points for the direct-space representation and $mpol$ and $ntor$ are the parameters for the Fourier representation. To be specific, the range of Fourier modes for this parameter set is $m = 0, \dots, mpol - 1$ and $n = -ntor, \dots, ntor$.

Case 2 utilizes the parameter set of a fixed boundary calculation in which the main emphasis is on a high resolution in the angle variables. The parameters used are $ns = 99, nu = 60, nv = 60, mpol = 20, ntor = 20$. Note that the speed of convergence, i.e. the number of iterations needed to reach the same convergence in terms of the force tolerance levels, decreases with increasing resolution due to timestep restrictions.

The measurements were performed on a compute server with two eight-core Intel "Sandy Bridge" sockets (Xeon E5-2670 CPU with a frequency of 2.6 GHz and 20 MBytes of L3 cache). The configuration of this server is very typical for compute nodes of state-of-the-art HPC clusters with x86 architecture such as the HPC system "Hydra" operated by RZG.

2.3 Optimization of prototypical routines

The VMEC internal timings showed, that the major fraction of the computing time is spent in four routines: `TOTZSP` and `TOMNSP`, which implement Fourier transforms, and the `BCOVAR` and `FORCES` routines (see Table 1).

	Case 1		**Case 2**	
Code section	T [s]	%	T [s]	%
VACUUM LOOP	142.75	33.9	-	-
TOTZSP	84.57	20.1	727.17	35.8
TOMNSP	78.78	18.7	662.61	32,6
FORCES	33.13	7.9	183.45	9.0
BCOVAR	45.30	10.8	233.86	11.5
RESIDUE	7.98	2.0	30.11	1.5
(REMAINDER) IN FUNCT3D	25.19	6.0	180.91	8.9
TOTAL TIME	420.64	100	2031.11	100

Table 1: Exclusive timings as reported by the VMEC code and relative contribution of the individual code parts to the total runtime for the two representative test cases, **Case 1** (free-boundary calculation with standard resolution), and **Case 2** (fixed-boundary calculation with high angular resolution).

Since the Fourier transform routines together with the routines `BCOVAR` and `FORCES` take a large fraction of the total runtime (about 60% for case 1 and almost 90% for case 2), we focus our analysis on those four routines. Moreover, the routines `BCOVAR` and `FORCES` are structurally very similar, so that working on one of the two routines (namely `BCOVAR`) is sufficient to assess the potential for optimization and parallelization. Because it is not relevant for all cases, we have not optimized the `VACUUM` routine at this stage.

In the following, we will focus on the more highly resolved case 2, performing 100 iterations with full resolution. The results are in principle transferrable to test case 1.

2.3.1 Fourier transforms

The `TOTZSP` and `TOMNSP` routines implement the two directions of the transformation between direct and Fourier-space representations in the angular coordinates of the various fields used in VMEC. Since the direct-space representation maintains the split between even and odd modes (i.e. the back-

transform from Fourier space is not complete), standard discrete Fourier transform (DFT) libraries can not be used. Instead, the transforms are implemented via convolutions with precomputed arrays.
The Fourier transforms are global operations in the two angular coodinates, but are completely decoupled in the different flux surfaces (the s coordinate). In the current data layout (with s being the first index) we found very limited potential for OpenMP parallelization. To exploit the parallelism in s and to improve the cache locality of the convolutions, we introduced transposes of all arrays on entry and exit of the `TOTZSP` and `TOMNSP` routines and computed the actual transforms on the arrays (m, n, s) and (θ, ζ, s).

After a complete restructuring of the routine, it was possible to rewrite the convolutions in the two angular directions using the highly efficient matrix-matrix multiplies (DGEMM) of the BLAS (Basic Linear Algebra Subroutines) library. We used the BLAS implementation of the Intel MKL library, which is highly tuned for the Sandy Bridge processor of the test system.
The speedup due to the rewrite was significant, even for the serial case. In addition, OpenMP parallization was trivial to implement in the new data layout. The convolutions are computationally intensive and the data locality in the new layout is optimal, which leads to a very good parallel efficiency, as can be seen in Fig. 1.

The transposes that are used to map between the original data layout

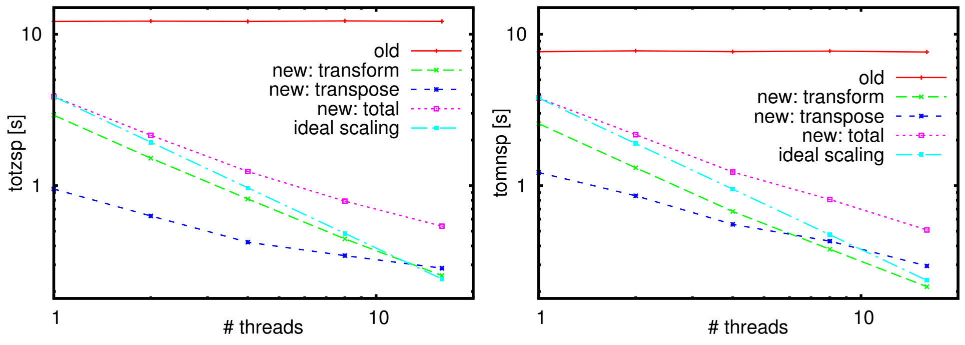

Figure 1: Runtime of the old and new implementions of the Fourier transform routines for test case 2.

used in VMEC and the new data layout used for the Fourier transforms add a significant overhead and reduce scalability. As can be seen in the plots (Fig. 1), the contribution of the transposes exceeds the computing time of the actual Fourier transform for high thread counts. Without the transposes, the speedup compared to the old implementation for 1, 8, 16 OpenMP threads is 4.2x, 27.4x, 47.8x for `TOTZSP` and 3.0x, 20.3x, 35.8x for `TOMNSP`, respectively. When the transposes to and from the old data layout are included, the speedups for 16 threads are still as large as 22.6x and 15x for `TOTZSP` and `TOMNSP`, respectively.

Since the Fourier transforms are decoupled in the flux surface label s, this dimension could also be used very efficiently for a future MPI parallelization of the `TOTZSP` and `TOMNSP` routines via domain decomposition. A domain decomposition in the angular dimensions is expected to be much less favourable – the global nature of the Fourier transforms would require a lot of data transfer, thus strongly reducing the parallel efficiency.

2.3.2 Subroutine `BCOVAR`

The subroutine `BCOVAR` computes the $3\mathrm{D_p}$ metric coefficients and other quantities. The dominant operation is element-wise multiply or add like e.g.

```
A(:)=B(:)*C(:)+D(:).
```

There are more than 30 of these $3\mathrm{D_p}$ arrays used in `BCOVAR`, and most of them are first used to store intermediate results of other computations before they are finally updated with their actual output values.

For the test case described above, the number of elements in the s dimension is $n_s = 99$, the combined angular coordinates have $n_{\zeta\theta} = n_\zeta * n_\theta = 1860$ elements. With double precision numbers (8 byte), this amounts to 792 bytes for a $1\mathrm{D_p}$ $(1 : n_s)$ array, 14.5 KB for a $2\mathrm{D_p}$ array describing one flux surface, i.e. with range $(1 : n_{\zeta\theta})$, and 1.4 MB for a full $3\mathrm{D_p}$ array.

The large number of $3\mathrm{D_p}$ arrays used in `BCOVAR` can not be held in cache at the same time, so that the current implementation leads to a lot of cache misses. Even when arrays are reused in consecutive lines of code,

```
A(:)=B(:)*C(:)+D(:)
B(:)=B(:)*E(:)+A(:)
```

only the L3 cache can be used, because the arrays are multiplied as a whole and are bigger than the L2 cache. Introducing blocking

```
do n=1,nblocks
  lb=(n-1)*blocksize+1
  ub=n*blocksize
  A(lb:ub)=B(lb:ub)*C(lb:ub)+D(lb:ub)
  B(lb:ub)=B(lb:ub)*E(lb:ub)+A(lb:ub)
  ..
end do
```

with appropriate block size helps to improve performance significantly, because the blocks now fit into cache and can efficiently be reused during the loop iteration. Blocking occurs naturally when the flux surface label s is reintroduced:

```
do n=1,nznt
  A(:,n)=B(:,n)*C(:,n)+D(:,n)
  B(:,n)=B(:,n)*E(:,n)+A(:,n)
  ..
end do
```

In this form, the operations can still be vectorized in the first index to exploit the SIMD units of the CPU, and this structure can also be exploited for OpenMP parallelization. The introduction of blocking and a restructuring of `BCOVAR` and the related subroutines resulted in a speedup of 2.1x on a single core and 10.4x on 8 cores (see Fig. 2).

A future restructuring of `BCOVAR` into smaller, independent sections, each

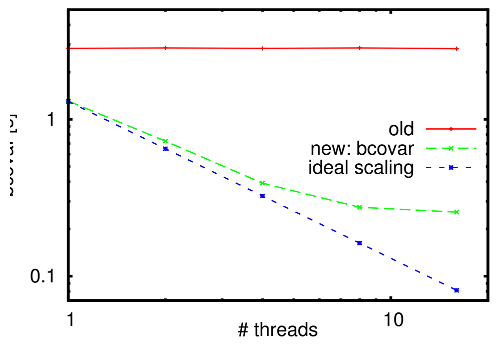

Figure 2: Runtime of the old and new implemention of the `BCOVAR` routine for test case 2, without the serial I/O part.

operating on a smaller data set, would probably improve performance (and parallel efficiency) further.

MPI parallelization of the `BCOVAR` routine would be possible for both the flux surface label and the angular coordinates. A domain decomposition in s (cf. Sect. 2.3.1) would require the introduction of one ghost cell at the left and right s boundary and corresponding updates via nearest-neighbour communication. The data to transfer is non-contiguous in the current data layout, but would consist of a single contiguous block in the transposed data layout described in the previous section. A domain decomposition in the angular coordinates would introduce some reduction operations with collective communication (MPI_Allreduce, see next section), but would nevertheless also be feasible for BCOVAR.

Because of the similarity to BCOVAR, we expect that the measures described above will lead to a similar speedup for the FORCES subroutine.

2.3.3 Computing sums over flux surfaces

An operation that occurs in several places across the code are sums over flux surfaces. The following example is taken from the fbal routine:

```
DO js = 2, ns
   buco(js) = SUM(bsubu(js:nrzt:ns)*wint(js:nrzt:ns))
   bvco(js) = SUM(bsubv(js:nrzt:ns)*wint(js:nrzt:ns))
END DO
```

Reintroducing the flux surface label, this corresponds to

```
DO js = 2, ns
   buco(js) = SUM(bsubu(js,:)*wint(js,:))
   bvco(js) = SUM(bsubv(js,:)*wint(js,:))
END DO
```

The $3D_p$ arrays bsubu, bsubv and wint are too big to fit into L2 cache, so the data has to be fetched from L3. The strides length is $n_{\zeta\theta}$. For each js, data scattered over the whole array has to be fetched.

Using the same data layout, this operation can be written as

```
  buco=0.
  bvco=0.
  DO l = 1, nznt
     buco(2:ns) = buco(2:ns) + bsubu(2:ns,l)*wint(2:ns,l)
     bvco(2:ns) = bvco(2:ns) + bsubv(2:ns,l)*wint(2:ns,l)
  END DO
```

which improves the access pattern for the arrays significantly. The $1D_p$ arrays buco and bvco are very small and remain in L1 cache, the big $3D_p$ arrays are accessed with stride 1 in a continuous stream from start to end. This improves the single core performance by a factor of 5. OpenMP parallelization in this $(n_s, n_{\zeta\theta})$-layout requires a reduction at the end, which

reduces scalability.
A future MPI parallelization of the s direction would not lead to any complications. A parallelization of the angular coordinates, by contrast, would require an `MPI_Allreduce` operation with negative effects on performance and scalability.
In a transposed data layout, the same operation can be implemented in a very natural and efficient way, without the need for any parallel reduction operations:

```
DO js = 2, ns
   buco(js) = SUM(bsubu(:,js)*wint(:,js))
   bvco(js) = SUM(bsubv(:,js)*wint(:,js))
END DO
```

2.3.4 Treatment of radial boundary conditions

Boundary conditions apply at the innermost and outermost flux-surface label. In the current data layout (s, ζ, θ), these points are spread across the entire array. In a transposed layout (ζ, θ, s), by contrast, the inner and outer boundaries are just indexed as $(:, :, 1)$ and $(:, :, ns)$, respectively, i.e. they are represented as small contiguous subarrays that even fit into L1 cache and thus can be handled very efficiently. A transposed data layout would speed up the special treatment of boundary conditions in routines like `BCOVAR`, but would probably also lead to big speedups for the `VACUUM` routine, which is used to compute the outermost flux surface ($s = ns$) in free-bondary calculations. So far, only a few OpenMP statements have been added to this routine, while the general structure has been left untouched.

2.4 Overall speedup

The aim of this work was to assess the potential for optimization and parallelization in the VMEC code. In order to get to a fully optimized, parallelized, and validated version, a considerable amount of work still remains to be done (cf. Sect. 3). Nevertheless, the improvements for the total runtime are quite significant already, as can be seen from the following measurements created with our optimized version on 8 cores. In particular, the measurements shown in Tabble 2 demonstrate that our optimized and parallelized VMEC variant could already run faster by an order of magnitude on a single multicore processor. Rather than just continuing by extending our optimization strategies to the remaining subroutines (which is conceptually straightforward) we propose to launch a coordinated effort together with the VMEC development team (see below).

	Case 1		**Case 2**	
Code section	T_8 [s]	S	T_8 [s]	S
VACUUM LOOP	84.83	1.68	-	-
TOTZSP	12.11	6.98	60.32	12.06
TOMNSP	11.23	7.02	60.55	10.94
FORCES	34.27	0.97	182.88	1.00
BCOVAR	4.49	10.09	21.89	10.68
RESIDUE	7.93	1.01	29.10	1.03
(REMAINDER) IN FUNCT3D	31.95	0.79	179.31	1.01
TOTAL TIME	189.69	2.21	547.07	3.71

Table 2: Internal timings (T_8) as reported by the VMEC code after prototypical optimization and relative speedup ($S = T/T_8$) with respect to the original code, using the two representative test cases, **Case 1** (free-boundary calculation with standard resolution), and **Case 2** (fixed-boundary calculation with high angular resolution). Runtimes, T_8 and T (see Tab. 1) were obtained with the new implementation on 8 cores of an Intel Xeon E5-2670 CPU, and with the original VMEC code on a single core of the same processor, respectively. Note that T_8 includes all overhead introduced by the transformations from and to the original data layout of VMEC. In a transposed data layout, the speedup for the Fourier transforms would be larger by a factor of two (cf. Fig. 1).

The corresponding source code with our modifications to VMEC 8.49 is available in a subversion repository with IPP-internal access and is ready for use in production applications.

3 Summary and Conclusions

We have presented an assessment of the computational performance of the VMEC code (version 8.49) using ”real-world” setups which are relevant for computing 3D-MHD equilibria at the Wendelstein 7-X (W7-X) experiment and have demonstrated a considerable potential for optimization on (clusters of) multicore CPUs. Speedups by one to two orders of magnitude are in reach, but the code requires significant restructuring.

Focussing on the three most time-consuming routines, namely `BCOVAR`, and the Fourier-transforms `TOMNSPS`, `TOTZSPS`, we have straightened out legacy data structures and have adapted the coding style, which — besides improving performance — significantly improves the readability of the code. In particular, we had to revert the "serialization" of three-dimensional arrays and heavy, temporary reuse of the resulting one-dimensional "vectors", the introduction of which presumably was motivated by the paradigms of the era of vector computers in the 1980's, when VMEC was originally written. These measures already lead to performance improvements by a factor of two to three, when comparing with the original code executed on a single core of the same processor (Intel Xeon E5-2670 with 8 cores). Adding an OpenMP parallelization layer to the three routines we achieve total speedups on the order of 10 which, for typical setups, translates to a twofold to fourfold overall speedup of the entire VMEC code. Production applications at W7-X could already take advantage of these improvements. The restructuring we have outlined above needs to be applied to the entire code, in particular to the remaining "hot spots" (`VACUUM`, `FORCES`) which are similar to `BCOVAR`. The parallel efficiency and thus overall speedup can be further improved by using a data layout which is transposed with respect to the current implementation. Moreover, such a transposed data layout is a prerequisite for taking the next optimization step towards a distributed MPI parallelization. Depending on the setup this will allow to utilize on the order of a hundred of processor cores with high parallel efficiency, thus enabling VMEC runs with significantly shorter computation times (up to two orders of magnitude are in reach) or with correspondingly larger resolution.

Due to the appreciable efforts which we expect for implementing the proposed changes throughout the entire code and for their validation a coordinated effort with the VMEC developers appears highly desirable. By contrast, the Fourier-transformation routines (`TOMNSPS`, `TOTZSPS`) are relatively encapsulated and the functionality is expected to remain static. Thus, these parts could be immediately integrated into the VMEC code base to allow the user community taking immediate advantage of the acceleration we have already achieved.

In conclusion we provide a concise list of specific recommendations which are thought to serve as a basis for initiating and planning a comprehensive optimization effort on the VMEC code, ideally conceived as a close collaboration between IPP (with application support by RZG), and the VMEC development team.

1. In order for VMEC to take advantage of the capabilities of modern multicore processors the code needs significant restructuring. With

the current structure (VMEC version 8.49) no major performance improvements can be expected from the forthcoming technology developments.

2. With the proposed restructuring and parallelization, typical VMEC simulations could run faster by a factor of ten on a single multi-core processor, and eventually up to two orders of magnitude using about 100 processor cores.

3. The restructuring should be done in close collaboration with the main developers of VMEC (S. Hirshman and coworkers) in order not to "fork off" a variant of VMEC which will be tedious and error-prone to keep in sync with regular functionality upgrades from the main development line.

4. A scientist either from IPP or from the VMEC development team who is familiar with the relevant numerics of the code and also the underlying physics should coordinate the project and actively engage in the restructuring and optimization effort.

5. A comprehensive test suite for automated and reliable code validation should be established in collaboration with the VMEC developers.

6. We recommend to implement the transposed data layout throughout the entire code immediately, rather than deferring this change to a subsequent optimization step. First, it is no less intrusive to the implementation than the straightening of the code which we consider as a prerequisite for all optimization efforts. Second, it helps to improve the efficiency of the OpenMP parallelization. Third, it is indispensable for achieving speedups beyond a factor of 10 by distributed parallelization.

References

[1] S. P. Hirshman and J. C. Whitson. Steepest-descent moment method for three-dimensional magnetohydrodynamic equilibria. Physics of Fluids, 26(12):3553–3568, 1983.

[2] S.P. Hirshman, W.I. van Rij, and P. Merkel. Three-dimensional free boundary calculations using a spectral green's function method. Computer Physics Communications, 43(1):143 – 155, 1986.

[3] L. F. Romero, E. M. Ortigosa, E. L. Zapata, L. Romero, E. M. Ortigosa, E. L. Zapata, and J. A. Jimenez. Parallelization strategies for the VMEC program. Lecture Notes in Computer Science, 1150:123–456, 1998.

[4] L. F. Romero, E. M. Ortigosa, and E. L. Zapata. Data-task parallelism for the VMEC program. Parallel Computing, 27(10):1347–1364, 2001.

[5] A. H. Reiman and H. Greenside. Calculation of three-dimensional mhd equilibria with islands and stochastic regions. Comp. Phys. Comm., 43(1):157–167, Dec 1986.

[6] Yasuhiro Suzuki, Noriyoshi Nakajima, Kiyomasa Watanabe, Yuji Nakamura, and Takaya Hayashi. Development and application of hint2 to helical system plasmas. Nucl. Fusion, 46:L19–L24, November 2006.

[7] S. P. Hirshman, R. Sanchez, and C. R. Cook. Siesta: A scalable iterative equilibrium solver for toroidal applications. Physics of Plasmas, 18(6):062504, 2011.

[8] S. R. Hudson, R. L. Dewar, G. Dennis, M. J. Hole, M. McGann, G. von Nessi, and S. Lazerson. Computation of multi-region relaxed magnetohydrodynamic equilibria. Physics of Plasmas, 19(11):112502, 2012.

[9] PPPL, http://vmecwiki.pppl.wikispaces.net/VMEC. VMEC Wiki.