Achieving Memory Scalability in the Gysela Code to Fit Exascale Constraints

Fabien Rozar[1,2], Guillaume Latu[1], Jean Roman[3]

[1] IRFM, CEA Cadarache, FR-13108 Saint-Paul-les-Durance
[2] Maison de la simulation, CEA Saclay, FR-91191 Gif sur Yvette
[3] Inria, Université de Bordeaux, CNRS, FR-33405 Talence

Abstract. Gyrokinetic simulations lead to huge computational needs. Up to now, the semi-Lagrangian code Gysela performed large simulations using a few thousands cores (65k cores). But to understand more accurately the nature of the plasma turbulence, finer resolutions are wished which make Gysela a good candidate to exploit the computational power of future Exascale machines. Among the Exascale challenges, the less memory per core issue is one of the must critical. This paper deals with memory management in order to reduce the memory peak, and presents an approach to understand the memory behaviour of an application when dealing with very large meshes. This enables us to extrapolate the behaviour of Gysela for expected capabilities of Exascale machine.

Keywords: Exascale, Memory scalability, Memory footprint reduction, Plasma Physics.

1 Introduction

The architecture of the supercomputers will considerably change in the next decade. Since several years, CPU frequency does not increase anymore. Consequently the on-chip parallelism is dramatically increasing to offer more performance. Instead of doubling the clock-speed every 18-24 month, the number of cores per compute node follows the same law. These new parallel architectures are expected to exhibit different levels of memory and one tendency of these machines is to offer less and less memory per core. This fact has been identified as one of the Exascale challenges [SDM11] and is one of our main concerns.

In the last decade, the simulation of turbulent fusion plasmas in Tokamak devices has involved a growing number of people coming from the applied mathematics and parallel computing fields [ÅCH+13]. These applications are good candidates to be part of the scientific applications that will be able to use the first generation of Exascale computers. The Gysela code already efficiently exploits supercomputing facilities [LGCDP11]. In this paper we especially focus on its memory consumption. This is a critical point to simulate larger physical cases while using a constrained available memory.

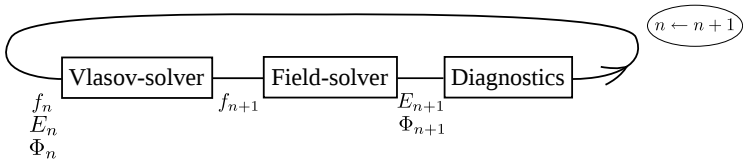


Fig. 1. Numerical scheme for one time step of Gysela

A module has been developed to provide a way to generate memory traces for the specific Gysela application. However, our final goal (not achieved yet) is to define a methodology and a versatile and portable library to help the developer optimize memory usage in scientific parallel applications.

The goal of the work presented here is to decompose and reduce the memory footprint of Gysela to improve its memory scalability. We present a tool which provides a visualization of the memory allocation/deallocation of Gysela in off-line mode. An other tool allows us to predict the memory peak depending on some input parameters. This is helpful to check whether future simulation memory needs fit into available memory.

This article is organized as follow. Section 2 describes shortly the Gysela code. Section 3 presents the memory consumption of Gysela. Section 4 presents the module implemented to generate a trace file of allocation/deallocation in Gysela. It also illustrates the visualization and prediction tool capabilities to handle the data of the trace file. Section 5 shows an example of reduction of the memory footprint and a study of the memory scalability thanks to the prediction tool. Section 6 concludes and presents some future works.

2 Overview of Gysela

This section gives an overview of the global Gysela algorithm and introduces the main data structures used.

Gysela is a global nonlinear electrostatic code which solves a gyrokinetic Vlasov-Maxwell system. Gysela is a coupling between a Vlasov solver that modelizes the motion of the ions inside a tokamak and a Maxwell solver that computes the electrostatic field which applies a force on the ions. The Vlasov equation is solved with a semi-Lagrangian method [GSG+08] and the Maxwell equation is reduced to the numerical solving of a Poisson-like equation [Hah88].

In this gyrokinetic model, the main unknown is a distribution function f which represents the density of ions at a given phase space position. The execution of Gysela is decomposed in the initialization phase, iterations over time, and the exit phase. Figure 1 illustrates the numerical scheme used during a time step. f_n represents the distribution function, Φ_n the electric potential and E_n the electric field which corresponds to the derivative of Φ_n. The Vlasov step performs the evolution of f_n over time and the Field-solver step computes E_n. Periodically, Gysela executes diagnostics which export meaningful values extracted from f_n, E_n and saves the results in HDF5 files.

The distribution function f is a 5 dimensions variable and evolves over time. The first 3 dimensions are in space, $\mathbf{x}_G = (r, \theta, \varphi)$ with r and θ the polar coor-

dinates in the poloidal cross-section of the torus, while φ refers to the toroidal angle. The two last coordinates are in velocity space: $v_\parallel$ the velocity along the magnetic field lines and μ the magnetic moment.

Let N_r, N_θ, N_φ, $\mathrm{N}_{v_\parallel}$ be respectively the number of points in each dimension r, θ, φ, $v_\parallel$. In the Vlasov solver, each value of μ is associated with a set of MPI processes (a MPI communicator). Within each set, a 2D domain decomposition allows us to attribute to each MPI process a sub-domain in (r, θ) dimensions. Thus, a MPI process is then responsible for the storage of the sub-domain defined by $f(r = [i_{start}, i_{end}], \theta = [j_{start}, j_{end}], \varphi = *, v_\parallel = *, \mu = \mu_{value})$. The parallel decomposition is initially set up knowing local values $i_{start}, i_{end}, j_{start}, j_{end}, \mu_{value}$. These 2D domains are derived from a classical block decomposition of the r domain into $\mathrm{p_r}$ sub-domains, and of the θ domain into p_θ sub-domains. The numbers of MPI processes used during one run is equal to $\mathrm{p_r} \times \mathrm{p}_\theta \times \mathrm{N}_\mu$. The OPENMP paradigm is then used in addition to MPI (#T threads in each MPI process) to use fine-grained parallelism.

3 Memory Bottleneck

3.1 Analysis

A GYSELA run needs a lot of amount of memory to be executed. During a run of GYSELA, each MPI process is associated with a μ value (Section 2) and sees the distribution function as a 4D array and the electric field as a 3D array. The remaining of the memory consumption is mostly related to arrays used to store precomputed values, MPI user buffers to concatenate data to send/receive and OPENMP user buffers to compute temporary results. Almost all the arrays are allocated during the initialization of GYSELA.

In order to better understand the memory behaviour of GYSELA, each allocation (allocate statement) is logged by storing: the array name, the type, and the size. Using these data we have done a *strong scaling* presented on the Table 1 (16 threads per MPI process). From the memory point of view, the *strong scaling* study consists in doing a run with a large enough mesh and evaluating the memory consumption for a process increasing step by step the number of MPI processes used for the simulation. If for a given simulation with n processes we use x Giga Bytes of memory per process, in the ideal case, one can hope that the same simulation with $2n$ processes would use $\frac{x}{2}$ Giga Bytes of memory per process. In this case, is it said that the memory *scalability* is perfect. But in practice, this is generally not the case because of parallelization memory overheads.

Table 1 shows the evolution of the memory consumption with the number of cores for a MPI process. The percentage of memory consumption compared with the total memory of the process is given for each type of data structures. The dimensions of the mesh are set to: $N_r = 1024$, $N_\theta = 4096$, $N_\varphi = 1024$, $N_{v_\parallel} = 128$, $N_\mu = 2$. This mesh is bigger than the meshes used in production nowadays, but match further needs, especially those expected for multi-species physics. The last case with 2048 processes requires 67.5 GB of memory per MPI process. We usually launch a single MPI process per node. One can notice the

Table 1. Strong scaling: static allocation sizes in (GB per MPI process) and percentage of the total for each kind of data

Number of cores Number of MPI processes	2k 128	4k 256	8k 512	16k 1024	32k 2048
4D structures	209.2 67.1 %	107.1 59.6 %	56.5 49.5 %	28.4 34.2 %	14.4 21.3 %
3D structures	62.7 20.1 %	36.0 20.0 %	22.6 19.8 %	19.7 23.7 %	18.3 27.1 %
2D structures	33.1 10.6 %	33.1 18.4 %	33.1 28.9 %	33.1 39.9 %	33.1 49.0 %
1D structures	6.6 2.1 %	3.4 1.9 %	2.0 1.7 %	1.7 2.0 %	1.6 2.3 %
Total per MPI process in GBytes	311.5	179.6	114.2	83.0	67.5

memory required is much more than the 64 GB of a Helios[4] node or than the 16 GB of a Blue Gene/Q node. Table 1 also illustrates that 2D structures and many 1D structures do not benefit of the domain decomposition. In fact, the memory cost of the 2D structures does not depend on the number of processes at all, but rather on the mesh size and the number of threads. On the last case with 32k cores, the cost of the 2D structures is the main bottleneck. It takes 49 % of the whole memory footprint.

In GYSELA, the memory overhead for large simulations is due to various reasons. Extra memory can be needed, for example to store some coefficients during an interpolation (for the Semi-Lagrangian solver of the Vlasov equation). MPI buffers appear also as memory overhead. The MPI subroutines accept as input 1D array which often requires to copy the data we want to send or receive in an appropriate way. We have reduced some of these memory overheads. It has improved the memory scalability and has allowed us to run bigger physical cases.

3.2 Approach

There are two ways to reduce the memory footprint of a parallel application. On the one hand one can increase the number of nodes used for the simulation. Since the size of structures which benefit of a domain decomposition will decrease along with the number of MPI processes. On the other hand, we can manage more finely the allocations of arrays in order to reduce the memory costs that do not scale with the number of threads/MPI processes and to limit the impact of all allocated data at the memory peak.

To achieve the reduction of the memory footprint and to push back the memory bottleneck, we choose to focus on the second approach.

In the original version of the code, most of the variables are allocated during the initialization phase. This approach is rightful for structures which are

[4] http://www.top500.org/system/177449

persistent variables in opposition to *temporary variables* that could be dynamically allocated. In this configuration, firstly, one can determine early the memory space required without actually executing a complete simulation. This allows a user to know if the case submitted can be run or not. Secondly, it avoids execution overheads due to dynamic memory management. But a disadvantage of this approach is that variables used locally in one or two subroutines consume their memory space during the whole execution of the simulation. As the memory space becomes a critical point when a large number of cores are employed, we have allocated a large subset of these as temporary variables with dynamic allocation. This has reduced the memory peak with a negligible impact on the execution time. Also, we notice that some persistent variables can be deallocated at the memory peak time which can decrease memory footprint. However, one issue with dynamic allocations is that we lost the two main advantages of the static allocations, and particularly the ability to determine in advance the memory space required to run a simulation.

4 Customised Modeling and Tracing Memory Tools

To follow the memory consumption of GYSELA and to measure the memory footprint reduction, three different tools has been developed: a FORTRAN module to generate a trace file of allocations/deallocations, and a visualization + prediction PYTHON script which exploits the trace file. The information retrieved from the execution of GYSELA thanks to the instrumentation module is a key component of our memory analysis. The implementation of these helpful tools is detailed in the following sections.

4.1 Trace File

Various data structures are used in GYSELA, and in order to handle their allocations/deallocations, a dedicated FORTRAN module was developed to log them to a file: the *dynamic memory trace*. As the MPI processes have almost the same dynamic memory trace, in the current implementation, we produce a single trace file for the allocations/deallocations of the MPI process `0`.

Overview. In the community of performance analysis tools dedicated to parallel application, different approaches exist. But almost all of them relies on *trace files*. A trace file collects information from the application to represent one aspect of its execution: execution time, number of MPI messages send, idle time, memory consumption and so on. But to obtain these information, the application have to be instrumented. The instrumentation can be made at 4 levels: in the source code, at the compilation time, at the linking step or during the execution (just in time).

The SCALASCA performance tool [GWW+10] is able to instrument at the compilation time. This approach has the advantage to cover all code parts of the application and it allows the customization of the retrieved information. This

systematic approach gives a full detailed trace but the record of information in all subroutine of the code may induce a consequent overhead in execution time. Also with an automatic instrumentation, it would be difficult to retrieve the expression of an allocation, like we do (cf. next section). The tool set EZ-TRACE [AMGRT13] offers the possibility to intercept calls to a set of functions. This tool can quickly instrument an application thanks to a link with third-party libraries at the linking step. Unlike our approach, this one does not need an instrumentation of the code but you cannot hope to retrieve the allocation expression in this approach. The tools PIN [LCM+05], DYNAMORIO [BGA03] or MAQAO [DBC+05] produce an instrumentation during the execution time. The advantage here is the generic aspect of the method. Any program can be instrumented this way, but unlike our approach, these ones often introduce a quite large overhead of execution time.

The tool we have developed allows us to measure the performance of GYSELA, from the memory point of view. A visualization tool has been developed to deal with the provided trace file. It offers a global view of the memory consumption and an accurate view around the memory peak to help the developer to reduce the memory footprint. The terminal output of the post processing script gives precious information about the arrays allocated at the memory peak. Given a trace file, we can also extrapolate the memory consumption in function of the input parameters. This allows us to investigate the memory scalability. As far as we know, there is no equivalent tool to profile the memory behaviour in the HPC community.

Implementation. A dedicated FORTRAN module of instrumentation has been developed. This instrumentation will generate a trace file. Then we practice a post-mortem analysis on it. The instrumentation module offers an interface, *take* and *drop*, which wraps the calls to `allocate` and `deallocate`. The `take` and `drop` subroutines perform the allocation and deallocation of the array handled and they log their memory action in the dynamic memory trace file.

For each allocation and deallocation, the module logs the name of the array, its type, its size and the expression of number of elements. The expression is required to make prediction. For example, the expression associated to this allocation:

```
integer, dimension(:,:), pointer  :: array
integer                           :: a0, a1, b0, b1
allocate(array(a0:a1, b0:b1))
```

is

$$(a1 - a0 + 1) \times (b1 - b0 + 1) \tag{1}$$

To be able to evaluate these allocation expressions, the variables inside them must be recorded. Either the value of the variable is logged, either an arithmetical expression depending on other recorded variables is logged. This is done respectively by the subroutines *write_param* and *write_expr*. The writing of expression saves the relationship between parameters in the trace file. This is essential for the prediction tool (4.3). The following code is an example of recording the parameters $a0, a1, b0, b1$:

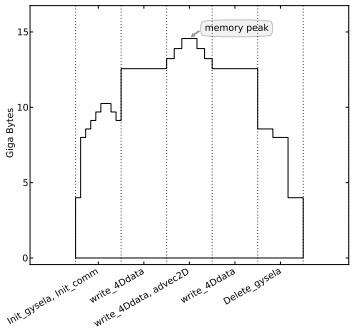


Fig. 2. Evolution of the dynamic memory consumption during GYSELA execution

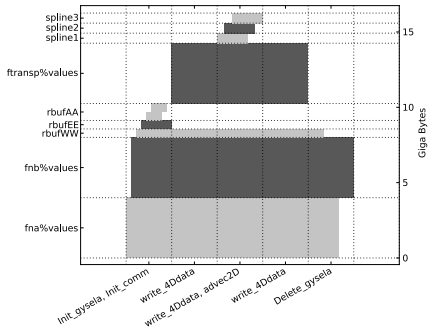


Fig. 3. Allocation and deallocation of arrays used in different GYSELA subroutines

```
call write_param('a0', 1); call write_param('a1', 10)
call write_param('b0', 1); call write_expr('b1', '2*(a1-a0+1)')
```

To retrieve the temporal aspect of the memory allocation, the entry/exit to selected subroutines is recorded by the interface *write_begin_sub* and *write_end_sub*. This allows us to localize where happen the allocations/deallocations which is an essential aspect for the visualization step.

4.2 Visualization

In order to address memory consumption, we have to identify the parts of the code where the memory usage reaches its peak. The log file can be large, some Mega Bytes. To manage this amount of data, a PYTHON script was developed to visualize them. This tool will help the developer to understand the memory cost of the handled algorithms, and so give him some hints how and where it is meaningful to decrease the memory footprint. These information are given thanks to two kinds of plot.

Figure 2 plots the *dynamic* memory consumption in GB along time. The X axis represents the chronological entry/exit of instrumented subroutines. The Y axis gives memory consumption in GB. Figure 3 shows which array is used in which subroutine. The X axis remains identical as previously and the Y axis shows a name of array. Each array is associated to a horizontal line of the picture. The allocation of an array matches a rectangular filled in dark or light grey color in its corresponding line. The width of rectangles depends on the subroutines where allocation/deallocation happens.

In Figure 2 one can locate in which subroutine the memory peak is reached. In Figure 3 one can then identify the arrays that are actually allocated when the memory peak is reached. Thanks to these information, we exactly know where to modify the code in order to reduce the global memory consumption.

4.3 Prediction

To anticipate our memory requirements to run a given simulation, we need to predict the memory consumption for a given input parameter set. Thanks to the expressions of array size and the value or expression of numerical parameters

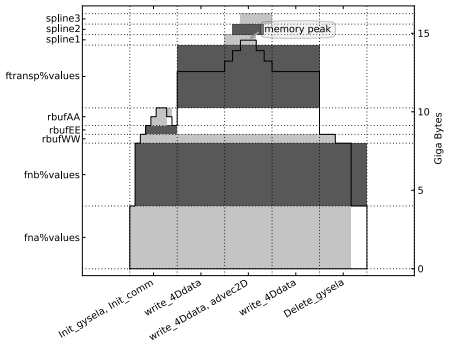


Fig. 4. First trace visualization

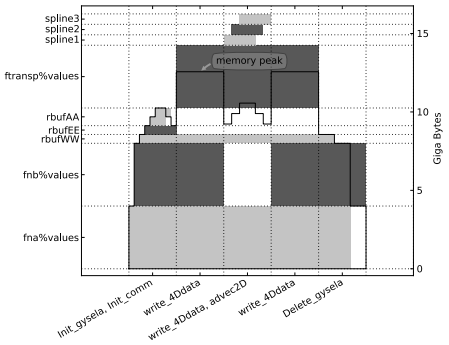


Fig. 5. Second trace visualization

contained in the trace file, we can model the memory behaviour off-line. The idea here is to reproduce allocations with any input set of parameters.

Sometimes, a parameter value cannot be expressed as a one line arithmetical expression (e.g. multi criteria optimization loop to determine the value). To manage this case and in order to be faithful, the FORTRAN piece of code which returns the value is call from PYTHON script. This is possible thanks to a compilation of the FORTRAN needed sources with `f2py` [Pet09].

By changing the value of input parameters, our prediction PYTHON tool offers the possibility to extrapolate the GYSELA memory consumption on greater meshes and even on supercomputer configurations which do not exist yet, as the Exascale ones. The results of this tool are presented in the Section 5.2.

5 Results

5.1 Memory footprint reduction

Reduce the memory footprint is equivalent to cut down the memory peak. The Figures 4 and 5 show the impact on memory of some modifications of the code. After analysis of the code, we noticed that during the memory peak, the transposition structure `ftransp%values` and the distribution function `fnb%values` contain the same data organized differently. We obtain the trace of the Figure 5 in deallocating `fnb%values` during the memory peak.

With this tool, one can see that depending on the size of the mesh and the number of MPI processes and OPENMP threads, the memory peak moves. This behavior can be explained by the dependencies between the size of some characteristic arrays and the value of some input parameters. For example, MPI buffer sizes are sensitive to parallelization parameters. In GYSELA, the sizes of temporary buffer are sensitive to the number of points in r and *theta* dimensions.

The visualization tool gives a new point of view of the source code. This tool helped us to iteratively reduce the memory overhead and thus to improve the memory scalability.

5.2 Prediction over large meshes

Scalability. The Tab. 2 presents the strong scaling test with the new dynamic allocations, and several algorithmic improvements we have done thanks visual-

ization tool (not detailed here). The prediction tool allows us to reproduce the Tab. 1 on the same mesh, i.e. $N_r = 1024$, $N_\theta = 4096$, $N_\varphi = 1024$, $N_{v_\parallel} = 128$, $N_\mu = 2$.

Table 2. Strong scaling: memory allocation size and percentage of the total for each kind of data at the memory peak moment

Number of cores Number of MPI processes	2k 128	4k 256	8k 512	16k 1024	32k 2048
4D structures	207.2 79.2%	104.4 71.5%	53.7 65.6%	27.3 52.2%	14.4 42.0%
3D structures	42.0 16.1%	31.1 21.3%	18.6 22.7%	15.9 30.4%	11.0 32.1%
2D structures	7.1 2.7%	7.1 4.9%	7.1 8.7%	7.1 13.6%	7.1 20.8%
1D structures	5.2 2.0%	3.3 2.3%	2.4 3.0%	2.0 3.8%	1.7 5.1%
Total per MPI process in GBytes	261.5	145.9	81.9	52.3	34.3

The Tab. 2 outputs the memory consumption at the memory peak. It is obtained by keeping the mesh size constant and changing the number of MPI processes and OPENMP threads. The prediction script replays the allocation/deallocation of trace file with the new parameters. As you can see on the bigger case (32k cores), the consumption of the 2D structures were reduced by 20.8%. Also the memory gain on this case is of **50.8%** on the global consumption relatively to Tab. 1. The 4D structures contain the most relevant data used during the computation, and they consume the major part of the memory as they should. The memory overheads have been globally reduced which improves the memory scalability of GYSELA and allows larger simulations to be run.

Investigation. By using the prediction tool, larger meshes can be investigated and the size of the machine required to handled this amount of data can be estimated. With the actual implementation, to run the mesh $N_r = 2048$, $N_\theta = 4096$, $N_\varphi = 2048$, $N_{v_\parallel} = 256$, $N_\mu = 2$, the number of cores needed is of 524k cores, with 64 GB per process and 16 threads per process.

6 Conclusion

The work described in this paper focuses on a memory modeling and tracing module and some post processing tools which enable one to improve the memory scalability. With this framework, the understanding of the memory footprint behaviour along time is accessible. Also, the generated trace file can be reused to extrapolate the memory consumption for different input sets of parameter

in off-line mode ; this aspect is important both for end-user who needs greater resolutions or features with greedy memory needs, and for developer to design algorithms for Exascale machine.

With these tools, a reduction of **50.8%** of the memory peak has been achieved and the memory scalability of the GYSELA has been improved. Our next objective is to implement a versatile C/Fortran library. The work presented in this paper is a first step toward building a methodology that helps developers to improve memory scalability of parallel applications.

References

ÅCH+13. JA Åström, Adam Carter, James Hetherington, K Ioakimidis, E Lindahl, G Mozdzynski, Rupert W Nash, P Schlatter, Artur Signell, and Jan Westerholm. Preparing scientific application software for exascale computing. In *Applied Parallel and Scientific Computing*, pages 27–42. Springer, 2013.

AMGRT13. Charles Aulagnon, Damien Martin-Guillerez, François Rué, and François Trahay. Runtime function instrumentation with eztrace. In *Euro-Par 2012: Parallel Processing Workshops*, pages 395–403. Springer, 2013.

BGA03. Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *[CGO 2003]*, pages 265–275. IEEE, 2003.

DBC+05. Lamia Djoudi, Denis Barthou, Patrick Carribault, Christophe Lemuet, Jean-Thomas Acquaviva, William Jalby, et al. Maqao: Modular assembler quality analyzer and optimizer for itanium 2. In *The 4th Workshop on EPIC architectures and compiler technology, San Jose*, 2005.

GSG+08. V. Grandgirard, Y. Sarazin, X. Garbet, G. Dif-Pradalier, Ph. Ghendrih, N. Crouseilles, G. Latu, E. Sonnendrucker, N. Besse, and P. Bertrand. Computing ITG turbulence with a full-f semi-Lagrangian code. *Communications in Nonlinear Science and Num. Sim.*, 13(1):81 – 87, 2008.

GWW+10. Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *CCPE*, 22(6):702–719, 2010.

Hah88. T. S. Hahm. Nonlinear gyrokinetic equations for tokamak microturbulence. *Physics of Fluids*, 31(9):2670–2673, 1988.

LCM+05. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

LGCDP11. G. Latu, V. Grandgirard, N. Crouseilles, and G. Dif-Pradalier. Scalable quasineutral solver for gyrokinetic simulation. In *PPAM (2)*, LNCS 7204, pages 221–231. Springer, 2011.

Pet09. Pearu Peterson. F2py: a tool for connecting fortran and python programs. *International Journal of Computational Science and Engineering*, 4(4):296–305, 2009.

SDM11. John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *High Performance Computing for Computational Science – VECPAR 2010*, LNCS 6449, pages 1–25. Springer, 2011.