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 the 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 mouth, the number of cores per compute node follows the same law. These new parallel architectures is expected to exhibit different levels of memory and one tendency of these machines is to offer less and less memory per core. This fact is known to be 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 [JMV+11,GLB+11,MII+11]. These applications are good candidate to become one of the scientific applications that will be able to use the first generation of Exascale computers. The Gysela code already exploits efficiently supercomputing facilities and in this paper we will especially focus on its memory consumption. This is a critical point to simulate larger physical cases while using a constrained available memory.

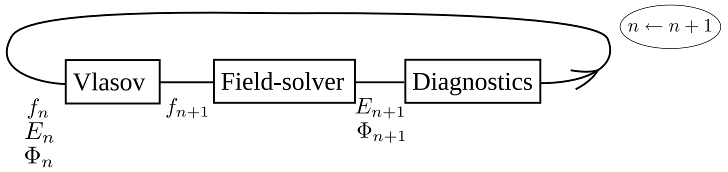

Fig. 1. Numerical scheme for one time step

We have designed a module that provides a way to generate memory management traces for a specific application. However, our final goal (not achieved yet) is to define a methodology and a versatile and portable library to help the developer to optimize memory usage in scientific parallel applications.

The goal of the work presented here is to decompose and to reduce the memory footprint of Gysela. Thus we will improve its scalability and we will be able to predict the required memory for a physical case. We present a tool to model the allocation/deallocation mechanisms in off-line mode. By replaying the allocations outside of a run, we can adjust some numerical parameters (for example, $N_r = 1024$, $N_\theta = 4096$, $N_\varphi = 1024$, $N_{v_\parallel} = 128$, $N_\mu = 2$) for a new physical case and ensure the simulation can fit into available memory. Parameters describing the parallel configuration are used too: usually the number of Mpi processes matches the number of nodes, the number of OpenMP threads matches the number of cores per processor.

Section 2 describes shortly the Gysela code. Section 3 presents the memory consumption of Gysela. Section 4 presents the tool implemented in Gysela to retrieve informations in a trace file for memory allocations/deallocations. It also illustrates the visualization and prediction tool capabilities to manage the data of the trace file. Section 5 illustrates two utilisations of the tools developed: the reduction of the memory footprint and the memory allocations extrapolation. 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 data structures used.

Gysela is a global nonlinear electrostatic code which solves a gyrokinetic Vlasov-Maxwell system. Gysela is a coupling between a Vlasov equation which describes the moves of the ions inside a tokamak and a Maxwell equation which expresses the electrostatic field which applied a force on the ions. The Vlasov equation is solved with a semi-Lagrangian method and the Maxwell equation is reduced to the numerical solving of a Poisson-like equation.

The solving of these equations is not the purpose of this article and we refer the reader to [LCGS07,GSG$^+$08] and to [LGCDP11,Hah88] for detailed descriptions.

Our gyrokinetic model considers as main unknown a distribution function $\bar{f}$ that represents the density of ions at a given phase space position. The execution of Gysela is decomposed in the initialisation phase, the time steps, and the exit phase. Figure 1 illustrates the numerical scheme used during a time step of Gysela. f_n represents the distribution function, Φ_n the electric potential and E_n the electric field which corresponds to the derived of Φ_n. The Vlasov box performs the time evolution of the distribution function and the Field-solver box the electric field. Periodically, Gysela does diagnostics which exports meaningful values extrated from f_n, E_n and saves the results in HDF5 files.

The distribution function $\bar{f}$ depends on time and on 5 other dimensions. First, 3 dimensions in space $\mathbf{x}_G = (r, \theta, \varphi)$ with r and θ the polar coordinates in the poloidal cross-section of the torus, while φ refers to the toroidal angle. Second, velocity space has two dimensions: $v_\parallel$ being the velocity along the magnetic field lines and μ the magnetic moment.

Large data structures are used in Gysela, the main ones are: the 5D data associated with $\bar{f}$, and the 3D data associated with the electric potential Φ. 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, we give the responsibility of each value of μ to a given set of Mpi processes (a Mpi communicator), and we fixed that there are always N_μ sets such that only one μ value is attributed to each communicator. Within each set, a 2D domain decomposition allows us to attribute to each Mpi Process a subdomain in (r, θ) dimensions. Thus, a Mpi process is then responsible for the storage of the subdomain defined by $\bar{f}(r = [i_{start}, i_{end}], \theta = [j_{start}, j_{end}], \varphi = *, v_\parallel = *, \mu = \mu_{value})$. The parallel decomposition is initially set up knowing local values $i_{start}, i_{end}, j_{start}, j_{end}, \mu_{value}$. They are derived from a classical block decomposition of the r domain into $\mathrm{p_r}$ pieces, and of the θ domain into p_θ subdomains. 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

During a run, Gysela uses a lot of memory to store the 5D distribution functions and the 3D electric field. The 5D distribution function is stored as a collection of 4D arrays, each Mpi process handles only one element along the 5^{th} dimension (μ variable) of the 5D array. The electric field is a 3D array which is distributed as 3D blocks handled by each process. The remaining of the memory consumption is mostly related to arrays used to store precomputed values, Mpi user buffers to concatenate data to send and OpenMP user buffers to compute temporary results. Most of the arrays are allocated during the initialisation of Gysela.

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

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

Table 1 shows the memory consumption of each MPI process for a strong scaling test with a varying number of processes. The percentage of memory consumption compared with the total memory of the process is given for each kind of data structures. The mesh size is $N_r = 1024$, $N_\theta = 4096$, $N_\varphi = 1024$, $N_{v_\parallel} = 128$, $N_\mu = 2$. This mesh is bigger than the meshes used in production nowadays, but we try to match further needs (such as those expected for multi-species physics). The last case with 2048 processes requires 67.5 GB of RAM; this is much more than the 16 GB of a Blue Gene/Q node or even than the 64 GB of a Helios[4] node. Table 1 also illustrates that 2D structures and many 1D structures do not scale. 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 with 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 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. The reduction of some of these memory overheads increases the memory scalability and has allowed us to run bigger physical cases.

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

3.2 Approach

There are two main ways to reduce the memory footprint. On the one hand we can increase the number of nodes used for the simulation. Since the main structures (4D and 3D) are well distributed between the cores, the size allocated for those structures on each node will tend to decrease with the number of 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.

During an execution of GYSELA, there are three main steps: the initialisation, the time steps and the exit phase. In the original version of the code, most of the variables are allocated during the initialisation phase, these are *persistent variables* in opposition to *temporary variables* that would be dynamically allocated throughout the simulation. This approach has two main advantages:

– it is possible to determine the memory space required without actually executing a complete simulation, thus allowing to determine valid input parameters and mesh size on a given machine in running only initialisation phase;
– it prevents any overhead related to dynamic memory management; in early versions of the code, this was slowing down the execution time by a factor two.

A disadvantage however is that variables used locally in one or two subroutines use memory during the whole execution of the simulation. As this 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 without impacting execution time. Also we notice that some persistent variables can be deallocated at the memory peak time which can benefit memory footprint. However, one issue with this approach is that we lost the two main advantages of the static allocations, and particularly the capability to determine the memory space required to run the program.

4 Customised Tool for Memory Tracing/Analysis

To follow the memory consumption of GYSELA and to measure our gain of memory, three different tools has been developed: a FORTRAN module to obtain a trace file of allocations/deallocations, and a visualization + prediction PYTHON script which exploits the trace file. The informations retrieved from the execution of GYSELA thanks to the instrumentation module is a key component of our memory analysis. The tools based on the trace file help to reduce the memory footprint of GYSELA on big cases by improving memory scalability.

The implementation of these helpful tools is detailed in the following sections.

4.1 Trace File

One main difference between the static and the dynamic allocation is the time evolution of the memory footprint. It is easy to establish the log file of the static allocations because they happen only in the beginning of the run, which is well localized in the source code. Contrary to static allocations, dynamic allocations can happen at any moment, close to the use of the array. Various data structures are used in Gysela, and in order to handle their allocations, a dedicated Fortran module was developed to log them to a file (the *dynamic memory trace*). Gysela is a hybrid Mpi/OpenMP parallel application. However, each Mpi process has the same allocation/deallocation trace file, so to avoid redundant informations, only the Mpi process `0` writes a trace file.

Overview. In the community of performance analysis tools dedicated to parallel application, different approaches exist. However, 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 informations, the application must be instrumented. The instrumentation can be made at 4 levels: in the source code, at the compilation time, at the linked 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 advantages to cover all code parts of the application and it allows the customization of the retrieved informations. An inconvenient can be the instrumentation of all subroutines of the code: it gives a lot of informations but can be expansive. Also with an automatic instrumentation, it would be difficult to get the expression of an allocation, like we do (cf. next section). The tool set EZTrace [AMGRT13] offers the possibility to intercept calls to a set of functions. This tool can quickly instrument an application thanks to a link with an alternative library at the linked step. Unlike our approach, this one does not need to recompile the code to instrument it. 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. We can investigate the memory scalability. A visualisation tool has been developed to deal with the provided trace file. It offers a large view of the memory consumption and an accurate view around the memory peak to help the developer to reduce the memory footprint of one application. The terminal output of the visualisation script gives precious informations about the arrays allocated at the memory peak. Given a trace file, we can also extrapolate the memory consumption in function of the entry parameters. As far as we know, there is no equivalent tool to profile the memory behaviour in the HPC community.

Implementation. Our work is based on an instrumentation at the source code level. We make a post-mortem analysis of the trace file generated thanks to the following subroutines.

This 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 name of the array, its type, its size and the expression used to evaluate its size. The expression of the size will be useful to make prediction. For instance, the expression associated to this allocation :

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

is

$$(a1 - a0 + 1) \times (b1 - b0 + 1) \qquad (1)$$

This expression gives the number of elements in a array.

Often, an expression depends on a set of variables or parameters. These must be recorded to allow the computation of the expression. Our instrumentation module offers an other interface, *write_param* and *write_expr* which write respectively the value associated to a parameter and the expression associated to a parameter. The following code is an example of record the parameters $a0, a1, b0, b1$:

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

To catch the temporal aspect of the memory allocation, the module offers an other interface, *write_begin_sub* and *write_end_sub* to instrument the code in order to record the entrance and exit of selected subroutines. This make it possible to identify in the trace file which array is used in which subroutine. This aspect is essential for the visualization.

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 contain a lot of lines (some Mega Bytes). To face and manage all these informations, a Python script was developed for interpretation and visualisation. It is important to know which structures are allocated at the memory peak. These data will help the developer to understand the memory cost of the algorithms, and so give him some hints to decrease the memory footprint. In order to provide all these informations, we have plotted two aspects of the memory consumption by analysing the dynamic memory trace.

Figure 2 plots the *dynamic* memory consumption in GB along time. The X axis is not linear in time but rather shows the name of the instrumented subroutines. Figure 3 shows which array is used in which subroutine. The X axis similarly shows the name of subroutines and the Y axis shows the name of each

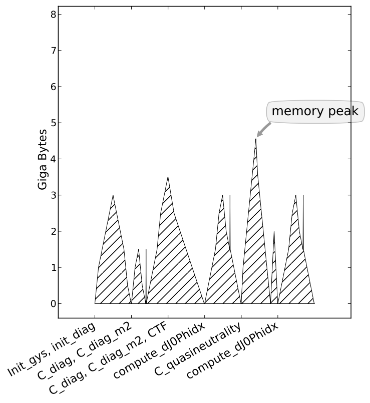

Fig. 2. Evolution of the dynamic memory consumption during Gysela execution

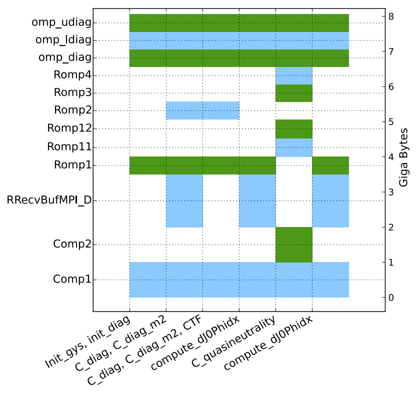

Fig. 3. Allocation and deallocation of arrays used in different Gysela subroutines

array. A square is drawn at the intersection of a subroutine column and array line if this specific array is actually allocated in the subroutine.

Using Figure 2 we can locate in which subroutine the memory peak is reached. Using Figure 3 we can then identify the arrays that are actually allocated when the memory peak is reached. Thank to this information, we now 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 entry set. The runs of Gysela are customizable through a data file (input) which specifies all the numerical and the physical parameters. For example it specifies the mesh size. The numerical parameters have a direct impact on the amount of memory needed because almost all array dimensions are deduced from these ones. As the expression of each array size and the numerical parameters are recorded, we can reproduce the memory behaviour off-line. The idea here is to reproduce allocation of the application with a different set of input parameters.

A difficulty of this step is to determine the relationship between an array bound and the set of entry parameters to ensure correct result of the computation of the expression. Sometimes, some relations are quite impossible to map to an expression (e.g. multi criteria optimization loop to determine the bound). So in this case, we decide to call the Fortran piece of code to get the right value of the array boundary handled. This is solved thank to a compilation of the Fortran sources with `f2py` [Pet09].

We can replay the trace file with any entry set of parameters. This feature offers us the possibility to extrapolate the Gysela memory consumption on greater meshes and even on supercomputer configurations which do not exist yet, as the Exascale one. The results of this tool are presented in the Section 5.2.

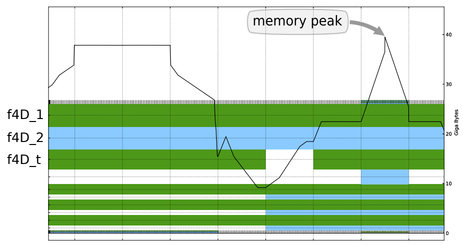

Fig. 4. Dynamic allocations

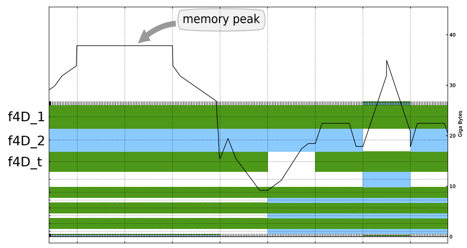

Fig. 5. Decreasing of the memory peak

5 Results

5.1 Memory footprint reduction

As shown on the Figures 2 and 3, the dynamic allocations give a memory footprint that depends on times. The reduction of the footprint resumes in cutting down the memory peak.

The Figures 4 and 5 show modifications of the code that impact the memory peak. We obtain this result thanks to the deallocation of the distribution function `f4D_2` during the memory peak. After analysis of the subroutine where the memory peak happens, we noticed that we use a transposition of the distribution function. Thus at the memory peak, the transposition structure and the distribution function contain the same data organized differently.

Our experience with this tool has shown us that depending on the size of the mesh and the number of Mpi processes and OpenMP threads, the memory peak moves. For example, on really large simulation, the Mpi buffers are much more bigger than on small simulation. In Gysela, depending on the number of point in the r and θ dimensions, the computation of spline coefficients during the interpolation requires big buffers. With the visualisation of the memory consumption, it is easy to notice and to understand these behaviours.

The terminal output of the visualisation script lists all arrays allocated during the memory peak, which makes the development work easier.

The visualization tool gives a new look of the source code to the developer. This tool helped us to reduce efficiently the memory overhead and thus improve the memory scalability.

5.2 Prediction over large meshes

The prediction tool allows us to reproduce the Tab. 1 with the new memory management on the same mesh, i.e. $N_r = 1024$, $N_\theta = 4096$, $N_\varphi = 1024$, $N_{v_\parallel} = 128$, $N_\mu = 2$. The Tab. 2 presents the strong scaling test with the new dynamic allocations, and several algorithmic improvements also (not detailed here).

The Tab. 2 outputs the measurement of the memory consumption at the memory peak. This peak is on the critical path using the dynamic management of allocations and can be compared with Tab. 1. The Tab. 2 is obtained by changing the numerical parameters in the beginning of the trace file (mesh size unchanged, only the number of processes and threads). The prediction script

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

replays the trace file with these new parameters. As we can see, on the big case (32k cores), the 2D structures consume only 20.8% of the memory. We also obtain a memory gain of **50.8%** on the global consumption relatively to Tab. 1. The major part of the memory consumption is due to the 4D structures. It is legitimate, they are the heaviest structures as they contain the most important data used during the computation. The memory overheads have been globally reduced with the new management of allocations. This improves the memory scalability of GYSELA and allow us to run greater physical cases.

Now we can investigate what kind of behaviour we would have on a very large mesh using the prediction tool. We find that with a computer of 32k processes (with 64 GB per process and 16 threads per process), the code would be able 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 for this configuration is of 524k cores (number of processes times number of threads per process).

6 Conclusion

The work described in this paper focuses on a memory management module and some post processing scripts that provide low level tools to improve memory scalability. With this framework, the developer is able to gain a better understanding of the memory footprint behaviour along time and to identify the data structures allocated at the memory peak. The trace file generated during execution can be reused to produce prediction of memory consumption for a different parameter set in off-line mode ; this aspect is important both for end-user designing physical cases on a given supercomputer, and for developer in order to predict memory consumption on Exascale machine.

With these tools, we manage to achieve a reduction of the memory peak and to improve the memory scalability of the GYSELA code. To this issue, algorithmic changes and dynamic memory management have been made. We expect to implement soon a more generic 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

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.

GLB$^+$11. T. Görler, X. Lapillonne, S. Brunner, T. Dannert, F. Jenko, F. Merz, and D. Told. The global version of the gyrokinetic turbulence code gene. *J. Comput. Physics*, 230(18):7053–7071, 2011.

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.

JMV$^+$11. S. Jolliet, B.F. McMillan, L. Villard, T. Vernay, P. Angelino, T.M. Tran, S. Brunner, A. Bottino, and Y. Idomura. Parallel filtering in global gyrokinetic simulations. *Journal of Computational Physics*, 2011. In press.

LCGS07. G. Latu, N. Crouseilles, V. Grandgirard, and E. Sonnendrucker. Gyrokinetic semi-Lagrangian parallel simulation using a hybrid OpenMP/MPI programming. In *Recent Advances in PVM and MPI*, LNCS 4757, pages 356–364. Springer, 2007.

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.

MII$^+$11. K. Madduri, E.-J. Im, K. Ibrahim, S. Williams, S. Ethier, and L. Oliker. Gyrokinetic particle-in-cell optimization on emerging multi- and manycore platforms. *Parallel Computing*, 37(9):501–520, 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.