Computational performance of a parallelized high-order spectral and mortar element toolbox

Roland Bouffanais [a,*], Vincent Keller [b], Ralf Gruber [b], Michel O. Deville [b]

[a] *Massachusetts Institute of Technology,*
Department of Mechanical Engineering,
77 Massachusetts Avenue, Bldg 5–326,
Cambridge, MA 02139

[b] *Laboratory of Computational Engineering,*
École Polytechnique Fédérale de Lausanne,
STI – ISE – LIN, Station 9,
CH–1015 Lausanne, Switzerland

Abstract

In this paper, a comprehensive performance review of a MPI-based high-order spectral and mortar element method C++ toolbox is presented. The focus is put on the performance evaluation of several aspects at the hardware and software levels with a particular emphasis on the parallel efficiency, the C++ implementation weaknesses and the influence of the concurrent and subsequent implementation layers in a multi-programming environment. The performance evaluation is analyzed and compared to predictions given by a heuristic model, the so-called Γ model. Three tailor-made CFD computation benchmark cases are introduced and used to carry out this review on different serial and parallel architectures, stressing the particular interest for commodity clusters. Conclusions are drawn from this extensive series of analyses and modeling leading to specific recommendations concerning such toolbox development and implementation.

Key words: Spectral and mortar element method, C++ toolbox, MPI, scalability, commodity clusters, compiler optimization, linking optimization, profiling.

* Corresponding author.
Email addresses: bouffana@mit.edu (Roland Bouffanais), vincent.keller@epfl.ch (Vincent Keller), ralf.gruber@epfl.ch (Ralf Gruber), michel.deville@epfl.ch (Michel O. Deville).

1 Introduction

This paper provides a detailed performance evaluation of the C++ toolbox named Speculoos (for Spectral Unstructured Elements Object-Oriented System). Speculoos is a spectral and mortar element analysis toolbox for the numerical solution of partial differential equations and more particularly for solving incompressible unsteady fluid flow problems. It was initiated by Prof. Michel O. Deville and Dr. Vincent Van Kemenade [1] in 1995 at the Laboratory of Fluid Mechanics of the École Polytechnique Fédérale de Lausanne (EPFL). The main architecture choices and the parallel implementation were elaborated and implemented by Van Kemenade and Dubois-Pèlerin [2, 3]. Subsequently, Speculoos' C++ code has been growing up with additional layers enabling to tackle and simulate more specific and arduous CFD problems: viscoelastic flows by Fiétier and Deville [4–6], fluid-structure interaction problems by Bodard and Deville [7], large-eddy simulations of confined turbulent flows by Bouffanais et al. [8] and free-surface flows by Bouffanais and Deville [9].

It is well known that spectral element methods are amenable easily to parallelization as they are intrinsically a natural way of decomposing a geometrical domain [10].

The numerous references previously given and the ongoing simulations based on Speculoos highlight the achieved versatility and flexibility of this C++ toolbox. Nevertheless, ten years have passed between the first version of Speculoos' code and now, and tremendous changes have occurred at both hardware and software levels: fast dual DDR memory, RISC architectures, 64-bit memory addressing, compilers improvement, libraries optimization, libraries parallelization, increase in inter-connecting switch performance, etc.

Back in 1995, Speculoos was commonly compiled and was running on HP, Silicon Graphics workstations and also on the Swiss-Tx machine, a commodity-technology based computer with enhanced interconnect link between processors [11]. Currently most of the simulations based on Speculoos are compiled and are running on commodity clusters. The workstation world experienced a technical revolution with the advent of 'cheap' RISC processors leading to the ongoing impressive development of parallel architectures such as massively parallel clusters and commodity clusters. As a matter of fact, Speculoos benefited from this fast technical evolution as it was originally developed as to run in a single program, multiple data mode (SPMD) on a distributed-memory computer. The performance evaluations presented here are demonstrating the correlation between the good performances measured with Speculoos and the adequation of this code structure with the current hardware and software evolutions.

This paper is organized as follows. In Section 2 we introduce the numerical context in which Speculoos was initiated, the software aspects related to its implementation and the variable-size benchmark test case used for the performance evaluation presented in the subsequent sections. In Section 3 we present the performance analysis carried out on single-processor architectures and involving different compilers and compilation parameters. Section 4 is devoted to the parallel performance analysis achieved on a RISC-based commodity cluster. It is followed by an advanced parallel benchmarking presented in Sec. 5. Finally, in Section 6 we draw some conclusions on the results obtained.

2 Speculoos numerical and software context

In this section, is gathered the necessary background information regarding the numerical method—namely the spectral and mortar element method—, the object-oriented concept and the parallel paradigm, essential roots embodied in Speculoos. The final Section 2.4 introduces the two simulations used throughout this study as benchmark evaluation test cases.

2.1 Spectral and mortar element method

The spectral element method (SEM) is a high-order spatial discretization method for the approximate Galerkin solution of partial differential equations expressed in weak forms. The SEM relies on expansions on Lagrangian interpolants bases used in conjunction with particular Gauss–Lobatto and Gauss–Lobatto–Jacobi quadrature rules [12, 13]. As high-order finite element techniques, the SEM can deal with arbitrary complex geometry where h-refinement is achieved by increasing the number of spectral elements and p-refinement by increasing the Lagrangian polynomial order within the elements. From a high-order precision viewpoint, SEM is comparable to spectral methods as an exponential rate-of-convergence is observed when smooth solutions to regular problems are sought.

$\mathcal{C}^0$-continuity across element interfaces requires the exact same interpolation in each and every spectral elements sharing a common interface. The associated caveat to such conforming configurations is the over-refinement meshing generated in low-gradient zones. The adopted remedy to such nuisance is a technique developed by Bernardi et al. [14] referred to as the mortar element method. Mortars can be viewed as variational patches of the discontinuous field along the element interfaces. They relax the $\mathcal{C}^0$-continuity condition while preserving exponential rate-of-convergence, and thus allow polynomial nonconformities along element interfaces. Geometrical nonconformities are not

implemented yet.

2.2 Object-oriented programming

As mentioned in [2], the reasons for choosing C++ as the implementation language for such a spectral and mortar element simulation toolbox are numerous [15]: object-oriented concepts, widespread, non-proprietary, portability, efficiency, possibility to interface with C and Fortran subroutines. Joyner [16] noted that C++ embodies several weak points: it is complex, cryptic, uselessly permissive, it has no garbage collector—in one word, low-level. Dubois-Pèlerin and Van Kemenade [2] have attempted to overcome these drawbacks in Speculoos by the use of a restrictive but sufficient set of different C++ instructions and of several programming guidelines and style conventions which had pervaded the implementation.

Nevertheless, as compared to other widespread efficient programming languages such as Fortran 90, C++ constitutes a high-level programming approach, facilitating the know how transfer from one programmer to the other. In the past decade, this crucial feature has proved its relevance and efficiency as attested by the afore-given references [2–6, 8, 9]. However it seems reasonable to assess the impact of the accumulation of several implementation layers by different programmers over the years on the current efficiency of this C++ toolbox.

2.3 Parallel implementation

The complexity and the size of the large three-dimensional problems tackled by numericists in their simulations require top computational performance accessible from highly parallelized algorithms running on parallel architectures. As mentioned in [2], the implementation of concurrency in Speculoos was based on the concept that concurrency is a painful implementation constraint going against the high-level object-oriented programming concepts introduced in Section 2.2. As a matter of consequence, Speculoos parallelization was kept very low-level. In most higher-level operations parallelism does not even show up.

From a computational viewpoint, systems discretized with a high-order spectral element method rely mainly on optimized tensor-product operations taking place at the spectral element level. The natural data distribution for high-order spectral element methods is based on an elemental decomposition in which the spectral elements are distributed to the processors available for the run. It is worth noting that for very large computations, the number of spec-

tral elements can become relatively important as compared to the number of processors available for the computation. The design of Speculoos makes it possible to have several elements sitting on a single processor. Nodal values on subdomain interface boundaries are stored redundantly on each processor corresponding to the spectral elements having this interface in common. Moreover, this approach is consistent with the element-based storage scheme which minimizes the inter-processor communications. Inter-processor communication is completed by MPI instructions [17].

2.4 Benchmark evaluation test cases description

As a common practice in performance evaluation, it is important to build a tailor-made benchmark based on a numerical simulation corresponding to a concrete situation for the numericist. Before proceeding to the first step of our performance evaluation, we have short-listed some key parameters that have the most significant impact on the performance of our toolbox: single-processor optimization on the three architectures described above, single-processor profiling analysis, parallel implementation and scalability (including speedup, efficiency, communication times) and parallel implementation and processor dispatching. The main characteristics of the different computer architectures are presented in Table 1. In the advanced parallel benchmarking presented in Sec. 5, other machines and computer architectures have been used and are presented in Table 6.

Two test cases have been developed for this benchmark and an additional one for the advanced parallel benchmarking, see Sec. 5. All these three test cases belong to the field of CFD and consist in solving the Navier–Stokes equations for a Newtonian incompressible fluid. Based on the problem at hand, it is always physically rewarding to non-dimensionalize the governing Navier–Stokes equations which take the following general form

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \boldsymbol{\nabla} \mathbf{u} = -\boldsymbol{\nabla} p + \frac{1}{\mathrm{Re}} \boldsymbol{\Delta} \mathbf{u} + \mathbf{f}, \qquad \forall (\mathbf{x}, t) \in \Omega \times I, \tag{1}$$

$$\boldsymbol{\nabla} \cdot \mathbf{u} = 0, \qquad \forall (\mathbf{x}, t) \in \Omega \times I, \tag{2}$$

where $\mathbf{u}$ is the velocity field, p the reduced pressure (normalized by the constant fluid density), $\mathbf{f}$ the body force per unit mass and Re the Reynolds number expressed as

$$\mathrm{Re} = \frac{UL}{\nu}, \tag{3}$$

in terms of the characteristic length L, the characteristic velocity U and the constant kinematic viscosity ν. The system evolution is studied in the time interval $I = [t_0, T]$. From the physical viewpoint, Eqs. (1)–(2) are derived from the conservation of momentum and the conservation of mass respectively. For

incompressible viscous fluids, the conservation of mass also called continuity equation, enforces a divergence-free velocity field as expressed by Eq. (2). Considering particular flows, the governing Navier–Stokes equations (1)–(2) are supplemented with appropriate boundary conditions for the fluid velocity $\mathbf{u}$ and/or for the local stress at the boundary. For time-dependent problems, a given divergence-free velocity field is required as initial condition in the internal fluid domain.

The first two test cases are based respectively on a two-dimensional (2D) and pseudo three-dimensional (3D) Navier–Stokes simulations of the incompressible flow of decaying vortices in the domain Ω. An analytical solution of these 2D and pseudo 3D Navier–Stokes problems are available and a numerical solution is sought in a 2D or 3D framework. The discretization errors can therefore be explicitly calculated. The accuracy of the results of this benchmark is monitored using relative errors based on the $H^1(\Omega)$-norm (resp. $L^2(\Omega)$-norm) for the velocity field $\mathbf{v}$ (resp. pressure field p):

$$\varepsilon_{\mathbf{v}} = \frac{\|\mathbf{v} - \mathbf{v}_{\text{exact}}\|_{H^1(\Omega)}}{\|\mathbf{v}_{\text{exact}}\|_{H^1(\Omega)}} \qquad \text{and} \qquad \varepsilon_p = \frac{\|p - p_{\text{exact}}\|_{L^2(\Omega)}}{\|p_{\text{exact}}\|_{L^2(\Omega)}} \ . \tag{4}$$

The unsteady flow of decaying vortices has the following exact solution in a 3D framework

$$u(x,y,t) = -\cos x \, \sin y \, e^{-2t} \ , \tag{5}$$
$$v(x,y,t) = +\sin x \, \cos y \, e^{-2t} \ , \tag{6}$$
$$w(x,y,t) = 0 \ , \tag{7}$$
$$p(x,y,t) = -\frac{1}{4} \left[\cos(2x) + \cos(2y)\right] \, e^{-2t} \ . \tag{8}$$

Computations are carried out in the domain $\Omega = [0,\pi]^d$, where $d = 2, 3$ is the space dimension, with a Reynolds number Re equals to unity. This flow is very well documented [18–22] and therefore constitutes a suitable choice to test the accuracy of numerical methods and boundary conditions. As the exact solution is exponentially time-decaying, it is important to run the calculations on a short time interval $[0, t_{\text{final}}]$ to prevent a too important analytical decay of the solution, leading to a convergent solution anyway. A choice of $t_{\text{final}} = 10^{-2}$ seems reasonable, associated with a time-step $\Delta t = 10^{-3}$.

All our computations were carried out using two time integrators: the implicit backward-differentiation formula (BDF) of order 2 for the treatment of the viscous diffusive term and an extrapolation scheme (EX) [23,24] of same order for the nonlinear convective term. One type of pressure decomposition mode, based on a fractional-step method using pressure correction namely `BP1` [25–27] is used.

	Pleiades	Pleiades2	ielnx2	Itanium
Processor	Pentium 4	Xeon (mono-proc)	Xeon (bi-proc)	Itanium 2 (bi-proc)
Nprocs	1	1	1	1
R_∞ [GFlops/s]	5.6	5.6	3.4	5.3
Memory addressing	32-bit	64-bit	32-bit	64-bit
Memory type	Dual DDR	Dual DDR	Fast Rambus	Fast Rambus
Memory size	2 GB	4 GB	4 GB	4 GB

Table 1. The computer architectures used.

	Pleiades	Pleiades2	ielnx2	Itanium
M_∞ [GWords/s]	0.8	0.8	0.538	0.3
V_m [Flops/Word]	7	7	6.32	8.67
Compilers	gcc, icc	gcc	gcc	ecc
T_5 [s]	15'749	13'827	35'658	50'131

Table 2. The computer architectures characteristics: gcc the GNU project *C/C++* compiler, icc (resp. ecc) the Intel *C++* compiler, 32-bit (resp. 64 bit) version. T_5 is the process duration for test case 5 defined in Table 3.

Speculoos uses a Legendre SEM [12, 13, 28] for the spatial discretization of the Navier–Stokes equations. For the sake of simplicity the same polynomial order has been chosen in the different spatial directions ($N_x = N_y = N = 10$). Moreover, to prevent any spurious oscillations in our Navier–Stokes computations, the choice of a staggered $\mathbb{P}_N - \mathbb{P}_{N-2}$ interpolation method for the velocity and pressure respectively, has been made [28, 29]. As a consequence of this choice of a staggered grid, the inner-element grid for the x- and y-component of the velocity field is a Gauss–Lobatto–Legendre grid made up with 121 ($= (N+1)^2$) quadrature (nodal) points and the the grid for the pressure is a Gauss–Legendre grid made up with 81 ($= [(N-2)+1]^2$) quadrature (nodal) points, in each spectral element. Given E_x (resp. E_y) the number of spectral elements (or subdomains in the finite element nomenclature) in the x-direction (resp. y-direction), the total number of degrees of freedom dof is

$$\text{dof} = 2 \times (10E_x + 1)(10E_y + 1) + 81E_x \times E_y.$$

Table 3 summarizes the seven cases of variable size studied throughout this appendix, ranging from around 0.1 million dofs up to around 5 millions of degrees of freedom.

Case	$E_x \times E_y$	**Nb. of elements**	**Total nb. of dof**	**Size in memory**
1	16×16	256	72'578	15.73 MB
2	16×32	512	144'834	31.39 MB
3	32×32	1'024	289'026	62.64 MB
4	64×32	2'048	577'410	125.14 MB
5	128×32	4'096	1'154'178	250.13 MB
6	256×32	8'192	2'307'714	500.13 MB
7	256×64	16'384	4'610'306	999.15 MB

Table 3. Information on the variable-size test cases studied.

Two major concerns regarding the specificities of this benchmark case need to be addressed before performing extensive series of tests and afterwards drawing conclusions. The first concern is related to the scalability of the problem, in terms of computer effort, with respect to the number of degrees of freedom of the problem when increasing its dimension from a two-dimensional case to a three-dimensional one. The second concern is also related to the same problem of scalability as before, but now with respect to the number of degrees of freedom when varying the polynomial order in the two directions of the 2D problem. These issues are concurrently dealt with by measuring, on a single-processor architecture, the process durations for variable-size jobs for two- and three-dimensional cases based on the unsteady flow of decaying vortices. The 2D problem described by (5)–(8) is straightforwardly extended to a three-dimensional one. To maintain a number of degrees of freedom comparable to the ones for the 2D cases presented in Table 3, the number of spectral elements in the third dimension was kept to the unity—$E_z = 1$—the

polynomial order in this direction, N_z, varying, with $N_x = N_y = 10$, just like in the 2D cases. The results of these computations are reported in Figure 1. The 2D cases correspond to $N_x = N_y = 10$, the size is adjusted by changing the number of spectral elements E_x and E_y. Two series of 3D cases have been computed, the first one—crosses on Figure 1—represents $32 \times 32 \times 1$ spectral elements, with $N_x = N_y = 10$ and N_z varying between 2 up to 6. Finally, the second series of 3D cases—squares on Figure 1—represents $64 \times 32 \times 1$ spectral elements, with $N_x = N_y = 10$ and N_z varying between 2 up to 6.

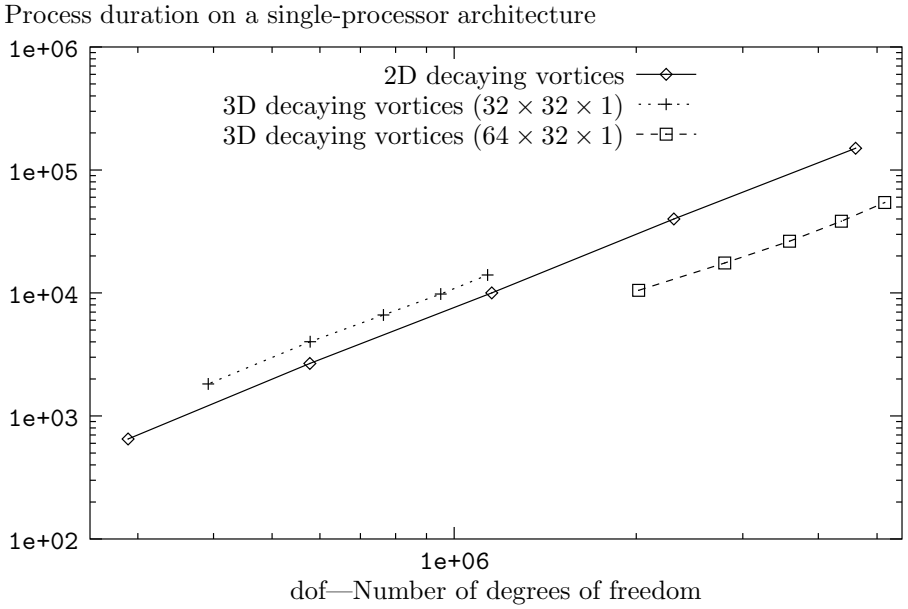

Fig. 1. Process duration for variable-size jobs running on a single-processor architecture (log-log scale). Diamonds, crosses and squares represent 2D cases with $N_x = N_y = 10$, 3D cases with $E_x = E_y = 32$ and $E_z = 1$, and 3D cases with $E_x = 64$, $E_y = 32$ and $E_z = 1$, respectively.

As expected from the tensor-product formulation of the SEM within each spectral element, Figure 1 shows a good scalability, in terms of computer effort, of Speculoos with respect to the number of degrees of freedom of the problem when changing both the dimension and/or the polynomial order in one or more directions. It is worth noting that the scalability may be affected by the use of specific preconditioners [23, 25].

The third test case is the fully three-dimensional simulation of the flow enclosed a in lid-driven cubical cavity at the Reynolds number of 12 000 placing us in the locally-turbulent regime. It corresponds to the case denoted under-resolved DNS (UDNS) in Bouffanais et al. [30]. The reader is referred to Bouffanais et al. [30] for full details on the numerical method and on the parameters used throughout the advanced parallel benchmark described in Sec. 5.

3 Single-processor performance analysis

Single-processor tests have been carried out and are presented in this section, relating to computer architectures, compiler optimization and finally profiling information.

3.1 Influence of the computer architecture

In this section, we will look into the hardware part of our performance analysis. The results described hereafter are obtained using test case 5, corresponding to 128×32 spectral elements. This choice of test case 5 is primarily based on the fact that it is an *intermediate case* in terms of memory resources and also in terms of computing time. In the sequel we will refer to this job run on a single-processor on the three different architectures described in Table 1 and Table 2.

The last row of Table 2 gives the process duration T_5 (for test case 5) obtained compiling in 32-bit mode with `gcc` (the GNU compiler, see 3.2) on `Pleiades` and on `ielnx2`, and compiling in 64-bit mode with `ecc` (the Intel C++ compiler) on Itanium 2. As expected from the architectures characteristics, `Pleiades` is the fastest platform: 2.3 times faster than `ielnx2` and 3.2 times faster than Itanium 2. These results are justified by the higher frequency of the Pentium 4 but also by its very efficient Dual DDR bus memory.

3.2 Compiler optimization

For the same C++ code, two compilers are available on the `Pleiades` cluster: `gcc` (or more precisely `g++` that is calling `gcc`) the GNU project C/C++ compiler and `icc` the Intel C++ compiler. Linking for both `gcc` and `icc` used the Intel `mkl_lapack 6.1`, `libmkl_def`, `libguide` and `libF90` libraries. Both of them have been tested with different options, mainly two families of options:

– *Optimization parameters*, offering different levels of optimizations from `O0` (no optimization) until `O3` where the code optimization is maximum;
– *Processor architecture and CPU type parameters*; on the cluster `Pleiades` where Pentium 4 CPUs are used, options `i686` and `pentium4` are notified to the compilers `gcc` and `icc` respectively.

Table 4 contains the results of several computations run on the same Navier–Stokes test case described in section 2.4 with 128×32 spectral elements (case 5—corresponding to approximately one million dofs). The second col-

umn shows the process duration in second for a single-processor run and the third column corresponds to the relative performance improvement for the run studied compared to the same run with the same options but with the other compiler (`gcc` if `icc` is considered and conversely).

Compiler optimization parameters	**Process duration (s)**	**R.P.I. g++/icc (%)**
`g++ (all options off)`	22'291	+ 6.8%
`g++ -O3 -march=i686`	16'189	+ 1.9%
`g++ -O3 -mcpu=i686`	16'192	+ 2.1%
`g++ -O3 -march=i686 -mcpu=i686`	15'749	+ 4.9%
`g++ -O3 -march=i686 -mcpu=i686 -finline-functions`	16'210	
`g++ -O3 -march=i686 -mcpu=i686 -finline-functions -msse2`	16'192	
`icc -O0`	23'918	- 7.3%
`icc (all options off)`	16'729	
`icc -O3`	16'553	
`icc -O3 -march=pentium4`	16'508	- 2.0%
`icc -O3 -mcpu=pentium4`	16'537	- 2.1%
`icc -O3 -march=pentium4 -mcpu=pentium4`	16'570	- 5.2%
`icc -O3 -march=pentium4 -mcpu=pentium4 -xW -ip`	16'560	

Table 4. Comparisons of the performances of two compilers `icc` and `gcc` (R.P.I standing for Relative performance improvement).

The results presented in the Table 4 lead us to the following remarks and conclusions:

– For all cases studied with different compiler options, the compiler `gcc` provides better results than `icc`. In the usual case, where the architecture and CPU types are defined and when the optimization is maximum (flag `-O3`), using `gcc` shortens the process duration by approximately 5% compared to the usage of `icc`. This performance improvement is non-negligible and for a run lasting 30 days, the choice of `gcc` will save one and a half day of computer resources;
– as `icc` is a compiler dedicated to Intel Pentium processors and architectures, it automatically detects the CPU and architecture types. This explains why the results of the computation with `icc` with all options off is close to the results where CPU and architecture types are specified. In addition, as the CPU and architecture types are automatically specified with `icc`, it is therefore possible to just optimize it by flagging with `-O3` when compiling. On the other hand, `gcc` requires the CPU or architecture type when flagging `-O3` for optimization purpose;
– we can also notice that by default `icc` optimizes the code (comparison of

cases with flags `-O0` and `-O3`) that is not the case of `gcc`. More importantly, one can notice that the difference between an aggressive optimization `-O3` and a size and locality optimization `-O1`—which corresponds by default to the case where all options are off—is only more than 1%. This suggests that Speculoos is well optimized, but indeed solely reflects the optimization of the libraries `Blas` and `Lapack`, which are substantially used by the code.

As a conclusion of this section on performance improvement due to compilation, we can say that with Speculoos, the GNU project compiler `gcc` is recommended, specifying both architecture and CPU type and with the maximum code optimization parameter on. To our knowledge, Speculoos is far from being the only C++ code producing more cost-effective simulations when compiled with `gcc` instead of its proprietary counterpart `icc`. The choice of `gcc` as compiler for Speculoos is primarily based on performance requirements but also on portability requirement, `gcc` being available on most of the current platforms—if not sources are available.

3.3 Linking optimization

Choosing `gcc` as compiler—see Section 3.2—, we have tested out linking the 32 Speculoos object files with three different groups of libraries. Speculoos requires `Blas`, `Lapack` and also a library ensuring the conversion from Fortran to C/C++. The same test case as in the previous section is used:

- Using `Blas 3.0`, `Lapack 3.0` and `libg2c` from the GNU project, the computing time is `18'680` s;
- Using the Intel `mkl_lapack 6.1`, `libmkl_def`, `libguide` and `libF90`, the computing time is `15'750` s;
- Using the Intel `mkl_lapack 6.1`, `libmkl_p4`, `libguide` and `libpthread`, the computing time is `26'833` s.

As a conclusion of this section, it appears clearly that the proprietary optimized libraries from Intel with the Fortran 90 converter, coupled with the non-proprietary compiler `gcc` produce the most efficient binary executable file. These results highlight the room for improvement for the GNU project in the framework of their optimized libraries associated to its efficient compiler.

3.4 Profiling

In annex is presented the most significant part of the profiling information, namely the flat profile. This information was gathered by `gprof` the GNU

profiler after compiling and linking on Pentium 4 of `Pleiades`, the 32 object files comprised by Speculoos using `gcc` together with the flag `-pg`.

For ease of presentation only the function calls representing the top 90% of the cumulative running time appears in the flat profile in annex.

In general the flat profile allows the programmer to spot the functions monopolizing the major part of the CPU resources. For scientific computations, these functions should correspond to fundamental operations such as multiplication-summation of vector-Matrix/Matrix-vector. Therefore optimizing these functions lead to a significant reduction of the computing time.

In our case, the intermediate test case 5 (128×32 elements) has been profiled and the most striking piece of information delivered by the flat profile is that no function represents a large and significant part of the running time. For instance the most costly function called, the first one in the profile

`RealVector` :: `Multiply`(`RealVector`$*$)

represents a bit less than 15% of the cumulative computing time. It means that optimizing this function could in the best case reduce its "cost" by 10 to 20%, representing finally a "saving" of only 1 to 3% of the total running time.

Knowing the complexity of the Speculoos C++ code—see Sec 2.2—, the optimization of the four first functions in the flat profile, namely:

(1) `RealVector::Multiply(RealVector*)`
(2) `FlatField::Multiply(Field*, int, int)`
(3) `Element::CopyAddValuesFrom(Element*)`
(4) `ElementaryField::MultiplyByWeights(int)`

corresponding each of them to more than 5% of the total computing time, is not worth the time to be invested by the programmer.

The second important piece of information revealed by the flat profile is gathered in the fourth column "`calls`" corresponding to the number of times a function was invoked by the code. A quarter of the functions listed in the flat profile (including the most costly (1): `RealVector::Multiply(RealVector*)`) are invoked a number of times reaching extremely high values with an order of magnitude of the billion! This remark highlights one of the weaknesses of Speculoos due to its implementation with very structured classes, templates and repeated function calls.

As a last point, it is important to notice that the flat profile obtained is somehow different from what could be expected from the profiling output of an optimized scientific computation. Several functions not involved in pure

calculations appear to occupy a non-negligible part of the cumulative time. From the flat profile in annex, 9 functions are of this type and cumulate about 12% of the total computing time. In addition, these functions are usually invoked a very large number of times as discussed earlier.

As a conclusion, the profiling information presented in this appendix is a proof of some weaknesses of Speculoos, in particular related to its excessive structured C++ implementation [15, 16].

4 Parallel implementation

In the sequel, we will assume that the reader is familiar with the basics of parameterization on a parallel machine. For a complete introduction to these notions we refer the reader to the following references [31, 32].

The speedup A of an application on a given parallel machine can be described as

$$A = \frac{\text{Computing time on one processor}}{\text{CPU plus communication times on } P \text{ processors}} = \frac{T_1}{T_P + T_C}. \qquad (9)$$

If we suppose that the computing effort strictly scales with P, then $T_1 = PT_P$ and the speedup can be written as

$$A = \frac{T_1}{T_P + T_C} = \frac{PT_P}{T_P + T_C} = \frac{P}{1 + \gamma_m/\gamma_a} = \frac{P}{1 + 1/\Gamma}, \qquad (10)$$

where γ_m and γ_a are introduced in [31], and $\Gamma = \gamma_a/\gamma_m$. The efficiency E of a parallel machine is defined by

$$E = \frac{A}{P} = \frac{1}{1 + 1/\Gamma}. \qquad (11)$$

The quantity $P_{1/2}$ denotes the number of processors for which the efficiency E is half, and $A_{1/2} = A(P_{1/2})$ is the speedup for $P = P_{1/2}$ processors.

4.1 Influence of processor dispatching on performance

In this section we will discuss the effect of specifying the way of dispatching the spectral elements—or subdomains—to the processors for a job running in parallel on a cluster architecture such as `Pleiades` for instance in our case.

With Speculoos, the first basic procedure to assign a processor to every spectral element is called

`DispatchElements`().

It is a non-optimized algorithm in which the spectral elements are assigned to a processor following a cycle, with a cycle-length equal to the number of processors available for the run. It is clear that this non-optimized method will increase the communications between processors.

In order to improve on this point and to optimize the communications between the processors a new procedure to assign a processor to every spectral element, called

`DispatchElementsByBlocks`(`nx`, `ny`),

has been implemented. The computational grid is assumed to be structured, in the sense that its elements are assumed to be generated by a `nx*ny` distribution on a face element. The elements are assigned by blocks, in order to minimize inter-processor communications.

The functions `DispatchElements()` and `DispatchElementsByBlocks(nx,ny)` have been tested on our intermediate test case 5 (128×32 spectral elements) with a number of processors ranging between 1 and 16. Figure 2 summarizes the results in terms of parallel efficiency E. The increase in efficiency E between the non-optimized and the optimized processor dispatching procedures is almost constant with the number of processors `Nprocs` and is equal to 7–8%.

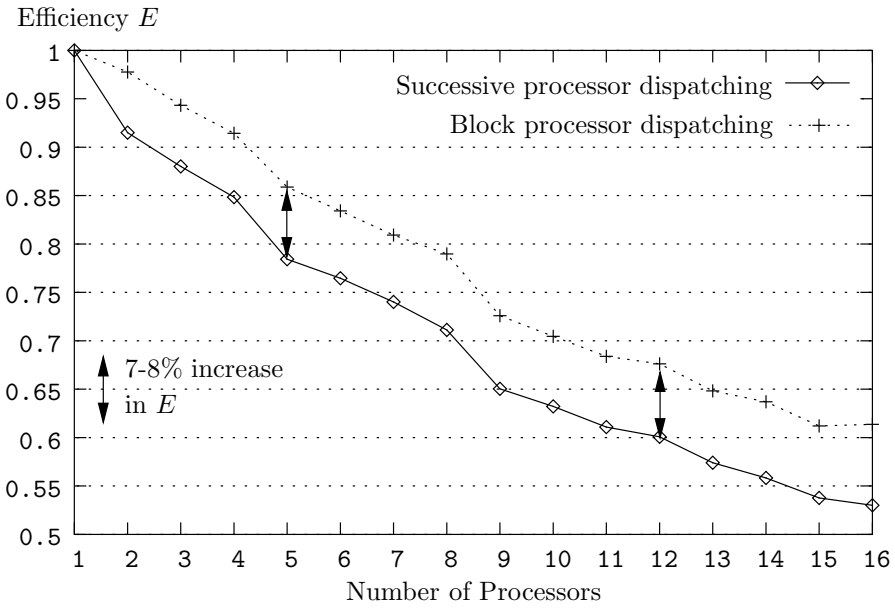

Fig. 2. Performance improvement due to an optimized dispatching of the spectral elements by blocks.

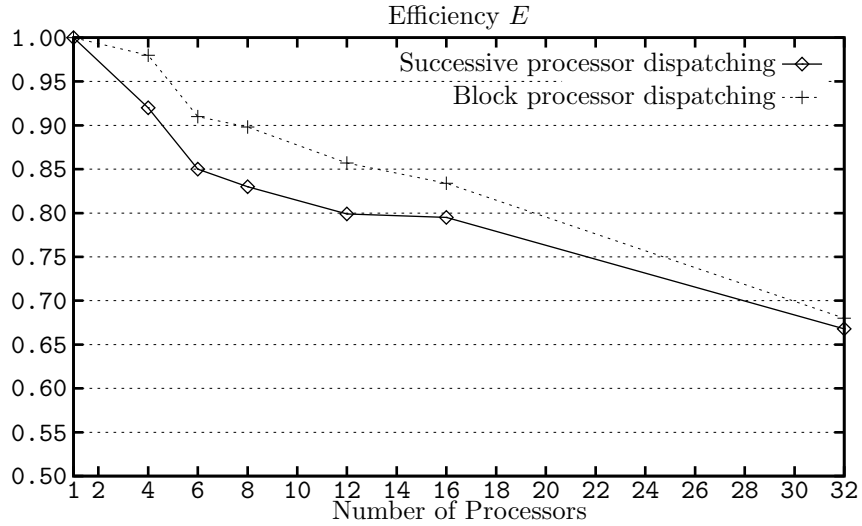

Fig. 3. Performance improvement due to an optimized dispatching of the spectral elements by blocks. `Pleiades2` Xeon

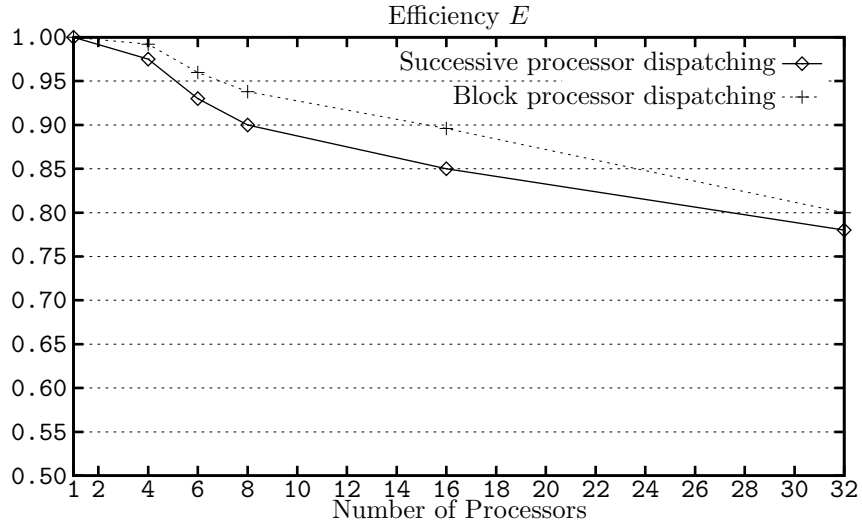

Fig. 4. Performance improvement due to an optimized dispatching of the spectral elements by blocks. `Pleiades1` MPICH2.

4.2 Scalability study and influence of the size of the problem

As expected from the theory, the scaling of our computations depends highly on the load-balancing of our cases among the processors available for the run. Therefore the smaller the case studied (in terms of memory and CPU resources) the longer the communications, reducing *ipso facto* the speedup A and the efficiency $E = A/P$ measuring the scalability.

As can be seen on Fig 6 and 7, cases 5–7 (128×32–256×64 elements) scale almost perfectly for a number of processors up to four. Cases 1–2 (16×16–16×32 elements) are too small and therefore do not load sufficiently the processors

and their associated shared memory. In these cases, parallel computations are not justified: $P_{1/2}$ is close to 4 and 6 (see Table 5), corresponding to an efficiency of only 50%. The waste of resources is due to communications and a detailed analysis is given in the next section.

Case	$E_x \times E_y$	$P_{1/2} \Leftrightarrow E(P_{1/2}) = 0.5$	$A_{1/2} = A(P_{1/2})$
1	16×16	4	$\gtrsim 2$
2	16×32	6	$\gtrsim 3$
3	32×32	9	$\simeq 4.5$
4	64×32	12	$\lesssim 7$
5	128×32	$\simeq 20$	$\simeq 11$
6	256×32	32	$\simeq 17$
7	256×64	> 32	> 20

Table 5. Values of $P_{1/2}$ and $A_{1/2}$ for the seven test cases.

For small values of the number of processors P ($P \leq 4$), the load-balancing is acceptable for all cases studied leading to efficiencies very close to 1 (see Fig. 9) and an almost perfect scalability.

For a larger number of processors ($16 \leq P \leq 32$) Fig. 8 (magnification on the left side) shows a very poor load-balancing for the small cases. The speedups comprised between $A(P = 16)$ and $A(P = 32)$ are smaller than the speedup $A(P = 8)$ and even $A(P = 4)$ for $A(P = 32)$. These observations translate into very poor efficiencies (smaller than 0.15) observed on Fig. 9.

For information, a dimensionalized graph showing the computing time T_P in seconds, with the number of processors varying between 1 and up to 32 is given on Fig. 5.

4.3 Communications

One major concern when running jobs on a parallel architecture is to have the inter-processor communications taking a too important part of the computing time. This is even more important when the network switch is like in the case of `Pleiades` a Fast Ethernet switch running at only 10 MB/s. Even with a Gigabit Ethernet switch, communications still remain a point of special attention.

Fig. 10 and Fig. 11 show that for the large cases, namely cases 6 and 7 (256×32 and 256×64 elements), the communications represent only a very small fraction 2–3% of the total computing time, for all values of `Nprocs` varying from 2 up to 32. For intermediate cases 4–5 (64×32 and 128×32 elements respectively), communications take less than one tenth of the total elapsed

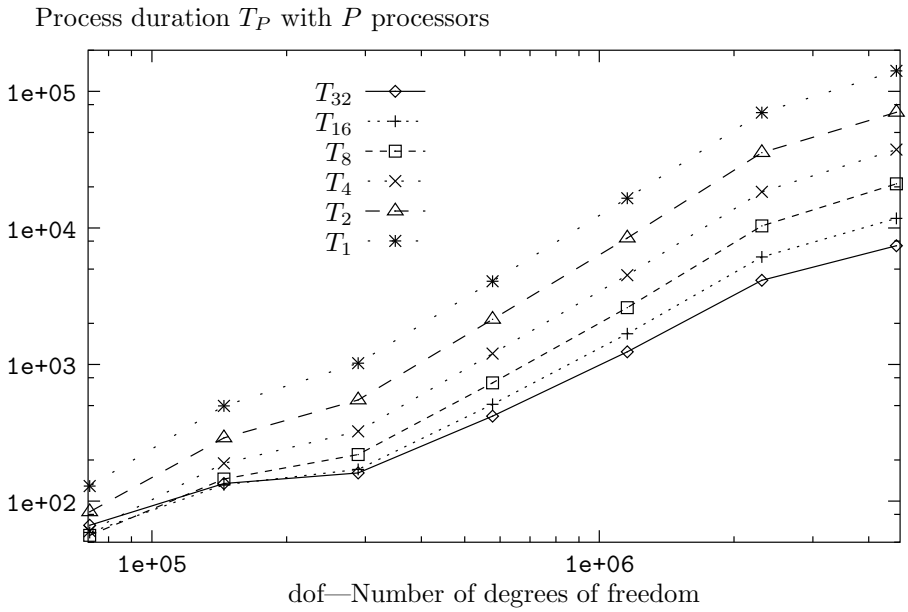

Fig. 5. Process durations for variable-size jobs running on 1 up to 32 processors (log-log scale).

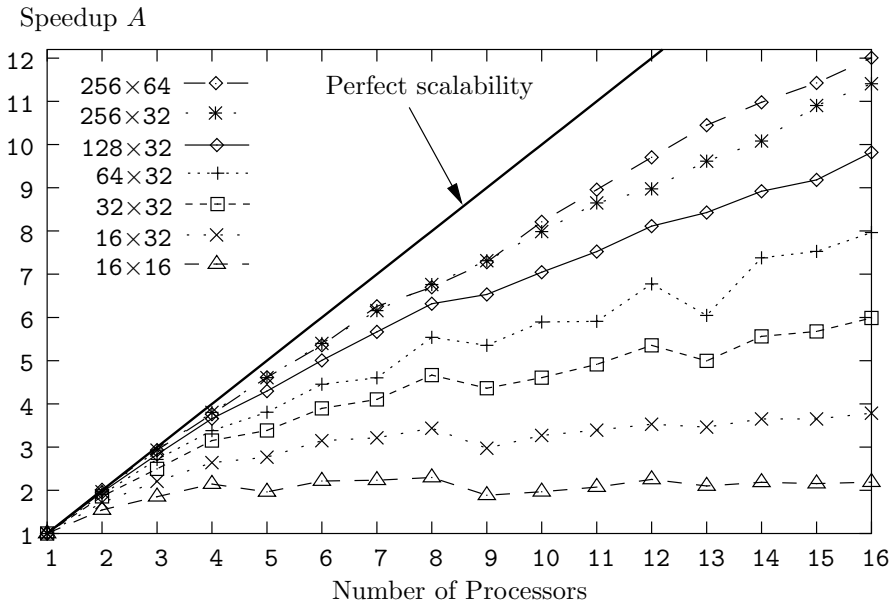

Fig. 6. Evolution of the speedup A against the number of processors `Nprocs` for the seven test cases 1–7.

time, that is still acceptable. Finally for small cases 1–3, the communication times reach unacceptable values and can even take over the computing time, like in the smallest case studied, 16×16 with `Nprocs` $= 32$.

One case of 'superlinearity' is observed, corresponding to the largest test case 7 (256×64 elements and dof $\simeq 5.10^6$) using `Nprocs` $= P = 2$ processors. This phenomenon of superlinearity corresponds to a speedup $A > P$ (see Fig. 6

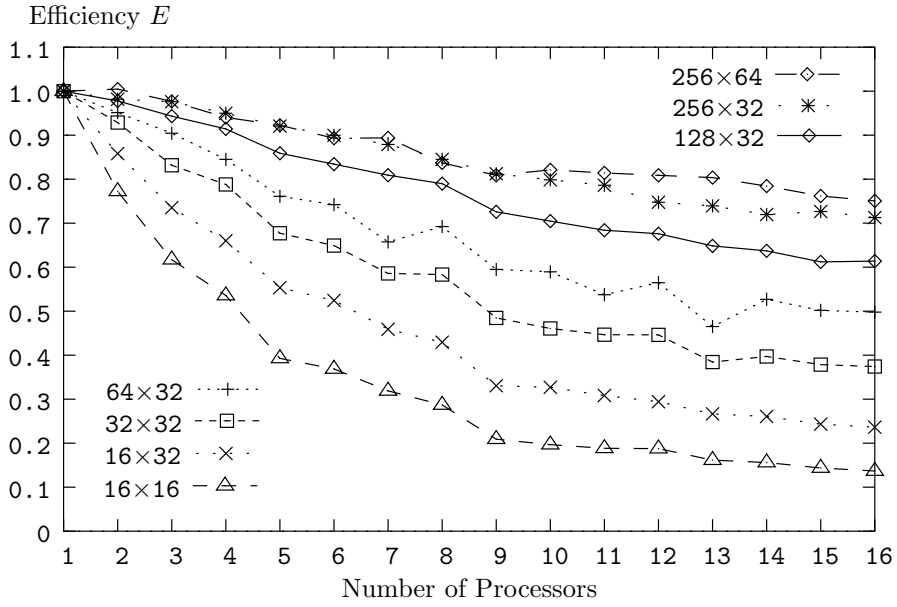

Fig. 7. Evolution of the efficiency E with the number of processors `Nprocs` for the seven test cases 1–7.

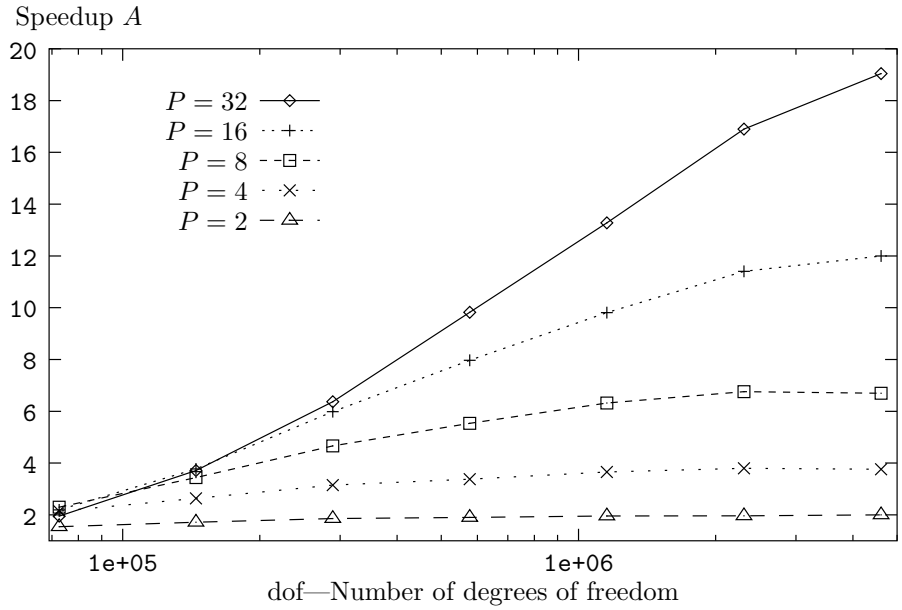

Fig. 8. Evolution of the speedup A as a function of the size of the test case, for `Nprocs`$= 2^k$, with $k = 1, .., 5$.

and Fig. 8), an efficiency $E > 1$ (see Fig. 7 and Fig. 9) and also to a negative communication time $T_C < 0$ (see Fig. 10). This so-called superlinearity can be explained by the fact that with Speculoos, for very large cases and small number of processors, the computing effort scales with `Nprocs` $= P$: $T_1 \neq PT_P$.

Speculoos code running on a single processor is slightly slowed down by the useless MPI commands leading to a single-processor computing time T_1(MPI) greater than the real T_1 that could be measured without using any MPI com-

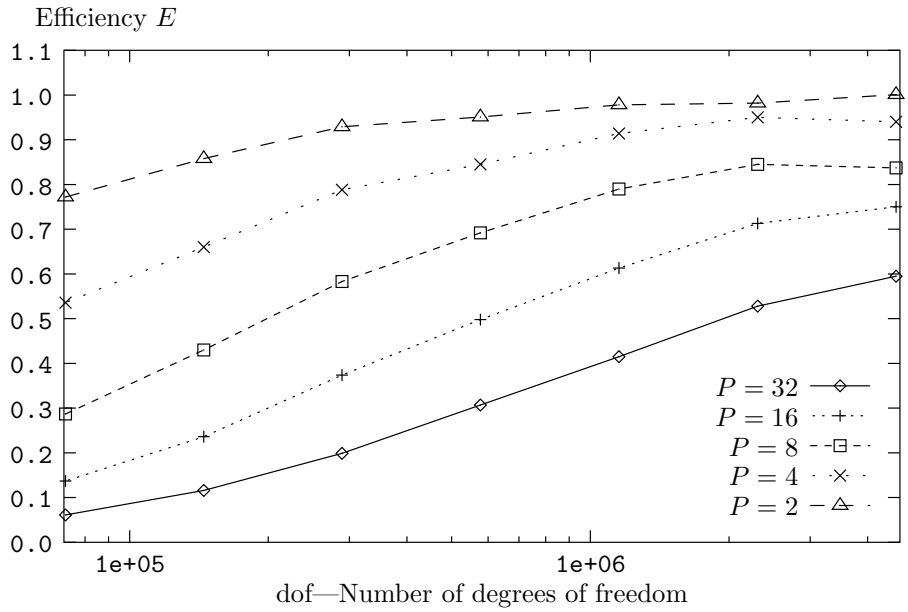

Fig. 9. Evolution of the efficiency E as a function of the size of the test case, for `Nprocs`$= 2^k$, with $k = 1, .., 5$.

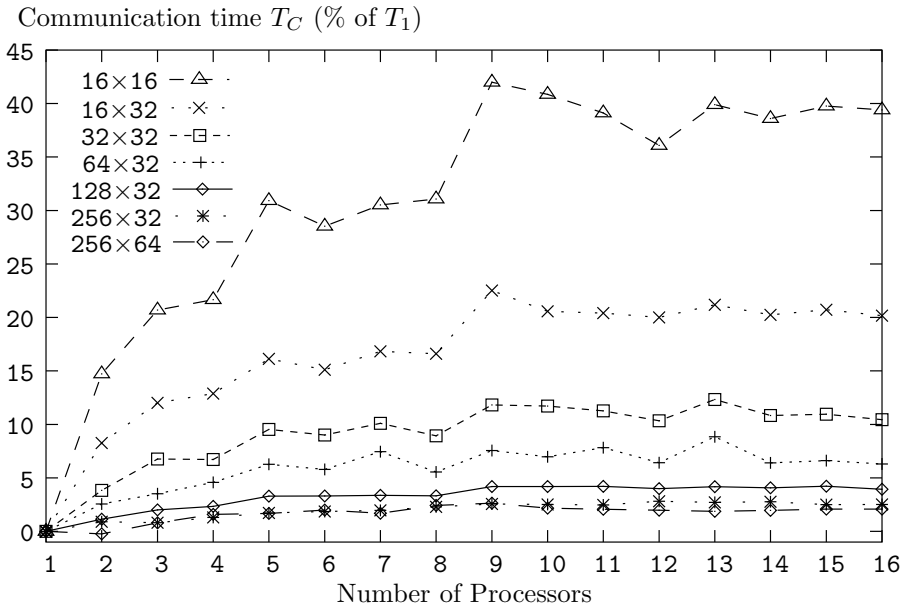

Fig. 10. Communication time T_C (as a percentage of the process duration with a single-processor T_1) as a function of `Nprocs` for the 7 test cases.

mands.

For cases not too large (dof $\leq 10^6$), Fig. 11 shows that the communications T_C/T_1 decreases linearly (in log-log scales) with the number of dof and in addition the corresponding negative slope is the same whatever the number of processors `Nprocs` $= P$. This observation translates into the following heuristic

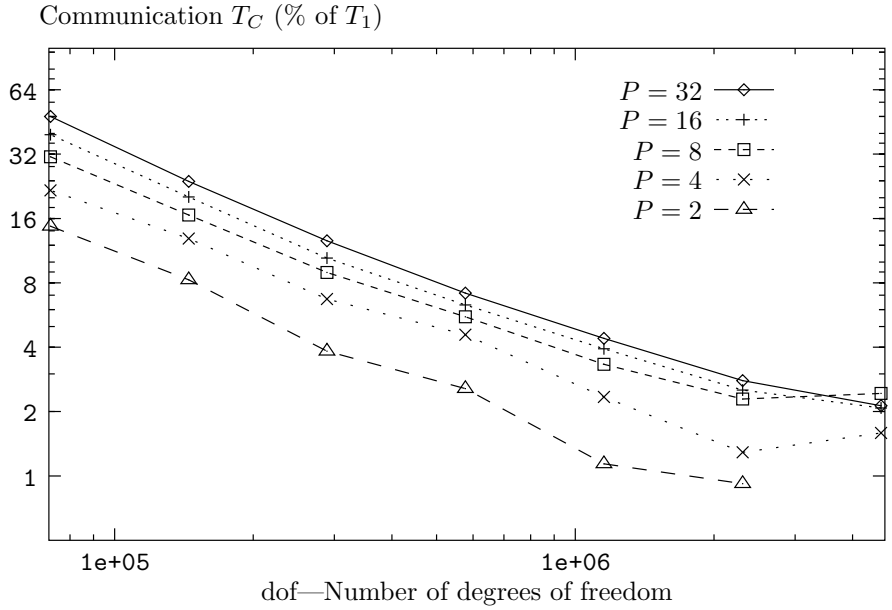

Fig. 11. Communication time T_C (as a percentage of the process duration with a single-processor T_1) as a function of the size of the test case, for `Nprocs`$= 2^k$, with $k = 1, .., 5$.

power-law scaling

$$\frac{T_C}{T_1} = K(P)\,\mathrm{dof}^{-\alpha}, \tag{12}$$

where $-\alpha$ is the slope measured on Fig. 11—approximately $\alpha = 0.9$—and K is a function of the number of processors P.

5 Advanced parallel benchmarking

As mentioned in Sec. 2.4, the test case used throughout this section is the simulation of the lid-driven cubical cavity flow, detailed in Bouffanais et al. [30].

5.1 Speculoos characteristics

Speculoos uses a small amount of main memory. Parallelization is made in order to reduce the high overall computing time. The performance measurements are made at time-step number 4, the first 3 time-steps also include initialization operations. The number of elements and the polynomial degrees in the three space directions are denoted by E_x, E_y, and E_z, and N_x, N_y, and N_z, respectively. The total number of independent variables per element is therefore $n_v \times (N_x + 1) \times (N_y + 1) \times (N_z + 1)$, where n_v is the number of

vector components per Gauss–Lobatto–Legendre (GLL) quadrature point. In addition, there are $E_x \times E_y \times E_z$ elements.

5.2 Hardware and software used

To perform the Speculoos code benchmark, the machines presented in Table 6 have been used.

Name	**Manufacturer**	**CPU type**	**Nodes**	**Cores**	**Interconnect**
`Gele`	Cray	Opteron DC	16	32	SeaStar
`Pleiades`	Logics	Pentium 4	132	132	FE
`Pleiades2`	Dell	Xeon	120	120	GbE
`Pleiades2+`	Dell	Xeon 5150	99	396	GbE

Table 6. Characteristics of the machines used for the benchmark. DC=Dual-Core.

As mentioned previously, the Speculoos code is written in C++, uses BLAS operations and implements the Message Passing Interface (MPI). The benchmarks have been performed using the compilers and libraries versions shown in Table 7.

Compiler	**Company**	**Machine**	**Version**
`icc`	Intel	`Pleiades`	9.0
`icc`	Intel	`Pleiades2`	9.1e
`icc`	Intel	`Pleiades2+`	9.1e
`g++`	Intel	`Pleiades`	4.2.0
`CC` (pgCC)	Cray (PGI)	`Gele`	6.1-4 64 bits
Library	**Company**	**Machine**	**Version**
MPICH	GNU	`Pleiades`	1.2.7
MPICH2	GNU	`Pleiades`	1.0.5
MPICH	GNU	`Pleiades2`	1.2.7
MPICH2	GNU	`Pleiades2`	1.0.5
MPICH	GNU	`Pleiades2`	1.2.7
MPICH2	GNU	`Pleiades2+`	1.0.5
MPICH2-nemesis	GNU	`Pleiades2`	1.0.5
MPICH2-nemesis	GNU	`Pleiades2+`	1.0.5
MPICH2	Cray	`Gele`	MPT 1.3
MKL	Intel	`Pleiades`	7.1
MKL	Intel	`Pleiades2`	8.1e
MKL	Intel	`Pleiades2`	9.0e
MKL	Intel	`Pleiades2+`	8.1e
MKL	Intel	`Pleiades2+`	9.0e
ACML	AMD	`Gele`	3.0
PAPI	GNU	`Gele`	3.2.1

Table 7. Characteristics and versions of the software used for benchmarking. MPT stands for Message Passing Toolkit.

The PAPI (Performance API) [33] available on the Cray XT3 machine was used to measure the numbers O of operations (in GFlops) and the MFlops/s rate of Speculoos. The VAMOS service available on the three Pleiades clusters [34] maps the hardware related data from the Ganglia monitoring tool [35] with the application and user related data (from cluster RMS and Scheduler). We used the most aggressive optimization flag on all machines (`-O3` flag).

5.3 Fixed problem size

The first measurements are done on `Pleiades2` with a fixed problem size, $E_x = E_y = E_z = 8$; $N_x = N_y = N_z = 8$; $O = 155.4$ GFlops, and varying the number P of processing elements from 1 to 60. The evolution of the runtime (for one time-step), the associated MFlops/s rate, and the efficiency E are given in Table 8. The speedup A as a function of the number of processors is plotted in Fig. 12. One sees that with 8 processors a speedup of 7 can be reached and a speedup of 20 with 26 processors.

P	**GFlops/s**	**Runtime (1 step)**	E
1	0.638	243.59	1.00
2	1.251	124.23	0.98
3	1.901	81.75	0.99
4	2.395	64.88	0.94
5	3.038	51.15	0.95
6	3.566	43.58	0.93
7	4.101	37.89	0.92
8	5.590	34.52	0.88
16	8.346	18.62	0.82
32	14.179	10.96	0.70

Table 8. Evolution of GFlops/s rate and runtime for fourth time-step. E: Efficiency.

5.4 Increase CPU performance

In this section, the number of processors on a Cray XT3 is kept fixed at the value $P = 4$. Then, we modify the polynomial degree and measure the MFlops/s rate. The MFlops/s rate performance metric for each process element is shown on Table 9. It increases as the problem size increases. As expected, one deduces that there is a limit on the number of processors that should be used in parallel.

$E_x - E_y - E_z$	$N_x - N_y - N_z$	**MFlops/s**	**Walltime**
$8-8-8$	$6-6-6$	1624	18.54
$8-8-8$	$7-7-7$	2580	29.79
$8-8-8$	$8-8-8$	3100	50.07
$8-8-8$	$9-9-9$	3700	83.12
$8-8-8$	$10-10-10$	4150	146.97
$8-8-8$	$11-11-11$	4390	257.36

Table 9. Evolution of MFlops/s rate and runtime for one time-step on 4 Cray XT3 dual-CPU nodes as a function of the polynomial degree.

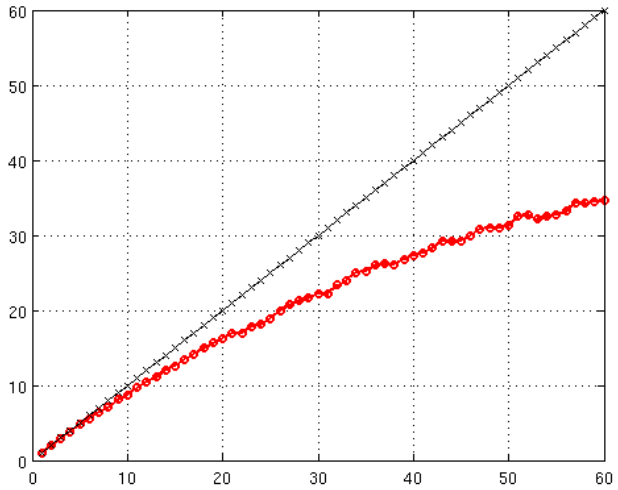

Fig. 12. Speedup of Speculoos code on the `Pleiades2` (Xeon CPU).

5.5 Varying the number of processing element P with problem size

A more common way to measure scalability, and to overcome Amdahl's law, is to fix the problem size per processor and to increase the number of processors with the overall problem size. In other words, one tries to fix Γ that measures the ratio between processor needs over communication needs. We show in Table 10 the scalability of Speculoos on the `Pleiades2+` cluster. It was compiled using MPICH2 and `icc` C++ compiler version 9.1e.

Table 10 (A) shows results obtained when all the 4 cores are active for $P > 1$. Note that one Woodcrest node with 2 dual-core processors (Table 10) is slightly faster than 4 dual-CPU nodes (Table 9) of the Cray XT3. When increasing the number of nodes with the problem size, the MFlops/s rate per core remains the same for all the cases. At this point, it is legitimate to determine if Speculoos is memory or processor bound. To find out, all the test cases in Table 10 have been resubmitted to the Woodcrest nodes, first (A) using all the 4 cores per node, then (B) restricting to two the maximal number of MPI threads per node. Thus, instead of 16 nodes, 32 nodes were used to run the 64-processor

$E_x - E_y - E_z$	$N_x - N_y - N_z$	**Nodes-Cores**	**Elem/Core**	**Walltime**
$4-4-4$	$8-8-8$	1 - 1	64	8.68
$8-8-8$	$8-8-8$	2 - 8	64	39.26
$16-16-16$	$8-8-8$	16 - 64	64	147.97

(A)

$E_x - E_y - E_z$	$N_x - N_y - N_z$	**Nodes-Cores**	**Elem/Core**	**Walltime**
$4-4-4$	$8-8-8$	1 - 1	64	8.68
$8-8-8$	$8-8-8$	4 - 8	64	33.50
$16-16-16$	$8-8-8$	32 - 64	64	111.71

(B)

Table 10. Scalability of Speculoos. Same polynomial degree, same number of elements on each compute node on `Pleiades2+` (Woodcrest) cluster. **(A):** with 4 MPI threads per node. **(B):** with $npernode = 2$, two MPI threads per node.

case (see Table 10 (B)). One sees that the overall CPU time has been reduced by 20%, but the number of nodes was doubled. This shows that Speculoos includes parts that are processor bound and others that are memory bound. As a consequence, using all 4 cores does not give a two fold speedup (as one expects for a processor bound program) but neither the speedup is zero (as for a main memory bound application). Therefore, it is always more efficient to run Speculoos on all the 4 cores per node.

5.6 CPU usage and the Γ *model*

CPU usage has been monitored by the VAMOS monitoring service [34] available on the Pleiades clusters. It provides information on the application's behavior. The higher the CPU usage is, the better the machine fits to the application. To perform that monitoring we took the same problem size ($E_x = E_y = E_z = 8$ and $N_x = N_y = N_z = 8$) during the same computing duration (10 hours = 36'000 seconds). The application is run for 10 hours and the number of iteration steps performed during this time is counted. With such a methodology, we ensure that each sample can perform a maximum of calculations in a given amount of time. It is equivalent to set the same number of iteration for each sample and to measure the walltime.

Figure 13 shows the different behavior of Speculoos on the three different architectures. The Γ value—introduced in Eq. (10) and, which reflects the "fitness" of a given application on a given machine [31]—is also computed. Results are reported in Table 11.

Using the notations introduced earlier, T, T_P, T_C, and T_L denote the total walltime, the CPU time for P processing elements, the time to communicate,

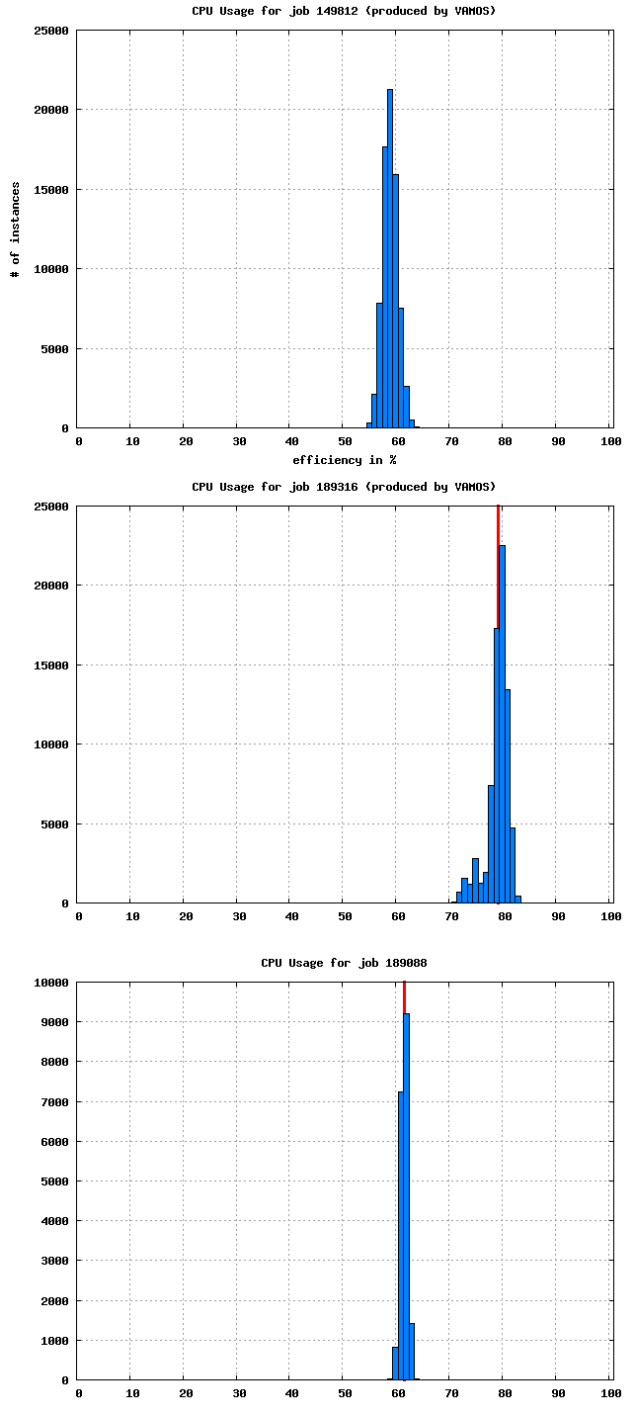

Fig. 13. CPU Usage of Speculoos on different machines. Top: `Pleiades` cluster (CPU usage average 51.05% ,$\Gamma = 1.04$).Middle: `Pleiades2` cluster (CPU usage average = 79.24%, $\Gamma = 3.81$). Bottom: `Pleiades2+` cluster (CPU usage average 61.6%, $\Gamma = 1.60$).

and the latency time per iteration step, respectively. Then,

$$T = T_P + T_C + T_L, \tag{13}$$

and the parameter Γ is easily expressed as

$$\Gamma = \frac{T_P}{T_C + T_L}. \tag{14}$$

	T [s]	Γ	**b [MB/s]**	**S [words]**	T_P [s]	T_C [s]	T_L [s]
`Pleiades`	23.01*	1.44*	12*	$180 * 10^6$	13.58	8.43	1
`Pleiades2`	9.55*	3.81*	101	$180 * 10^6$	7.56	0.98	1
`Pleiades2+`	12.89*	1.60*	101	$180 * 10^6$	7.93	3.96	1

Table 11. Measured (*) and computed quantities using the Γ model.

It is possible to measure the total time T by means of an interpretation of the CPU usage plots (see Fig. 13). Indeed, the middleware Ganglia determines for every time interval of 20 seconds the average CPU usage (or efficiency E) for each processing element. This information has to be put into relation to the Speculoos application. This is done via the middleware VAMOS. In the plots in Fig. 13, are added up all the values of E that lie in between x and $x + 0.01$, where x is the percentile represented on the abscissae of the plots. The efficiency E is related to the Γ through

$$\Gamma = \frac{E}{1 - E}. \tag{15}$$

What can also be estimated are the network bandwidths b of the GbE switch (between $b = 90$ and 100 MB/s per link), the network bandwidth of the Fast Ethernet switch (between $b = 10$ and 12 MB/s per link) and the latency ($L = 60$ μs for both networks). First, a consistency test of those quantities is performed. Assuming that the Fast Ethernet switch has a fix bandwidth of $b_1 = 12$ MB/s, and for the GbE switch $b_2 = \alpha b_1$, with α unknown. Another unknown is the number of words S that is sent per node to the other nodes, and $T_C = S/b$. Based on the previous assumptions, the three Γ values for the three machines and the two networks is expressed as

$$\Gamma_1 = \frac{T_{P_1}}{S/b_1 + T_L}, \tag{16}$$

$$\Gamma_2 = \frac{T_{P_2}}{S/b_2 + T_L}, \tag{17}$$

$$\Gamma_3 = \frac{T_{P_3}}{S/b_2 + T_L}. \tag{18}$$

These constitute a set of three equations for three unknown variables, namely S, α, and T_L. Solving for these variables leads to $T_L = 1$, $S = 180$ MWords, and $\alpha = 8.43$. The value of $b_2 = 101$ MB/s corresponds precisely to the one measured. This means that the model is well applicable.

5.7 Modification of the number of running threads per SMP node

To demonstrate that Speculoos is dominated by inter-node communications, Figure 14 shows the result of two runs of the same problem size ($E_x = E_y = E_z = 8$ and $N_x = N_y = N_z = 8$) made respectively on 4 and 8 Woodcrest nodes during the same period of time (1h = 3600 seconds) and counting the number of iteration steps. The first sample was launched forcing 2 MPI threads on each node and the second with 4 MPI threads on each node.

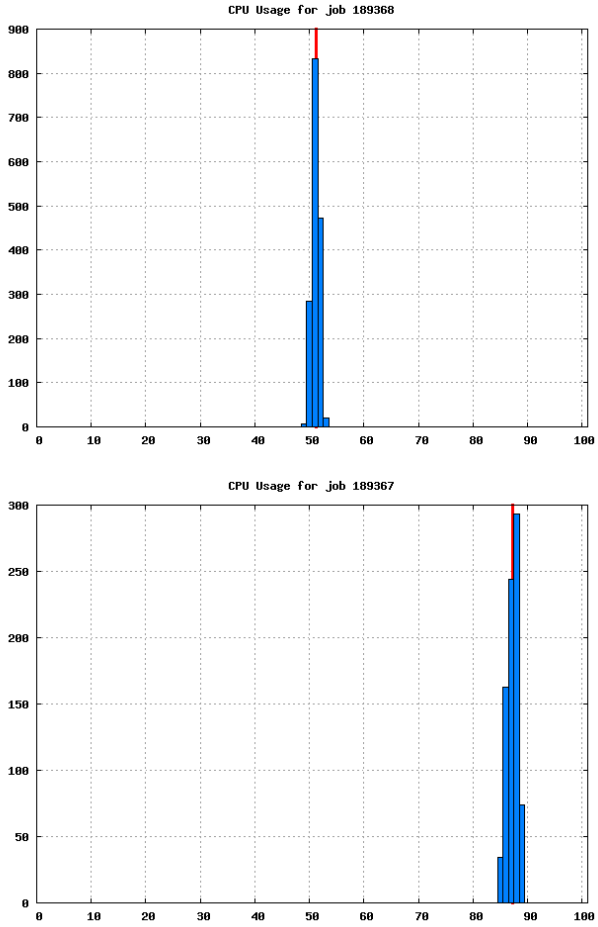

Fig. 14. CPU Usage on the 5100-series SMP node of `Pleiades2+` cluster. 16 processing elements were required. 8 nodes/2 cores with 2 MPI threads per nodes in the upper case, 4 nodes/4 cores with 4 MPI threads per node in the lower case.

We have to note that the CPU usage (system+user+nice) monitored by Ganglia is the sum of all the process elements. For instance, for a dual-processor machine, when Ganglia measures 50% CPU usage, it means that each processor run at 100%. In Figure 14, when 2 MPI threads are blocked per node, we get a CPU usage of 51.13% while 157 iteration loops have been performed during one hour; when 4 MPI threads run on each node, we get a CPU usage of 87.25% while only 117 iteration loops have been performed during one

hour. Thus, the real CPU usage for the sample with 2 MPI threads per node is above 100% (2 cores are unused).

6 Conclusions

The extensive performance review presented in this paper for the high-order spectral and mortar element method C++ toolbox, Speculoos, has shown that good performances can be achieved even with relatively common available software and hardware resources—small commodity clusters with non-proprietary compilers installed on it. However, universal and commonly employed non-proprietary libraries such as `Blas` and `Lapack` seem to require a further improvement in their optimization as compared to proprietary ones.

As a complement to the previous partial conclusions provided at each step of this performance evaluation, we can conclude that the main implementation choices made a decade ago reveal their promises. Even though those choices could have been questionable ten years ago, they are now in line with the current trend in computer architecture developments with the generalization of commodity and massively parallel clusters.

Moreover the analysis reveals some weaknesses of our C++ code. As presented in Section 2.2, those weaknesses are inherent to C++ itself and cannot be circumvented. A proper compiler optimization and a parallel implementation could at least balance them. The trade-off between the development and implementation advantages of the object-oriented paradigm, and the computational efficiency of a lower-level programming language—Fortran 90 for instance—is not easily accessible.

The parallel implementation of Speculoos based on MPI has shown to be efficient. Reasonable scalability and efficiency can be achieved with a high loading of the cluster nodes. In addition, a smart assignment of the spectral elements to the cluster nodes leads to an additional increase in the parallel performance of Speculoos. These results support the original choices made in Speculoos parallel implementation by keeping it at a very low-level. Nevertheless, it would be interesting to further investigate and evaluate the scalability for a number of computer nodes corresponding to hundreds or even thousands. Therefore the part of this performance evaluation devoted to the parallelism of Speculoos will soon be completed by carrying out the presented benchmark test cases on the IBM eServer Blue Gene Solution, massively parallel computer comprising 8'192 processors acquired by EPFL.

Acknowledgements

This research is being partially funded by a Swiss National Science Fundation Grant (No. 200020–101707) and by the Swiss National Supercomputing Center CSCS, whose supports are gratefully acknowledged.

The results were obtained on supercomputing facilities at the Swiss National Supercomputing Center CSCS and on Pleiades clusters at EPFL–ISE.

References

[1] V. Van Kemenade, Incompressible fluid flow simulation by the spectral element method, Tech. rep., "Annexe technique projet FN 21-40'512.94", IMHEF–DGM, Swiss Federal Institute of Technology, Lausanne (1996).

[2] Y. Dubois-Pèlerin, V. Van Kemenade, M. Deville, An object-oriented toolbox for spectral element analysis, J. Sci. Comput. 14 (1999) 1–29.

[3] Y. Dubois-Pèlerin, Speculoos: an object-oriented toolbox for the numerical simulation of partial differential equations by spectral and mortar element method, Tech. Rep. T-98-5, EPFL–LMF (1998).

[4] N. Fiétier, Detecting instabilities in flows of viscoelastic fluids, Int. J. Numer. Methods Fluids 42 (2003) 1345–1361.

[5] N. Fiétier, M. O. Deville, Linear stability analysis of time-dependent algorithms with spectral element methods for the simulation of viscoelastic flows, J. Non-Newtonian Fluid Mech. 115 (2003) 157–190.

[6] N. Fiétier, M. O. Deville, Time-dependent algorithms for the simulation of viscoelastic flows with spectral element methods: applications and stability, J. Comput. Phys. 186 (2003) 93–121.

[7] N. Bodard, M. O. Deville, Fluid-structure interaction by the spectral element method, J. Sci. Comput. 27 (2006) 123–136.

[8] R. Bouffanais, M. O. Deville, P. F. Fischer, E. Leriche, D. Weill, Large-eddy simulation of the lid-driven cubic cavity flow by the spectral element method, J. Sci. Comput. 27 (2006) 151–162.

[9] R. Bouffanais, M. O. Deville, Mesh update techniques for free-surface flow solvers using spectral element method, J. Sci. Comput. 27 (2006) 137–149.

[10] P. F. Fischer, A. T. Patera, Parallel spectral element solution of the Stokes problem, J. Comput. Phys. 92 (1991) 380–421.

[11] R. Gruber, A. Gunzinger, The Swiss-Tx supercomputer project, EPFL Supercomputing Review 9 (1997) 21–23.

[12] Y. Maday, A. T. Patera, Spectral element methods for the incompressible Navier–Stokes equations, State-of-the-Art Survey on Computational Mechanics, A. K. Noor & J. T. Oden Eds., ASME, New-York, 1989, pp. 71–142.

[13] A. T. Patera, Spectral element method for fluid dynamics: laminar flow in a channel expansion, J. Comput. Phys. 54 (1984) 468–488.

[14] C. Bernardi, Y. Maday, A. T. Patera, A new nonconforming approach to domain decomposition: The mortar element method, Vol. 299 of Pitman Res. Notes Math. Ser., Nonlinear partial differential equation and their applications, Collège de France Seminar, 11 (Paris, 1989–1991), Longman Sci. Tech., Harlow, 1994, pp. 13–51.

[15] V. Shtern, Core C++: A Software Engineering Approach, Prentice Hall PTR, Upper Saddle River, New Jersey, 2000.

[16] I. Joyner, C++??: A Critique of C++ and Programming and Language Trends in the 1990s, 3rd Edition, Available online at http://www.elj.com/cppcv3/, 1996.

[17] W. D. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, Cambridge, Massachusetts, 1999.

[18] B. R. Hodges, R. L. Street, On simulation of turbulent nonlinear free-surface flows, J. Comput. Phys. 151 (1999) 425–457.

[19] B. R. Hodges, Numerical simulation of nonlinear free-surface waves on a turbulent open-channel flow, Ph.D. thesis, Department of Civil Engineering, Stanford University (1997).

[20] Y. Zang, R. L. Street, J. R. Koseff, A non-staggered grid, fractional step method for time-dependent incompressible Navier–Stokes equations in curvilinear coordinates, J. Comput. Phys. 114 (1994) 18–33.

[21] J. Kim, P. Moin, Application of a fractional-step method to incompressible Navier–Stokes equations, J. Comput. Phys. 59 (1985) 308–323.

[22] S. E. Rodgers, D. Kwak, C. Kiris, Paper 89-0463, AIAA (1989).

[23] W. Couzy, Spectral element discretization of the unsteady Navier–Stokes equations and its iterative solution on parallel computers, Ph.D. thesis, no. 1380, Swiss Federal Institute of Technology, Lausanne (1995).

[24] G. E. Karniadakis, M. Israeli, S. A. Orszag, High-order splitting methods for the incompressible Navier–Stokes equations, J. Comput. Phys. 97 (1991) 414–443.

[25] W. Couzy, M. O. Deville, Spectral-element preconditioners for the Uzawa pressure operator applied to incompressible flows, J. Sci. Comput. 9 (1994) 107–112.

[26] J. B. Perot, An analysis of the fractional step method, J. Comput. Phys. 108 (1993) 51–58.

[27] J. B. Perot, Comments on the fractional step method, J. Comput. Phys. 121 (1995) 190–191.

[28] M. O. Deville, P. F. Fischer, E. H. Mund, High-order methods for incompressible fluid flow, Cambridge University Press, Cambridge, 2002.

[29] Y. Maday, A. T. Patera, E. M. Rønquist, The $\mathbb{P}_N \times \mathbb{P}_{N-2}$ method for the approximation of the Stokes problem, Tech. Rep. 92009, Department of Mechanical Engineering, MIT, Cambridge, MA (1992).

[30] R. Bouffanais, M. O. Deville, E. Leriche, Large-eddy simulation of the flow in a lid-driven cubical cavity, Phys. Fluids 19 (2007) Art. 055108.

[31] R. Gruber, P. Volgers, A. DeVita, M. Stengel, T.-M. Tran, Parametrisation to tailor commodity clusters to applications, Future Generation Computer Systems 19 (2003) 111–120.

[32] R. Gruber, T.-M. Tran, Scalability aspects of commodity clusters, EPFL Supercomputing Review 14 (2004) 12–17.

[33] Performance API, website, `http://icl.cs.utk.edu/papi/index.html` (2007).

[34] Veritable Application MOnitoring Service, website, `http://pleiades.epfl.ch/ vkeller/VAMOS` (2006).

[35] The Ganglia Monitoring Tool, website, `http://ganglia.sourceforge.net` (2007).

7 Annex: Profiling information

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self
 time   seconds   seconds    calls      name
 14.88    987.08   987.08 3461217386   RealVector::Multiply(RealVector*)
  9.86   1641.04   653.96   307891     FlatField::Multiply(Field*, int, int)
  8.99   2237.57   596.53 3603304192   Element::CopyAddValuesFrom(Element*, int, ...
  6.02   2636.75   399.18 916811776    ElementaryField::MultiplyByWeights(int)
  4.14   2911.52   274.77 177840128    Edge::CopyAddValuesFromFace(Face*, int, int, ...
  3.91   3170.74   259.22 1778679808   Quad::CopyAddValuesFromEdge(Edge*, int, int, ...
  3.72   3417.45   246.71 1820727360   Vertex::CopyAddValuesFromEdge(Edge*, int, int, ...
  3.09   3622.50   205.05   123303     FlatField::Add(Field*, double)
  2.63   3797.05   174.55 35926016     ElementaryField::GetWork()
  2.40   3956.46   159.41   108636     FlatField::SetValues(double)
  2.35   4112.24   155.78 1813419648   Edge::CopyAddValuesFromVertex(Vertex*, int, int, ...
  2.34   4267.58   155.34   111474     FlatField::CopyFrom(Field*)
  2.23   4415.67   148.09    51311     FlatField::SetToWeakDivergence(FlatField*, FlatField*)
  1.76   4532.48   116.81   153676     FlatField::Dot(Field*)
  1.76   4649.07   116.59 1833762816   ElementaryField::CopyInterpolateFrom(ElementaryField*, ...
  1.68   4760.74   111.67    51311     FlatField::SetToWeakGradient(FlatField*, FlatField*)
  1.66   4870.88   110.14 307981560    RealVector::MultiplyAndAdd(double, RealVector*, double)
  1.43   4965.67    94.79 420339712    RealVector::MultiplyAndSwitchSigns(RealVector*)
  1.33   5053.58    87.91 891158528    TensorMatrix::MatTransVec(RealVector*, RealVector*)
  1.29   5138.83    85.25 891789312    TensorMatrix::MatVec(RealVector*, RealVector*)
  1.28   5223.80    84.97    60070     FlatField::Multiply(Field*)
  1.28   5308.40    84.60   240852     FlatField::CopyAddValuesFrom(FlatField*, int, int, ...
  1.21   5388.57    80.17 2311449449   ElementaryField::SetValues(double, int)
  1.09   5461.13    72.56    60030     FlatField::MultiplyAndAdd(double, Field*, double)
  1.06   5531.42    70.29 3565895681   ListMatrices::Search(ParentElement*)
  0.97   5595.57    64.15 2283535745   ElementaryField::HasSameInterpolationAs(ElementaryField*)
  0.96   5659.26    63.69 471113728    ElementaryField::SetToGradient(ElementaryField*, int, ...
  0.90   5718.74    59.48 470818816    ElementaryField::SetToGradientT(ElementaryField*, int, ...
  0.90   5778.22    59.48    54453     FlatField::Multiply(double)
  0.83   5832.98    54.76 471146496    ElementaryField::SetToGradientOnParents(ElementaryField*,...
  0.76   5883.25    50.27 3834749408   Vector<Edge*>::GetIndexOf(Edge*)
  0.71   5930.67    47.42 35926016     ElementaryField::Retrieve(ElementaryField*)
  0.64   5973.25    42.58 729517856    ElementaryField::CopyFrom(ElementaryField*, int, int)
  0.60   6012.88    39.63 445841408    ElementaryField::CopyInterpolateTFrom(ElementaryField*,...
  0.60   6052.41    39.53 2546965353   RealVector::SetValuesZero()
  0.56   6089.23    36.82 356589567    Vector<ParentElement*>::GetIndexOf(ParentElement*)


%          the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
seconds    for by this function and those listed above it.

self       the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.
```