QMC=Chem: a quantum Monte Carlo program for large-scale simulations in chemistry at the petascale level and beyond

Anthony Scemama[1], Michel Caffarel[1], Emmanuel Oseret[2], and William Jalby[2]

[1] Lab. Chimie et Physique Quantiques, CNRS-Université de Toulouse, France.
[2] Exascale Computing Research Laboratory, GENCI-CEA-INTEL-UVSQ
Université de Versailles Saint-Quentin, France.

Abstract. In this work we discuss several key aspects for an efficient implementation and deployment of large-scale quantum Monte Carlo (QMC) simulations for chemical applications on petaflops infrastructures. Such aspects have been implemented in the QMC=Chem code developed at Toulouse (France). First, a simple, general, and fault-tolerant simulation environment adapted to QMC algorithms is presented. Second, we present a study of the parallel efficiency of the QMC=Chem code on the Curie machine (TGCC-GENCI, CEA France) showing that a very good scalability can be maintained up to 80 000 cores. Third, it is shown that a great enhancement in performance with the single-core optimization tools developed at Versailles (France) can be obtained.

1 Introduction

Quantum Monte Carlo (QMC) is a generic name for a large class of stochastic approaches solving the Schrödinger equation by using random walks. In the last forty years they have been extensively used in several fields of computational physics and are in most cases considered as state-of-the-art approaches. However, this is not yet the case in the important field of computational chemistry where the two "classical" computational methods are the Density Functional Theory (DFT) and post-Hartree-Fock methods. For a review of QMC and its status with respect to the standard approaches, see *e.g.* [1]. In the recent years several applications for realistic chemical problems have clearly demonstrated that QMC has a high potential in terms of accuracy and in ability of dealing with (very) large systems. However, and this is probably the major present bottleneck of QMC, simulations turn out to very CPU-expensive. The basic reason is that chemical applications are particularly demanding in terms of precision: the energy variations involved in a chemical process are typically several orders of magnitude smaller than the total energy of the system which is the quantity computed with QMC. Accordingly, the target relative errors are typically 10^{-7} or less and the Monte Carlo statistics needed to reach such a chemical accuracy can be tremendously large.

Now, the key point for the future is that this difficulty is expected to be largely overcome by taking advantage of the remarkable property of QMC methods (not valid for standard methods of chemistry) of being ideally suited to HPC and, particularly, to massive parallel computations. In view of the formidable development of computational

platforms this unique property could become in the near future a definite advantage for QMC over standard approaches.

The stochastic nature of the algorithms involved in QMC enables to take advantage of today and tomorrow's computer architectures through the following aspects: i) Data structures are small inducing a fairly small memory footprint (less than 300 MiB per core for very large systems) and data accesses are organized to maximize cache usage: spatial locality (stride-one access) and temporal locality (data reuse), ii) Most of the computation can be efficiently vectorized making full use of the vector capabilities of recent processors, iii) Network communications can be made non-blocking, iv) Access to persistent storage is negligible and can be made non-blocking, v) Different parallel tasks can be made independent of each other so as to run asynchronously, vi) Fault-tolerance can be naturally implemented. All these features which have been implemented in the QMC=Chem code developed at Toulouse[2] are required to take advantage of large-scale computing grids[3] and to achieve a very good parallel efficiency on petascale machines.

In section 2 a short overview of the mathematical foundations of the QMC method employed here is presented. For a more detailed presentation the reader is referred to [1] and references therein. Section 3 is devoted to the presentation of the general structure of the simulation environment employed for running QMC=Chem on an arbitrary computational platform. A preliminary version of such an environment has been presented in [3]. Here, we present an improved implementation where the network communications are now fully handled by a client-server implementation and the computational part is isolated in multiple instances of a single-core Fortran program. These modifications allow the program to survive failures of some compute nodes. In section 4 the results of our study of the parallel efficiency of the QMC=Chem code performed on the Curie machine (TGCC-GENCI, France) thanks to a PRACE preparatory access[4] are presented. In section 5 the results of the optimization of the single-core performance are discussed. The optimization was performed after a static assembly analysis of the program and a decremental analysis.

2 Overview of a QMC Simulation

In the simulations discussed here, the basic idea is to define in the $3N$-dimensional electronic configuration space a suitable Monte Carlo Markov chain combined with a birth-death (branching) process to sample a target probability density from which exact (or high-quality) quantum averages can be evaluated. In our simulations we employ a variation of the Fixed-Node Diffusion Monte Carlo (FN-DMC) method, one of the most popular versions of QMC. In short, we aim at solving the electronic Schrödinger equation written as

$$\mathscr{H}\Psi_0(\mathbf{r}_1,\ldots,\mathbf{r}_N) = E_0\Psi_0(\mathbf{r}_1,\ldots,\mathbf{r}_N) \tag{1}$$

where $\mathscr{H}$ is the molecular Hamiltonian operator, $\Psi_0(\mathbf{r}_1,\ldots,\mathbf{r}_N)$ the N-electron wave function, and E_0 the total electronic energy.

To do that, the basic idea is to construct a stochastic process having the density

$$\pi(\mathbf{r}_1,\ldots,\mathbf{r}_N) = \frac{\Psi_0(\mathbf{r}_1,\ldots,\mathbf{r}_N)\Psi_T(\mathbf{r}_1,\ldots,\mathbf{r}_N)}{\int\ldots\int \mathrm{d}\mathbf{r}_1\ldots\mathrm{d}\mathbf{r}_N\Psi_0(\mathbf{r}_1,\ldots,\mathbf{r}_N)\Psi_T(\mathbf{r}_1,\ldots,\mathbf{r}_N)} \tag{2}$$

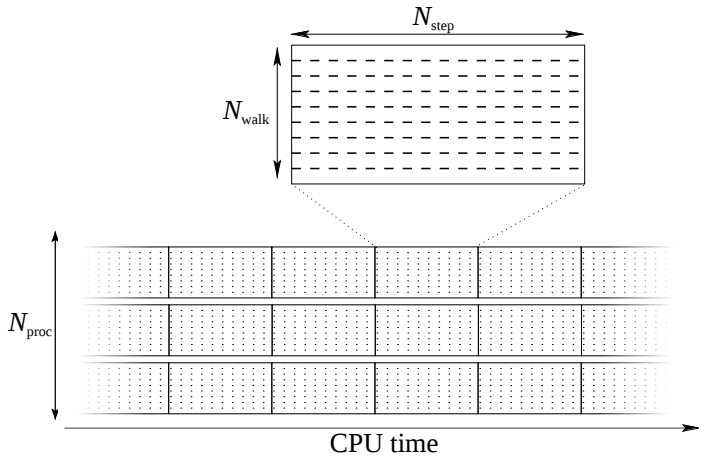

Fig. 1. Graphical representation of a QMC simulation. Each process generates blocks, each block being composed of N_{walk} walkers realizing N_{step} Monte Carlo steps.

as stationary density. Here, Ψ_T — called the trial wavefunction — is some good known (computable) approximation of Ψ_0. The role played by the trial wavefunction is central since it is used to implement the "importance sampling" idea, a fundamental point at the heart of any *efficient* Monte Carlo sampling of a high-dimensional space. In the important case of the total energy, it can be shown that the exact ground-state energy E_0 may be expressed as the average of the so-called local energy defined as $E_L(\mathbf{r}_1,\ldots,\mathbf{r}_N) \equiv \frac{\mathscr{H}\Psi_T}{\Psi_T}$ over the density π.

In brief, the stochastic rules employed are as follows:

1. Use of a standard Markov chain Monte carlo chain based on a drifted Brownian motion. The role of the drift term is to push the configurations (or "walkers" in the QMC terminology) towards the regions where the trial wavefunction takes its largest values (importance sampling technique).
2. Use of a birth-death (branching) process: the walkers are killed or duplicated a certain number of times according to the magnitude of the local energy (low values of the local energy are privileged).

It can be shown that by iterating these two rules for a population of walkers the stationary density π (Eq. 2) is obtained. Note that in the actual implementation in QMC=Chem, a variation of the FN-DMC method working with a *fixed* number of walkers is used[5].

Denoting as $\mathbf{X} = (\mathbf{r}_1,\ldots,\mathbf{r}_N)$ a walker in the $3N$-dimensional space, the random trajectories of the walkers differ from each other only in the initial electron positions $\mathbf{X}_0$, and in the initial random seed S_0 determining the entire series of pseudo-random numbers.

The main computational object is a *block*. In a block, N_{walk} independent walkers realize random walks of length N_{step}, and the energy is averaged over all the steps of each random walk. N_{step} is set by the user, but has to be taken large enough such that

the positions of the walkers at the end of the block can be considered independent from their initial positions. A new block can be sampled using the final walker positions as $\mathbf{X}_0$ and using the current random seed as S_0. The block-averaged energies are Gaussian distributed and the statistical properties can be easily computed. The final Monte Carlo result is obtained by super-averaging all the block-averages. If the block-averages are saved to disk, the final average can be calculated by post-processing the data and the calculation can be easily restarted at any time. As all blocks are completely independent, different blocks can be computed asynchronously on different CPU cores, different compute nodes and even in different data centers. Figure 1 shows a pictorial representation of three independent CPU cores computing blocks sequentially, each block having different initial conditions.

The core of QMC=Chem is a single-core Fortran executable that computes blocks as long as a termination event has not been received. At the end of each block the results are sent in a non-blocking way to a central server, as described in the next section.

3 The Client/Server Layer

In the usual MPI implementations, the whole run is killed when one parallel task will not be able to reach the *MPI_Finalize* statement. This situation occurs when a parallel task is killed, often due to a system failure. For deterministic calculations where the result of every parallel task is required, this mechanism is convenient since it immediately stops a calculation that will not give the correct result. In our case, as the result of the calculation of a block is a Gaussian random variable, removing the result of a block from the simulation is not a problem since doing that does not introduce any bias in the final result. Therefore, if one compute node fails, the rest of the simulation should survive.

A second disadvantage of using MPI libraries is that all resources need to be available for a simulation to start. In our implementation, as the blocks can be computed asynchronously we prefer to be able to use a variable number of cores during the simulation in order to reduce the waiting time in the batch queue.

These two main drawbacks lead us to write a lightweight TCP client/server layer in the Python language to handle all the network communications of the program and the storage of the results in a database. The Python program is divided into three distinct tasks shown in figure 2, the first and second tasks running only on the master compute node.

The first task is a *manager* that watches periodically the database associated with the running simulation. The manager computes the running averages and error bars, and checks if the stopping condition of the calculation is reached. The stopping condition can be for instance a threshold on the error bar of the energy, a condition on the total execution time, *etc*. If the stopping condition is reached, a stopping flag is set in the database.

The second task is a *data server*. This task contains two threads: one network thread and one I/O thread. The network thread periodically sends a UDP packet to the connected clients to update their stopping condition and to check that they are still running. The network thread also receives the block averages and puts them in a queue. Simulta-

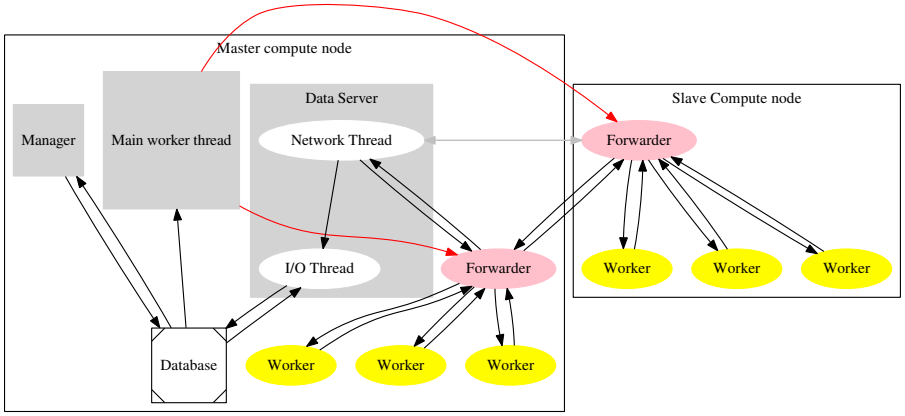

Fig. 2. The communication architecture of QMC=Chem.

neously, the I/O thread empties this queue by storing in the database the block averages. At any time, a new client can connect to the data server to request the input files and participate to a running calculation. Another way to increase the number of running cores is to submit another run reading and writing to the same database. As the managers read periodically the content of the database, each simulation is aware of the results obtained by the other simulations. This allows the use of multiple managers and data servers that can be submitted to the batch scheduler as independent jobs in order to gather more and more computing resources as they become available.

The third task is a *forwarder*. Each compute node (as well as the master node) has only one instance of the forwarder. The forwarder has different goals. The first goal is to spawn the computing processes (single-core Fortran executables) and to collect the results via Unix pipes after a block has been computed. Then, it sends the results to the data server while the computing processes are already computing the next block. If every compute node sends directly the results to the data server, the master node is flooded by small packets coming from numerous sources. Instead the forwarders are organized in a binary tree structure, and the second goal of a forwarder is to collect results from other forwarders and transfer them in a larger packet to its parent in the binary tree. Using this structure, the data server has much fewer connected clients, and receives much larger packets of data. All the nodes of the forwarder tree can possibly connect to all their ancestors if the parent forwarder does not respond.

On massively parallel machines an MPI launcher is used to facilitate the initialization step. The launcher sends, via the MPI library, the input files and Python scripts to all the slave nodes allocated by the batch scheduler. The files are written in a RAM disk on every node (the */dev/shm* location). The reason for this copy is to avoid too many simultaneous I/O on the shared file system, and also to avoid I/O errors if the shared file system fails, if a local disk fails or is full. The MPI launcher then forks to an instance of a forwarder that connects to the data server.

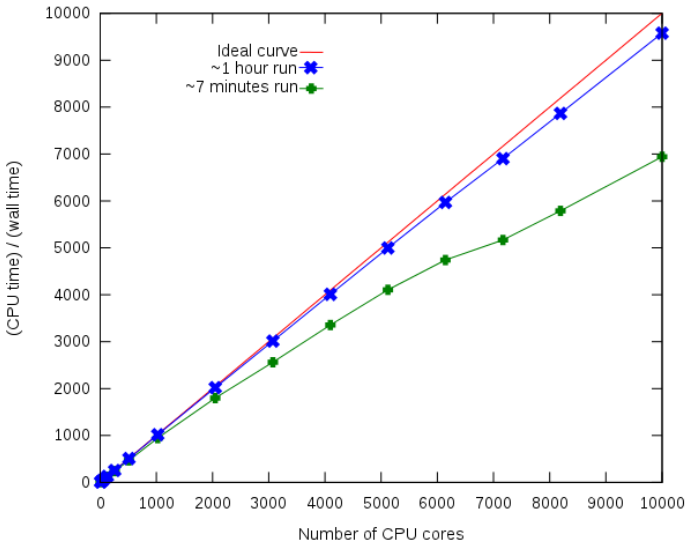

Fig. 3. Number of computed blocks as a function of the number of CPU cores for a fixed computation time.

4 Parallel Efficiency

Using the design described in the previous section, the parallel section of the program is expected to display a parallel efficiency of about 100% since there is no blocking statement. In this section, we investigate in some detail how the initialization and finalization steps impact the parallel efficiency of QMC=Chem.

For that, a small benchmark was set up during a PRACE preparatory access on the Curie machine (TGCC-GENCI, France).[4] On this machine, all the compute nodes were equipped with the same processors, namely four Intel Nehalem 8-core sockets. Two important parameters are to be kept in mind. First, we chose to run a short simulation for which the average CPU time required to compute one block is 82 seconds.[3] Second, we have chosen to send the stopping signal after 300 seconds (during the fourth block). When the forwarders receive the stopping signal, they wait until all the working CPU cores finish their current block. Hence, in this study the ideal wall time for perfect scalability should be 328 seconds.

The additional time $T(N_{\text{core}})$ with respect to the ideal time can be expressed as

$$T(N_{\text{core}}) = W(N_{\text{core}}) - \frac{C(N_{\text{core}})}{N_{\text{core}}} \tag{3}$$

where N_{core} is the number of cores, $W(N_{\text{core}})$ is the wall time and $C(N_{\text{core}})$ is the CPU time, both measured for N_{core} cores. With 10 000 cores, 149 seconds are needed for the

[3] This is not representative of a real simulation since it is far too small: the time spent in network communications will be over-estimated compared to real simulations where the time to compute one block is typically much greater (10 minutes and more).

initialization and finalization steps. For this 7 minutes benchmark, a parallel efficiency of 69% was obtained. However, as the parallel section has an ideal scaling, one can extrapolate the parallel efficiency one would obtain for a one hour run. If the stopping signal occurs after one hour, each core would have computed 44 blocks. The total CPU time would be $\tilde{C}(N_{\text{core}}) = 11C(N_{\text{core}})$. As the additional time $T(N_{\text{core}})$ does not depend on the number of computed blocks, the wall time would be $\tilde{W}(N_{\text{core}}) = T(N_{\text{core}}) + \tilde{C}(N_{\text{core}})/N_{\text{core}}$. A parallel efficiency of 96% would be obtained for a one-hour run on 10 000 CPU cores (figure 3).

More recently, we were given the opportunity to test QMC=Chem on the thin nodes of the Curie machine (80 000 Sandy Bridge cores), for a real application on a biological molecule made of 122 atoms and 434 electrons (the largest application ever realized using all-electron Diffusion Monte Carlo). Using 51 200 cores for 3 hours, the parallel efficiency was 79%. After the runs were finished, we realized that for such a large molecular system, the CPU time needed to compute one block had quite large fluctuations due to the implementation the dense-sparse matrix product presented in the next section. This implementation considerably reduces the total wall time (which is what the end user wants), but slightly reduce the parallel efficiency. This problem has been solved by making the number of Monte Carlo steps per block non-constant. Nevertheless, these runs confirm that a good scaling can still be obtained for a real simulation.

5 Single-Core Efficiency

Our choice in the implementation of the QMC algorithms was to minimize the memory footprint. This choice is justified first by the fact that today the amount of memory per CPU core tends to decrease and second by the fact that small memory footprints allows in general a more efficient usage of caches. Today, the standard size of the molecular systems studied by QMC methods and published in the literature usually comprise less than 150 electrons. For a 158 electron simulation, the binary memory footprint (including code and data) per core is only 9.8 MiB. To check the memory footprint of much larger systems, a few Monte Carlo steps were performed successfully on a molecular system containing 1731 electrons; such a large system only required 313 MiB of memory per core. For a system beyond the largest systems ever computed with all-electron QMC methods, the key limiting factor is only the available CPU time and neither the memory nor disk space requirements. This feature is well aligned with the current trends in computer architecture for large HPC systems.

As the parallel scaling is very good, single-core optimization is of primary importance: the gain in execution time obtained on the single-core executable will also be effective in the total parallel simulation. The Fortran binary was profiled using standard profiling tools (namely gprof[6] and Scalasca[7]). Both tools exhibit two major hot spots in the calculation of a Monte Carlo step. The first hot spot is a matrix inversion, and the second hot spot is the product of a constant matrix A with five different matrices $B_1 \dots B_5$. These two bottlenecks have been carefully optimized for the x86 micro-architectures, especially the for the AVX instruction set of the Sandy Bridge processors.

To measure the performance of the matrix inversion and the matrix products, small codelets were written. The final results are given in table 1, compared to the performance of the single core executable, with different molecular system sizes. Computational complexity (with respect to FP operations) of the matrix inversion is $\mathcal{O}(N_e^3)$ where N_e is the number of electrons. Exploiting the sparse structure of the right matrices in the matrix matrix products, computational complexity of such products is reduced to $\mathcal{O}(N_e^2)$ with a prefactor depending on N_{basis}, the size of the basis set used to describe the wave function.

The matrix inversion is performed in double precision (DP) using the Intel MKL library, an implementation of the LAPACK[9] and BLAS[10] APIs. To maximize MKL efficiency, arrays were padded to optimize array alignment (lined up on 256 bit boundaries) and the leading dimension of the array is chosen to be a multiple of 256 bits to ensure that all the column accesses are in turn properly aligned.

For the matrix matrix products, similar alignment/padding techniques were used. Loops were rearranged in order to use full vector length stride-one access on the left dense matrix and then blocked to optimize temporal locality (i.e. cache usage).

An x86_64 version of the MAQAO framework[11] was used to analyze the binary code and to generate best possible static performance estimates. This technique was used not only on the matrix matrix products but also on all of the hottest loops (i.e. accounting at least for more than 1% of the total execution time). This allowed us to detect a few compiler inefficiences and to fix them by hard coding loop bounds and adding up pragmas (essentially for allowing use of vector aligned instructions).

Then, the DECAN tool[12] was used to analyze performance impact of data access. For that purpose, for each loop, two modified binaries were automatically generated: i) FPISTREAM: all of AVX load instructions are replaced by PXOR instructions (to avoid introduction of extra dependencies) and all of the AVX store instructions were replaced by NOP instructions (issuing no operation but preserving the binary size). FPISTREAM corresponds to the ideal case where all of the data access are suppressed. ii) MISTREAM: all of the AVX arithmetic instructions were replaced by NOP instructions. By comparing cycle counts of FPISTREAM, MISTREAM binaries with the original binary, potential performance problems due to data access (essentially cache misses)

Table 1. Single core performance (GFlops/s) of the two hot routines: inversion (DP), matrix products (SP), and of the entire single-core executable. Measurements were performed on an Intel Xeon E31240, 3.30GHz, with a theoretical peak performance 52.8 GFlops/s (SP) and 26.4 GFlops/s (DP). The values in parenthesis are the percentages with respect to the peak. The turbo feature was turned off, and the results were obtained using Likwid performance tools.[8]. [1]As the matrix to invert is block-diagonal with two $N_e/2 \times N_e/2$ blocks, the inversion runs on the two sub-matrices.

System sizes	Matrix inversion[1]	Matrix products	Overall performance
$N_e = 158$, $N_{\text{basis}} = 404$	6.3 (24%)	26.6 (50%)	8.8 (23%)
$N_e = 434$, $N_{\text{basis}} = 963$	14.0 (53%)	33.1 (63%)	11.8 (33%)
$N_e = 434$, $N_{\text{basis}} = 2934$	14.0 (53%)	33.6 (64%)	13.7 (38%)
$N_e = 1056$, $N_{\text{basis}} = 2370$	17.9 (67%)	30.6 (58%)	15.2 (49%)
$N_e = 1731$, $N_{\text{basis}} = 3892$	17.8 (67%)	28.2 (53%)	16.2 (55%)

could be easily detected. Such an analysis revealed that for most of the hot loops, data access was accounting for less than 30% of the original time, indicating an excellent usage of the caches. Measurement of the binaries were performed directly with the whole application running allowing to take into account runtime context for the loops measured.

6 General Conclusion

In December 2011, GENCI gave us the opportunity to test our program on Curie (TGCC-GENCI, France) while the engineers were still installing the machine. At that time, up to 4 800 nodes (76 800 cores) were available to us for two sessions of 12 hours. As the engineers were still running a few benchmarks, our runs were divided into 3-hour jobs using 400 nodes (6 400 cores). In this way the engineers were able to acquire resources while our job was running. This aspect points out the importance of our flexible parallel model, but makes it impossible to evaluate rigorously the parallel efficiency. At some point, all the available nodes were running for our calculation during several hours. As we had previously measured a sustained performance of 200 GFlops/s per node for this run, we can safely extrapolate to a sustained value of $\sim$960 TFlops/s (mixed single/double-precision) corresponding to about 38 % of the peak performance of the whole machine for a few hours. As the machine was still in the test phase, we experienced a few hardware problems and maintenance shutdowns of some nodes during the runs. Quite interestingly, it turns out to be an opportunity for us to test the robustness of our program: it gave us the confirmation that our fault-tolerant scheme is indeed fully functional.

In this work we have presented a number of important improvements implemented in the QMC=Chem program that beautifully illustrate the extremely favorable computational aspects of the QMC algorithms. In view of the rapid evolution of computational infrastructures towards more and more numerous and efficient processors it is clear that such aspects could be essential in giving a definite advantage to QMC with respect to other approaches based on deterministic linear algebra-type algorithms.

Acknowledgments. AS and MC would like to thank ANR for support through Grant No ANR 2011 BS08 004 01. This work was possible thanks to the generous computational support from CALMIP (Toulouse) under the allocation 2011-0510, GENCI, CCRT (CEA), and PRACE through a preparatory access (No PA0356).

References

1. M. Caffarel. Quantum monte carlo methods in chemistry. In Björn Engquist, editor, *Encyclopedia of Applied and Computational Mathematics*. Springer, 2011. http://qmcchem.ups-tlse.fr/files/caffarel/qmc.pdf.
2. See web site: "Quantum Monte Carlo for Chemistry@Toulouse", http://qmcchem.ups-tlse.fr.
3. A. Monari, A. Scemama, and M. Caffarel. Large-scale quantum monte carlo electronic structure calculations on the egee grid. In Franco Davoli, Marcin Lawenda, Norbert Meyer, Roberto Pugliese, Jan Wglarz, and Sandro Zappatore, editors, *Remote Instrumentation for eScience and Related Aspects*, pages 195–207. Springer New York, 2012. http://dx.doi.org/10.1007/978-1-4614-0508-5_13.

4. M. Caffarel and A. Scemama. Large-scale quantum monte carlo simulations for chemistry. Technical Report PA0356, PRACE Preparatory Access Call, April 2011. http://qmcchem.ups-tlse.fr/files/scemama/Curie.pdf.
5. R. Assaraf, M. Caffarel, and A. Khelif. Diffusion monte carlo methods with a fixed number of walkers. *Phys. Rev. E*, 61, 2000.
6. The GNU profiler, http://sourceware.org/binutils/docs-2.18/gprof/index.html.
7. F. Wolf. Scalasca. In *Encyclopedia of Parallel Computing*, pages 1775–1785. Springer, October 2011.
8. J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *39th International Conference on Parallel Processing Workshops (ICPPW)*, pages 207 –216, sept. 2010.
9. E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: a portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.
10. J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14:1–17, March 1988.
11. L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W. Jalby. MAQAO: Modular assembler quality Analyzer and Optimizer for Itanium 2. In *Workshop on EPIC Architectures and Compiler Technology, San Jose, California, United-States*, March 2005.
12. S. Koliai, S. Zuckerman, E. Oseret, M. Ivascot, T. Moseley, D. Quang, and W. Jalby. A balanced approach to application performance tuning. In *LCPC*, pages 111–125, 2009.