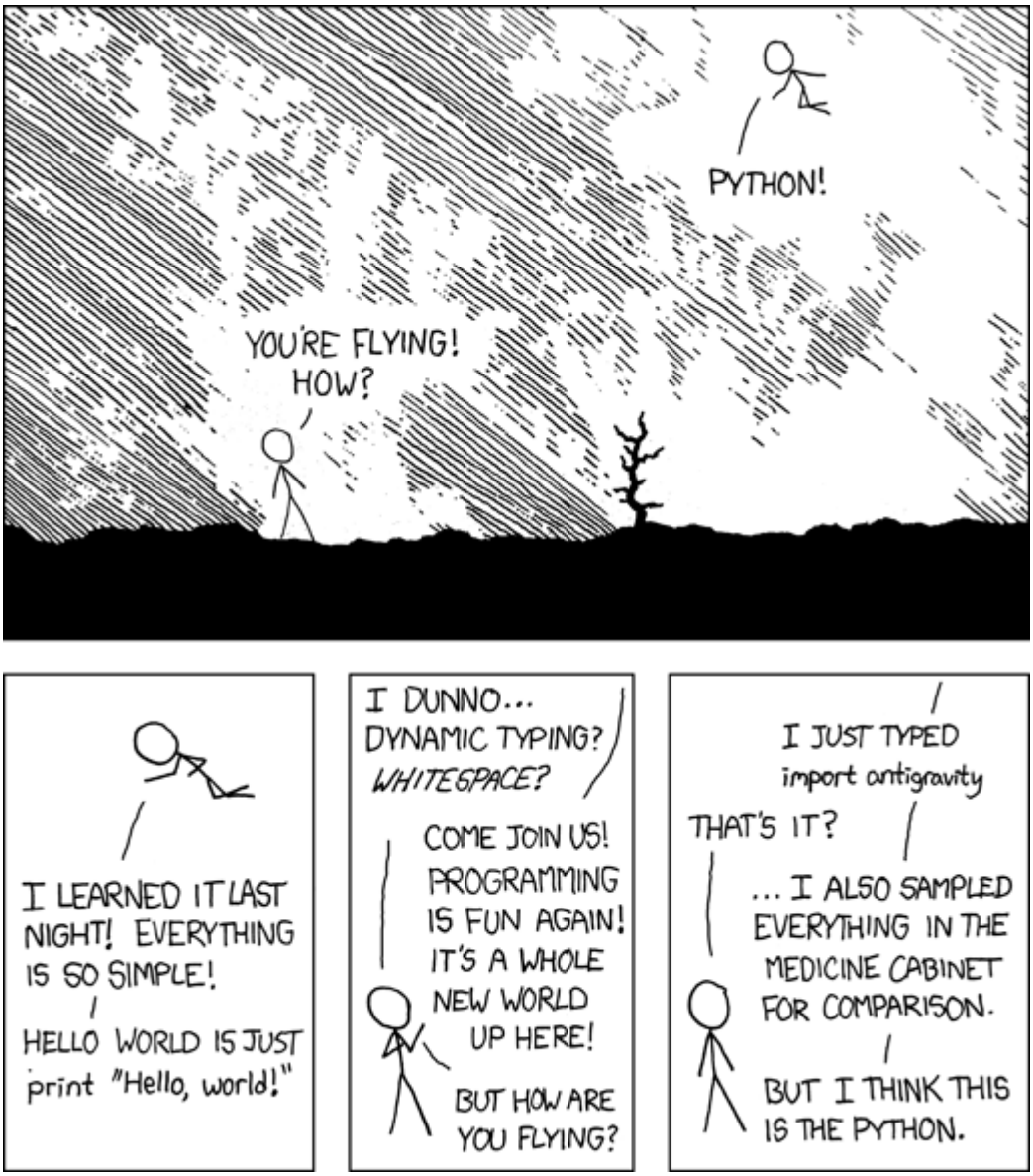
Quantum Monte Carlo and Data Management in Grid Middleware

Jon Kerr Nilsen
Department of Physics
University of Oslo
Norway

Dissertation presented for the degree of
Philosophiae Doctor (PhD) in Physics

March 2010

Contents



(From `http://xkcd.com/353/`.)

From `http://en.wiktionary.org/wiki/Pythonesque`: Pythonesque (comparative more Pythonesque, superlative most Pythonesque)

1. (*of humour*) Surreal or absurd.
2. (*computing, informal*) Typical of, or suited to, the Python programming language.

Chapter 1

Introduction

The task of this thesis is twofold. First, the thesis considers a genre of physics problems and efficient algorithms for solving them. Second, the thesis considers distributed computing resources and how to connect and utilize them efficiently. While the two tasks at first thought may seem unrelated, each task has been a motivating factor for the other in science in the last half century.

Nuclear and particle physics strives to understand how the world is built by exploring its smallest building blocks and how they are connected. This question spawns a large number of computational problems with no upper limits to the amount of computing resources needed. In many cases the advances in the physics field are limited by the amount of computing resources, thus motivating the work of connecting computing resources to gain more computing power.

On the other hand, advances in distributed computing technology are driven by the demand for computing power. While the physics community is of course not the only scientific nor non-scientific community with growing needs for computing resources, increasingly large physics experiments have generated demands that would not even have been considered possible for smaller experiments. As an example, the European Organization for Nuclear Research (CERN) created the world wide web as a means to simplify communication and collaboration between scientists and is now in the forefront of developing distributed computing as a consequence of the computing needs of the largest physics experiment ever built, the Large Hadron Collider.

In the remainder of this introductory chapter we will give a short introduction to the two main topics of this thesis aimed at the non-expert, before presenting the outline of the thesis.

1.1 Quantum Monte Carlo

Already in the nineteenth century a sharp distinction began to appear between two mathematical methods for treating physical phenomena [11]. Problems involving only a few particles were studied in classical mechanics, through the study of systems of ordinary differential equations. On the other hand, to describe systems with very many particles one used the entirely different methodology of statistical mechanics. Here, probability theory is applied to describe properties of a set of particles by relating the microscopic properties of individual atoms to the macroscopic properties of the observed material.

However, in an intermediate situation where the number of particles is moderate, both of these methods come short. Classical mechanics cannot describe interactions between three or more interacting particles and a statistical-mechanical approach would also be unrealistic. For example, consider a high-energy electron entering an electromagnetic calorimeter. Due to bremsstrahlung the electron emits a photon. The photon interacts with the matter through pair production, converting into an electron-positron pair. The electron and positron, still having relatively high energy, emit photons, which again convert to electron-positron pairs in a so-called electromagnetic shower. The probability of producing a given particle with a given energy in a given event depends on the energy of the incoming particle. Additionally, there is a probability distribution for the direction of the motion. This process is an illustration of a Markov chain, i.e., a chain of events where the next event depends only on the current event. The tool for studying such chains is matrix theory. To obtain a mathematical analysis one would have to multiply a large number of $(n \times n)$ transition matrices containing the state of the system.

Now, a full analysis of all the possible transitions in an electromagnetic shower is of course not feasible. However, one can get a quite reasonable description of the shower by performing a finite number of "gedankenexperiments" to obtain a set of possible outcomes. These experiments will not be performed with any physical instruments, but merely as theoretical experiments on a computer. Assuming that the probability of each possible event is known, we can run a large number of experiments to study the shower empirically. This method is referred to as the Monte Carlo method, or, when applied to quantum mechanical phenomena, the quantum Monte Carlo method.

1.2 Grid

The use of the term *grid* as a technology for connecting computing resources is based on how the power grid works. The power grid is a rather complicated structure. Power plants generate electricity by, e.g., burning fossil fuel, nuclear reactions or converting wind, water or sun power to electric power. The power is transported by cables over great distances and across borders, and it is bought and sold on international markets with complicated pricing schemes. However, for an end user, all that is needed to utilize this power is to plug whatever device into a socket and pay a bill to a single supplier.[1]. The idea of the computing grid is then that computers, laptops and other devices should only need to plug into a socket to get access to worldwide computing and storage resources.

While the comparison with the power grid gives a broad idea of what a computing grid is (or rather, should be) a perhaps more instructive comparison is with a single desktop. A typical desktop contains a central processing unit (CPU), random access memory (RAM), a graphical processing unit (GPU), a hard-drive, a DVD player (or some other optical device) and a motherboard where these units are connected. Additionally the motherboard has different ports for attaching external devices such as keyboard, mouse, external hard-drives, memory sticks, and so forth. These units and devices need to be combined to do the work a desktop is expected to do. For example, when writing a Word document, the CPU calculates how many letters can fit into one line, checks if the words entered exist in a dictionary (and sometimes if they are grammatically correct), etc., the GPU converts the text to a graphical image and sends it to the screen, the text is written to RAM for fast access and to the hard-drive for backup. To do all these tasks, Word relies on the operating system, which acts as a middleware between the software (Word) and the hardware (the desktop).

In this comparison the grid resembles a desktop. Its hardware is a bit more complicated as it may contain hundreds of thousands of CPU's and hard-drives, all interconnected through network cables rather than through a single motherboard, and not all the CPU's can necessarily carry out the same tasks. Additionally, moving data over the network raises several security issues as, in theory, anyone can pick up the data stream between the devices. However, the main ideas are the same: A user who needs to cary out a task, use some software to run the task, the software send this task to the grid middleware, the grid middleware discovers the hardware suitable for the

[1] Unless, of course, the socket is at a workplace or in a public building where someone else pays the bill.

task and send the task there. When the task is finished the middleware makes the result available to the software which brings the result to the user. An example of such grid middleware is the Advanced Resource Connector (ARC), the platform for one of the main results of this thesis.

1.3 Outline

The main results of the thesis are given in the seven publications in Appendix A; "Vortices in atomic Bose-Einstein condensates in the large-gas-parameter region" (Appendix A.1), "MontePython: Implementing Quantum Monte Carlo using Python" (Appendix A.2), "Simplifying Parallelization of Scientific Codes by a Function-Centric Approach in Python" (Appendix A.3), "ARC middleware: evolution towards standards-based interoperability" (Appendix A.4), "Chelonia - A Self-healing Storage Cloud" (Appendix A.5), "Chelonia - distributed cloud storage" (Appendix A.6) and "Parallel Monte Carlo simulations on an ARC-enabled computing grid" (Appendix A.7).

As already mentioned the task of this thesis is twofold. This is also reflected in the outline of the thesis. While Chapter 2 gives an overview of the physical problem which sets the stage and/or is the motivational factor for all my contributions to the publications in this thesis, Part I presents the Monte Carlo theory needed for the publications in Appendices A.1, A.2 and A.3. Part II gives the needed background and main results in distributed computing (Chapter 4) and distributed data management (Chapter 5) related to the publications in Appendices A.4, A.5 and A.6 before Parts I and II are brought together in Part III, presenting GaMPI, a framework for running Monte Carlo simulations using distributed computing technologies, and giving the main result from the publication in Appendix A.7. In Chapter 7 we give some concluding remarks and look at some possible future directions for GaMPI and Chelonia.

Chapter 2

Physics Motivation

The dramatic achievement of trapping alkali-metal gases at ultra-low temperatures to observe Bose-Einstein condensation (BEC) [1, 2, 3] stimulated an intense experimental and theoretical activity on confined Bose systems. Of interest is the fraction of condensed atoms, the nature of the condensate, the excitations above the condensate, the atomic density in the trap as a function of temperature and the critical temperature of BEC, T_c. The extensive progress made up to early 1999 is reviewed by Dalfovo et al. [4].

A key feature of the trapped alkali and atomic hydrogen systems is that they are dilute. The characteristic dimensions of a typical trap for ^{87}Rb is $a_{h0} = (\hbar/m\omega_\perp)^{\frac{1}{2}} = 1 - 2 \times 10^4$ Å. The interaction between ^{87}Rb atoms can be well represented by its s-wave scattering length, a_{Rb}. This scattering length lies in the range $85 < a_{Rb} < 140a_0$ where $a_0 = 0.5292$ Å is the Bohr radius. The definite value $a_{Rb} = 100a_0$ is usually selected and for calculations the ratio of atom size to trap size $a_{Rb}/a_{h0} = 4.33 \times 10^{-3}$ is usually chosen [4]. A typical ^{87}Rb atom density in the trap is $n \simeq 10^{12} - 10^{14}$ atoms/cm^3 giving an inter-atom spacing $\ell \simeq 10^4$ Å. Thus the effective atom size is small compared to both the trap size and the inter-atom spacing, the condition for diluteness (i.e., $na_{Rb}^3 \simeq 10^{-6}$ where $n = N/V$ is the number density). In this limit, although the interaction is important, dilute gas approximations such as the Bogoliubov theory[5], valid for small na^3 and large condensate fraction $n_0 = N_0/N$, describe the system well. Also, since most of the atoms are in the condensate (except near T_c), the Gross-Pitaevskii equation[6, 7] for the condensate describes the whole gas well. Effects of atoms excited above the condensate have been incorporated within the Popov approximation[8].

Going beyond the dilute limit, the Gross-Pitaevskii approximation will loose it's validity, and other methods like quantum Monte Carlo simulations are needed to properly study Bose-Einstein condensation, taking into account inter-particle interactions. While the Gross-Pitaevskii equation is a

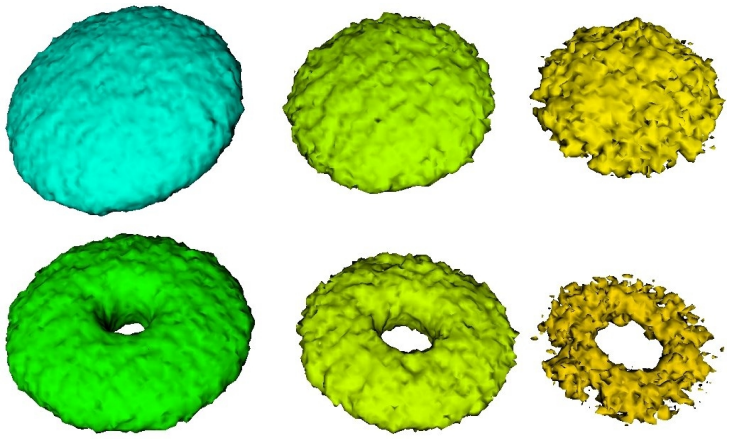


Figure 2.1: The figures show where particles are detected. We see the expectation values for finding, from left to right, 1, 2 and 3 particles. The topmost figure corresponds to the ground state, while the bottom figure corresponds to a state with one vortex in the center of the trap. (Taken from "MontePython: Implementing Quantum Monte Carlo using Python", Appendix A.2.)

mean-field approximation, quantum Monte Carlo can take into account the movements of each individual particle in the gas. While this increases the precision in the computations of actual physical systems, it also considerably increases the need for computational resources.

A key feature of Bose-Einstein condensates is the way they react under rotation. The second derivative of the free energy with respect to the rotational frequency is proportional to the superfluid density and vanishes for a normal fluid [9]. This provides a way to distinguish between superfluids and normal fluids. In contrast to a normal fluid, which rotates like a rigid body at thermal equilibrium, the thermodynamically stable state of a superfluid does not rotate at low frequency. At higher frequencies the angular momentum appears as vortex fillaments where the superfluid density vanishes. The ^{87}Rb condensate provide a system for which this superfluid behaviour can be studied in the weak-coupling regime. Figure 2.1 shows a quantum Monte Carlo simulation of the change in the ground state when inserting one vortex on the z-axis. The repulsive nature of the vortex pushes the particles away from the z-axis, decreasing the maximum density when compared to the ground state.

While being a fascinating study in itself, the Bose-Einstein condensate (BEC) also serves as an interesting computational example. Due to the symmetric properties of the underlying wave functions, there exists numerically

exact Monte Carlo algorithms for simulating bosons, making it possible to compare results from experiments and simulations in a direct manner. The articles included in Appendices A.1, A.2 and A.3 all demonstrate different aspects of computational physics using BEC as an example: In Appendix A.1, we investigate the applicability of the Gross-Pitaevskii equation to dilute BEC by comparing with results from a variational Monte Carlo simulation, in Appendix A.2 we use BEC to demonstrate use of the programming language Python [10] mixed with compiled code to create an efficient computational program and in Appendix A.3 we use BEC as one of three examples for demonstrating a framework for automatic parallelization of serial code in Python.

In Part I we will introduce the Monte Carlo methods used in the publications, while in Part we describe the computational infrastructure which enables running Monte Carlo simulations at a very large scale.

Part I

Quantum Monte Carlo

Chapter 3

Monte Carlo

While the main principles for Monte Carlo are the same as in 1949, many variants of Monte Carlo have emerged as access to computer resources has improved. The usefulness and generality of the Monte Carlo method makes it applicable even beyond physics and chemistry, with applications in economy [12] and political science [13] (see also Appendix A.3). While the different variants all have in common that random numbers are involved, they can broadly be divided in two areas with separate goals:

- Direct simulation of stochastic processes (e.g., electromagnetic showers, parton scattering, roll-call voting).
- Calculation of (many-dimensional) integrals (e.g., many-body problems).

We will in the remainder of this chapter describe different aspects of the Monte Carlo method, from simulation of processes which are stochastic in nature (e.g., scattering processes), to Monte Carlo integration methods in general and the method of diffusion Monte Carlo in particular, before discussing MontePython, an implementation of diffusion Monte Carlo.

3.1 Random numbers

The main requirement for all Monte Carlo algorithms is a way of generating uncorrelated, unique random numbers. Since computers are deterministic and handle only a finite set of numbers, truly random numbers do not exist on computers. Instead, pseudo-random number generators (PRNG's) are used. A PRNG generates a deterministic set of numbers that appears non-deterministic to anyone not knowing the underlying algorithm.

A good random generator should satisfy the following criteria [14]:

- Good distribution: The numbers should be distributed according to the probability distribution function the PRNG claims to produce, without detectable correlations.

- Long period: The sequence of numbers generated by a PRNG will always have a finite length, known as the period of the PRNG. To avoid correlations, a calculation should never come close to exhausting this period.

- Repeatability: For testing and development purposes, it may be necessary to repeat the sequence generated in the previous calculation. Furthermore, a PRNG should allow for repeating a part of a calculation. This requires the ability to store the state of the PRNG.

- Long disjoint sub-sequences: To be able to perform independent sub-simulations and later combine them, the results need to be statistically independent. This can be achieved if the PRNG can offer disjoint sub-sequences.

- Portability: The PRNG should generate the exact same sequence regardless of hardware and operating system.

- Efficiency: The generation of the random numbers should not be too time-consuming.

Note that these criteria should be weighted against the expected use of the PRNG. If only a relatively short sequence is needed, efficiency and good distribution is more important than a long period. When results from sub-sequences are to be used (e.g., for parallel Monte Carlo simulations) many PRNG's satisfying all the criteria except long disjoint sub-sequences may need to be discarded due to long-term correlations appearing as short-term correlations in a parallel setting [15, 16].

To discover non-deterministic features of a PRNG, the PRNG should be thoroughly tested. A detailed description of some of the classic tests can be found in Knuth [17], but we will mention some of them here to give a general idea.

Frequency test: The most basic requirement that a uniform PRNG must meet is that its numbers are uniformly distributed between zero and one. The test is as follows: Generate a large sequence of numbers between zero and one. Divide the interval into n bins. Count the number of random numbers that falls into bin j $(0 \leq j \leq n)$. Calculate χ^2 assuming that the

numbers are truly random to get a probability that the generated distribution is compatible with a random distribution.

Gap test: Another test is used to examine the length of "gaps" between numbers that falls in the same bin. Let α and β be two real numbers with $0 \leq \alpha \leq \beta \leq 1$. Given a sequence of supposedly random numbers, one considers lengths of consecutive sub-sequences $u_j, u_{j+1}, \ldots, u_{j+r}$ in which u_{j+r} lies between α and β while the other numbers do not. Having determined the gaps of length $0, 1, \ldots, t$, one applies the frequency test to this empirical sequence.

Tests on sub-sequences: Consider an external program which uses a PRNG. For example, if the program works with three random variables at a time it may consistently invoke the generation of three random numbers at a time. In such applications it is important that the sub-sequences of every *third* term in the original sequence is random. If the program requires q numbers at a time, the sequences

$$u_0, u_q, u_{2q}, \ldots; u_1, u_{q+1}, u_{2q+1}, \ldots; \ldots; u_{q-1}, u_{2q-1}, u_{3q-1}, \ldots$$

should undergo the same testing as the sequence $u_0, u_1, \ldots, u_n$.

3.2 Selecting from a Distribution

Quantum mechanics introduces a concept of randomness in the behavior of physical processes. The idea of an event generator is to simulate this behavior by using Monte Carlo techniques. While many techniques are used, we will here only highlight a few of them, in particular the most basic techniques used in, e.g., high energy physics event generators like HERWIG [18] and PYTHIA [19].

One of the most common situations is to have a function $f(x)$ which is non-negative in the allowed x range $x_{min} \leq x \leq x_{max}$. The wish is to select an x at random so that the probability in a small region dx around x is proportional to $f(x)\,dx$. Here $f(x)$ might be, e.g., a parton fragmentation function, a differential cross section or any of a number of probability density functions.

If it is possible to find a primitive function $F(x)$ for which the inverse $F'(x)$ is known, a random x can be found as follows:

$$\int_{x_{min}}^{x} f(x)\,dx = R \int_{x_{min}}^{x_{max}} f(x)\,dx \rightarrow x = F^{-1}(F(x_{min}) + R(F(x_{max}) - F(x_{min}))). \tag{3.1}$$

Here R is the fraction of the total area under $f(x)$ to the left of x. Unfortunately, functions of interest are rarely nice enough for this method to work, and more involved schemes need to be considered. Sometimes special tricks can be found. For example, the generation of a Gaussian, $f(x) = \exp(-x^2)$. This function is not integrable, but by combining it with the Gaussian of a second variable y, a transformation to polar coordinates (r, θ) yields

$$f(x)f(y), dxdy = \exp(-x^2 - y^2)\, dx, dy = r \exp(-r^2)\, drd\theta. \tag{3.2}$$

Now the r and *theta* distributions are easily generated and combined to yield x. Additionally, y is also a Gaussian-distributed number which can be used. For the generation of transverse momenta in parton fragmentation this is very convenient since there are two transverse degrees of freedom.

The main problems in generating representative samples of a function $f(x)$ occurs if it has singularities close to or within the range $x_{min} \leq x \leq x_{max}$. In this case it may be necessary with one or several variable transformations to make a function smoother. Examples of this can be found in, e.g., Sjöstrand et al. [19].

3.3 Monte Carlo Integration

Monte Carlo integration may be easiest explained by looking at conventional numerical integration methods. In conventional methods evaluation points are chosen and the integrand for every point is weighted to get

$$\int_\Omega f(\mathbf{r})\, d\Omega = \sum_{i=1}^{m} \omega_i f(r_i). \tag{3.3}$$

The values of the weights are chosen according to how the evaluation points are chosen. In one dimension the simplest form of Eq. (3.3) is made by choosing the evaluation points with equal spacing over the integration area. The weights then become the length of the integration region divided by the number of integration points so that

$$\int_0^1 f(x)\, dx = \frac{1}{m}\left(\sum_{i=1}^{m} f(x_i) + \mathcal{O}\left(\frac{1}{m}\right)\right). \tag{3.4}$$

Similarly, a two dimensional integration gives

$$\int_0^1 \int_0^1 f(x,y)\, dxdy = \frac{1}{m^2}\left(\sum_{i=1}^{m}\sum_{j=1}^{m} f(x_i, y_j) + \mathcal{O}\left(\frac{1}{m}\right)\right) \tag{3.5}$$

with an equal number of integration points in both dimensions. So by adding one dimension, the number of evaluation points has increased from m to m^2 to obtain the same order of accuracy. For n dimensions we have to carry out m^n evaluations. For an n-dimensional quantum N-body system we have nN degrees of freedom in the spatial case. The integral then becomes an integral over nN dimensions. For example, for a system with 500 particles in 3 dimensions we need to evaluate 10^{1500} points to get an accuracy of 10^{-1}.

Obviously, conventional integration methods are not usable in such a scenario. Monte Carlo integration, however, does not depend on the number of dimensions to get a reasonably accurate result. In Monte Carlo integration, the integrand is evaluated at random points $\mathbf{r}_i$ taken from an arbitrary probability distribution $\rho(\mathbf{r})$ (see for example Kent, [20]),

$$\int_\Omega f(\mathbf{r})\, d\Omega = \int_\Omega \frac{f(\mathbf{r})}{\rho(\mathbf{r})}\rho(\mathbf{r})\, d\Omega = \int_\Omega g(\mathbf{r})\rho(\mathbf{r})\, d\Omega = \sum_{i=1}^{m} g(\mathbf{r}_i) + \mathcal{O}\left(\frac{1}{\sqrt{m}}\right). \tag{3.6}$$

This is exact in the limit $m \to \infty$, but in a numerical approach one has to truncate the summation at some finite value m.

By choosing $\rho(\mathbf{r}) = 1$, the integrand is sampled uniformly at random points. If the function varies considerably within the integration domain, the variation of the individual samples will be significant. It is therefore advisable to choose $\rho(\mathbf{r})$ to duplicate the behavior of $f(\mathbf{r})$. If choosing $\rho(\mathbf{r}) = f(\mathbf{r})/\mathcal{N}$, the fraction $g(\mathbf{r}) = f(\mathbf{r})/\rho(\mathbf{r}) = \mathcal{N}$, merely a constant which is the true value of the integral (here the normalization constant $\mathcal{N}$), yielding the exact answer with only one sample. Of course, this would require knowing the answer before calculating it, but it illustrates how the Monte Carlo integration can be optimized by choosing a good probability distribution. This optimization is often referred to as *importance sampling*.

The main advantage of the Monte Carlo integration scheme is that it is independent of the number of dimensions. The evaluation time of the integrand depends only on the functional form, and the variance of the integral estimate depends solely on how much the integrand varies. However, a convergence of order $O(1/\sqrt{m})$ is not really satisfactory. While there are several techniques to improve the efficiency of Monte Carlo integration, the perhaps most well-known in quantum Monte Carlo is the Metropolis-Hastings algorithm.

3.3.1 Metropolis-Hastings

The Metropolis-Hastings algorithm [21, 22] (often shortened to the *Metropolis algorithm*) generates a stochastic (random) sequence of phase space points

that sample a given probability distribution. In quantum Monte Carlo methods each point in phase space represents a vector $\mathbf{R} = \{\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N\}$ in Hilbert space. Here $\mathbf{r}_i$ represents all degrees of freedom for particle i. Coupled with a quantum mechanical operator, each point can be associated with physical quantities (like kinetic and potential energy). The fundamental idea behind the Metropolis algorithm is that the sequence of individual *samples* of these quantities can be combined to arrive at average values which describe the quantum mechanical state of the system. The Metropolis algorithm then provides the sample points. We will refer to the randomized walk through the phase space as a *random walk*. From an initial position in phase space a *proposed move* is generated and the move is either *accepted* or *rejected* according to the Metropolis algorithm. In this way a random walk generates a sequence

$$\{\mathbf{R}_0, \mathbf{R}_1, \ldots, \mathbf{R}_i, \ldots\} \tag{3.7}$$

of points in the phase space. An important requirement for the random walk is that it is *ergodic*, which means that all points in phase space are reachable from any initial point. Taking a sufficient number of trial steps then ensures that all of phase space is explored and the Metropolis algorithm makes sure that the points are distributed according to the required probability distribution.

Given a probability distribution $\rho(\mathbf{R})$ to draw the points from, Metropolis et al. [21] showed that the sampling is most easily achieved if the points $\mathbf{R}$ form a *Markov chain*. A random walk is Markovian if each point in the chain depends only on the position of the previous point. A Markov process may be completely specified by choosing values of the *transition probabilities* of moving from $\mathbf{R}$ to $\mathbf{R}'$, $P(\mathbf{R}', \mathbf{R})$[1]. The Metropolis algorithm works by choosing the transition probabilities in such a way that the sequence of points generated by the random walk sample the required probability distribution.

To properly understand the Metropolis algorithm it is necessary to work out the statistical properties of the points on the Markov chain. Consider a large ensemble of random walkers, all evolving simultaneously. All the walkers move step by step in accordance with the transition probabilities. At a given time, the number of walkers at a point $\mathbf{R}$ is $N(\mathbf{R}, t)$. As the Markov chains evolve in time the number of walkers develops according to the master equation,

$$\frac{d}{dt}N(\mathbf{R}, t) = -\sum_{\mathbf{R}'} P(\mathbf{R}', \mathbf{r})N(\mathbf{R}, t) + \sum_{\mathbf{R}'} P(\mathbf{R}, \mathbf{R}')N(\mathbf{R}', t). \tag{3.8}$$

[1]It is convention to write the final position to the left of initial position.

As $t \to \infty$ the derivative $dN(\mathbf{R}, t)/dt \to 0$ so that

$$\sum_{\mathbf{R}'} P(\mathbf{R}', \mathbf{R}) N(\mathbf{R}) = \sum_{\mathbf{R}'} P(\mathbf{R}, \mathbf{R}') N(\mathbf{R}') \tag{3.9}$$

where $N(\mathbf{R}, t) \to N(\mathbf{R})$. Metropolis et al. [21] then realized that the distribution of walkers would end up in the required distribution $\rho(\mathbf{R})$ as long as the transition probabilities obeyed the equation of *detailed balance*,

$$P(\mathbf{R}', \mathbf{R}) \rho(\mathbf{R}) = P(\mathbf{R}, \mathbf{R}') \rho(\mathbf{R}'). \tag{3.10}$$

Imposing the condition of detailed balance is a necessary requirement for a random process to sample the phase space with the probability density $\rho(\mathbf{R})$. By imposing the condition of detailed balance on the transition probabilities we get

$$\sum_{\mathbf{R}'} P(\mathbf{R}', \mathbf{R}) \left(N(\mathbf{R}) - \frac{\rho(\mathbf{R})}{\rho(\mathbf{R}')} N(\mathbf{R}') \right) = 0 \tag{3.11}$$

so that

$$\frac{\rho(\mathbf{R})}{\rho(\mathbf{R}')} = \frac{N(\mathbf{R})}{N(\mathbf{R}')}, \tag{3.12}$$

showing that the number of walkers in the state $\mathbf{R}$ becomes proportional to the steady state distribution $\rho(\mathbf{R})$ to be sampled.

There is still some freedom in choosing the transition probabilities, which are not uniquely defined by the detailed balance condition. In the Metropolis approach, the walk is generated by starting from a point $\mathbf{R}$ and making a trial move to a new point $\mathbf{R}'$ somewhere in nearby phase space. The way to choose the trial moves is not crucial, as long as it satisfies the detailed balance. One such approach is to choose

$$P_T(\mathbf{R}', \mathbf{R}) = P_T(\mathbf{R}, \mathbf{R}') \tag{3.13}$$

and then accept or reject according to the rule

$$P_A(\mathbf{R}', \mathbf{R}) = min \left(1, \frac{\rho(\mathbf{R}')}{\rho(\mathbf{R})} \right). \tag{3.14}$$

Note that since this approach involves the ratio of probabilities there is no need to worry about normalization of the distribution $\rho(\mathbf{R})$. Combining the trial and acceptance probabilities we get

$$\frac{P(\mathbf{R}', \mathbf{R})}{P(\mathbf{R}, \mathbf{R}')} = \frac{P_T(\mathbf{R}', \mathbf{R}), P_A(\mathbf{R}', \mathbf{R})}{P_T(\mathbf{R}, \mathbf{R}') P_A(\mathbf{R}, \mathbf{R}')} = \frac{\rho(\mathbf{R}')}{\rho(\mathbf{R})} \tag{3.15}$$

and the condition of detailed balance is satisfied.

3.3.2 Variational Monte Carlo

Variational Monte Carlo is a quantum Monte Carlo method combining Markov-chain Monte Carlo[2] (MCMC) and the quantum mechanical variational principle.

Finding the ground-state energy of a many-body system of N particles and n dimensions is equivalent to minimizing of the integral

$$\langle H \rangle = E[\psi] = \frac{\int \psi^*(\mathbf{R}) H \psi(\mathbf{R})\, d\mathbf{R}}{\int \psi^*(\mathbf{R})\psi(\mathbf{R})\, d\mathbf{R}}. \tag{3.16}$$

According to the variational principle the energy will be at a minimum for the exact wave function Ψ_0. The functional $E[\psi]$ thus provides an upper bound to the ground state energy.

Rewriting eq. (3.16),

$$\langle H \rangle = \int E_L(\mathbf{R}) \frac{|\psi(\mathbf{R})|^2}{\int |\psi(\mathbf{R})|^2\, d\mathbf{R}}\, d\mathbf{R} \tag{3.17}$$

with the local energy

$$E_L = \frac{1}{\psi(\mathbf{R})} H\psi(\mathbf{R}), \tag{3.18}$$

the square of the wave function divided by its norm can be interpreted as the probability distribution of the system, arriving at

$$\langle H \rangle = \int E_L(\mathbf{R}) \rho(\mathbf{R})\, d\mathbf{R}, \tag{3.19}$$

where

$$\rho(\mathbf{R}) = \frac{|\psi(\mathbf{R})|^2}{\int |\psi(\mathbf{R})|^2\, d\mathbf{R}}. \tag{3.20}$$

The integral (3.19) may then be carried out with MCMC by moving walkers randomly through phase space according to the Metropolis algorithm, and sampling the local energy in each move.

3.3.3 Diffusion Monte Carlo

In the Diffusion Monte Carlo (DMC) method [23], the Schrödinger equation is solved in imaginary time,

$$-\frac{\partial d\psi(\mathbf{R}, t)}{\partial t} = [H - E]\psi(\mathbf{R}, t). \tag{3.21}$$

[2]Due to the use of Markov chains, Monte Carlo methods using the Metropolis-Hastings algorithm are often referred to as Markov-chain Monte Carlo methods.

The formal solution of (3.21) is

$$\psi(\mathbf{R},t) = e^{-[H-E]t}\psi(\mathbf{R},0), \tag{3.22}$$

where $e^{[-(H-E)t]}$ is called the *Green's function*, and E is a convenient energy shift.

The wave function $\psi(\mathbf{R},t)$ in DMC is represented by a set of random vectors $\{\mathbf{R}_1,\mathbf{R}_2,\ldots,\mathbf{R}_M\}$, in such a form that the time evolution of the wave function is actually represented by the evolution of a set of walkers. The wave function is positive definite everywhere, as it happens with the ground state of a bosonic system, so it may be considered as a probability distribution function.

The DMC method involves Monte Carlo integration of the Green's function by every walker. The time evolution is done in small time steps τ, using the following approximate form of the Green's function:

$$G = e^{-[H-E]t} = \prod_{i=1}^{n} e^{-[H-E]\tau}, \tag{3.23}$$

where $\tau = t/n$. Assume that an arbitrary starting state can be expanded in the basis of stationary states,

$$\psi(\mathbf{R},0) = \sum_{\nu} C_\nu \phi_\nu(\mathbf{R}), \tag{3.24}$$

we have

$$\psi(\mathbf{R},t) = \sum_{\nu} e^{-[E_\nu - E]t} C_\nu \phi_\nu(\mathbf{R}), \tag{3.25}$$

in such a way that the lowest energy components will have the largest amplitudes after a long elapsed time, and in the limit of $t \to \infty$ the most important amplitude will correspond to the ground state (if $C_0 \neq 0$)[3].

The Green's function, Eq. (3.23), is approximated by splitting it up in a diffusional part

$$G_D = (4\pi D\tau)^{-3N/2} \exp\{-(\mathbf{R}' - \mathbf{R})^2/4D\tau\}, \tag{3.26}$$

which has the form of a Gaussian, and a branching part

$$G_B = \exp\{-((V(\mathbf{R}) + V(\mathbf{R}'))/2 - E_T)\tau\}, \tag{3.27}$$

[3]This can easily be seen by replacing E with the ground state energy E_0 in (3.25). As E_0 is the lowest energy, we will get $\lim_{t\to\infty} \sum_\nu \exp[-(E_\nu - E_0)t] C_\nu \phi_\nu = C_0\phi_0$.

so that

$$G \approx G_D G_B. \tag{3.28}$$

While diffusion is taken care of by a Gaussian random distribution, the branching is simulated by creation and destruction of walkers with a probability G_B. The idea of DMC computation is quite simple; once an appropriate approximation of the short-time Green's function is found and a starting state is determined, the computation consists in representing the starting state by a collection of walkers and letting them independently evolve in time. That is, the walker population is repeatedly updated, until a large enough time when all other states than the ground state are negligible.

An important improvement to the DMC scheme above is the use of *importance sampling*. In problems with singularities in the potential (e.g., the Coulomb potential) the Green's function $\exp[-(H-E)t]$ will reach unbounded values, leading to an unstable algorithm. Even without singularities the scheme above is inefficient. This is due to the fact that no restrictions are imposed as to where the walkers will walk.

To impose such a restriction, the wave function $\psi(\mathbf{R}, t)$ is substituted with a new quantity $f(\mathbf{R}, t) = \psi_T(\mathbf{R})\psi(\mathbf{R}, t)$ where $\psi_T(\mathbf{R})$ is a time-independent trial wave function, which should be as close as possible to the true ground state[4]. This substitution can be shown [24, p. 92] to lead to a transformed Hamilton operator which may be written as a sum of three terms $H = K + F + L$, where

$$K = -D\nabla^2, \quad F = -D(\nabla \cdot \mathbf{F}(\mathbf{R})) + \mathbf{F}(\mathbf{R}) \cdot \nabla, \quad L = E_L(\mathbf{R}), \tag{3.29}$$

corresponding respectively to the kinetic part, the drift part and the local energy part.

An $\mathcal{O}(\tau^2)$ approximation of the Green's function is given by [25]:

$$\begin{aligned}\langle \mathbf{R}'|G|\mathbf{R}\rangle =& \frac{1}{(4\pi D\tau)^{3N/2}} e^{-[\mathbf{R}'-\mathbf{R}-D\tau\mathbf{F}(\mathbf{R})]^2/4D\tau} \times \\ & e^{E\tau-[E_L(\mathbf{R}')+E_L(\mathbf{R})]\tau/2} + \mathcal{O}(\tau^2).\end{aligned} \tag{3.30}$$

while an $\mathcal{O}(\tau^3)$ approximation of the Green's function is obtained from [26]

$$G = e^{E\tau} e^{-L/2\tau} e^{-F/2\tau} e^{-K\tau} e^{-F/2\tau} e^{-L/2\tau} + \mathcal{O}(\tau^3). \tag{3.31}$$

[4]The trial wave function can be found using, e.g., variational Monte Carlo.

3.4 On mixed-language programming

When developing a scientific software a frequent question is which programming language to use. On one hand there are high-level, interpreted programming languages like R, MATLAB® and Octave, where features like clean syntax, interactive command execution, integrated simulation and visualization and rich documentation shortens development time, leaving more time to focus on the scientific problem. On the other hand there are (relatively) low-level, compiled programming languages like FORTRAN, C and C++ which are faster (if programmed optimally), but generally less readable and, due to the overhead of compile time, slower to debug. To complicate the question further, one may have inherited an old software package (e.g., from a supervisor or the previous developer in the project), seemingly being left with the choice of either continuing to develop in the same language or translate the entire code to a new language.

The programming language Python [10] opens up a third possibility. When extended with numerical and visual modules like SciPy [27], it offers most of the functionality of MATLAB. However, one of the advantages of Python is that it is designed to be easily extendable with compiled code. With freely available tools like F2PY [28] and SWIG [29], old, well-tested FORTRAN and C functions can be directly reused in Python as is, rather than making wrappers and calling the functions as external processes with an extra, error-prone layer of input/output parsing.

While reusing old code is a great motivation in itself, it can be beneficial to look at code extensions from the opposite side as well. If one is to develop software for a given scientific problem from scratch, a high-level, interpreted programming language will yield the shortest period of development. Unfortunately, interpreted languages are slower than compiled languages, partially because compiled languages can do low-level optimalisations on e.g. loops during compilation, and partially because, after all, the interpreted language is just a layer on top of a compiled set of libraries. However, in a scientific program most of the time of the computation is usually spent in a small part of the program code (e.g. a matrix operation or an iteration over a dataset). The idea is then to make a profile of where in the code the time is spent, and rewrite the most time-consuming part in a compiled language[5].

[5]The time to rewrite the code should of course be weighed against the amount of time the code will actually be running.

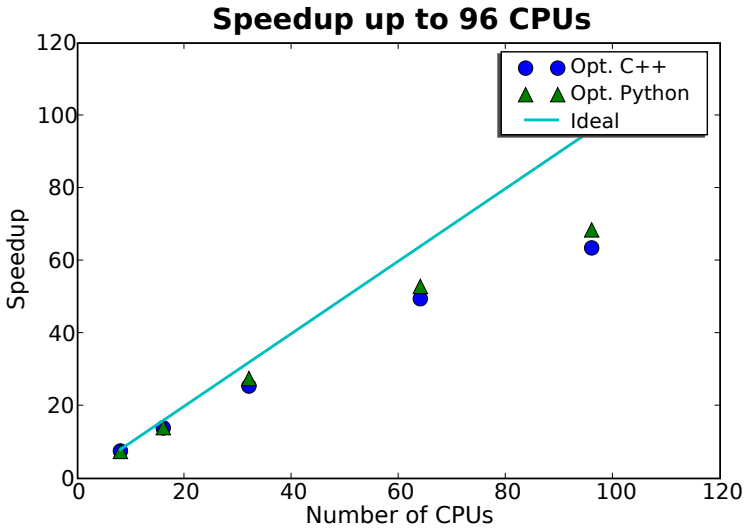


Figure 3.1: The figure shows the speedup for the simulation as a function of the number of CPUs used. The serial run took about 209 minutes for C++, 225 minutes for Python and was run with an initial 4800 walkers moved in 1750 time steps. (Taken from "MontePython: Implementing Quantum Monte Carlo using Python", Appendix A.2.)

3.4.1 MontePython – mixing Python and C++

While the article in Appendix A.2 presents the implementation of a general framework for simulating bosons using diffusion Monte Carlo (DMC), one of the main goals of the presented software (MontePython) was to illustrate the applicability of mixed-language programming for scientific computing. For the actual implementation of DMC we refer to the article in Appendix A.2. We will, however, repeat the main result here.

The main argument against programming in Python is the poor performance. One would naturally assume that this applies to a mixed-language program as well. Figure 3.1 show the speedup when running the same simulation on an increasing number of CPU's. Circles show the speedup for a pure C++ implementation of DMC while triangles shows the mixed-language implementation where the most costly computations are implemented in C++ while the main part of the DMC algorithm is implemented in Python. Note that that the speedup is actually better for Python for higher numbers of CPUs. The speedup is calculated as the ratio of the wall-time of the serial

simulation to the wall-time of the parallel simulation. Since the serial version of C++ is faster than the serial version of Python, when increasing the number of CPUs the walltime of the Python version gets closer to the walltime of the C++ version, and already for 32 CPUs the difference in walltimes is less than 1%. It should be noted that the speedup curve flattens out when going to large numbers of CPUs. This is due to the small number of random walkers on each node when having a constant global number of walkers.

One of the main challenges of the mixed-language approach used in MontePython was that the main data structure needed to be accessible both from Python and C++. The Python module NumPy (which is part of SciPy) provides an efficient array library highly capable of containing the data of the walkers. However, in the C++ implementation the walkers were represented as separate objects. While this representation makes sense in an object oriented implementation, it is not optimal performance-wise due to memory fragmentation. To kill two birds with one stone we decided to let the walkers be stored in a NumPy array while the C++ walker objects only contained pointers to the corresponding elements in the array.

While this concludes our brief look at implementing Monte Carlo, a key part for running Monte Carlo simulations has been left out. As the Monte Carlo algorithm is suitable for simulating systems too large for conventional methods, Monte Carlo is mostly interesting for simulating very large systems, thus requiring large amounts of computing resources. The remainder of this thesis will be devoted to exploring ways of acquiring such resources.

Part II

Data Management in Grid Middleware

Chapter 4

Distributed Computing

The physicist's need for computational power can easily outgrow the limitations of a cluster confined between the walls of a computing center. How to deal with such computations falls into the category of distributed computing . While the main work of this thesis is related to distributed data management, it is worth getting a brief overview of the machinery that supports and depends on the distributed storage.

Computing can, broadly speaking, be divided into two groups; high-performance computing (HPC) and high-throughput computing (HTC). While HPC focuses on relatively short-lived (rarely longer than a month), tightly coupled parallel jobs, the main focus of HTC is to run as many serial jobs as possible within a reasonable time frame to solve a larger problem. To take the problems described in Appendix A.3 as examples, the problem of ocean wave propagation is studied with partial differential equations (PDE's), voting in legislatures is studied with the use of Markov chain Monte Carlo (MCMC) and a Bose-Einstein condensate is studied with diffusion Monte Carlo (DMC). While parallelizing PDE's involves splitting up a lattice graph and dealing with inter-process communication on the borders of the sub-lattices, MCMC requires only communication at the start and end of the simulation. Hence, PDE's belong to the HPC range of problems while MCMC typically falls into the HTC category. DMC, however, is not so easily classified. Being a Monte Carlo method, one may jump to conclusions and classify DMC as a HTC problem. However, the algorithm requires a global update of the observables at each time step to correctly calculate the branching term (see Section 3.3.3). While this involves much less communication than for PDE's, the communication is far from negligible. The question of which category (if any) DMC fits into is a chapter in itself and will be discussed further in Chapter 6.

The term distributed computing relates to running jobs on the wide area

network (WAN), as opposed to cluster jobs which are confined to a local area network (LAN). Distributed computing is therefore closely related to high-throughput computing, and the main paradigms for distributed computing are grids and clouds, as will be explored in Section 4.2 after looking at a physical experiment in need of such technologies in Section 4.1.

4.1 From Proton to Dataset: Computing needs of the ATLAS experiment

4.1.1 The Large Hadron Collider

Figure 4.1 shows an overview of the accelerator complex at the European Organization for Nuclear Research (CERN) which includes the Large Hadron Collider (LHC), the largest, highest energy particle accelerator ever built [31, 32]. LHC provides high-energy particles to the four experiments shown on the figure; A Large Ion Collider Experiment (ALICE), A Toroidal LHC ApparatuS (ATLAS), Compact Muon Solenoid (CMS) and LHC beauty (LHCb) and two smaller experiments; the TOTal Elastic and diffractive cross section Measurement (TOTEM) (close to CMS) and LHC forward (LHCf) (near ATLAS).

While these experiments are scientifically independent, they all depend on high-energy particle collisions provided by the LHC. In a synchrotron, particles are accelerated using a magnetic field. This magnetic field must increase synchronously with the energy. Since a magnet has a limited range where the magnetic field can be increased linearly and precisely enough, in order to gain very high energies, it is necessary to accelerate the particles through a succession of increasingly sized accelerators. To gain the high energies required to be inserted in LHC, the particles go through a series of synchrotron accelerators. While the CERN accelerator complex accelerates Lead ions as well, the acceleration of the proton serves as a good example. Starting up from hydrogen, the electrons and protons are separated and the protons accelerated to 50 MeV to be injected from the linear accelerator Linac2 to a booster. The booster accelerates the protons to 1.4 GeV, before injecting them to the Proton Synchrotron (PS) where they are further accelerated to 25 GeV and passed on to the Super Proton Synchrotron (SPS) for further acceleration. At 450 GeV the protons are injected in both directions to the Large Hadron Collider, where they are accelerated up to 7 TeV to generate proton collisions with up to 14 TeV center-of-mass energy.

When accelerated and at full intensity, the protons circulate in LHC in 2808 bunches in each direction, each containing 10^{11} protons. With a bunch

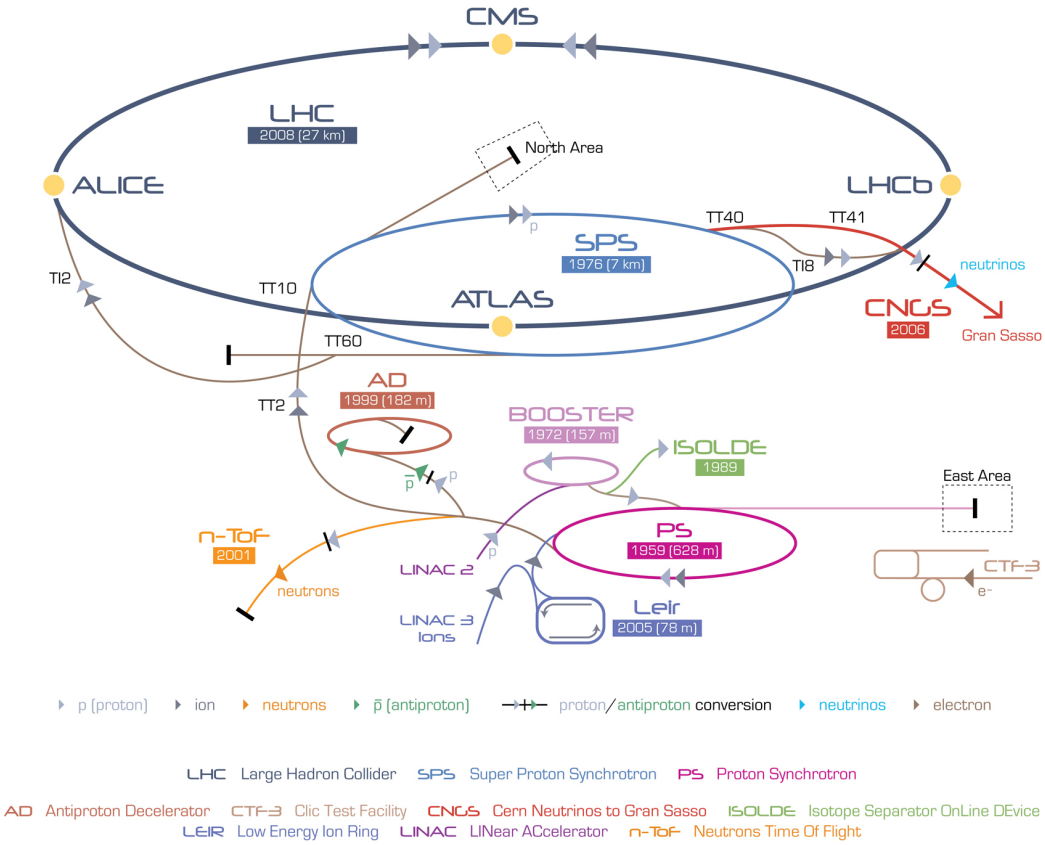


Figure 4.1: Overview of the CERN accelerator complex. Protons are injected from Linac2 to the Booster, and then further accelerated in the Booster, PS (Proton Synchrotron) and the SPS (Super Proton Synchrotron) before injection to LHC (Large Hadron Collider) in both directions. Lead ions are injected from Linac3 to LEIR (Low Energy Ion Ring) before being accelerated in PS and SPS before injected to LHC. (Illustration taken from CERN Web pages.)

crossing rate up to 40 MHz and around 25 collisions per bunch (depending on lumnosity) LHC produces up to ∼1 billion collisions per second. Consider the data flow of, e.g., the ATLAS detector. Here, one collision (or event) takes about 1.6 MB. To store all the events would require a sustained bandwith of 12.8 Pb/s, which with today's technology is unrealistic. However, searching for new physics discoveries, most of these events are not interesting. The ATLAS Trigger and Data Aquisition (TDAQ) uses a three level trigger system to reduce the number of events. The first level (LVL1), which is hardware-based, searches in a subset of the data for specific combinations of high transverse momentum, electrons/photons, hadrons and jets and missing transverse energy. The second trigger (LVL2) is software-based and refines the search even further by taking into account data from the inner detector and a region-of-interest mechanism. While the LVL1 and LVL2 triggers are based on specialized algorithms on highly selective data and have latencies of ∼2 μs and ∼10 ms, respectively, the last level event filter accesses the full event data to run event reconstruction and has an expected latency of a few seconds.

While this trigger mechanism reduces the event ratio to 200 Hz, it still requires 320 MB of raw data to be stored per second during data-taking. Taking into account planned breaks in production, ATLAS alone will produce 3.2 PB of raw data per year. Even though all of this will be stored at CERN, the needed computing resources to reconstruct the events and do the data analysis needed to discover new physics cannot fit into the physical area of the computing center of CERN[1]. Additionally, to avoid loss of data in case of natural disasters, terrorist attacks or human errors at CERN, the data need to be replicated elsewhere.

4.1.2 The ATLAS Computing Model

The data from the event filter comes in form of a byte-stream reflecting the format delivered from the detector, and is not very suitable for physics analysis. To maintain reproducibility while ordering the event data for efficient searching and analysis, the ATLAS Computing Model introduces an event store, i.e., a set of successively derived event representations [33], of which we will mention the most important here. The Raw data (RAW) from the event filter are stored in files of up to 2 GB size with unordered events. The RAW data are reconstructed to get an consecutive, ordered presentation of the events, the Event Summary Data (ESD). The ESD is further derived to make Analysis Object Data (AOD). The AOD is a reduced event repre-

[1]Not to mention the required electrical power and cooling of the computers.

sentation containing physics objects and other elements of analysis interest. While the AOD contains all the physics data needed for analysis, the Tag Data (TAG) contains event level metadata, information about events to be able to identify and select events of interest to a given analysis. Further derived from AOD are the Derived Physics Data (DPD). These are n-tuples of events generated by physicists to analyze and present specific types of events. Additionally there are representations of simulated data from Monte Carlo simulations.

In the ATLAS computing model, computing and storage resources are distributed in a hierarchical system of tiers as shown in Figure 4.2. The tiers have different roles:

- The **Tier-0** facility is located at CERN and is responsible for archiving and distributing the primary RAW data from the event filter and performing first-pass processing. The derived data (ESD, primary AOD and TAG sets) are distributed to the Tier-1 facilities. The access to Tier-0 is restricted to people in the central production group and those providing the first-pass calibration.

- The **Tier-1** facilities have the responsibility of hosting the RAW data (about one tenth each) and of providing long term access to the RAW data. Additionally, the Tier-1 facilities are responsible of providing ATLAS-wide access to the derived data sets, hosting a secondary copy of ESD's, AOD's and TAG's from another Tier-1 and simulated data from Tier-2 facilities to improve access and provide fail-over. Access to Tier-1 facilities is restricted to the production managers of the working groups and to the central production group for reprocessing.

- The **Tier-2** facilities support calibration, simulation and analysis activities. They provide analysis capacity for physics working groups and subgroups and therefore host all TAG samples, some AOD's and some of the physics group DPD samples. All members of the ATLAS virtual organization have access to all Tier-2's.

An interesting feature of the ATLAS computing model is that it puts no restrictions on which middleware that should be used on the different tiers, as long as they can provide the required computing power and storage space. As a result the tiers of ATLAS are run on three different grids with distinct middleware running together as one, those being Enabling Grids for E-sciencE [34] (EGEE) with gLite middleware [35], the Open Science Grid [36] (OSG) with the Virtual Data Toolkit middleware [37] and NorduGrid [38] with the Advanced Resource Connector (ARC) middleware [39]

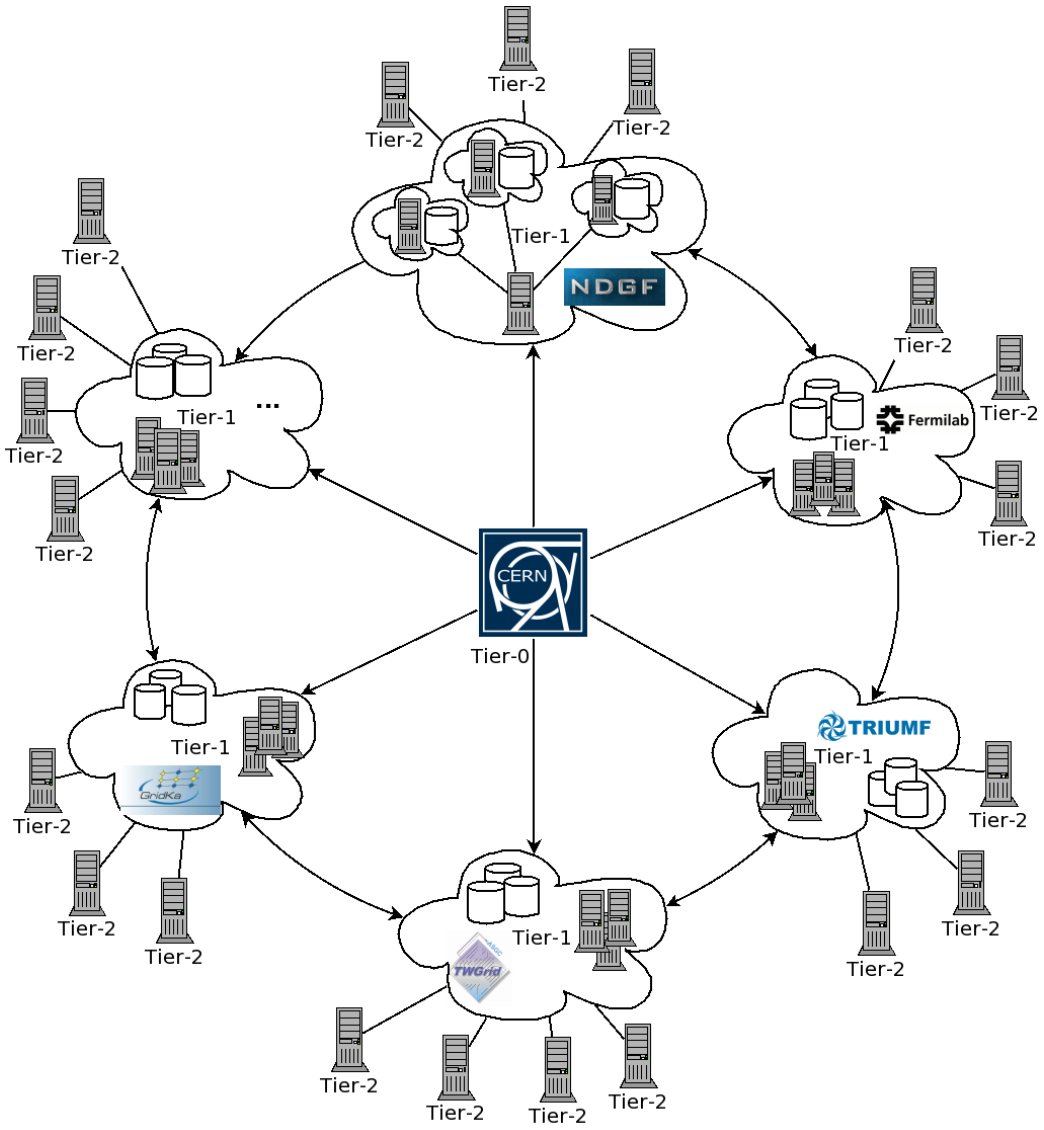


Figure 4.2: Overview of the Tiers of ATLAS. The Tier-0 facility at CERN is located in the middle, connecting to Tier-1 facilities (clouds). The Tier-1 facilities connects to each other, while the Tier-2 facilities only connect to assigned Tier-1 facilities.

(Appendix A.4). While the management of distributing the data sets between the tiers will be discussed in Section 5.1, the term grid and the middleware running on the NorduGrid will be explained in Section 4.2.

4.2 Grids and Clouds – Distributed Technologies

The idea of having on-demand access to computing, data and services is not entirely new. Leonard Kleinrock, one of the internet pioneers who contributed to setting up the ARPANET, envisioned "the spread of computer utilities, which, like present electric and telephone utilities, will service individual homes and offices across the country" [40]. While this vision may have been somewhat premature at the time, the evolution of new standards and technologies have brought us a great deal closer to realizing the vision. In the seventies and eighties, internet standards like TCP/IP [41, 42] and the "vague but interesting" proposal of Tim Berner-Lee in 1989 (on the management of general information about accelerators and experiments at CERN [43]) describing the Hyper-Text Markup Language (HTML) and the Hyper-Text Transfer Protocol (HTTP) laid the foundations for what was to become known as the World Wide Web (WWW). While WWW in the early nineties consisted of static information, the introduction of Web 2.0 [44] in the last decade with interactive web services like wiki, blog and RSS feeds, has opened the possibility of using the web as a computing platform. The introduction of the Service Oriented Architecture (SOA) with web services communicating using the Simple-Object Access Protocol (SOAP) standard provided means for connecting world-wide resources into grids and later clouds.

Lately, there has been some level of confusion as to the difference between computing grids and computing clouds. Some argue that the term *cloud* is just another word for *grid* that appeared as a response to the slower-than-promised progress in grid technology. Others argue that there is a fundamental difference between the concepts of grids and clouds, the first evolving from an academic need for sharing resources and the latter evolving from a commercial urge to sell resources. Much of the confusion is probably due to the fact that grids and clouds share the same goal of utilizing available computing resources distributed over a large geographical area. We will in the remainder of this section briefly touch upon the concepts of grids and clouds, before discussing a grid middleware implementation to try to pinpoint the conceptual difference between grids and clouds.

4.2.1 Grid

While several attempts have been made to define the grid (see, for example [45, 46, 47]) probably the most commonly accepted definition is the three point checklist by Ian Foster [48]: A grid is a system that

1. *coordinates resources that are not subject to centralized control...* (Non-centralized control of resources is often connected with the term Virtual Organization (VO). A VO is a set of individuals or institutions with a common set of rules for sharing resources. The individuals/institutions are typically geographically distributed in several countries.)

2. *... using standard, open, general-purpose protocols and interfaces...* (Open standards are vital for enabling different communities to share resources. As an example, the success of WWW is for a large part due to the open standards HyperText Transfer Protocol (HTTP) and HyperText Markup Language. While there is a great number of web browsers on the market, thanks to the open standards, all of them are able to present a HTML document in the same way.)

3. *... to deliver nontrivial qualities of service.* (The utility of the combined system should be significantly better than that of the sum of its parts.)

Being mainly developed for scientific computing needs, an important aspect of a grid is the sharing of resources. Commonly, the resources behind the grid deployments are based on public funding and organizations provide resources to share by installing a grid middleware.

Similar to the way an operating system provides an abstraction layer on top of the hardware in a computer, a grid can be viewed as a set of increasingly abstracted layers between hardware and users. Figure 4.3 shows the typical grid abstraction layers. At the bottom, computers, data network, storage systems, data sources and scientific instruments (e.g., the ATLAS experiment as described in Section 4.1) represent the hardware resources in the grid, i.e., the *grid fabric*. These are connected as networked resources across organizations. On top of the hardware, operating systems, queuing systems, libraries, application kernels and internet protocols represent the local abstraction of the hardware. To connect the distributed grid fabric to one unit, the *grid middleware* provides services to handle security, information, data management, resource access and quality of service. To protect the grid fabric from unauthorized access, all the services need to go through a security layer to access the grid fabric. For applications to access the grid middleware, a set of *grid tools*, or client tools, provide high-level interfaces to the middleware. As the grid tools are middleware dependent, the layer

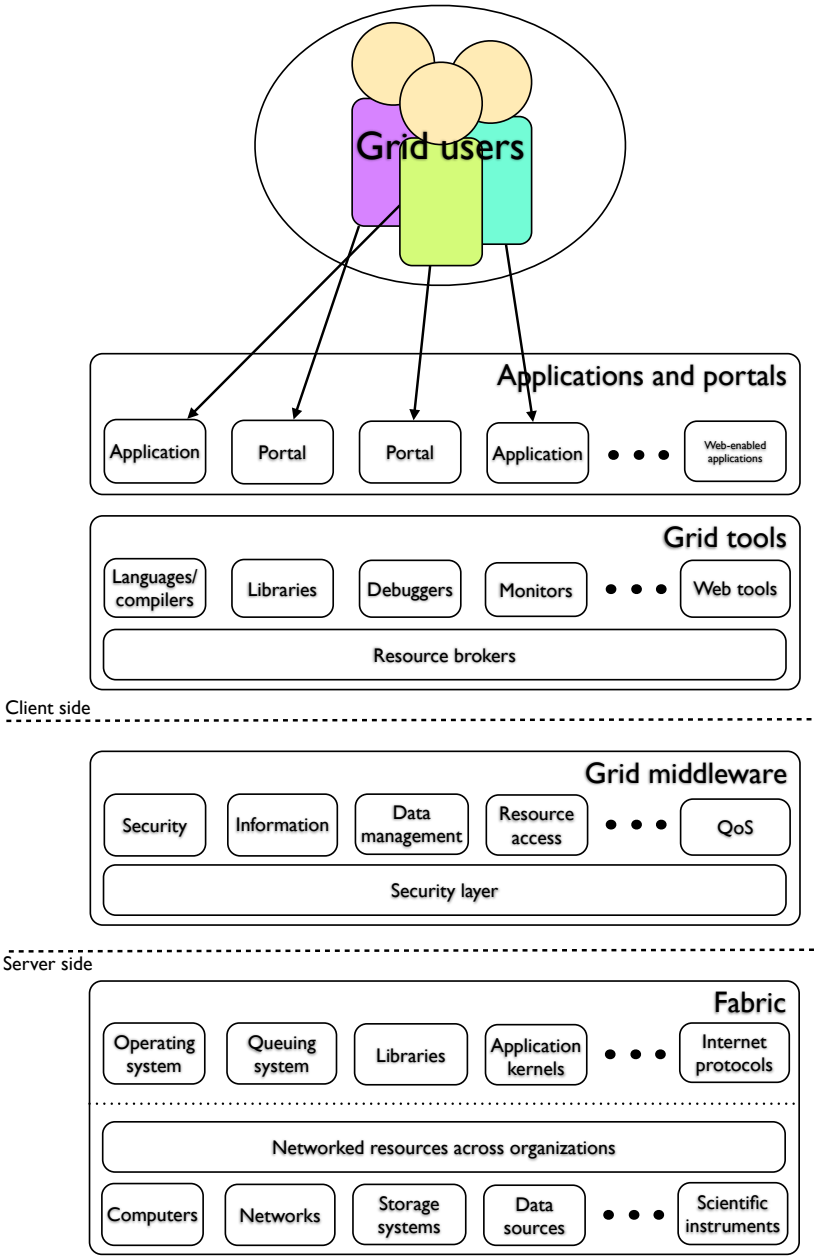


Figure 4.3: Schematic view of the grid abstraction layers. The users use *applications and portals* to interact with the grid. The applications acts as a frontend for the users and uses the *grid tools* to communicate with the *grid middleware* which is an abstraction layer in front of the *fabric* consisting of the hardware and the software needed to communicate with the hardware.

of applications and portals on top of the grid tools are often designed to work with grid tools from various grid middlewares. An example of such an application is Ganga, which will be used in Chapter 6.

4.2.2 Cloud

Cloud computing can be considered a rather more immature concept than grid computing. As an illustration, Gartner's hype cycle [49] characterizes technologies through a series of phases "*from over-enthusiasm through a period of disillusionment to an eventual understanding of the technology relevance and role in a market or domain*". While grid computing hit the peak of inflated expectation in 2002 (to continue into a period of disillusionment), cloud computing reached the same peak in 2009. This is not to say that the idea of clouds should be discarded. However, being at the peak of the hype, it can be difficult to extract the basics of what a cloud is from the diverse solutions embracing the cloud paradigm.

However, some early attempts on defining the cloud paradigms have been made, of which one of the most notable is "*Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized Service Level Agreements.*" [50].

The main building blocks of a cloud are the services. The cloud actors access the services both to add resources and utilize resources. The services provides quality-of-service (QoS) guarantees through service level agreements. By combining services, the service user can create a customized virtual platform. Conceptually the basic services can be divided into three classes.

- **Hardware as a Service** (HaaS) provides customizable, scalable hardware resources, typically as a pay-per-use subscription service. Examples of HaaS are Amazon EC2, IBM's Blue Cloud project, Nimbus and Eucalyptus.

- **Data as a Service** (DaaS) provides data in various formats and from multiple resources to be accessed over the Internet. Examples of DaaS are Amazon Simple Storage Service (S3) and to some extent Google Docs and Adobe Buzzword.

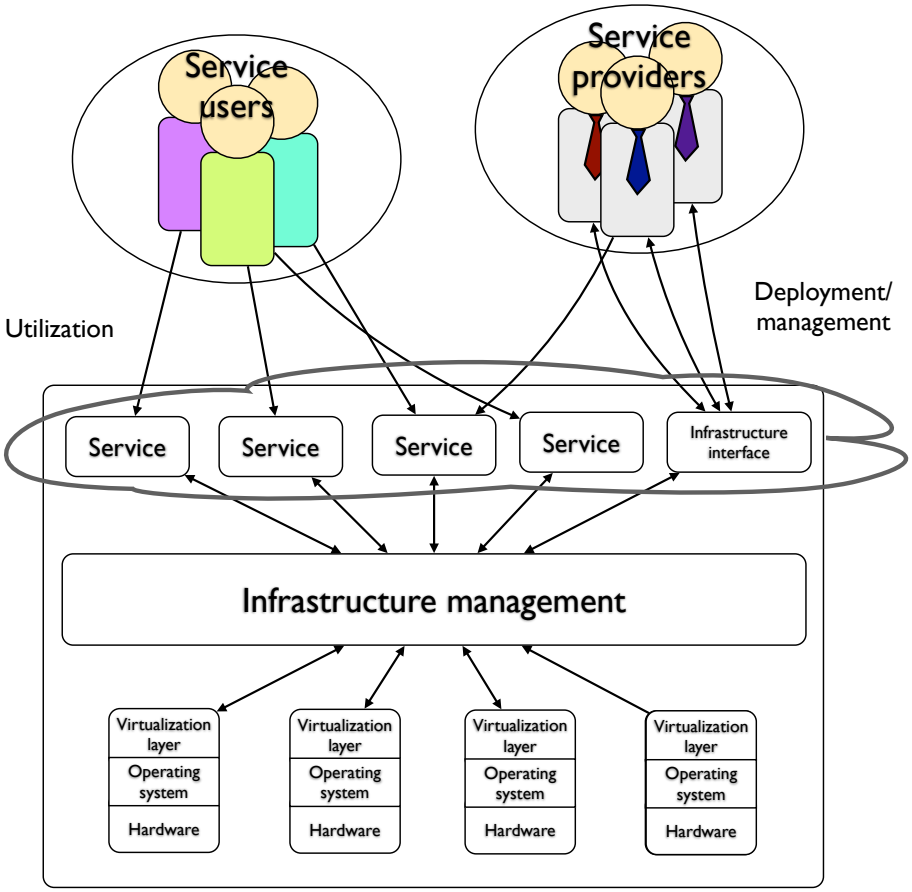


Figure 4.4: Schematic view of the cloud actors; the *service users* and *service providers*. Both users and providers need to go through the *infrastructure management* to use and provide hardware through a *virtualization layer*.

- **Software as a Service** (SaaS) offers software applications as web services, usually accessible through standard Web browses, thus eliminating the need to install and run the application on the client machine. Examples here are office applications provided by, e.g., Google Docs and Microsoft Office Live.

Additionally, concepts like Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) combines the above mentioned concepts to offer higher-level services. For example, Google Wave is a web-based service, computing platform and communication protocol, combining HaaS, DaaS and SaaS in one service.

Figure 4.4 shows the relationship between the cloud actors, i.e., the service users and the service providers, and the hardware resources. When compared to Figure 4.3 clouds have fewer abstraction layers, and service providers, being at the same side of the infrastructure management as the users, have more control in adding and removing services. However, the hardware is isolated through separated virtualization layers. While in grid one security layer spans the entire grid middleware, security in clouds is provided through isolation of the different virtualization layers. Note also that where grids provides high-level services like metadata searches and data services through the grid tools, the clouds leave these issues to the cloud applications. This may be due to the current lack of federated clouds, as grids were in a similar state before the need for federated grids forced a need for standardization of grid tools.

4.2.3 The Advanced Resource Connector

A detailed overview of the Advanced Resource Connector middleware is given in the article in Appendix A.4 .

Since the Condor project [51] set out on their hunt for idle workstations in 1988, many have joined the hunting party. Where Condor developed a computing environment with heterogeneous distributed resources, the appearance of the Globus Toolkit [52] and its Grid Security Infrastructure (GSI) created a layer for secure, coordinated access to geographically distributed computing and storage resources in the late nineties.

Meanwhile, in late 2000, the high-energy physics (HEP) community in the Nordic countries, due to their participation in LHC experiments, e.g., the ATLAS project, needed urgently to coordinate their available computing power in order to contribute to the large computing needs of the LHC collaboration. A peculiarity of the scientific and academic computing resources in the Nordic countries is that it consist of geographically scattered small to

medium sized facilities of different kinds and ownership. The only viable solution to meet these computing needs was to join these facilities into a unified structure. Studies and tests conducted by the NorduGrid collaboration [38] revealed that the existing solutions at the time were either not ready or did not meet the requirements of the Nordic HEP community [39].

To meet these requirements NorduGrid set out to develop what is now known as the Advanced Resource Connector (ARC) (Appendix A.4). Building on the by then *de facto* standard Globus Toolkit 2002, ARC was developed and put in production in 2002. While being in non-stop production since 2002, the development of ARC is a continuously ongoing effort. This has been possible due to a set of strict rules for the middleware development [39]:

1. A grid system, based on ARC, should have no single point of failure, no bottlenecks.

2. The system should be self-organizing with no need for centralized management.

3. The system should be robust and fault-tolerant, capable of providing stable round-the-clock services for years.

4. Grid tools and utilities should be non-intrusive, have small footprint, should not require special underlying system configuration and be easily portable.

5. No extra manpower should be needed to maintain and utilize the Grid layer.

6. Tools and utilities respect local resource owner policies, in particular, security-related ones.

Supported by the European Union (EU) Framework Package 6, "Grid-enabled Know-how Sharing Technology Based on ARC Services and Open Standards" [53] (KnowARC) started up in June 2006 to improve and extend the ARC middleware while complying with the requirements of grid standardization and interoperability. Up to the end of the project in November 2009, KnowARC has redesigned and implemented a Web-Service based ARC solution. To keep the continuity of the ARC production, the new implementation is intended to be introduced gradually after thorough testing and validation by NorduGrid.

Adding new services gradually is possible due to the plug-able architecture of ARC. Figure 4.5 shows the message flow between the client side and the

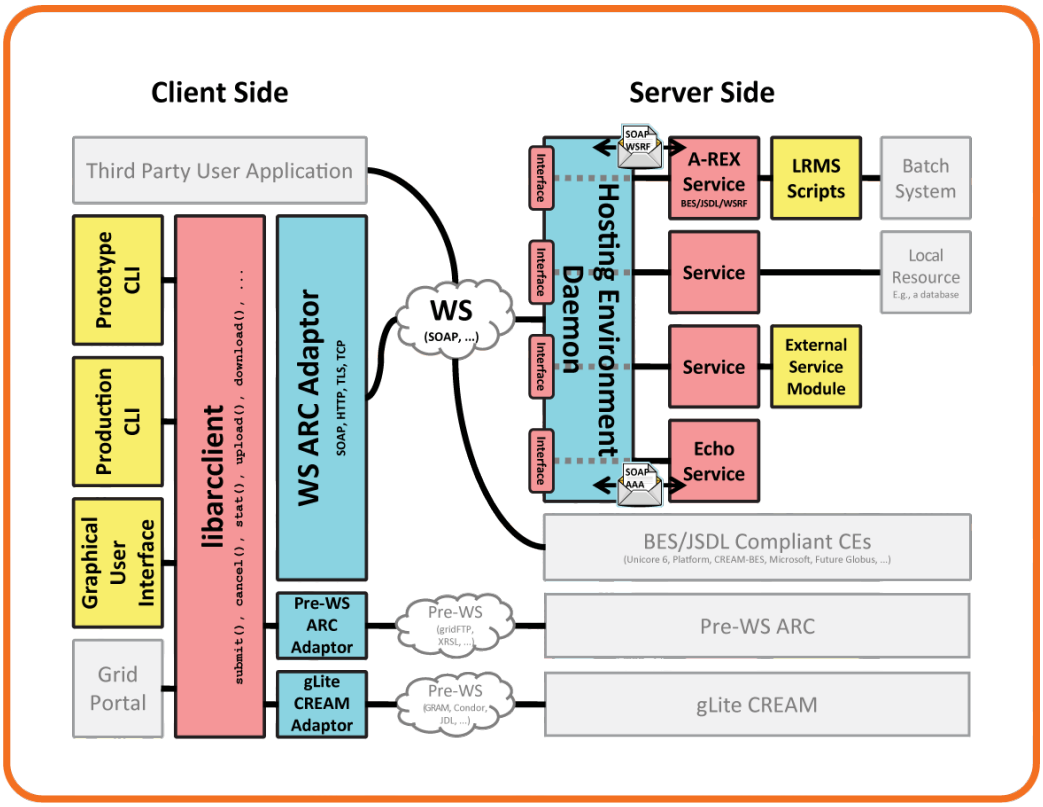


Figure 4.5: Overview of the client side and the server side of Web Service (WS) ARC middleware. From left to right, client tools either connects through the ARC client library (`libarcclient`) or directly to the WS cloud or pre-WS clouds for backwards compatibility. If the WS cloud is used, depending on the SOAP message, the request goes to the Hosting Environment Daemon (HED) which routes the message to the requested service, which in turn does the required work and sends the result back to the sequester. (Illustration taken from KnowARC web pages.)

server side. User applications (command line interfaces (CLI's), graphical user interfaces, grid portals) can access the server side through either the WS cloud, using the Simple Object Access Protocol (SOAP) or, for backwards compability, with pre-WS standard protocols like GridFTP, XRSL, Gram, Condor, JDL[2]. The client tools provided by ARC build on the ARC client library (`libarcclient`) which provides low-level commands such as `submit()`, `cancel()`, `upload()`, `download()` and so forth. These commands again build on an adaptor layer which converts the request to a WS message through a chain of SOAP, HTTP, TLS and TCP. In the case of an ARC SOAP request, the message is sent through the WS cloud to the Hosting Environment Daemon (HED), a minimal, plug-able WS container. The HED parses the message and routes to the requested service which does its work and returns the result to the requester.

The HED container is worth some extra attention. It contains a set of plug-able components which provides an interface to connect the services to HED. There are four kinds of plug-able components with well defined tasks: Data Management Components (DMC) are used to transfer data using various protocols, Message Chain Components (MCC) are responsible for the communication within clients and services as well as between the clients and the services, ARC Client Components (ACC) are plugins used by clients to connect to different Grid flavors, and Policy Decision Components (PDC) are responsible for the security model within the system. Figure 4.6 illustrates how a message is propagated through the chain of MCC's in HED, from the Transfer Level Protocol (TCP) MCC, through the Transfer Level Security (TLS) MCC, the HTTP MCC, via a plexer to the SOAP MCC. The SOAP MCC invokes the wanted service and the result is returned back through the same chain.

The most important services included in the final release of the KnowARC project are the ARC Resource-coupled Execution Service (A-Rex), providing execution of computational tasks, Chelonia, providing data storage (see Section 5.3), Hopi, providing data transfer, ISIS, a distributed information service and Charon, providing remote policy decision. ARC is a grid middleware, intended for connecting resources to form a grid. However, combining the above mentioned services the likeness to cloud concepts such as Platform as a Service and Infrastructure as a Service is quite striking. The main difference lays in the way resources are provided. In clouds the resources are provided by the infrastructure provider, so the service provider can gain flexibility and reduce cost [50]. In an ARC deployment the resources are owned

[2]One may argue that the problem of the early grid solutions is not the *lack* of standards, but rather the overflow of overlapping standards.

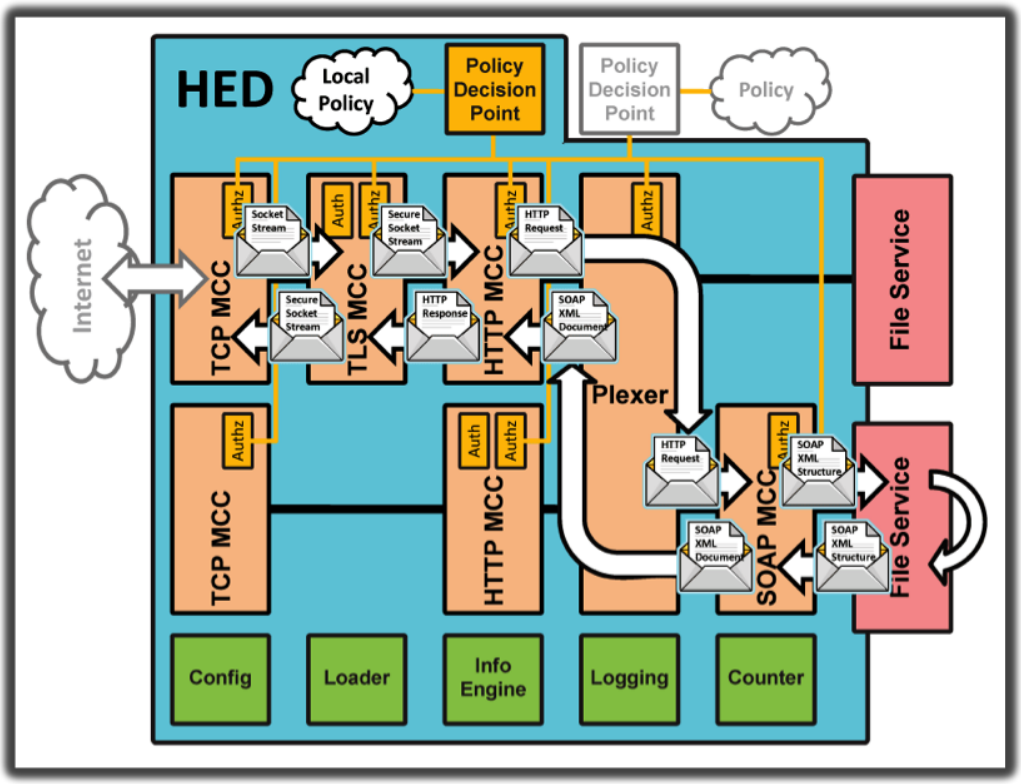


Figure 4.6: Overview of the Message Chain Components (MCC's) of the Hosting Environment Daemon (HED). The MCC's parses layers of a message and after confirming with the Policy Decision Component (PDC) passes the message to the next MCC in the chain.

by the service providers who share the resources through the infrastructure. For example, a resources owner who wants to share the resource, e.g. a computing element, will install and run a HED instance with the A-Rex service which is registering to a remote ISIS service. While this is a sensible way to share resources, it makes it difficult to provide quality-of-service guarantees as the service provider do not necessarily control the infrastructure the service is connected through.

It should, however, be emphasized that neither the definition of grid nor the definition of cloud is widely agreed upon and that they should be viewed as descriptions of currently available solutions. ARC came as a response to missing capabilities in other grid middleware solutions and still differs from most grid solutions in, e.g., the lack of centralized control. It also differs from cloud solutions in that it is interoperable with other distributed solutions and in that the service providers share the infrastructure responsibility. The common goal of grids and clouds is to connect available, but geographically distributed resources to lighten the burden of local resources. Be it for a grid or a cloud, ARC may simply be viewed as an advanced resource connector.

Chapter 5

Distributed Data Management

Distributed data management covers all aspects in the life cycle of distributed data, including topics like data transfer, distributed storage, data localization, data security, data availability and data access. While most of these topics are touched upon in this chapter, the main focus of this chapter (and in fact the thesis) are on distributed storage. However, it may be worthwile to take a look at a real life example of distributed data management before delving into the aspects of distributed storage. The remainder of this chapter is organized as follows. In Section 5.1 we have a brief look at the system used in the ATLAS Distributed Data Management model and the Nordic Tier-1 in particular before we look closer at distributed storage and the ARC storage solution in Section 5.2.

5.1 Don Quijote – ATLAS Distributed Data Management

In full operation, with a luminosity of 10^{34} $\mathrm{cm}^{-2}\mathrm{s}^{-1}$ and 7 TeV beams, the ATLAS experiment will generate tens of petabytes of data per year, to be aggregated and distributed globally between the tiers of ATLAS as described in Section 4.1. To handle the flow and replication of these data the ATLAS Distributed Data Management (DDM) project was established in 2005 to develop the system Don Quijote 2 [54] (DQ2). DQ2 is a centralized service at CERN providing a data management interface to the Tier-0 facility, most Tier-1 facilities and their associated Tier-2 facilities. The tiers are spread globally and run on three different grid flavours. The task of DQ2 is to manage the complete data flow of the ATLAS experiment from raw data archiving through global managed production and analysis to individual physics analysis at the various home institutions.

Like in the ATLAS computing model, the primary concept in DQ2 is the dataset. A dataset is an aggregation of data from a set of files that are processed together in a computation or data acquisition. Examples of datasets are the RAW data, ESD's, AOD's and DPD's mentioned in Section 4.1. A dataset has two distinct properties:

- A dataset is *versioned*, meaning that a dataset which is altered (by adding or removing a file) is tagged with a version number. The tag 0 points to the last version of the dataset.

- A dataset has a *mutability* state which can be *open*, *closed* or *frozen*. If a dataset is open, new data can be added to it, thus creating a new version. In a closed dataset, new data cannot be added, but file transactions which have been started before the dataset was closed may change the state back to open again by creating a new version. A dataset is frozen, or immutable, when the last version is closed and no new version can be added to it.

In DQ2, the file is the basic system unit. It is identified both by a Globally Unique Identifier (GUID) and a human-readable Logical File Name (LFN). A file is immutable and cannot be updated. For each update a new GUID and LFN must be created. A file may be added to a dataset version and removed in a later version. However, per default, it will still exist in the system. One of the reasons for this is that both GUID-based and LFN-based lookup is required. To achieve adequate performance, GUID-LFN associations needs to be recorded in many places, and both removing and renaming a file require costly synchronizations.

The DQ2 system consists of a set of command-line tools, end-user tools and a production system layered on top of a set of centralized catalogues and a set of site-based services for communicating with the different grid middlewares:

- The DQ2 tools provide logical organization at dataset level, supporting the data aggregations by which data is replicated, discovered and analyzed. The tools for end-users and production are separated since the usage patterns of the two are quite different. While production tasks are well-defined and managed at the collaboration or physics group level, the user activities are more chaotic in nature, requesting subsets of production data and uploading data produced by individual users.

- The centralized catalogues store file metadata such as GUID, LFN, size and checksum and dataset metadata such as locations of dataset

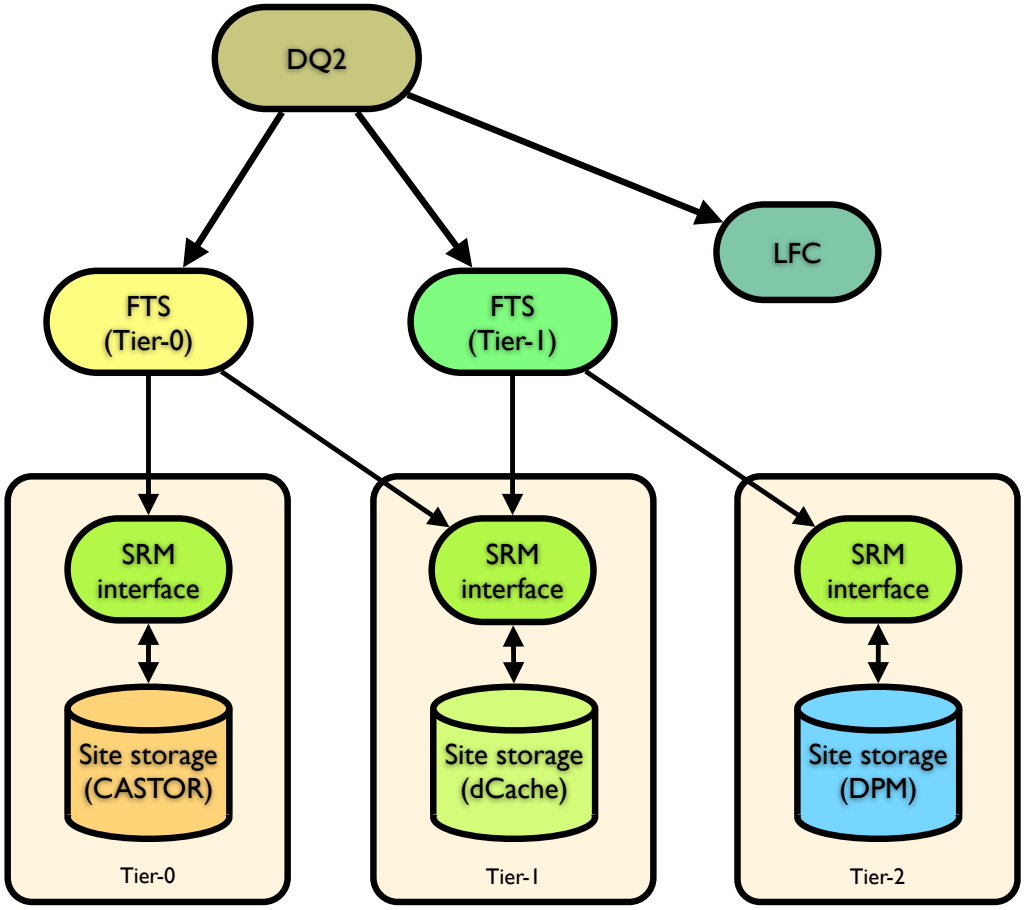


Figure 5.1: The ATLAS DDM topology. DQ2 uses LFC (one at each site) for file look-up and FTS's at Tier-0 and Tier-1's for file transfers. The FTS's communicate with the storages at each site through SRM interfaces.

replicas and users' dataset requests. Being centralized, the catalogues need to handle massive amounts of lookups and registrations.

- The local site services deals with communication between DQ2 and the grid middlewares. All managed data movement in DQ2 is automated using subscriptions, i.e., when a site subscribes to a dataset, the DQ2 site service agent acts to pull the dataset to the site as well as to keep the site up-to-date on later changes to the dataset. Data movement is triggered from the destination side so that local uploading can be done using site-specific mechanisms if desired.

Figure 5.1 shows the ATLAS DDM topology with the relations between DQ2, the LHC Computing Grid (LCG) File Catalog (LFC), the File Transfer

Service (FTS), the Storage Resource Manager (SRM) and the site storages. Each ATLAS cloud (Tier-0 and Tier-1's) runs a file catalog and a transfer service. Each LFC is a centralized service maintaining a separate namespace for the data stored in its particular cloud. To upload a file to a site, DQ2 first register it at the LFC at the corresponding cloud, then contacts the corresponding FTS which redirects the file transfer to the SRM service of the Tier-2. For file replication DQ2 registers and uploads the file to two different clouds, and to delete a replicated file all the clouds with a replica need to be contacted. As there is no direct communication neither directly between the LFC's nor between the LFC and the SRM at each site, manual work is needed to make sure the different namespaces are not fragmented (e.g., removing file registrations to non-existent files and registering or removing non-registered files).

The ATLAS DDM model considers a Tier-1 facility as a single entity and requires that it provides a single entry point. This is quite natural, as each of the EGEE and OSG based Tier-1 facilities is situated at a large data center. On the other hand, while the Nordic countries together have more than enough resources and a solid data infrastructure, no single data center can provide the required computing and storage resources to host a Tier-1 facility. NDGF is a meta-centre connecting several existing scientific computing resources and Grid facilities in the Nordic countries. One of the tasks of NDGF is to host a Nordic Tier-1 center and manage the distribution of data and computing tasks between various computing centers in the Nordic countries [55].

The NDGF Tier-1 builds on the ARC middleware which provides the fundamental grid services, such as information services, resource discovery and monitoring, job submission and management, brokering and data management and resource management. All these services run on the frontends of the different resources or, in the case of the brokering, on the client machines. However, the ARC middleware does not yet provide a production-ready, distributed storage solution scalable to the levels required by a Tier-1 facility. Instead, NDGF manages a distributed dCache [56] installation, where the physical data are stored in storage pools distributed between computing centers in the Nordic countries (e.g., at the IT-center at the University of Oslo (USIT)) while the indexing service and entry point are located at central machines in Örestaden near Copenhagen. To avoid having all the traffic passing through a central endpoint the actual file transfer is redirected to go directly between the source point and the selected data pool.

5.2 Distributed Storage

An essential part of distributed data management is of course how to store the data. To create a distributed storage solution involves facilitating for data transfer and data access, in a secure manner, i.e., in a manner so that at any point in the lifetime of the data only users with read access can access entries and only users with write access can change the entries. In the same manner as one can distinguish between high-performance computing and high-throughput computing, one can distinguish between distributed storage for high performance and distributed storage for high throughput. While the first is designed to be deployed in a Local Area Network (LAN), inside the same firewall and with full control of the network connections, the latter is designed to work over the Wide Area Network (WAN) without control of the network connection, high network latency and high probability of failing connections. While it is quite common for high throughput storage to use high performance storage solutions as local storage backends[1], we will here focus on the high throughput variant of distributed storage, and the term *distributed storage* should be read as *distributed storage for high throughput computing* in the remainder of this chapter.

Due to the need for connecting over the WAN, distributed storage solutions need to be exposed to the Internet, both internally and externally. This poses several challenges:

- **Security:** When being confined to a single LAN, a malicious user will have to break into the LAN (or otherwise gain access to the LAN) to pick up data transfers. File transfers over WAN are exposed to any user and sensitive data need to be secured through encryption.

- **Access:** In a LAN environment there is a limited number of users, all of them accessing the system with a local username and password. In a WAN environment there is a much greater number of potential users and traditional access control methods are not designed to scale to huge numbers of users.

- **Fault tolerance:** While a LAN environment is not void of errors, the potential sources of errors increase in a WAN environment as more connection points need to be passed between the resources. Additionally, the infrastructure (switches, routers, cables, etc.) is not physically controlled by one system administrator. Hence, one should expect a

[1] For example, the dCache pool at the University of Oslo uses GPFS to physically store the data.

higher fault rate in a WAN environment, and needs to design the storage system accordingly.

- **Performance:** Where in a LAN environment latency can be down to microseconds and bandwidth can reach levels near internal harddrive performance, in a WAN environment it is not unusual with latency up to a second and bandwidth in the order of megabits per second. A flow of messages which performs very well in a LAN environment does not necessarily perform optimally in a WAN environment.

- **Scalability:** While having a storage system distributed over WAN drastically increases the possibility of storing huge amounts of data, it also increases the needed efforts in managing and monitoring the data. Existing solutions for managing and monitoring data in a LAN are most likely not designed to work in a distributed environment.

- **Metadata handling:** Accessing a storage system, be it for localizing files, downloading files or uploading files, requires accessing metadata. The metadata are stored in a database which can grow very large when the number of entries in the system increases. To achieve a reasonable performance in the system, this meta-database needs to be very efficient. To have a reliable system the meta-database needs to be consistent. Having the metadata on a single machine can provide consistency, but will limit the performance on database lookups, make a single point of failure and limit the scalability of the system. As the need to distribute a storage system is closely connected to the question of scalability, a major challenge in distributed storage is how to consistently and efficiently replicate the metadata over several machines.

All these challenges are quite demanding and their solutions depend, to a certain degree, on the intended use of the storage system at hand. For example, if the storage system is intended to be used for a large scale physics experiment, high performance and scalability may be prioritized while lowering the security demands. On the other hand, a multinational company in need of sharing their customer database between the local offices may set security much above performance. It is not given that a storage solution that is considered very good in one case will even be considered usable in the other case.

As a result of the diverging needs of potential users, many distributed storage solutions have emerged over the years, usually based on either grid or cloud ideas. While many of these solutions are highly specialized to solve specific tasks, some solutions are intended to be used for more generic data

storage. Examples of such solutions are dCache and DPM/LFC (see below), both used at ATLAS Tier-1 facilities, and Amazon Simple Storage Service (S3) which is a storage cloud guaranteeing "infinite" storage space on a pay-per-use basis.

dCache is a service-oriented storage system which combines heterogeneous storage elements to collect several hundreds of terabytes in a single namespace. Originally designed to work on a local area network, dCache has proven to be useful also in a Grid environment, with the distributed NDGF dCache installation as one large example[2]. While proved to be suitable for storing huge amounts of data, dCache suffers from rather complex installation and maintenance. The fact that all metadata need to be stored in a single LAN environment introduces a single point of failure.

By combining the Disk Pool Manager (DPM) and LFC[3] it is possible to set up a relatively lightweight storage system with distributed metadata by registering all files uploaded to any DPM to several LFC's. However, just like in the ATLAS DDM topology, DPM and LFC have no direct coupling and registration and replication of files are left to the client tool. If, e.g., a file is removed from a DPM without updating LFC, the namespace of the storage system becomes inconsistent and file lookup inefficient.

Amazon S3 is a generic cloud storage system where storage space is provided by Amazon, thus freeing the users from the efforts and costs of buying and maintaining storage hardware. While this offers great savings for a company in need of storage space, it also raises a security issue. Where in traditional grid storage solutions, both the users and services need to trust the same, third-party, certificate authority, the security model of S3 requires users to trust the storage provider entirely.

A large part of my work in this thesis has been devoted to address the above mentioned challenges and short-comings. The result, a light-weight, fault-tolerant and generic storage solution without single points of failures, and how the short-comings are addressed is described below, in Section 5.3.

5.3 Chelonia, a Self-healing Storage Cloud with Grid Capabilities

The basic components needed for creating a distributed storage system are the storage elements (SE's) where the data are stored, a protocol for trans-

[2]With disk pools in Norway, Sweden, Finland, Denmark and Slovenia.

[3]Note that LFC is used both as an ATLAS cloud-wide file catalog and as a file catalog for DPM on local sites.

ferring the data to the SE's and between the SE's, a database to keep track of where the data are physically stored and a structure in which the data can be organized.

Chelonia, being one of the main components in the KnowARC final release, is a file-based storage system with the files organized in collections in a global, hierarchical namespace. Chelonia is a self-healing storage system in the sense that all files can be replicated and Chelonia handles both file replication and discovery of broken and missing file replicas internally. Additionally, to avoid single points of failure, all the services can be replicated. While an overview of Chelonia is given in the papers in Appendices A.5 and A.6 the most important parts will be summarized here.

Chelonia consists of a set of services communicating through the HED message chain components (see Section 4.2.3). Figure 5.2 shows an overview of the Chelonia architecture. The architecture consists of four services, each with well defined tasks: The A-Hash (blue space ship) consistently stores the metadata (e.g. file location, size and time of creation) of each entry , the Librarian (orange book) manages the hierarchical namespace, the Shepherd (light blue staff) manages the SE and the Bartender (red cup) provides the interface to users and third-party services.

In Chelonia every entry (files, collections, mount points) is associated with a Global Unique ID (GUID) and a Logical Name (LN). These are managed by the Librarian and stored in the A-Hash. Whereas the collections and mount points are only logical entities, the files have physical data stored at the SE's as well. The location of a physical file is uniquely defined by combining the Uniform Resource Locator (URL) of the Shepherd and the local UID of the file. The locations of the physical files are stored in the metadata of the corresponding LN. Thus, every physical file is uniquely and consistently defined. The Shepherd manages the SE together with a transfer service (e.g., a HTTP(S) or GridFTP service) and a storage backend. By checking the files, first when they are uploaded to the transfer service and then periodically, the Shepherd will notice if files are inconsistent and report it to a Librarian. If there are valid replicas of the same file in the system, the Shepherd will delete the inconsistent file, and the Shepherds together will generate a new replica of the file. The Bartender service provides a high-level interface to the storage storage system, both for clients and for Shepherds, and makes the decision of which Shepherds will host the replicas of an incoming file.

Additionally, the Bartenders support gateways which provide the possibility of including third-party storage services or federated Chelonia clouds

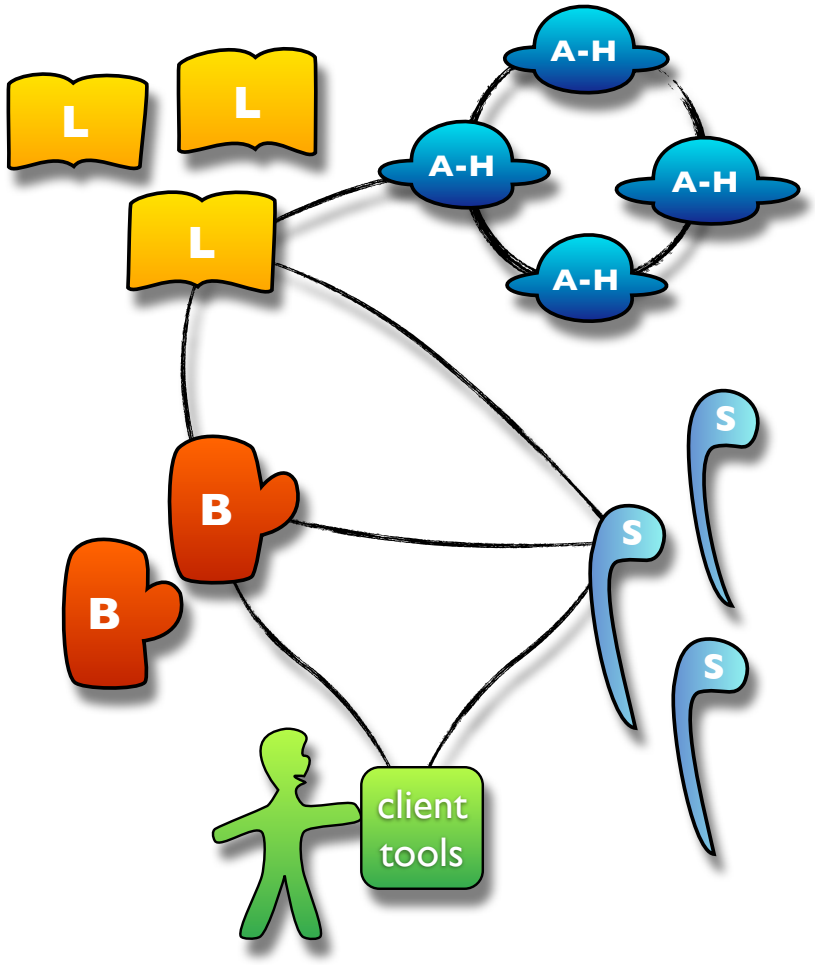


Figure 5.2: Schematic of the Chelonia architecture. The figure shows the main services of the Chelonia; The Bartender (cup), the Librarian (book), the A-Hash (space ship) and the Shepherd (staff). The communication channels are depicted by black lines. (Taken from "Chelonia - distributed cloud storage", Appendix A.6.)

in the global namespace[4]. The gateway feature is useful in several scenarios. One scenario appears when introducing Chelonia to a new community. Some of the users may already have data stored in an existing storage solution. With a gateway they can access these data through the Chelonia namespace. Another scenario is related to performance. In Linux operating systems it is quite common to partition the hard drive with some partitions optimized for many small files (e.g., user home areas) and some partitions optimized for larger files (e.g., for multimedia files). This scenario is directly transferable to Chelonia, where different policies on e.g. file access and file replication can be configured according to the expected usage of the partition.

The A-Hash is a vital part of Chelonia, as this is where the entire state of Chelonia is stored. A metadatabase which is replicated by design is rather uncommon in generic storage systems and deserves extra attention. The A-Hash is replicated using the Oracle Berkeley DB [57] (BDB), an open-source database library wih a replication API. The replication is based on a single master, multiple clients framework where all clients can read from the database and only the master can write to the database.

While a single master ensures that the database is consistent at all times, it raises the problem of having a single point of failure in the master. If the master is unavailable, the database cannot be updated, files and entries cannot be added to Chelonia and file replication will stop working. The possibility of a failing master cannot be completely avoided and to ensure high availability, means must be taken to find a new master if the first master becomes unavailable. BDB uses a variant of the Paxos algorithm [58] to elect a master amongst peer clients: Every database update is assigned an increasing number. In the event of a master going offline, the clients send a request for election, and a new master is elected amongst the clients with the highest numbered database update.

While the automatic election of a master between the A-Hashes reduces the time of unavailability, it does also make it impossible to know *a priori* which A-Hash is the master. For the Librarians to discover which A-Hash to write to and which to read from, the master A-Hash maintains a list of URL's to the currently available master and clients. The list is stored in the A-Hashes in a system entry which is readable for every Librarian, as shown in Figure 5.3. As long as a Librarian knows about and has read access to at least one A-Hash during start-up, it will download the list and set up a connection to the master and one client. The Librarian refreshes this list both periodically and in the event of a failing connection.

BDB comes with a replication manager taking care of replicas, election

[4]Gateways can be compared to mount points in UNIX operating systems.

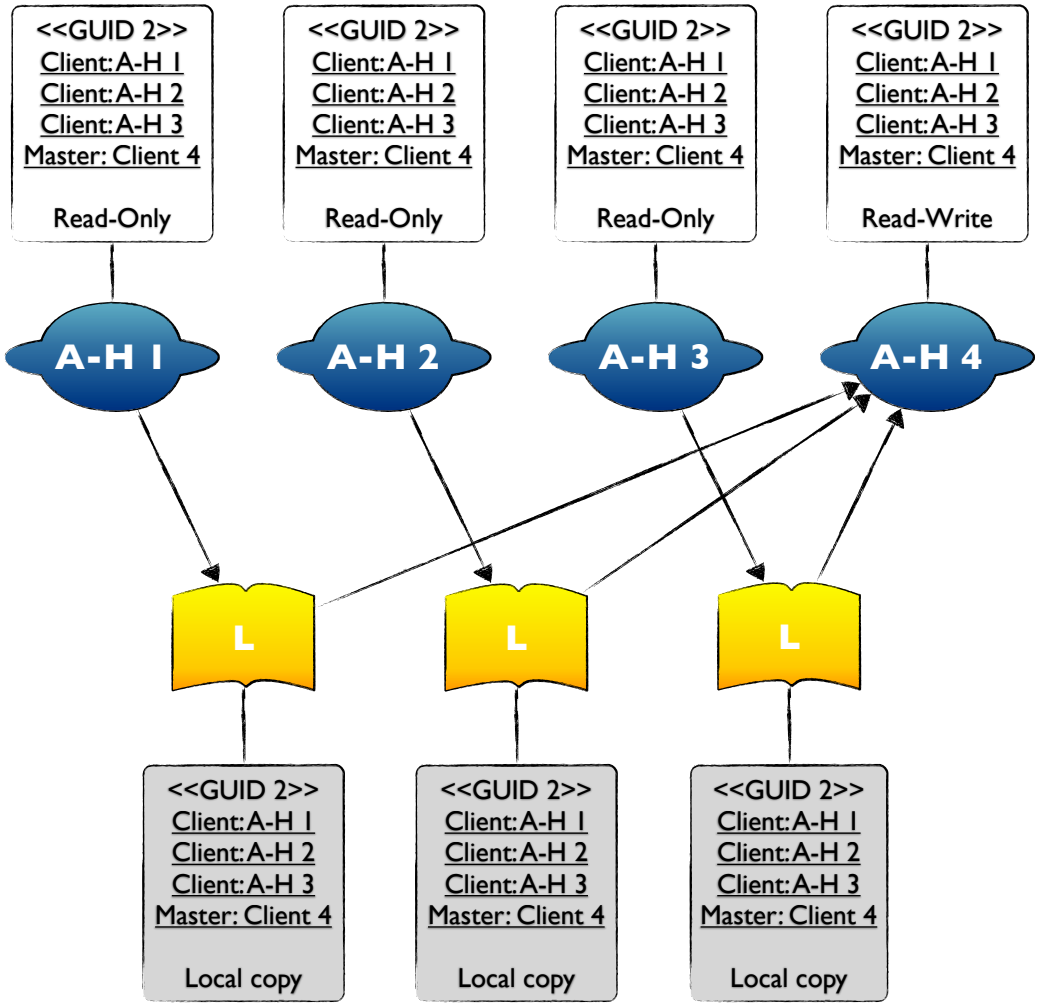


Figure 5.3: Relationship between A-Hash master and clients and Librarian. All Librarians write to the master (A-Hash 4) and read from any of the clients (A-Hashes 1, 2 and 3). The list of client and master URL's (GUID 2) are maintained by the master A-Hash and replicated to the client A-Hashes. Additionally, the Librarians have local copies of the list.

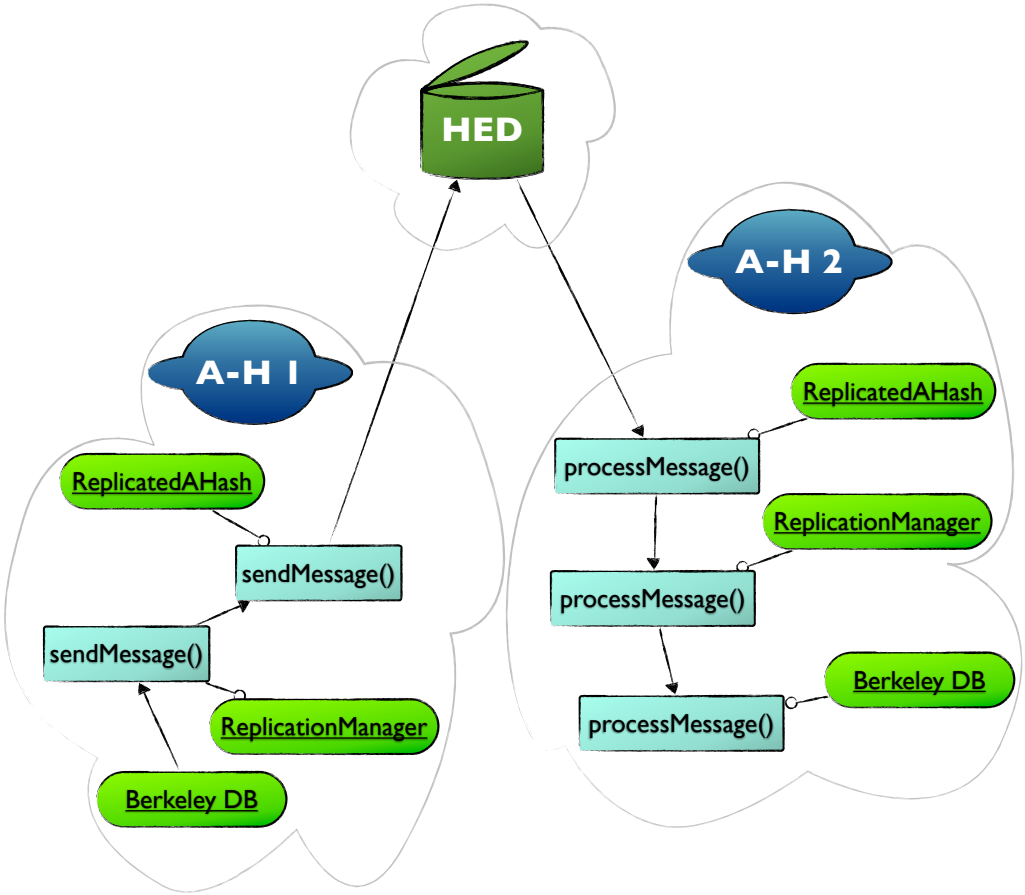


Figure 5.4: Message flow of a replication message (arrow) between two A-Hashes.

of master and replication messages. Unfortunately, this manager relies on a communication framework for sending messages using unsecured TCP messages and is thus unsuitable when exposing the A-Hashes to a WAN environment. On the other hand, the ARC HED provides a secure communication framework designed for a grid environment. For the A-Hash to be able to use the replication framework of BDB, BDB requires a call-back function for sending messages and a function for processing messages and forwarding them to BDB. Figure 5.4 shows how a replication message is sent from BDB at A-Hash 1 to BDB at A-Hash 2. When BDB sends a replication message, the message is passed to the business logic class `ReplicatedAHash` of A-Hash 1, converted to a SOAP message and sent through HED resulting in a call to the method `processMessage()` in the business logic class of A-Hash 2. This invokes `processMessage()` of the A-Hash 2 replication manager, which in turn sends the message to the `process()` function of BDB.

While storing data safely in a distributed fashion in a self-healing system without single points of failure may be important in itself, a user of the system may find the accessibility of the system even more important. To provide access to files, file systems must provide an Application Programming Interface (API). While the implementation of the API depends on the file system, newer file systems aim towards following the standard API set by the Portable Operating System Interface [for UNIX] [59] (POSIX) standard. Examples of system interfaces for file operations are `creat`[5] and `remove` for creating and removing a file, `fopen` and `fclose` for opening and closing files, and `fread` and `fwrite` for reading and writing files.

In addition to a command line interface and a specialized ARC protocol supported by the HED data management component ARC DMC, Chelonia provides a module for mounting the Chelonia namespace as a local directory on the user's local machine. The File System in Userland [60] (FUSE) provides an interface to the POSIX commands and makes it possible to translate the interface of the Bartender to a near-POSIX interface, thus enabling the possibility of mounting the Chelonia namespace into the namespace of almost any operating system supporting POSIX file systems (e.g. Mac OS X, Linux and other UNIX-like systems). In this way, features like graphical file browsers and drag-and-drop functionality to upload and download files are provided by the operating system on the user's machine, as exemplified by the screen shot from a Mac OS X computer in Figure 5.5.

Chelonia is a storage system intended to be lightweight and easy to in-

[5]Ken Thompson, winner of the 1983 Turing award for his work on designing UNIX, was once asked what he would have done differently. His reply: "I'd spell creat with an e."

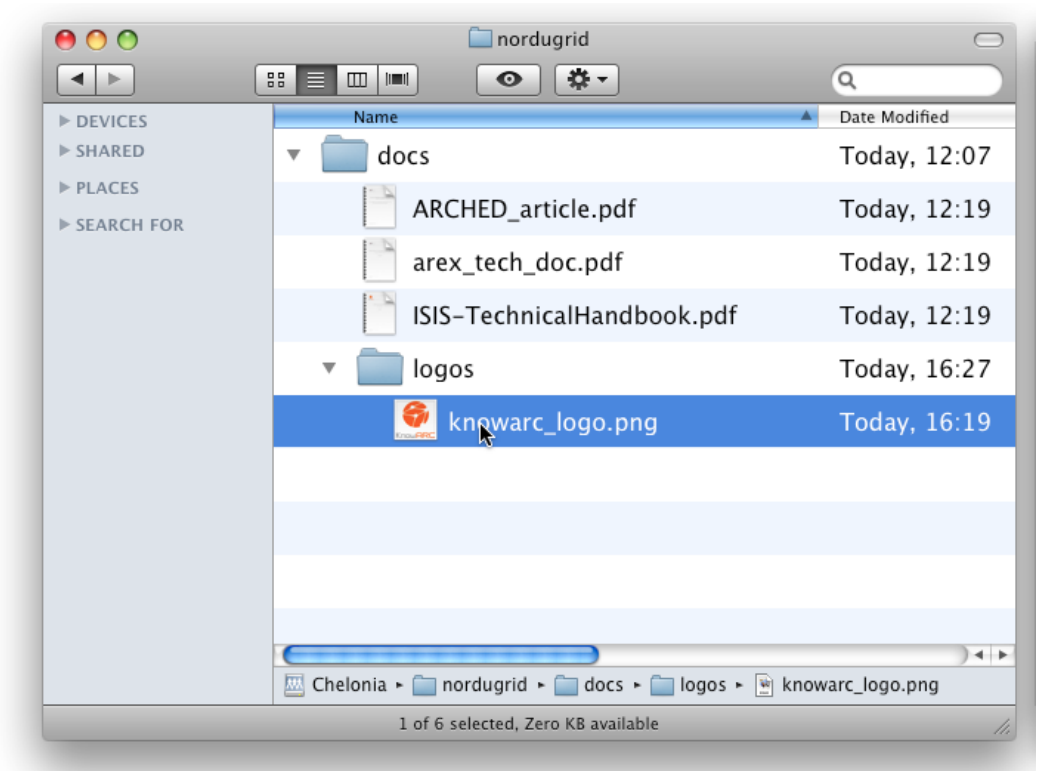


Figure 5.5: Screenshot of the Chelonia FUSE module in use. Through the FUSE module Chelonia offers user a drag and drop functionality to upload or download files to the storage cloud. (Taken from "Chelonia - distributed cloud storage", Appendix A.6.)

stall and use. It shares features with both traditional storage solutions and distributed data management tools. Being a generic storage system, it can be used in scenarios not at all envisioned when designing it. One such scenario is described in the following chapter, Chapter 6, where Chelonia plays a central role in a grid-based parallel computing framework.

Part III

Putting the Parts Together

Chapter 6

Parallel Monte Carlo – High Performance or High Throughput?

The idea of running Monte Carlo in parallel is far from new. As stated by Metropolis and Ulam on the Monte Carlo method [11]:

> *The Markov chain procedure itself is serial, and in general one does not shorten the time required for a solution of the problem by the use of more than one computer. On the other hand, the statistical methods can be applied by many computers working in parallel and independently.*

The most standard way of parallelizing a computing task in High Performance Computing (HPC) is by incorporating the Message Passing Interface (MPI). In many cases this can be done almost automatically, as explained in the article in Appendix A.3. MPI provides a way to split a task into smaller sub-tasks, which communicate by passing messages between each other. Even on a Local Area Network (LAN) with low latency and high bandwidth, the communication between sub-tasks is associated with lowered performance. On a Wide Area Network (WAN), where high latency and often low and unstable bandwidth is common, the cost of communication between subtasks can soon make the parallelized computing task slower than the serial version of the same task, thus making the utility of the combined resources significantly worse than that of the sum of its parts.

As mentioned in the introduction of Chapter 4, computing tasks involving partial differential equations are examples of tasks that require a high degree of communication and hence belong to the HPC category of computing tasks.

It was also mentioned that most Monte Carlo (MC) methods belong the High Throughput Computing (HTC) category, where resources are connected through the WAN. In most MC methods the computing tasks can be split into many independent sub-tasks with no need for communication during the computation.

However, as always there are exceptions which do not properly fit into either of the two categories. Diffusion Monte Carlo (DMC) uses a set of random walkers to simulate the ground state of a quantum mechanical system. The simulation is time-dependent, and in each time-step the observables of the system need to be updated to be used in the next time-step. The precision of the simulation increases with the number of walkers, and a natural parallelization of DMC is to split the set of walkers into walker subsets. This requires only communicating the observables of the system at the end of each time-step. Additionally, in the DMC algorithm the number of walkers is dynamic (see Section 3.3.3), thus raising a need to occasionally redistribute the walkers between the sub-tasks. Depending on the size of the simulated system and the number of walkers, DMC can fit into either of the HPC and HTC categories.

This observation raises (at least) one question: Is it feasible to run a parallelized version of DMC using the widely distributed resources of a grid? Restricting the problem to MPI, the answer is most likely negative due to latency and security issues related to passing messages over WAN. However, using some alternative means of communication, is it possible to run a large scale DMC computation?

6.1 GaMPI Architecture

The article in Appendix A.7 explores this question by means of a pull-based compute model based on the ARC middleware, Chelonia and the computational task-management tool Ganga [61]. Similar to a common parallelization method in MPI programming, the presented framework (GaMPI) uses a single master/multiple slaves method to divide the work. While the master process takes care of splitting the task into sub-tasks and updating the tasks after each time-step, the actual work is carried out by the slaves. However, where in MPI both the master and slaves can communicate directly through well established connections, in GaMPI the master and the slaves have no way of communicating directly with each other; while the slaves are running on the grid, the master resides on the grid user's local machine.

The workflow of GaMPI can be summarized as follows:

1. The master is running inside Ganga, which creates, submits and moni-

tor the grid jobs. After initializing the submission of the grid jobs (the slaves) the master leaves the management of the grid jobs to Ganga.

2. The master splits the required number of walkers into blocks. The number of blocks is independent of the number of slaves. The walker blocks are uploaded, named with the prefix `walker_block`, to a *walker pool* in Chelonia. Additionally, the master uploads an empty file named `do_timestep`.

3. The slaves checks if the file `do_timestep` exists and, if it does, try to download a block of walkers, chosen randomly from the files with prefix `walker_block`. When all the walkers in the block are moved and diffusion and branching is done on the walker block, the slave uploads the walker block with the prefix `moved_walker_block`.

4. The main variables of interest after a time step is the number of walkers (which is dynamic due to the branching part of DMC) and the ground state energy after moving the walkers. Chelonia supports arbitrary metadata for a file, enabling these two variables to be appended to the metadata of the walker block file. This way, the master can simply do a `stat` on the walker block files to calculate the global energy and number of walkers, without actually downloading all the walkers.

5. The master monitors the walker pool, and as soon as there are only moved walker blocks in the walker pool, the master removes the file named `do_timestep`, thus telling the slaves that they can poll the walker pool less frequently.

6. The master then checks the metadata of the files to get the ground state energy and the number of walkers, thus avoiding a full download of all walker blocks. The branching term of DMC (Equation 3.27, repeated here for clarity),

$$G_B = \exp\{-((V(\mathbf{R}) + V(\mathbf{R}')/2 - E_T)\tau\}, \tag{6.1}$$

involves a trial energy, E_T, which in this case is a best estimate of the ground state energy. The best estimate of the ground state energy is the mean of the energy per walker. For every time step the master acquires this mean from the walker blocks and refreshes the energy in the metadata. When the energy is updated, the master renames the walker blocks with prefix `walker_block` and again uploads the `do_timestep` file.

7. After repeating steps (1) to (5) for the required number of time steps, the master uploads the file `stop`. As soon as the slaves see the `stop` file, they end their work and exit.

To avoid that more than one slave works on the same walker block, the slaves will try to rename the walker block to have the prefix `.walker_block` before downloading it. In Chelonia, renaming is an atomic operation and it is not allowed to overwrite an already existing file. Hence, only the first slave to rename the walker block will get the walkers, thus limiting data transfer and avoiding multiple slaves working on the same task. It should be mentioned that a way to further reduce the amount of network load could be to introduce caching on the slaves where each slave maintained a local walker pool, and tried to reserve these files first. However, GaMPI being in a proof-of-concept state at the moment, caching is not yet implemented and the gain remains to be seen.

After a number of time steps, the walker blocks may be unbalanced. The master, knowing the number of walkers per walker block, checks after each time step if any walker block is more than 50% above or below the mean of the block-sizes. If so, the master downloads the walker blocks and iterates through them, taking as much from the largest walker block as needed for the smallest walker block to be at the mean of walker blocks. To avoid superfluous file transfers the iteration stops as soon as no walker blocks are above or below 10% of the mean.

Figure 6.1 shows an example of the communication flow between the components of GaMPI. Here, Ganga and the grid jobs communicate by exchanging files through Chelonia. Chelonia itself is set up with two sets of Bartender-Librarian-Shepherd and a ring of three A-Hash replicas. In this setup, any of the two Bartenders can be used and will yield the same result on queries, and uploading a file, the file can end up on any of the two Shepherds. However, each of the Shepherds contacts only one Bartender and one Librarian. The reason for such a setup is that a Shepherd will reuse the connection to the first Bartender/Librarian it successfully contacts. With only two instances of each service, there is a great chance of both Shepherds constantly communicating with the same Librarian and/or Bartender. Thus, to balance the load between the services, each Shepherd gets assigned one Librarian and Bartender. This means that if one Librarian or one Bartender stops, one of the Shepherds will stop. While this is not optimal in a production setup, it may make sense in a simple setup where one set of Shepherd, Librarian and Bartender is run on the same machine, as is the case for the results shown below.

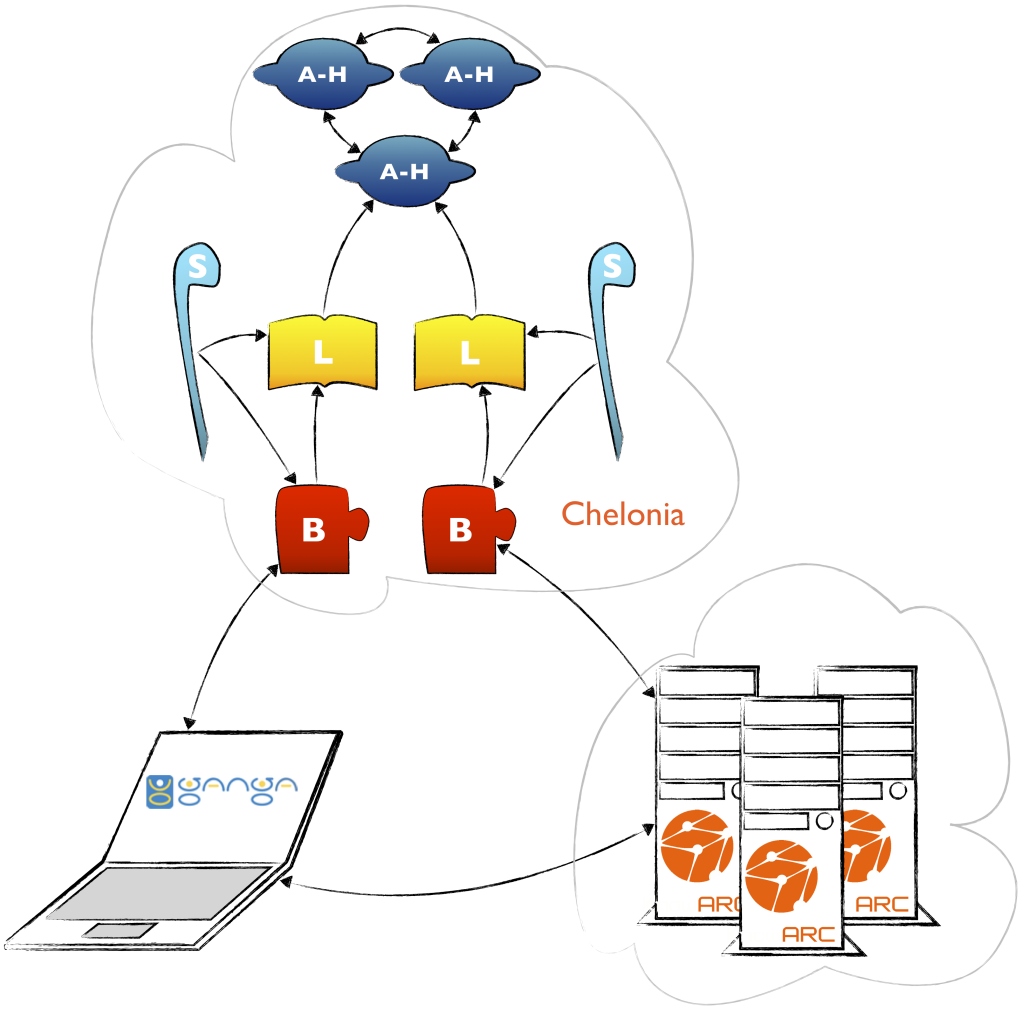


Figure 6.1: The figure shows an example of the communication flow between Ganga (laptop), Chelonia (top cloud) and ARC (bottom cloud), and between the services internally in Chelonia. Communication between Ganga and the grid is carried out by exchanging files through the Chelonia cloud. Internally, Chelonia is set up with two sets of Bartender-Librarian-Shepherd and a ring of three A-Hash replicas. (Taken from "Parallel Monte Carlo simulations on ARC connected grid resources, using Chelonia storage and GANGA job interface", Appendix A.7.)

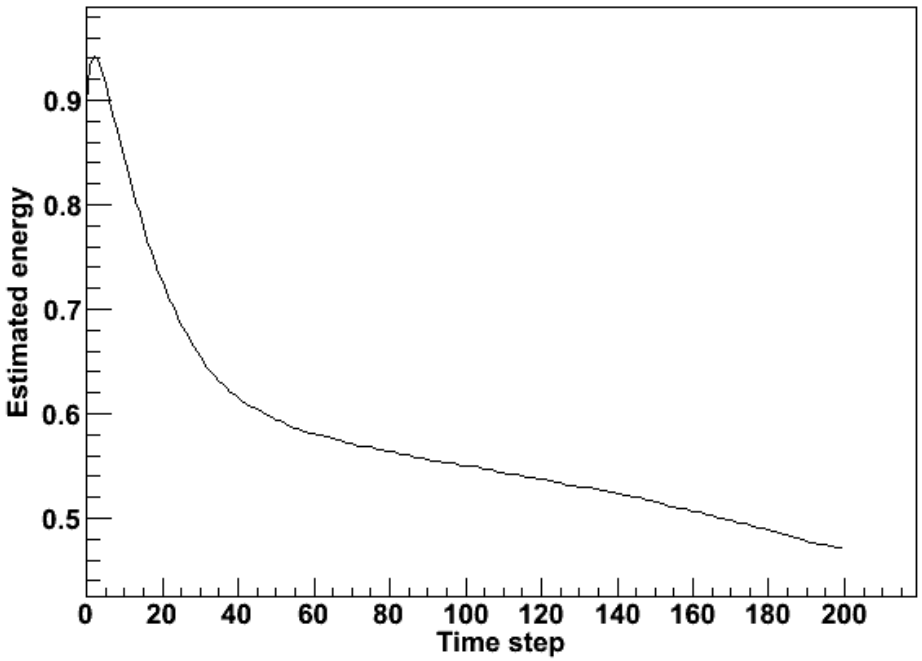


Figure 6.2: Trial energy as a function of time step.

6.2 Results

As can be seen in Equation 6.1 the trial energy plays an important role in the evolution of the number of walkers in diffusion Monte Carlo. If the trial energy is too high, the branching ratio will be too large and the number of walkers will grow very fast. If the trial energy is too low the number of walkers will fall to zero in only a few iterations. In fact the trial energy is strongly correlated to the number of walkers. An estimate of the trial energy is given by [24],

$$E_T = E_0 - \frac{1}{2\tau} \sum_i \ln \left(\frac{s_i}{s_{i-1}} \right), \tag{6.2}$$

where E_0 is an initial guess of the trial energy, τ is the time step length and s_i is the number of walkers at time step i. This trial energy may be updated at each time step. If the number of walkers is larger than in the previous step the trial energy is decreased, and if the number of walkers decreases, the trial energy increase. Thus, feeding Equation 6.2 into Equation 6.1, the trial energy decreases the variation in the number of walkers.

Figure 6.2 shows the trial energy as a function of time steps. E_0 was set

Time per iteration for 30000 walkers, 16 CPUs				
Case	Description	*Minimum (s)*	*Average (s)*	*Maximum (s)*
a	MPI, single cluster	735	790	838
b	GaMPI, single cluster	545	778	1191
c	GaMPI, three clusters	738	885	1157

Table 6.1: Timings per iteration (time-step) for running 50 iteration of diffusion Monte Carlo using regular MPI on a single cluster, using GaMPI on the same cluster, and running GaMPI with grid jobs distributed between three clusters in three different countries. In all runs 16 CPUs were used in parallel. Note that the timings does not take into account the number of walkers in each time-step. Values are for timesteps 30 — 50. (Taken from "Parallel Monte Carlo simulations on ARC connected grid resources, using Chelonia storage and GANGA job interface", Appendix .)

to 0.8. Initially, the walkers, each consisting of 100 particles, were distributed randomly with a gaussian distribution in three dimensions with mean 0 and standard deviation 1. The walkers were then moved in 200 time steps in an external potential $V_{ext}(\mathbf{r}) = \mathbf{r}^2$ corresponding to a spherical harmonic oscillator (similar to the system described in Chapter 2). As the particles initially are distributed with a rather high potential, the energy is higher than E_0. As the particles, due to the potential, move closer to the center of the trap, the energy decreases as expected, first rapidly then more gradually. It should be noted that while the trial energy estimate of Equation 6.2 serves as a good example of stabilizing the number of walkers, it is not optimal for giving an estimate of the actual energy of the system. To obtain a precise estimate of the energy, the importance sampling described in Section 3.3.3 needs to be introduced, thus complicating the diffusion Monte Carlo implementation beyond what is needed for testing the GaMPI framework.

Table 6.1 shows the timings for a simple diffusion Monte Carlo implementation run in three different environments. All three simulations used 16 CPUs and the same input parameters. The first two columns show the timings per time step when running on the same cluster in the same machine room as the Chelonia machines. While the timings in the first row is with running with MPI using the framework described in Appendix A.3, the second row is when using the GaMPI framework. The third row shows timings of simulations using the GaMPI framework, now with grid jobs distributed between three clusters in Norway, Sweden and Slovenia.

The perhaps most surprising in these results is that the single-cluster GaMPI simulation is actually slightly faster than the MPI simulation. All of the timings are wall-time timings, i.e., taking the time from one time step

ends to the next time step ends. For the GaMPI simulations this involves getting and updating the metadata of the walker blocks and, if the load is not balanced, downloading and uploading entire walker blocks, thus introducing an extra layer of file I/O and data transfer over WAN when compared to the MPI simulation. The differences in performance are not very large and can be explained by the stochastic nature of the number of walkers and the fact that the simulations were run in heterogeneous environments where both hardware and simultaneously running simulations can influence the performance. It does however show that a high throughput computation may run just as fast as a high performance computation.

Chapter 7

Conclusions and Outlooks

This thesis has been equally divided into two parts, with one part rooted in physics, studying efficient parallel algorithms for exploring Bose-Einstein condensates, and one part rooted in computational science, working with distributed computing and ways to improve distributed data management. While being two seemingly unrelated topics, both have given motivation and inspiration for the other and, in the end, the two topics play vital roles in the same article.

The Monte Carlo algorithm was presented and formed the basis for several of the presented publications. Being an *ab initio* method with individual treatment of particles, quantum Monte Carlo serves well as a "benchmark" algorithm for which to compare faster, but approximate mean-field algorithms. Being a relatively straight-forward method to implement, quantum Monte Carlo also serves well as an example implementation to present ideas on efficient programming methods and implementations, as was shown with the mixed-language MontePython implementation.

An overview of distributed technologies, specifically the grid and cloud technologies, and their use in a real-world physics experiment was given. The ATLAS experiment at LHC is one of the largest physics experiments ever built, and the need for computing and storage resources is far greater than what can be managed by one site. This raised the need for distributed technologies such as grids and clouds and led to middleware solutions such as the Advanced Resource Connector, a grid middleware for which the candidate has contributed significantly to development as part of the thesis.

A main part of this thesis was related to distributed data management and the main result was the generic storage solution Chelonia for which the candidate gave several important contributions, specifically the replicated database for metadata storage and the FUSE module for mounting Chelonia on the user's desktop.

Large scale computing problems are often split into two categories; high-performance computing (HPC) and high-throughput computing (HTC). While many problems clearly fall into one of the categories, it was shown that diffusion Monte Carlo is more of a borderline case, making it an ideal candidate for comparing the categories. Specifically MontePython was used to run on a HPC cluster using MPI, while GaMPI was used for HTC by running on an ARC grid. Interesting here was that the diffusion Monte Carlo implementation was identical in both cases, and only the communication protocols were different. The main result was that the run times where surprisingly similar, even though the interprocess communication was expected to be far slower for in the HTC case.

GaMPI is currently at the state of a prototype, with lots of room for improvements and adjustments, and the framework is only tested with a simple diffusion Monte Carlo implementation. Nevertheless, the idea of a parallel framework for grid computing seems promising and a natural way forward is to include and test more use-cases. One potential use-case is the Markov-chain Monte Carlo implementation for roll-call voting described in Appendix A.3. This is interesting both for attracting social scientists to grid computing and because of it's dependency of the programming language R, making it a mixed-language application putting more demands on software installed on the grid.

GaMPI has proven to be a good test-case for Chelonia, as it is quite different from the usecases originally considered when Chelonia was designed. The GaMPI simulations have revealed several weaknesses in the current Chelonia implementation and the underlying ARC middleware. Perhaps most noteworthy are the problems occuring when more than 64 users (or GaMPI processes) access Chelonia at the same time. While these weaknesses may require some work, it is very important to sort out as many of them as possible before deploying Chelonia for new users. In this respect, GaMPI can serve as a very good substitute for many (impatient) users.

Chelonia has so far been implemented under the KnowARC umbrella. As the KnowARC project is now ended, new projects are coming where Chelonia could be developed and hardened further. The perhaps most promising project is the European Middleware Initiative (EMI) in the EU Framework Programme 7. Here, the three major European grid middlewares ARC, gLite and UNICORE will join forces to create one European middleware package. Chelonia is one of three storage solutions mentioned in the EMI proposal. While nothing is settled yet, there is a strong hope that Chelonia will continue to be developed and hardened towards being a production-ready storage system, be it under the EMI umbrella or as part of other future projects.

Bibliography

[1] J.R. Anderson, M.R. Ensher, M.R. Matthews, C.E. Wieman, and E.A. Cornel. Observation of Bose-Einstein condensation in a dilute atomic vapor. *Science*, 269:198, 1995.

[2] K. B. Davis, M. O. Mewes, M. R. Andrews, N. J. van Druten, D. S. Durfee, D. M. Kurn, and W. Ketterle. Bose-einstein condensation in a gas of sodium atoms. *Phys. Rev. Lett.*, 75(22):3969–3973, Nov 1995.

[3] C. C. Bradley, C. A. Sackett, J. J. Tollett, and R. G. Hulet. Evidence of bose-einstein condensation in an atomic gas with attractive interactions. *Phys. Rev. Lett.*, 75(9):1687–1690, Aug 1995.

[4] F. Dalfovo, S. Giorgini, L. P. Pitaevskii, and S Stringari. Theory of bose-einstein condensation in trapped gases. *Rev. Mod. Phys.*, 71:463, 1999.

[5] N.N. Bogoliubov. On the theory of superfluidity. *J. Phys. (USSR)*, 11:23, 1947.

[6] E.P. Gross. Structure of a quantized vortex in boson systems. *Nuovo Cimento*, 20:454, 1961.

[7] L.P. Pitaevskii. Vortex lines in an imperfect bose gas. *Sov. Phys. JETP*, 13:451, 1961.

[8] D. A. W. Hutchinson, E. Zaremba, and A. Griffin. Finite temperature excitations of a trapped bose gas. *Phys. Rev. Lett.*, 78(10):1842–1845, Mar 1997.

[9] S. Ghosh. Vortices in atomic Bose-Einstein condensates: an introduction. *Phys. Rev. A*, 2004.

[10] Guido van Rossum et al. The Python programming language. `http://www.python.org/`, 1991–. Web site.

[11] N. Metropolis and S Ulamm. The monte carlo method. *J. of the American Statistical Association*, 44(247):335–341, 1949.

[12] D. Creal. A survey of sequential monte carlo methods for economics and finance. Serie Research Memoranda 0018, VU University Amsterdam, Faculty of Economics, Business Administration and Econometrics, 2009.

[13] Achim Zeileis, Christian Kleiber, and Simon Jackman. Regression models for count data in R. *Journal of Statistical Software*, 27(8), 2008.

[14] Stefan Weinzierl. Introduction to monte carlo methods. arXiv:hep-ph/0006269, 2000.

[15] M. Mascagni and A. Srinivasan. Sprng: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software*, 26:436–461, 2000.

[16] A. Srinivasan, M. Mascagni, and D. Ceperley. Testing parallel random number generators. *Parallel Computing*, 29:69–94, 2003.

[17] D.E. Knuth. *The Art of Computer Programming, Second Edition*. Addison-Wesley Publishing Company, 1981.

[18] G. Corcella, I. G. Knowles, G. Marchesini, S. Moretti, K. Odagiri, P. Richardson, and B. R. Webber. Herwig 6: an event generator for hadron emission reactions with interfering gluons (including supersymmetric processes).

[19] Torbjorn Sjostrand, Stephen Mrenna, and Peter Skands. Pythia 6.4 physics and manual, May 2006.

[20] P. R. C. Kent. *Techniques and Applications of Quantum Monte Carlo*. PhD thesis, (Robinson College, Cambridge, 1999).

[21] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. M. Teller, and E. Teller. Equations of state calculations by fast computing machines. *J. Chem. Phys.*, 21:1087, 1953.

[22] W. K. HASTINGS. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[23] R. Guardiola. Monte Carlo methods in quantum many-body theories. In J. Navarro and A. Polls, editors, *Microscopic Quantum Many-Body Theories and Their Applications*, volume 510 of *Lecture Notes in Physics*, pages 269–336. Springer Verlag, 1998.

[24] B.L. Hammond, W.A. Lester Jr, and P.J. Reynolds. *Monte Carlo Methods in Ab Initio Quantum Chemistry*. World Scientific, 1994.

[25] P. J. Reynolds, D. M. Ceperley, B. J. Alder, and W. A. Lester Jr. Fixed-node quantum Monte Carlo for molecules. *J. Chem. Phys.*, 77:5593–5603, 1982.

[26] A. Sarsa, J. Boronat, and J. Casulleras. Quadratic diffusion Monte Carlo and pure estimators for atoms. *J. Chem. Phys.*, 116:5956–5962, 2002.

[27] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. `http://www.scipy.org/`, 2001–. Web site.

[28] Pearu Peterson. F2py: a tool for connecting fortran and python programs. *Int. J. Comput. Sci. Eng.*, 4(4):296–305, 2009.

[29] David Beazley et al. SWIG: Simplified wrapper and interface generator. `http://www.swig.org`, 1995–. Web site.

[30] Pearu Peterson. PyVTK: Tools for manipulating vtk files in python. `http://cens.ioc.ee/projects/pyvtk/`, 2001–. Web site.

[31] C. Amsler et al. (Particle Data Group). Review of Particle Physics. *Physics Letters*, B667:1+, 2008.

[32] CERN video productions and CERN video productions. Video news release: Lhc energy record. lhc sets new world record. Nov 2009.

[33] *ATLAS computing: Technical Design Report*. Technical Design Report ATLAS. CERN, Geneva, 2005. revised version submitted on 2005-06-20 16:33:46.

[34] Enabling Grids for E-sciencE. `"http://www.eu-egee.org/"`. Web site.

[35] gLite, Lightweight Middleware for Grid Computing. `"http://glite.web.cern.ch/glite/"`. Web site.

[36] Open Science Grid. `"http://www.opensciencegrid.org/"`. Web site.

[37] Virtual Data Toolkit. `"http://vdt.cs.wisc.edu/"`. Web site.

[38] NorduGrid Collaboration. `"http//:www.nordugrid.org/"`. Web site.

[39] M. Ellert et al. Advanced Resource Connector middleware for lightweight computational Grids. *Future Gener. Comput. Syst.*, 23(1):219–240, 2007.

[40] L. Kleinrock. Ucla to be first station in nationwide computer network. Technical report, UCLA Press Release, 1969.

[41] V. Cerf, Y. Dalal, and C. Sunshine. Specification of internet transmission control program, 1974.

[42] Requirements for internet hosts - communication layers, 1989.

[43] Tim Berners-Lee. Information management: A proposal. Technical report, CERN, March 1989.

[44] Tim O'Reilly. What is web 2.0? design patterns and business models for the next generation of software. www.oreilly.com, September 2005.

[45] Miguel L. Bote-lorenzo, Yannis A. Dimitriadis, and Eduardo Gomez-Sanchez. Grid characteristics and uses: a grid definition. In *Across Grids 2003, LNCS 2970*, pages 291–298, 2003.

[46] Heinz Stockinger. Defining the grid: a snapshot on the current view. *J. Supercomput.*, 42(1):3–17, October 2007.

[47] Heba Kurdi, Maozhen Li, and Hamed A. Raweshidy. A classification of emerging and traditional grid systems. *IEEE Distributed Systems Online*, 9(3):1, 2008.

[48] Ian Foster. What is the grid? - a three point checklist. *GRIDtoday*, 1(6), July 2002.

[49] Gartner. Gartner's 2009 hype cycle special report evaluates maturity of 1,650 technologies. August 2009.

[50] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

[51] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[52] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[53] EU KnowARC project. `"http://www.knowarc.eu/"`. Web site.

[54] Mario Lassnig, Miguel Branco, David Cameron, Benjamin Gaidioz, Vincent Garonne, Birger Koblitz, Massimo Lamanna, Ricardo Rocha, and Pedro Salgado. Managing ATLAS data on a petabyte-scale with DQ2. In *Journal of Physics: Conference Series*, Bristol, England, September 2008. Institute of Physics Publishing.

[55] G Behrmann, D Cameron, M Ellert, J Kleist, and A Taga. Atlas ddm integration in arc. *Journal of Physics: Conference Series*, 119(6):062015 (8pp), 2008.

[56] M de Riese, P Fuhrmann, T Mkrtchyan, M Ernst, A Kulyavtsev, V Podstavkov, M Radicke, N Sharma, D Litvintsev, T Perelmutov, and T Hesselroth. *dCache Book*.

[57] Oracle Berkeley DB. `http://www.oracle.com/technology/products/berkeley-db/index.html`. Web site.

[58] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[59] Standard for information technology - portable operating system interface (posix). shell and utilities. Technical report, 2004.

[60] Filesystem in Userspace. `http://fuse.sourceforge.net/`. Web site.

[61] J.T. Moscicki, F. Brochu, J. Ebke, U. Egede, J. Elmsheuser, K. Harrison, R.W.L. Jones, H.C. Lee, D. Liko, A. Maier, A. Muraru, G.N. Patrick, K. Pajchel, W. Reece, B.H. Samset, M.W. Slater, A. Soroko, C.L. Tan, D.C. van der Ster, and M. Williams. Ganga: A tool for computational-task management and easy access to grid resources. *Computer Physics Communications*, 180(11):2303 – 2316, 2009.

Appendix A

Collection of publications

A.1 Vortices in atomic Bose-Einstein condensates in the large-gas-parameter region

Article published in Physical Review A 71, 053610 (2005).

The article "Vortices in atomic Bose-Einstein condensates in the large-gas-parameter region" was a product of a collaboration with a group at Universitat de Barcelona.

The main idea with the article was to explore the dilute limits where the mean-field equations of Gross-Pitaevskii (GP) and Modified Gross-Pitaevskii (MGP) are still valid. The variational Monte Carlo (VMC) application I developed as part of my Master thesis, being an *ab inito* method, was ideal to use as a "real" physics experiment for which to compare the mean-field approximations.

My contributions to this article was to implement the VMC application, run the VMC simulations and get the results needed for comparing with the mean-field results.

Vortices in atomic Bose-Einstein condensates in the large-gas-parameter region

J. K. Nilsen,[1] J. Mur-Petit,[2] M. Guilleumas,[2] M. Hjorth-Jensen,[1,3,4,5] and A. Polls[2]
[1]*Department of Physics, University of Oslo, N-0316 Oslo, Norway*
[2]*Departament d'Estructura i Constituents de la Matèria, Universitat de Barcelona, E-08028 Barcelona*
[3]*Center of Mathematics for Applications, University of Oslo, N-0316 Oslo, Norway*
[4]*PH Division, CERN, CH-1211 Geneve 23, Switzerland*
[5]*Department of Physics and Astronomy, Michigan State University, East Lansing, Michigan 48824, USA*
(Received 10 December 2004; published 19 May 2005)

In this work we compare the results of the Gross-Pitaevskii and modified Gross-Pitaevskii equations with *ab initio* variational Monte Carlo calculations for Bose-Einstein condensates of atoms in axially symmetric traps. We examine both the ground state and excited states having a vortex line along the z axis at high values of the gas parameter and demonstrate an excellent agreement between the modified Gross-Pitaevskii and *ab initio* Monte Carlo methods, both for the ground and vortex states.

DOI: 10.1103/PhysRevA.71.053610 PACS number(s): 03.75.Hh, 03.75.Lm, 02.70.Uu

I. INTRODUCTION

Most theoretical studies of Bose-Einstein condensates (BEC) in gases of alkali atoms confined in magnetic or optical traps have been conducted in the framework of the Gross-Pitaevskii (GP) equation [1]. The key point for the validity of this description is the dilute condition of these systems, i.e., the average distance between the atoms is much larger than the range of the interatomic interaction. In this situation the physics is dominated by two-body collisions, well described in terms of the s-wave scattering length a. The crucial parameter defining the condition for diluteness is the gas parameter $x(\mathbf{r})=n(\mathbf{r})a^3$, where $n(\mathbf{r})$ is the local density of the system. For low values of the average gas parameter $x_{\mathrm{av}}\leqslant 10^{-3}$, the mean-field Gross-Pitaevskii equation does an excellent job (see, for example, Ref. [2] for a review). However, in recent experiments, the local gas parameter may well exceed this value due to the possibility of tuning the scattering length in the presence of a Feshbach resonance [3,4].

Under such circumstances it is unavoidable to test the accuracy of the GP equation by performing microscopic calculations. If we consider cases where the gas parameter has been driven to a region were one can still have a universal regime, i.e., that the specific shape of the potential is unimportant, we may attempt to describe the system as dilute hard spheres whose diameter coincides with the scattering length. However, the value of x is such that the calculation of the energy of the uniform hard-sphere Bose gas would require to take into account the second term in the low-density expansion [5] of the energy density

$$\frac{E}{V}=\frac{2\pi n^2 a\hbar^2}{m}\left[1+\frac{128}{15}\left(\frac{na^3}{\pi}\right)^{1/2}+\ldots\right], \tag{1}$$

where m is the mass of the atoms treated as hard spheres. For the case of uniform systems, the validity of this expansion has been carefully studied using diffusion Monte Carlo [6] and hypernetted-chain techniques [7].

The energy functional associated with the GP theory is obtained within the framework of the local-density approximation (LDA) by keeping only the first term in the low-density expansion of Eq. (1)

$$E_{\mathrm{GP}}[\Psi]=\int d\mathbf{r}\left[\frac{\hbar^2}{2m}|\nabla\Psi(\mathbf{r})|^2+V_{\mathrm{trap}}(\mathbf{r})|\Psi|^2+\frac{2\pi\hbar^2 a}{m}|\Psi|^4\right], \tag{2}$$

where

$$V_{\mathrm{trap}}(\mathbf{r})=\tfrac{1}{2}m(\omega_\perp^2x^2+\omega_\perp^2y^2+\omega_z^2z^2) \tag{3}$$

is the confining potential defined by the two angular frequencies $\omega_\perp$ and ω_z. The condensate wave function Ψ is normalized to the total number of particles.

By performing a functional variation of $E_{\mathrm{GP}}[\Psi]$ with respect to Ψ^* one finds the corresponding Euler-Lagrange equation, known as the Gross-Pitaevskii (GP) equation

$$\left[-\frac{\hbar^2}{2m}\nabla^2+V_{\mathrm{trap}}(\mathbf{r})+\frac{4\pi\hbar^2 a}{m}|\Psi|^2\right]\Psi=\mu\Psi, \tag{4}$$

where μ is the chemical potential, which accounts for the conservation of the number of particles. Within the LDA framework, the next step is to include into the energy functional of Eq. (2) the next term of the low-density expansion of Eq. (1). The functional variation gives then rise to the so-called modified GP equation (MGP) [8]

$$\left[-\frac{\hbar^2}{2m}\nabla^2+V_{\mathrm{trap}}(\mathbf{r})+\frac{4\pi\hbar^2 a}{m}|\Psi|^2\left(1+\frac{32a^{3/2}}{3\pi^{1/2}}|\Psi|\right)\right]\Psi=\mu\Psi. \tag{5}$$

The MGP corrections have been estimated in Ref. [8] in a cylindrical condensate in the range of the scattering lengths and trap parameters from the first JILA experiments with Feshbach resonances. These experiments took advantage of the presence of a Feshbach resonance in the collision of two ^{85}Rb atoms to tune their scattering length [3]. Fully microscopic calculations using a hard-spheres interaction have also been performed in the framework of variational and diffusion Monte Carlo methods [10–13].

 ©2005 The American Physical Society

In this work we compare the results of the GP and MGP equations discussed above, Eqs. (4) and (5), with variational Monte Carlo (VMC) calculations for axially symmetric traps in a region ($x > 10^{-3}$), where the validity of the GP equation is not clear. We examine both the ground state and excited states having a vortex line along the z axis.

In the next section we present our numerical approaches together with a discussion of ground-state properties. In Sec. III we proceed to study several trial wave functions to describe the excited state with one vortex. A comparison between VMC and the GP and MGP equations is done. We summarize our results in Sec. IV.

II. NUMERICAL APPROACHES AND GROUND-STATE PROPERTIES

The starting point of the Monte Carlo calculations is the Hamiltonian for N trapped interacting atoms given by

$$H = -\frac{\hbar^2}{2m}\sum_{i=1}^{N}\nabla_i^2 + \sum_{i=1}^{N}V_{\text{trap}}(\mathbf{r}_i) + \sum_{i<j}^{N}V_{\text{int}}(|\mathbf{r}_i - \mathbf{r}_j|). \quad (6)$$

The two-body interaction $V_{\text{int}}(|\mathbf{r}_i - \mathbf{r}_j|)$ between the atoms is described by a hard-core potential of radius a, where a is the scattering length. The atoms are thus treated as hard spheres. The next step is to define a trial wave function

$$\Psi_T(1, \ldots, N) = F(1, \ldots, N)\Psi_{\text{MF}}(1, \ldots, N), \quad (7)$$

where $F(1,\ldots,N)$ is a many-body correlation operator applied to the mean-field wave function Ψ_{MF}. The advantage of using a correlated trial wave function lies in the fact that nonperturbative effects, as the short-range repulsion between atoms may be directly incorporated into the trial wave function. The simplest correlation operator has the Jastrow form [14],

$$F(1, \ldots, N) = \prod_{i<j} f(r_{ij}). \quad (8)$$

In our variational calculations we use a two-body correlation function, which is the solution of the Schrödinger equation for a pair of atoms at very low energy interacting via a hard-core potential of diameter a. The ansatz for the correlation function $f(r)$ reads

$$f(r) = \begin{cases} (1 - a/r) & r > a \\ 0 & r \leqslant a. \end{cases} \quad (9)$$

This type of correlation, besides being physically motivated, has been successfully used in Refs. [10,11] to study both spherically symmetric and deformed traps. These authors have also explored the quality of this correlation function by comparing variational Monte Carlo (VMC) and diffusion Monte Carlo (DMC) calculations for the case of spherically symmetric traps [12], with a good agreement between the VMC and DMC results.

The deformation of the trap is incorporated in the mean-field wave function Ψ_{MF}, which is taken as the product of N single-particle wave functions

$$\varphi(\mathbf{r}) = A(\alpha)\lambda^{1/4}\exp\left[-\tfrac{1}{2}\alpha(x^2 + y^2 + \lambda z^2)\right], \quad (10)$$

where α is taken as the variational parameter of the calculation, and $A(\alpha) = (\alpha/\pi)^{3/4}$ is the normalization constant. The parameter $\lambda = \omega_z/\omega_\perp$ is kept fixed and set equal to the asymmetry of the trap. In this way the mean-field wave function Ψ_{MF} has all the particles in the condensate, the latter being described by the wave function φ.

The evaluation of the expectation value of the Hamiltonian with this correlated trial wave function provides an upper bound to the ground-state energy of the system

$$E_T = \frac{\langle\Psi_T|H|\Psi_T\rangle}{\langle\Psi_T|\Psi_T\rangle}. \quad (11)$$

This expectation value has been evaluated by the Metropolis Monte Carlo method of integration [15,16].

The energy obtained with the Hamiltonian of Eq. (6) can be directly compared to the output of the GP and MGP equations, see Eqs. (4) and (5). The Gross-Pitaevskii equations represent a mean-field description, with all the atoms in the condensate. In fact, the additional correlations, which are taken into account in the second-order term of the low-density expansion of the energy [see Eq. (1)], are incorporated in the density functional and, therefore, in the solution of the MGP equation. In contrast, the Monte Carlo calculation explicitly incorporates the interatomic correlations, and therefore one could, in principle, find the natural orbits and extract the occupation of the condensate [10].

The GP and MGP equations have been solved by the steepest descent method [17] for the deformed harmonic oscillator trap previously described in Eq. (3). An initial deformed trial state is projected onto the minimum of the functional by propagating it in imaginary time. In practice, one chooses a small time step Δt and iterates the equation

$$\Psi(\mathbf{r}, t + \Delta t) \approx \Psi(\mathbf{r}, t) - \Delta t H\Psi(\mathbf{r}, t) \quad (12)$$

by normalizing Ψ at each iteration. When the gas parameter becomes large, the time step, which governs the rate of convergence, should be taken accordingly small. Convergence is reached when the chemical potential becomes a constant independent of the position, see Eqs. (4) and (5).

For the comparison of the results obtained with the different GP-type equations and the variational Monte Carlo calculations, we consider a disk-shaped trap with $\lambda = \omega_z/\omega_\perp = \sqrt{8}$, see Ref. [18]. We have fixed the scattering length to $a = 35a_{\text{Rb}}$, with $a_{\text{Rb}} = 100a_0$, a_0 being the Bohr radius. We set the number of confined atoms to $N = 500$ in order to keep the amount of computing time acceptable when using the Monte Carlo method. All the numerical results are given in units of the harmonic oscillator length $a_\perp = (\hbar/m\omega_\perp)^{1/2}$ and the harmonic oscillator energy $\hbar\omega_\perp$.

First we analyze the GP and MGP results reported in Table I. For a scattering length $a = 35a_{\text{Rb}}$, the corrections of the MGP approach to the chemical potential are of the order of 20%. The energy corrections are also relevant, and it is interesting to study the different contributions to the energy. The kinetic energy is given by

TABLE I. Chemical potential and energies in units of $\hbar\omega_\perp$ from the GP, MGP, and VMC calculations for the ground state. The scattering length is $a=35a_{\mathrm{Rb}}=0.151\,55a_\perp$, $\lambda=\sqrt{8}$, $N=500$.

	μ	E/N	E_{kin}/N	E_{HO}/N	E_1/N	E_2/N
GP	12.980	9.496 836	0.394 95	5.619 11	3.482 7765	
MGP	15.453	11.061 08	0.353 53	6.940 92	2.516 691	1.249 938
VMC		11.121 09(14)	4.215 20(24)	6.905 90(19)		

$$E_{\mathrm{kin}}=\frac{\hbar^2}{2m}\int d\mathbf{r}|\nabla\Psi(\mathbf{r})|^2, \tag{13}$$

while the harmonic oscillator energy due to the trapping potential reads

$$E_{\mathrm{HO}}=\frac{m}{2}\int d\mathbf{r}(\omega_\perp^2(x^2+y^2)+\omega_z^2z^2)|\Psi(\mathbf{r})|^2 \tag{14}$$

and the interaction energies E_1 and E_2 are given by

$$E_1=\frac{2\pi\hbar^2a}{m}\int d\mathbf{r}|\Psi(\mathbf{r})|^4, \tag{15}$$

$$E_2=\frac{2\pi\hbar^2a}{m}\frac{128}{15}\left(\frac{a^3}{\pi}\right)^{1/2}\int d\mathbf{r}|\Psi(\mathbf{r})|^5. \tag{16}$$

The virial theorem is used to establish a relation between the different contributions to the energy, viz.,

$$2E_{\mathrm{kin}}-2E_{\mathrm{HO}}+3E_1+\tfrac{9}{2}E_2=0, \tag{17}$$

which serves as a proof of the numerical accuracy of the solution of the GP equations. The results in Table I show that this test is well satisfied by all calculations.

Note that the kinetic energy associated with the mean-field descriptions is not negligible, indicating that the regime where the Thomas-Fermi approximation to the GP equation is valid has not been reached. In this limit, the chemical potential is

$$\mu_{\mathrm{TF}}=\tfrac{1}{2}(15\bar{a}N\lambda)^{2/5}\hbar\omega_\perp, \tag{18}$$

where $\bar{a}=a/a_\perp$ is the dimensionless scattering length, and the energy per particle $E_{\mathrm{TF}}/N=5\mu_{\mathrm{TF}}/7$. In this approach we have $E_{\mathrm{TF}}/N=9.03\hbar\omega_\perp$ and $\mu_{\mathrm{TF}}=12.64\hbar\omega_\perp$. Both these values differ from the values reported in Table I. However, this approximation can still be used to estimate the peak value of the gas parameter, namely,

$$x_{\mathrm{TF}}^{pk}=n(0)a^3=\frac{1}{8\pi}(15\bar{a}N\lambda)^{2/5}\bar{a}^2, \tag{19}$$

which yields $x_{\mathrm{TF}}^{pk}=0.023$. At this rather large value of the diluteness parameter, the corrections brought by the MGP equation to the GP results are expected to be relevant [6,7,9]. However, x is low enough to allow for a mean-field approach (as it is the case of the MGP equation). For such density regimes, a mean-field approach provides a rather good description when compared to a microscopic calculation [8].

The variational Monte Carlo results are also given in Table I and show a close agreement with the results provided by the MGP equation. Note that in this approach, and using the Hamiltonian of Eq. (6), the potential energy is zero since the wave function is strictly zero inside the core. The total energy in this case is distributed between E_{HO} and the true kinetic energy. Actually the only energies that can be directly compared to the GP results are the total and the harmonic oscillator energies.

The Monte Carlo results obtained with the Metropolis algorithm take into account the energy of 27 000 configurations, grouped in 90 blocks of 300 movements. At each Monte Carlo step we move all the particles and the acceptance is around 58%. A thermalization process is incorporated at the beginning of the Monte Carlo process and before each block. In the Monte Carlo calculation we have used the Pandharipande-Bethe prescription for the kinetic energy [16], which produces a smaller variance. To get a feeling for the numerical accuracy of our VMC results, we list here GP, MGP, and VMC results in the dilute limit. We employ $N=500$ particles and a scattering length for ^{87}Rb considered by Dalfovo and Stringari [19], which in units of the oscillator parameter perpendicular to the z axis is 4.33×10^{-3}. We obtain energies in units of the oscillator energy of 3.303 2151, 3.308 0392, and 3.324 1881 (10) for GP, MGP, and VMC calculations, respectively. The VMC results are for an optimum variational parameter $\alpha=0.475$. Taking into account that the two-body correlation has been kept fixed, and that the only variational parameter is α, these results indicate that our ansatz for the variational wave function is a viable one. Actually, as the reader will note from the discussion below, this discrepancy of roughly 0.5% is of the same relative order as for the higher density cases reported here.

In the minimization process we keep fixed the parameter λ in the single-particle wave function of Eq. (10), i.e., we assume that the deformation of the trap is transferred to the wave function, and vary only α. At the minimum, $\alpha=0.7687$. One can also explore the effects of the correlations in the density profiles. These profiles, which represent a column density defined according to

$$n_c(r_\perp)=\int dz\, n(r_\perp,z) \tag{20}$$

and normalized such that $2\pi\int dr_\perp r_\perp n_c(r_\perp)=1$, are shown in Fig. 1 for the various approximations used in this work. The repulsive character of the correction term of the MGP equation translates into a decrease of the value of the column density at the origin and an increase of the size of the condensate [8,9]. This gives a slightly more extended profile for the MGP approach compared to both the GP and the VMC results. As one can see from Fig. 1, there is a much better

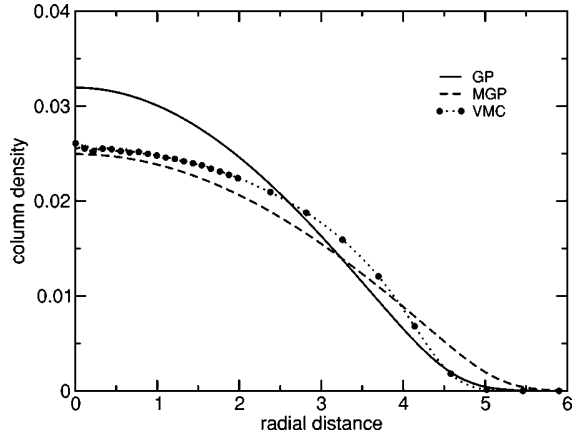


FIG. 1. Ground-state column density $n_c(r_\perp)$ as a function of the distance to the z axis, for $N=500$ particles, comparing the GP (solid line) and MGP (dashed line) results for $a=35a_{\mathrm{Rb}}=0.151\,55a_\perp$. Also shown are the results of variational Monte Carlo calculations (line with symbols). The deformation $\lambda=\sqrt{8}$ and the oscillator lengths are defined as in Refs. [18,19]. The radial distance is given in units of $a_\perp=(\hbar/m\omega_\perp)^{1/2}$. The column density is dimensionless. See text for further details.

agreement between the Monte Carlo and MGP profiles than with the corresponding profile from the GP calculation, particularly at small values of the radial distance where the density is larger.

The good agreement between VMC and MGP does not guarantee that these methods give a good description of the system. However, as it was shown in Ref. [11] for the case of spherical traps, the improvements introduced in the trial wave function by a diffusion Monte Carlo calculation, which, in principle, allows for an exact solution of the many-body problem, are rather small and the variational wave function of Eq. (10) provides a very good description of the system. Therefore we assume that the same will be true for deformed traps. Furthermore, for these values of the diluteness parameter, the MGP equation is very useful to calculate the energy, chemical potential, and density profiles of the ground state of the system for condensates with larger number of particles, which would be computationally prohibitive for a Monte Carlo calculation.

III. VORTEX STATES

The existence of these excited condensate states is crucial to studies of the superfluid behavior of trapped atomic condensates. In this section we study the effects of correlations in vortex states. We consider a singly quantized vortex line along the z axis. This means that all the atoms rotate around the z axis with a quantized circulation. The GP equation can easily be generalized to describe this kind of vortex states [2] by using the following ansatz for the condensate wave function

$$\Psi(\mathbf{r}) = \psi(\mathbf{r})\exp[i\kappa\phi], \tag{21}$$

where ϕ is the angle around the z axis and κ is an integer. This vortex state has a tangential velocity

$$v_\phi = \frac{\hbar}{mr_\perp}\kappa, \tag{22}$$

where $r_\perp=\sqrt{x^2+y^2}$ is the distance to the symmetry axis of the vortex. The number κ represents the quantum of circulation, and the total angular momentum along the z axis is given by $N\kappa\hbar$. Introducing the wave function of Eq. (21), in the GP energy functional of Eq. (2), one gets the corresponding GP energy functional for the vortex state

$$E_{\mathrm{GP+vor}}[\Psi] = \int d\mathbf{r}\left[\frac{\hbar^2}{2m}|\nabla\psi(\mathbf{r})|^2 + \frac{\hbar^2}{2m}\frac{\kappa^2}{r_\perp^2}|\psi|^2 + V_{\mathrm{trap}}(\mathbf{r})|\psi|^2 + \frac{2\pi\hbar^2 a}{m}|\psi|^4\right], \tag{23}$$

which incorporates a centrifugal term in the density functional, arising from the quantized flow of atoms around the vortex core. This term defines a rotational energy

$$E_{\mathrm{rot}} = \frac{\hbar^2}{2m}\int d\mathbf{r}\frac{\kappa^2}{r_\perp^2}|\psi(\mathbf{r})|^2. \tag{24}$$

The corresponding nonlinear Schrödinger equation obtained by functional variation is

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + \frac{\hbar^2}{2m}\frac{\kappa^2}{r_\perp^2} + V_{\mathrm{trap}}(\mathbf{r}) + \frac{4\pi\hbar^2 a}{m}|\psi|^2\right]\psi = \mu\psi. \tag{25}$$

Adding E_2 to the density functional and after performing a functional variation one gets the corresponding MGP equation for the vortex state.

Based on the virial theorem, one can again derive a relation between the different contributions to the energy

$$2E_{\mathrm{kin}} - 2E_{\mathrm{HO}} + 3E_1 + \tfrac{9}{2}E_2 + 2E_{\mathrm{rot}} = 0. \tag{26}$$

The thermodynamic critical angular frequency Ω_c required to produce a vortex of vorticity κ is obtained by comparing the energy of the system in the rotating frame with and without the vortex [20]

$$\Omega_c = \frac{1}{N\hbar\kappa}[E_\kappa - E_0]. \tag{27}$$

A main feature of a vortex state is the hole (core of the vortex) that appears in the center of the density profile along the rotation axis. From Eq. (25), it is clear that the solution of this equation has to vanish on the z axis because of the presence of the centrifugal term. The size of the core is characterized by the healing length.

For the microscopic description of the vortex state we use an Onsager-Feynman-type trial wave function [21]

$$\Psi_F(1,\ldots,N) = e^{i\kappa\Sigma_j\phi_j}\prod_j f(r_{\perp,j})\Psi_0(1,\ldots,N), \tag{28}$$

where $\Psi_0(1,\ldots,N)$ is the ground-state wave function. The phase factor $\kappa\Sigma_j\phi_j$ depends on the angular variables of the particles and is the equivalent to the phase factor introduced in the mean-field description of Eq. (21). The function $f(r_\perp)$ modulates the density as a function of the radial coordinate

TABLE II. Chemical potential and energies in units of $\hbar\omega_\perp$ from the GP and MGP calculations for the one-vortex state with the vortex line along the z axis. The scattering length is $a=35a_{\mathrm{Rb}}=0.151\,55a_\perp$, $\lambda=\sqrt{8}$, $N=500$.

	μ	E/N	E_{kin}/N	E_{HO}/N	E_1/N	E_2/N	E_{rot}/N
GP-1v	13.187	9.783 5936	0.425 08	5.742 71	3.403 871		0.211 93
MGP-1v	15.623	11.305	0.376 92	7.037 74	2.482 418	1.223 280	0.184 92

$r_\perp$. We examine three types of $f(r_\perp)$. In the first ansatz we use the simple option

$$f_1(r_\perp) = r_\perp. \tag{29}$$

In the second case we consider,

$$f_2(r_\perp) = 1 - \exp(-r_\perp/d), \tag{30}$$

where d is a variational parameter. Note that for $d=1$, the behavior of $f_2(r_\perp)$ for small $r_\perp$ coincides with the behavior of $f_1(r_\perp)$. Finally, the third function is that of Ref. [22], which has been used in the context of quantum liquids,

$$f_3(r_\perp) = 1 - \exp(-(r_\perp/d)^2), \tag{31}$$

where d is again a variational parameter.

These three trial wave functions describe a singly quantized vortex state ($\kappa=1$), whose axis lies in the z direction and with a tangential velocity field $v_\phi=\hbar/mr_\perp$. The evaluation of the expectation value of the Hamiltonian [Eq. (6)] with these wave functions is equivalent to calculate the mean value of the Hamiltonian

$$H = -\frac{\hbar^2}{2m}\sum_{i=1}^{N}\nabla_i^2 + \sum_{i=1}^{N}\frac{\hbar^2}{2m}\frac{\kappa^2}{r_{\perp,i}^2} + \sum_{i=1}^{N}V_{\mathrm{trap}}(\mathbf{r}_i) + \sum_{i<j}^{N}V_{\mathrm{int}}(|\mathbf{r}_i-\mathbf{r}_j|), \tag{32}$$

with $\Psi(1,\ldots,N)=\Pi_j f(r_{\perp,j})\Psi_0(1,\ldots,N)$. In this way the rotational contribution to the energy has been directly incorporated in the Hamiltonian. Minimizing this new problem provides the best energy and wave functions inside this subspace of wave functions. In the context of liquid ^{4}He there have been attempts to perform a full minimization allowing for a more general phase function. The analysis indicates that the present procedure provides very accurate results [23].

We start by discussing the GP and MGP results (obtained by the steepest descent method [17] as done for the ground state as well) with an initial condensate wave function

$$\psi(\mathbf{r}) \propto f_1(r_\perp)\Psi_0(\mathbf{r}). \tag{33}$$

It is worth mentioning, as a check of the numerical procedure, that starting with $f_2(r_\perp)$ or $f_3(r_\perp)$ to modulate the condensate wave function we converge to the same results as with $f_1(r_\perp)$.

As expected, the presence of the vortex increases the chemical potential. Also E_{HO} has a small increase, related to the enlargement of the profile because of the presence of the vortex hole. These results are listed in Table II. Although the MGP corrections to the energy are sizable and of the same order as those in the ground state, the critical frequency, $\Omega_{\mathrm{GP}}=0.29\omega_\perp$, is barely affected as both energies, the energy of the vortex state and the ground-state energy, are shifted by similar amounts, yielding $\Omega_{\mathrm{MGP}}=0.24\omega_\perp$.

The GP and MGP profiles for the vortex state are shown in Fig. 2. As a consequence of the repulsive character of the MGP corrections, the central density of the GP ground-state density profile is higher than the MGP one and, therefore, the depth of the hole around the z axis is larger in the GP approach. However, the healing length is almost the same.

As can be seen from Table III, the Monte Carlo results for the energies are in good agreement with the MGP ones for all the trial wave functions considered. This table shows two types of calculations. In the first three rows we list the energies obtained by keeping Ψ_0 equal to the ground-state wave function and performing the minimization with respect to the parameter d in the modulating function, except in the case of f_1, which has no variational parameters. In the second set of results, we perform a minimization allowing to vary also the harmonic oscillator parameter α of the wave function Ψ_0. The changes in α and d do not yield significant changes in the computed energy.

The density profiles seem to be more sensitive to the modulating function, as one can see from Fig. 2. These profiles correspond to the case where the ground-state wave

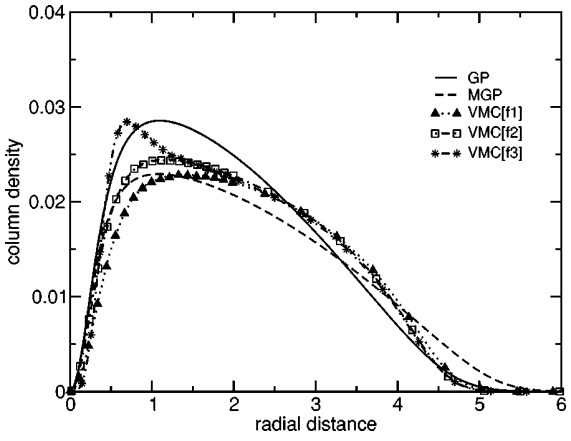


FIG. 2. Vortex column density $n_c(\mathbf{r}_\perp)$ as a function of the distance to the z axis, for $N=500$ particles, comparing GP (solid line) and MGP (dashed line) results for $a=35a_{\mathrm{Rb}}=0.151\,55a_\perp$. Also shown are the results of variational Monte Carlo calculations (lines with symbols) using the different Onsager-Feynman ansatzes. The deformation $\lambda=\sqrt{8}$ and the oscillator lengths are defined as in Refs. [18,19]. The radial distance is given in units of $a_\perp=(\hbar/m\omega_\perp)^{1/2}$. The column density is dimensionless. See text for further details.

TABLE III. Variational Monte Carlo results obtained with different Onsager-Feynman ansatzes. The results labeled VMC[f1], VMC[f2], and VMC[f3] stand for the modulating wave functions in Eqs. (28), (30), and (31), respectively.

	α	d	E/N	E_{kin}/N	E_{HO}/N	E_{rot}
VMC[f1]	0.7685		11.334 32(18)	4.148 04(26)	7.021 75(19)	0.164 527(44)
VMC[f2]	0.7685	1.175	11.362 73(18)	4.235 94(31)	6.939 87(24)	0.186 912(58)
VMC[f3]	0.7685	0.425	11.391 71(18)	4.288 45(30)	6.913 68(23)	0.189 580(30)
VMC[f1]	0.775		11.334 15(17)	4.186 34(29)	6.982 13(22)	0.165 679(60)
VMC[f2]	0.745	1.425	11.354 57(15)	4.078 16(31)	7.096 96(25)	0.179 446(93)
VMC[f3]	0.745	0.550	11.386 83(19)	4.149 02(33)	7.064 46(26)	0.173 350(26)

function Ψ_0 is kept fixed when we minimize the energy of the vortex state. For f_3 we obtain a radial structure, which is not present in the mean-field approach [22]. The MGP profile shows a broader surface region than the VMC profiles. In the core of the vortex, the MGP profile looks very similar to the VMC results with the modulating function f_2 of Eq. (30). These two results exhibit a smaller healing length than the VMC calculation which employs f_1.

From the variational point of view, the best description of the vortex should correspond to the wave function that provides the minimum energy. According to this criterion, this corresponds to the trial wave function built with the modulating function f_1 of Eq. (29).

Finally, in Fig. 3 we plot the density profiles for all VMC calculations, with and without vortices. We note that they all provide a similar healing length and that the asymptotic behavior is almost equal for both the ground state and the vortex states.

IV. CONCLUSIONS

We have compared the results of the Gross-Pitaevskii (GP) and the modified Gross-Pitaevskii (MGP) equations to *ab initio* variational Monte Carlo calculations for Bose-Einstein condensates of atoms in deformed traps. We have studied both the ground state and excited states having a vortex line along the z axis. The interatomic potential has been characterized by a hard-sphere potential with a radius that coincides with the scattering length used in the GP and MGP equations.

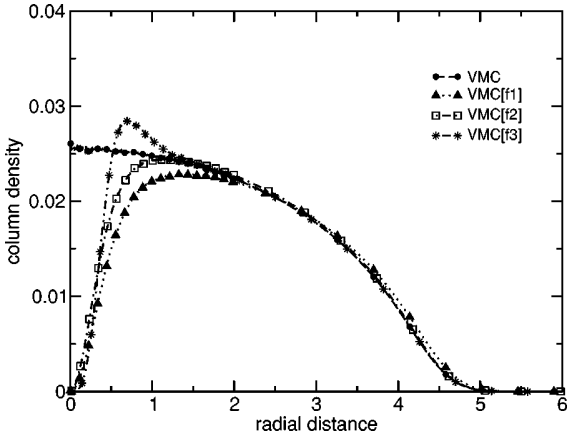


FIG. 3. Column density $n_c(\mathbf{r}_\perp)$ as a function of the distance to the z axis, comparing the VMC profiles for the vortex state corresponding to the different Onsager-Feynman ansatzes (lines with symbols as in Fig. 2) and the ground state (dashed line with full circles). The trap parameters and the scattering length are the same as in the two preceding figures. The radial distance is given in units of $a_\perp=(\hbar/m\omega_\perp)^{1/2}$. The column density is dimensionless. See text for further details.

We have performed the calculations for 500 particles. The parameters characterizing the trap and the scattering length have been chosen to reach values of the gas parameter where the MGP calculations provide corrections of the order of 20% compared to the GP results. It is indeed very interesting that even at such values of the gas parameter one can still describe the system in terms of mean-field approaches. We find, for example, an excellent agreement between the MGP and VMC results, especially for the energies of the ground state and the vortex states. The MGP and VMC density profiles for the ground state are also in good agreement. The situation is different for the vortex state. Three different trial wave functions produce similar energies but slightly different profiles. In the core of the vortex, the MGP profile is close to the profiles obtained with the ansatzes f_1 and f_2 of Eqs. (29) and (30), respectively. These functions yield also the lowest energies. Whether a diffusion Monte Carlo (DMC) calculation will show a similar trend remains to see. We are planning DMC studies of the systems discussed here. Our preliminary DMC calculations for the energy of the ground state show little change with respect to the VMC results and, hence, a very good agreement with the MGP results.

In summary, we would like to point out that the good agreement between the VMC and MGP is rather encouraging and allows for further MGP explorations of vortex states in condensates with both a larger number of interacting atoms and large scattering lengths.

ACKNOWLEDGMENTS

The authors are grateful to Professor A. Fabrocini and Professor J. Boronat for many useful discussions. This research was also partially supported by DGICYT (Spain) Project No. BFM2002-01868 and from Generalitat de Catalunya Project No. 2001SGR00064. J. Mur-Petit acknowledges support from the Generalitat de Catalunya. Support from the Research Council of Norway is acknowledged.

[1] L. P. Pitaevskii, Zh. Eksp. Teor. Fiz. **40**, 646 (1961) [Sov. Phys. JETP **13**, 451 (1961)]; E. P. Gross, Nuovo Cimento **20**, 454 (1961).
[2] F. Dalfovo, S. Giorgini, L. Pitaevskii, and S. Stringari, Rev. Mod. Phys. **71**, 463 (1999).
[3] S. L. Cornish, N. R. Claussen, J. L. Roberts, E. A. Cornell, and C. E. Wieman, Phys. Rev. Lett. **85**, 1795 (2000).
[4] B. GossLevi, Phys. Today **53**(9), 46 (2000).
[5] A. L. Fetter, and J. D. Walecka, *Quantum Theory of Many-Particle Systems* (McGraw-Hill, New York, 1971).
[6] S. Giorgini, J. Boronat, and J. Casulleras, Phys. Rev. A **60**, 5129 (1999).
[7] F. Mazzanti, A. Polls, and A. Fabrocini, Phys. Rev. A **67**, 063615 (2003).
[8] A. Fabrocini and A. Polls, Phys. Rev. A **60**, 2319 (1999).
[9] A. Fabrocini and A. Polls, Phys. Rev. A **64**, 063610 (2001).
[10] J. L. Dubois and H. R. Glyde, Phys. Rev. A **63**, 023602 (2001).
[11] A. R. Sakhel, J. L. Dubois, and H. R. Glyde, Phys. Rev. A **66**, 063610 (2002).
[12] J. L. Dubois and H. R. Glyde, Phys. Rev. A **68**, 033602 (2003).
[13] D. Blume and C. H. Greene, Phys. Rev. A **63**, 063601 (2001).
[14] R. Jastrow, Phys. Rev. **98**, 1479 (1955).
[15] N. Metropolis, A. E. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, J. Chem. Phys. **21**, 1087 (1953).
[16] R. Guardiola, in *Microscopic Quantum Many-Body Theories and their Applications*, edited by J. Navarro and A. Polls (Springer-Verlag, Berlin, 1998), Lecture Notes in Physics Vol. 510.
[17] K. T. R. Davies, H. Flocard, S. Krieger, and M. S. Weis, Nucl. Phys. A **342**, 111 (1980).
[18] M. H. Anderson, J. R. Ensher, M. R. Matthews, C. E. Wieman, and E. A. Cornell, Science **269**, 198 (1995).
[19] F. Dalfovo and S. Stringari, Phys. Rev. A **53**, 2477 (1996).
[20] A. L. Fetter and A. A. Svidzinsky, J. Phys.: Condens. Matter **13**, R135 (2001).
[21] L. Onsager, Nuovo Cimento, Suppl. **6**, 249 (1949); R. P. Feynman, in *Progress in Low Temperature Physics*, edited by C. Gorter (North-Holland, Amsterdam, 1955), Vol. I.
[22] G. V. Chester, R. Metz, and L. Reatto, Phys. Rev. **175**, 275 (1968); S. A. Vitiello, L. Reatto, G. V. Chester, and M. H. Kalos, Phys. Rev. B **54**, 1205 (1996).
[23] S. Giorgini, J. Boronat, and J. Casulleras, Phys. Rev. Lett. **77**, 2754 (1996).

A.2 MontePython: Implementing Quantum Monte Carlo using Python

Article and source code published in Computer Physics Communications, doi:10.1016/j.cpc.2007.06.013.

The article "MontePython: Implementing Quantum Monte Carlo using Python" is a writeup of the software MontePython, a software for simulating Bose-Einstein condensates with Monte Carlo methods in parallel.

The main ideas behind this article was to (1) illustrate how mixed-language programming can be used to make efficient, and readable, scientific code and (2) present an object-oriented approach to implementing a generic diffusion Monte Carlo simulator.

I was the sole author of both the article and the source code.

Available online at www.sciencedirect.com
ScienceDirect
Computer Physics Communications 177 (2007) 799–814
Computer Physics Communications
www.elsevier.com/locate/cpc

MontePython: Implementing Quantum Monte Carlo using Python ☆

Jon Kristian Nilsen

[a] *USIT, Postboks 1059 Blindern, N-0316 Oslo, Norway*
[b] *Fysisk institutt, Postboks 1048 Blindern, N-0316 Oslo, Norway*

Received 30 August 2006; received in revised form 24 April 2007; accepted 16 June 2007

Available online 29 June 2007

Abstract

We present a cross-language C++/Python program for simulations of quantum mechanical systems with the use of Quantum Monte Carlo (QMC) methods. We describe a system for which to apply QMC, the algorithms of variational Monte Carlo and diffusion Monte Carlo and we describe how to implement theses methods in pure C++ and C++/Python. Furthermore we check the efficiency of the implementations in serial and parallel cases to show that the overhead using Python can be negligible.

Program summary

Program title: MontePython
Catalogue identifier: ADZP_v1_0
Program summary URL: http://cpc.cs.qub.ac.uk/summaries/ADZP_v1_0.html
Program obtainable from: CPC Program Library, Queen's University, Belfast, N. Ireland
Licensing provisions: Standard CPC licence, http://cpc.cs.qub.ac.uk/licence/licence.html
No. of lines in distributed program, including test data, etc.: 49 519
No. of bytes in distributed program, including test data, etc.: 114 484
Distribution format: tar.gz
Programming language: C++, Python
Computer: PC, IBM RS6000/320, HP, ALPHA
Operating system: LINUX
Has the code been vectorised or parallelized?: Yes, parallelized with MPI
Number of processors used: 1–96
RAM: Depends on physical system to be simulated
Classification: 7.6; 16.1
Nature of problem: Investigating ab initio quantum mechanical systems, specifically Bose–Einstein condensation in dilute gases of ^{87}Rb
Solution method: Quantum Monte Carlo
Running time: 225 min with 20 particles (with 4800 walkers moved in 1750 time steps) on 1 AMD Opteron™ Processor 2218 processor; Production run for, e.g., 200 particles takes around 24 hours on 32 such processors.
© 2007 Elsevier B.V. All rights reserved.

PACS: 03.75.Hh; 03.75.Lm

Keywords: Python; C++; Quantum Monte Carlo; Bose–Einstein condensation; MPI

☆ This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

E-mail address: j.k.nilsen@usit.uio.no.

0010-4655/$ – see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.cpc.2007.06.013

1. Introduction

In scientific programming there has always been a struggle between computational efficiency and programming efficiency. On one hand, we want a program to go as fast as possible, resorting to low-level programming languages like FORTRAN77 and C which can be difficult to read and even harder to debug. On the other hand, we want the programming process to be as efficient as possible, turning to high-level software like Matlab, Octave, Maple, R and S+. Here features like clean syntax, interactive command execution, integrated simulation and visualization and rich documentation make us feel more productive. However, if we have some well tested and fast routines written in low-level language, interfacing these routines with, e.g., Matlab is rather cumbersome. Most often, we will end up using similar Matlab routines, which are often written as generally as possible at the cost of computing efficiency.

Recently, the programming language Python [1] has emerged as a potential competitor to Matlab. Python is a very powerful programming language which, when extended with numerical and visual modules like SciPy [2], shares many of the features of Matlab. In addition, Python was designed to be extendible with compiled code for efficiency and several tools are available for doing so.

In this paper we will demonstrate how Python can be extended with compiled code to yield an efficient scientific program. Specifically, we will start with a Monte Carlo simulator written in C++ and, with the help of SWIG [3], reuse the C++ code in a Python Monte Carlo simulator. We will show that this porting from low-level to high-level code can be achieved without significant loss of efficiency.

The remainder of this paper is organized as follows. In Section 2 we define the system we apply the Monte Carlo simulator to. Section 3 discuss briefly the diffusion Monte Carlo algorithm. Next, we go through the implementations of diffusion Monte Carlo, both in C++ and Python, in Section 4. Furthermore, Sections 5 and 6 compare the efficiency of C++ and Python for varying numbers of CPUs and Section 7 visualize the output from diffusion Monte Carlo with the use of Python. Finally, we round off with some remarks in Section 8.

2. The system

Quantum Monte Carlo (QMC) has a wide range of applications, for example studies of Bose–Einstein condensates of dilute atomic gases (bosonic systems) [4] and studies of so-called quantum dots (fermionic systems) [5], electrons confined between layers in semi-conductors. In this paper we will focus on a model which is meant to reproduce the results from an experiment by Anderson et al. [6]. Anderson et al. cooled down 4×10^6 ^{87}Rb to temperatures in the order of 100 nK to observe Bose–Einstein condensation in the dilute gas. Our physical motivation in this paper is to model numerically this fascinating experiment. This should be done in an as general as possible way, so that we can expand our computations to systems not yet explored in experiments. We will in this section go through the steps needed to put the experiment into the framework of QMC.

In QMC the goal is to solve the Schrödinger equation

$$i\hbar \frac{\partial}{\partial t}\Psi(\mathbf{R}, t) = H\Psi(\mathbf{R}, t), \tag{1}$$

or rather the time independent version

$$H\Psi(\mathbf{R}) = E\Psi(\mathbf{R}). \tag{2}$$

Thus, to model the experiment above using Quantum Monte Carlo methods, all we need is a Hamiltonian and a trial wave function. The Hamiltonian for N trapped interacting atoms is given by

$$H = -\frac{\hbar^2}{2m}\sum_{i=1}^{N}\nabla_i^2 + \sum_{i=1}^{N}V_{\text{ext}}(\mathbf{r}_i) + \sum_{i<j}^{N}V_{\text{int}}\big(|\mathbf{r}_i - \mathbf{r}_j|\big). \tag{3}$$

Taking advantage of the fact that the gas is dilute, we can describe the two-body interaction $V_{\text{int}}(|\mathbf{r}_i - \mathbf{r}_j|)$ by a hard-core potential of radius a, where a is the scattering length, thus treating the atoms as hard spheres [7].

We define the trial wave function by

$$\Psi_T = \prod_i g(\mathbf{r}_i)\prod_{i<j} f(r_{ij}), \tag{4}$$

where $g(\mathbf{r}_i)$ describes the interaction between one particle and the external potential, V_{ext}, while the two-body correlation function $f(r_{ij})$ describes the interaction between two particles. The function $f(r_{ij})$ is the solution of the Schrödinger equation for a pair of atoms at very low energy interacting via a hard-core potential of radius a. The ansatz for $f(r)$ reads

$$f(r) = \begin{cases} (1 - a/r) & r > a, \\ 0 & r \leqslant a. \end{cases} \tag{5}$$

Besides being physically motivated, this type of correlation has been successfully used in Refs. [8] and [4] to study both spherically symmetric and deformed traps. In the experiment of Anderson et al., the particles were trapped in a disk-shaped harmonic oscillator potential. This corresponds to using an external potential

$$V_{\text{ext}} = \frac{m}{2}\left(\omega_\perp x^2 + \omega_\perp y^2 + \omega_z z^2\right), \tag{6}$$

If we neglect the particle–particle interaction and insert the potential of Eq. (6) into Eq. (2) we obtain

$$g(\mathbf{r}) = A(\alpha)\lambda^{1/4}\exp\left(-\alpha\left(x^2 + y^2 + \lambda z^2\right)\right), \tag{7}$$

where α is taken as the variational parameter of the calculation and $A(\alpha) = (2\alpha/\pi)^{3/4}$ is a normalization constant. The parameter $\lambda = \omega_z/\omega_\perp$ is kept constant and set equal to the asymmetry of the trap. Still following Anderson et al. we let $\lambda = \sqrt{8}$ throughout this paper.

3. The Diffusion Monte Carlo algorithm

In this section we will go through the Diffusion Monte Carlo algorithm used in this paper. Diffusion Monte Carlo is built on Monte Carlo integration and the Metropolis algorithm [9] and usually needs input from a Variational Monte Carlo algorithm. The interested reader may find more information on these algorithms in, e.g., [10].

In Diffusion Monte Carlo (DMC) we seek to solve the Schrödinger equation in imaginary time. This involves Monte Carlo integration of a Green's function. As the Green's function is approximated by splitting it up in a diffusional part (which has the form of a Gaussian) and a branching part we also need a Gaussian random generator and a way to create and destroy walkers.

3.1. Basic ideas of DMC

The basic ingredients of DMC are [11]:

(1) It considers the Schrödinger equation in imaginary time,

$$-\frac{\partial\psi(\mathbf{R},t)}{\partial t} = [H - E]\psi(\mathbf{R},t), \tag{8}$$

where $\mathbf{R}$ represents the set of all coordinates. The formal solution of (8) is

$$\psi(\mathbf{R},t) = e^{-[H-E]t}\psi(\mathbf{R},0), \tag{9}$$

where $\exp[-(H-E)t]$ is called the *Green's function*, and E is a convenient energy shift.

(2) The wave function is positive definite everywhere, as it happens with the ground state of a bosonic system, so it may be considered as a probability distribution function. (This assumption leads to difficulties when we consider fermionic systems, where the wave functions are anti-symmetric and special care needs to be made.)

(3) The wave function is represented by a set of random vectors $\{R_1, R_2, \ldots, R_M\}$, in such a form that the time evolution of the wave function is actually represented by the evolution of the set of walkers.

(4) The actual computation of the time evolution is done in small time steps τ, and the Green's function is approximated accordingly,

$$e^{-[H-E]t} = \prod_{i=1}^{n} e^{-[H-E]\tau}, \tag{10}$$

where $\tau = t/n$.

(5) The imaginary time evolution of an arbitrary starting state $\psi(\mathbf{R},0)$, once expanded in the basis of stationary states of the Hamilton operator

$$\psi(\mathbf{R},0) = \sum_\nu C_\nu\phi_n u(\mathbf{R}) \tag{11}$$

is given by

$$\psi(\mathbf{R},t) = \sum_\nu e^{-[E_\nu - E]t} C_\nu\phi_\nu(\mathbf{R}), \tag{12}$$

in such a way that the lowest energy components will have the largest amplitudes after a long elapsed time, and in the $t \to \infty$ limit the most important amplitude will correspond to the ground state (if $C_0 \neq 0$).[1]

[1] This can easily be seen by replacing E with the ground state energy E_0 in Eq. (12). As E_0 is the lowest energy, we will get $\lim_{t\to\infty}\sum_\nu \exp[-(E_\nu - E_0)t]\phi_\nu = C_0\phi_0$.

(6) An improvement of this scheme is the introduction of *importance sampling*.

The scheme is quite simple; once we have found an appropriate approximation for the short-time Green's function and determined a starting state, the job consists in representing the starting state by a collection of walkers and letting them evolve in time, i.e. obtaining a collection of walkers from the old collection of walkers, up to a time large enough so that all other states than the ground state are negligible.

3.2. Importance sampling

An important improvement to the DMC scheme above is, as mentioned above, the use of *importance sampling*. In problems with singularities in the potential (e.g., the Coulomb potential) the Green's function $\exp[-(H-E)t]$ will reach unbounded values, leading to an unstable algorithm. Even without singularities the scheme above is inefficient. This is due to the fact that we have imposed no restrictions as to where the walkers will walk.

To impose such a restriction, we substitute our wave function $\psi(\mathbf{R},t)$ with a new quantity $f(\mathbf{R},t)=\psi_T(\mathbf{R})\psi(\mathbf{R},t)$ where $\psi_T(\mathbf{R})$ is a time-independent trial wave function, which should be as close as possible to the true ground state. This substitution can be shown [10, p. 92] to lead to a transformed Hamilton operator which may be written as a sum of three terms $H=K+F+L$, where

$$K=-D\nabla^2, \qquad F=-D\big(\nabla\cdot\mathbf{F}(\mathbf{R})\big)+\mathbf{F}(\mathbf{R})\cdot\nabla, \qquad L=E_L(\mathbf{R}), \tag{13}$$

corresponding, respectively, to the kinetic part, the drift part and the local energy part.

An $\mathcal{O}(\tau^2)$ approximation of the Green's function is given by [12]:

$$\langle\mathbf{R}'|G|\mathbf{R}\rangle=\frac{1}{(4\pi D\tau)^{3N/2}}e^{-[\mathbf{R}'-\mathbf{R}-D\tau\mathbf{F}(\mathbf{R})]^2/4D\tau}e^{E\tau-[E_L(\mathbf{R}')+E_L(\mathbf{R})]\tau/2}+\mathcal{O}(\tau^2), \tag{14}$$

while an $\mathcal{O}(\tau^3)$ approximation the Green's function is obtained from [13]

$$G=e^{E\tau}e^{-L/2\tau}e^{-F/2\tau}e^{-K\tau}e^{-F/2\tau}e^{-L/2\tau}+\mathcal{O}(\tau^3). \tag{15}$$

3.3. DMC algorithm

In Algorithm 1 we state the DMC algorithm corresponding to Eq. (14). The algorithm corresponding to Eq. (15) is similar except that the move is split into four parts due to the splitting of the drift operator. ξ in the move part of Algorithm 1 is drawn from the multivariate Gaussian distribution with null mean and $\sigma=\sqrt{2D\tau}$, the solution of the kinetic Green's function.

4. The implementations

In the previous sections we have identified a physical system to simulate and found algorithms to use in the simulations. One important question that remains is how we implement the system and the algorithms. In this section we will propose three different approaches. They all use the same algorithms, they solve the same systems, with identical results, and they are all written in an object oriented way. In fact, most of the code is the same for all three approaches. The only difference in the implementations is the amount of time spent in low level, compiled language (represented by C++) versus time spent in high level, interpreted language

```
Generate an initial set of random walkers with the Metropolis algorithm
for 0 to time
  for 0 to N_walkers
    Diffusion:
      for 0 to particles
          propose move r' = r + Dτ F(r) + ξ
      Branching; calculate replication factor:
      n = int(exp{τ(E_L(R)/2 + E_L(R')/2 − E)})
      if n = 0
          Kill the walker
      if n > 0
          Allow the walker to make n − 1 clones
      Remove dead walkers, and make new clones
      Check walker population and adjust trial energy sample contributions to observable
```

Algorithm 1. DMC algorithm

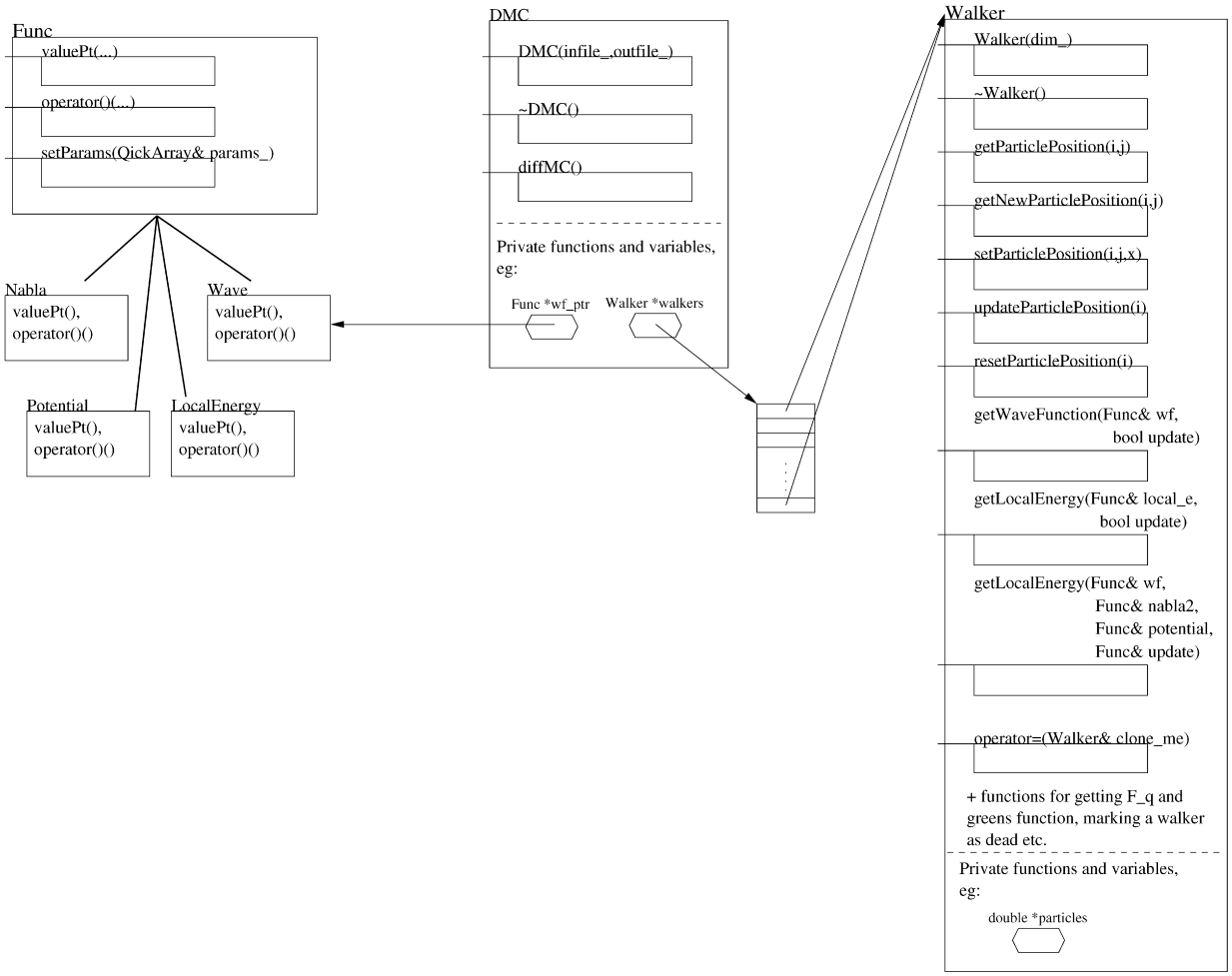


Fig. 1. Class diagram of DMC.

(represented by Python). The assumption is that compiled code is faster while interpreted code is clearer, easier to debug and easier to expand. We will in this section go through a pure C++ implementation, a straight forward Python approach and a more involved Python approach.

4.1. C++ implementation

The base of our implementations is a serial diffusion Monte Carlo (DMC) solver written in C++. The Python solvers are both heavily based on this code. We will therefore first go through the C++ implementation of DMC. In Figs. 1 and 2 we present the class diagram and float diagram of the C++ implementation.

In Fig. 1 we show three classes, class DMC, class Func and class Walker.

- The class DMC contains the DMC algorithm, implemented in the function diffMC() (and helper functions to clean up the code). It also contains a pointer to the class Func and an array of walker objects (or just walkers).
- The class Func contains functors, i.e. classes whose only purpose is to receive a set of numerical values and transform these to numerical output (not unlike mathematical functions). Specifically, Func contains different wave functions (with corresponding analytic local energies and quantum forces if implemented) along with generic functions for the gradient and Laplace operator. The different functions of the systems are subclasses derived from general functions to ensure that the functions of all the systems have the same input and output.
- The class Walker contains all the physical information of a walker, that is, its position in phase space (and function for setting and getting the position) and functions for getting physical values like the energy of the walker and the wave function of the walker.

The advantage of this division of the program is quite clear. The class DMC contains the DMC algorithm and may easily be replaced with other Monte Carlo algorithms, like the already mentioned variational Monte Carlo, Green's Function Monte Carlo and

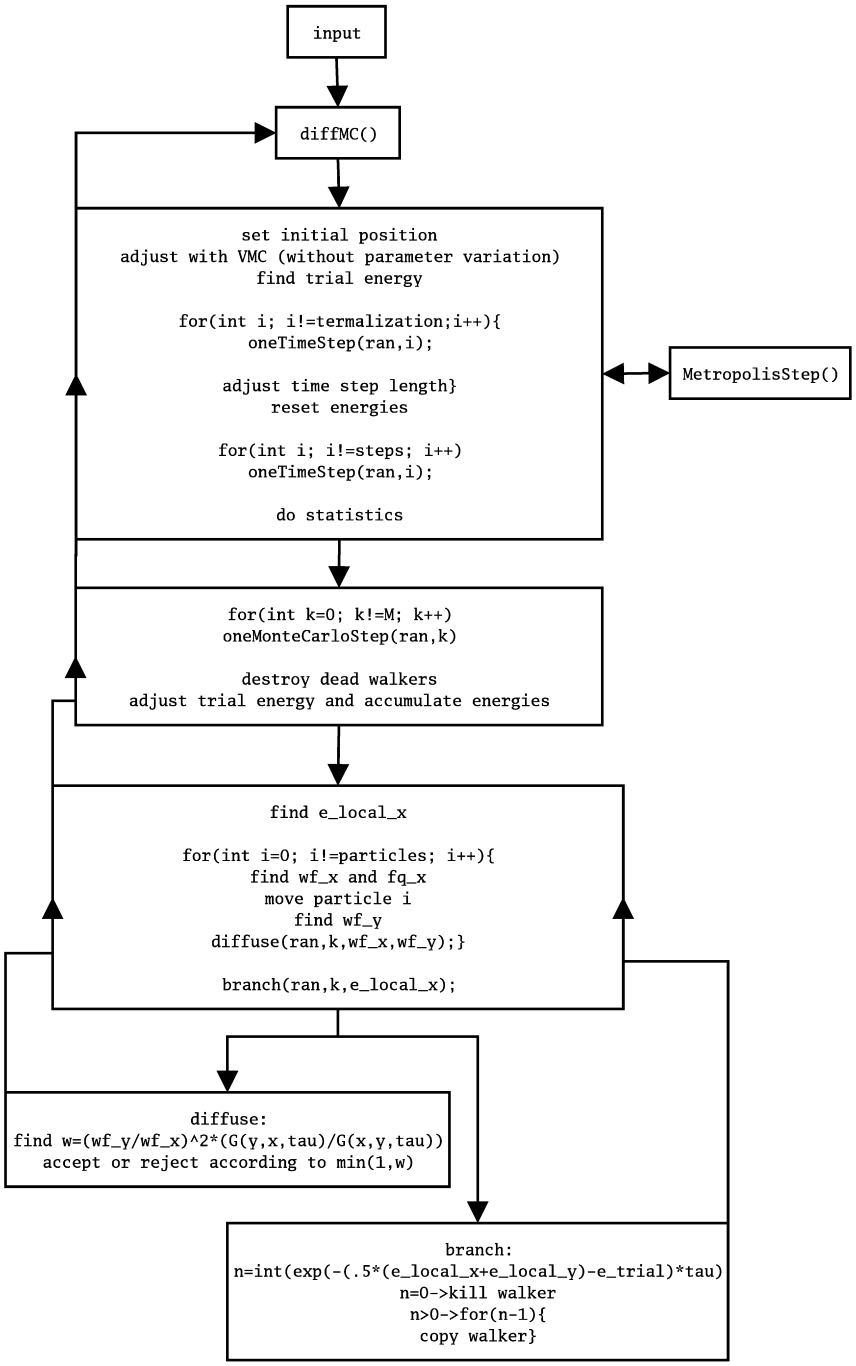


Fig. 2. Float diagram of DMC.

so on. These replacements will neither affect the systems implemented in class Func nor the physical information of the walkers. Likewise, new systems may be implemented without changing the code of the algorithm.[2] The wave function and the potential (or optionally an analytic expression of the local energy and the quantum force) are sent to the walkers as pointers to Func objects and are as such not known to the walkers at compile time.

Fig. 2 shows the float diagram of the DMC program. The algorithm is divided into functions so that, e.g., the function diffMC() contains a loop calling the function oneTimeStep(), which in turn loops over oneMonteCarloStep() and so on. Each such function is represented by a box in the float diagrams.

[2] Except, of course, that the DMC class has to know that the new system exists.

Looking at the float diagram, Fig. 2, it is easy to realize that most of the time of computation is spent in the bottom boxes of the diagram. When implementing the DMC in Python the bottom boxes should be kept in C++ while only diffMC() (which is in broad lines the hole DMC algorithm) will be in Python code.

4.1.1. Checkpointing

One aspect which is frequently forgotten when writing a scientific program is that of checkpointing. A Monte Carlo simulation may easily take several days, or even weeks and months. This would be infeasible without some way to stop and start the simulation in case of computer crashes, power losses or overeager computer managers. In checkpointing we store all information needed to resume the computation at given steps of the simulation. The challenge is to identify the steps at which the checkpointing should be made. The checkpoints should be made frequently enough to save time compared to starting all over, but not so frequently that the simulation is significantly slowed down.

In variational Monte Carlo it suffices to write a new initialization file where the number of steps is reduced to what is remaining of the original number of steps and a random seed so that we continue the random stream we have started. The latter is important if we want to get reproducible results. As the amount of data stored in a checkpoint is so small, we can do it after every step without reducing speed. However, making a checkpoint during the movement of the particles would be quite cumbersome and the amount of data needed to store the checkpoint would increase dramatically.

Again, diffusion Monte Carlo is more challenging. As generating a starting state in effect takes a variational Monte Carlo run, we have to store all the walkers at every checkpoint. This involves storing all particle positions, the last calculated local energy and quantum force (which is a vector) and so on, for every walker. In the C++ implementation this is realized by the functions getBuffer and setBuffer in the Walker objects. In a call to getBuffer the walker puts all it's information into a character array. In a checkpoint these arrays are concatenated and dumped to file. When restarting the program, the arrays are read from the file and sent to the walkers through setBuffer. The checkpoints are, as for variational Monte Carlo, made after every time step.

4.1.2. Generating random numbers

Central to a Monte Carlo method is the random number generator. The Monte Carlo integration depends on a walker's ability to reach all points in phase space from its starting point. If the random numbers determining the movement of the walker are in some way correlated, the walker will lose this ability. A good random number generator is therefore of great importance. Consider the simple one-dimensional definite integral

$$F = \int_0^1 f(x)\,\mathrm{d}x. \tag{16}$$

To solve this equation numerically, we approximate F in terms of F_N:

$$F = \lim_{N\to\infty} F_N, \tag{17}$$

where

$$F_N = \frac{1}{N}\sum_{i=1}^{N} f(x_i). \tag{18}$$

When we solve Eq. (16) using Monte Carlo integration, we draw the sample points $\{x_i\}$ randomly from a given probability density function. However, as a computer only has a finite sized set of numbers available, we have to use random numbers generated from a pseudo-random number generator (PRNG). For every PRNG there is a finite number of pseudo-random numbers, known as the cycle length of the PRNG. When this cycle length is reached Eq. (17) will cease to converge. This may not seem like a serious problem as the cycle length can be made quite large by using better PRNGs. However, we have to take care to choose a good PRNG. To take an example, the PRNG `ran0` (see [14]) has a cycle length of about 2.1×10^9. On a 3.40 GHz Intel(R) Xeon(TM) CPU `ran0` takes 40 seconds to run through one cycle. It is obvious that using this (widely used) PRNG will lead to problems when a Diffusion Monte Carlo simulation takes several days of CPU time.

The generation of random numbers is a science in itself and, though of great importance to Monte Carlo methods, we will not go through this aspect in detail. We can advise the interested reader to read the introduction of [15]. In the simulations in this paper we have used a 64-bit linear congruential generator with prime addend [16,17] which has a period of 2^{64}. Linear congruential generators may have correlations between numbers that are separated by a power of 2. We should therefore take care to avoid using this generator in batches of powers of 2.

4.2. Parallelizing the C++ implementation

To parallelize Variational Monte Carlo (VMC) is embarrassingly easy. As long as you ensure that all the calculations use different sets of random numbers (and thereby ensuring that the calculations are uncorrelated) the algorithm is parallelized by running an

independent calculation on each node. The communication between the nodes is restricted to spreading the input parameters before the calculations and collecting the output after calculation. The parallel efficiency is essentially 100%, and the calculation can theoretically use any number of nodes without efficiency loss.

The parallelization of Diffusion Monte Carlo (DMC) is more cumbersome. This is due to the branching part in Algorithm 1 where walkers are killed or reproduced. If we had parallelized DMC in a straight forward way, i.e. by starting one DMC run per node with different sets of random numbers and collected the results at the end, the walkers would be unevenly distributed among the processes, leading to an inefficient DMC code. For a DMC code to function properly it needs an as large as possible number of walkers to get a good representation of the wave function. A lot of unconnected DMC simulations will basically yield a set of not-so-good wave functions. We therefore have to collect all the walkers, remove dead walkers and make copies of the more virile walkers according to the branching process and then redistribute the walkers at every time step.

The parallelization is realized by a division of the walker array. A master node stores an array of the full number of walkers and distribute these walkers evenly between the slave nodes where the walkers are stored in smaller walker blocks. The preparation to sending and receiving the walker blocks is identical to the checkpoint procedure mentioned above, apart from the file writing and reading. In fact we use the MPI_Pack procedure to pack the walkers for checkpointing, even in the serial program. The only difference is that we send and receive the walkers instead of writing to and reading from file.

The main problem left is then to ensure that the sets of random numbers in fact are independent.

4.2.1. Generating random numbers in parallel

Generating random numbers in parallel is not as straightforward as one may think. A common first approach is to start the same random generator on every node, varying the seed with the rank of the node as a factor to get independent streams and hoping that these streams are uncorrelated. The main problem with this approach is that random generators often have long-term correlations which is of little importance in the serial case, but may appear as short-term correlations in a parallel case [16,17]. In the extreme case, we may chose seeds yielding random numbers separated with exactly one cycle. In this case we will end up with N_{CPU} identical streams, yielding N_{CPU} identical simulations and extremely good (but wrong) statistics in the results. Several approaches to get safe streams in parallel are suggested in [16,17] and implemented in the SPRNG library which we use in our simulations.

4.3. Python implementation I

The C++ implementation uses about 90% of the time in the walker objects and most of this time in computing local energies (N^3 operations where N is the number of particles) in functions located in Func. In the python implementation of diffusion Monte Carlo (pyDMC) the classes Walker and Func are therefore linked into a shared library readable from Python, through a thin wrapper module, together with the functions from the DMC class below the function oneTimeStep() in Fig. 2.

The main obstacle in implementing pyDMC is the handling of the walkers. In a straight forward approach we put the walker objects in a native Python array. This is a very tempting approach; we can leave the entire problem of creating and killing walkers[3] to Python. Another approach is to make a walker array class in C++. This way we can avoid explicit looping in the Python code, but we are again left to take care of varying array sizes in C++.

To understand the first approach, we must have a look at how to put the walkers into a native array. To get a C++ class visible from Python it has to be compiled and linked into a shared library. This step is taken care of by the use of SWIG [3]. The walker array is then realized by the function warray:

```
def warray(self,size,particles,dim):
    w = []
    for i in range(size):
        w += [Walker()]
        w[i].pyInitialize(particles,dim)
    return w
```

A great advantage with Python is that you can expand a class in run-time (or in fact build an entire class in run-time). Utilizing this advantage, we have inserted the function warray into the class Walker where it naturally belongs, as can be seen in the function funcToMethod:

```
def funcToMethod(func, clas, method_name=None):
    setattr(clas, method_name or func.__name__, func)
funcToMethod(warray,Walker) # insert function warray in class Walker
```

[3] Which is not a straight forward problem with C++ arrays.

This approach is particularly handy if we want to expand the Func class with new physical systems, enabling us to write the new functors in pure Python code. However, as the functors are where most of the computation time takes place, this approach will severely hinder the effectiveness of the simulation.

In the native array approach most of the parallelization is realized with the functions spread_walkers and gather_walkers. spread_walkers is as follows:

```
def spread_walkers(self):
    if self.master:
       displace = self.loc_walkers[self.master_rank]
       for i in range(1,self.numproc):
          send_w = self.w[displace:displace+self.loc_walkers[i]]
          send_buff = walkers2py(send_w)
          self.pypar.send(send_buff,i)
          displace += self.loc_walkers[i]
    else:
       recv_buff = self.pypar.receive(self.master_rank)
       w_args = [self.loc_walkers[self.myrank], self.particles, self.dimensions]
       self.w_block = py2walkers(recv_buff, *w_args)
```

If the master node calls spread_walkers, a walker buffer is made for each slave node and then sent. If the caller is a slave, it receives the relevant buffer and add it to the local walker block. The function gather_walkers is very similar, except that the slaves sends the buffers and the master receives and concatenates the buffers to the global walker array. The functions walkers2py and py2walkers are functions for converting a walker to a NumPy array and back again, taking advantage of the functions getBuffer and setBuffer in the Walker class.

A simple Python script for parallel DMC computations is as follows. We start with defining a function for one time step, or iteration:

```
def timestep(i_step):
    M = d.no_of_walkers
    d.spread_walkers()
    for walker in d.w_block:
       d.monte_carlo_step(walker)
    d.gather_walkers()
    d.update = False
    if d.master:
      # kill and replicate walkers
      for i in range(M-1,-1,-1):
         if d.w[i].isDead():
             d.w[i:i+1] = [] # removing walker
         else:
             while d.w[i].tooAlive():
                baby_walker = d.copy_walker(d.w[i])
                baby_walker.calmWalker()
                d.w += [baby_walker]
                d.w[i].madeWalker()
      d.no_of_walkers = len(d.w)
    d.no_of_walkers = d.pypar.broadcast(d.no_of_walkers, d.master_rank)
    d.refresh_w_blocks()
    d.spread_walkers()
    d.num_args[-1] = d.update
    if d.master:
       # find energy for this state (details skipped)
       # adjust trial energy (and no. of walkers)
       nrg = -.5*math.log(float(d.no_of_walkers)/float(M))/d.tau
       d.e_trial += nrg
    d.e_trial = d.pypar.broadcast(d.e_trial,d.master_rank)
    d.no_of_walkers = d.pypar.broadcast(d.no_of_walkers, d.master_rank)
```

This function may be divided into three steps. We start with running through all the walkers, doing a regular Monte Carlo step. Next we kill strayed walkers and replicate good walkers defined by the replication factor in Algorithm 1. Last, we calibrate the trial energy for the next time step. Given the `timestep` function the DMC script is as follows:

```
import pypar,math
from DMC import DMC

d = DMC(pypar)

# set initial walker positions:
d.uni_dist()

# do initial termalization using metropolis:
for i in range(d.metropolis_termalization):
    for walker in d.w_block:
        d.metropolis_step(walker)

# find initial energy (details skipped)

# do Monte Carlo iterations:
for i in range(d.steps):
    timestep(i)

if d.master:
    # calculate final energy and store to file (details skipped)
pypar.Finalize()
```

Here we generate a DMC object, setup an initial uniform walker distribution, do a termalization to distribute the walkers along the ground state of the system, make an estimation for the trial energy, and then start the production by iterating through the time steps. This implementation is capable of doing DMC computations, albeit not very efficiently, and, as you will see, there is room for improvement.

4.4. Python implementation II

When thinking performance of arrays in Python the add-on package *Numerical Python* (NumPy) springs to mind as an obvious choice. The fact that the module pypar (which we are going to use in the parallelization) supports sending NumPy arrays directly, is of course helping in that choice. However, even though NumPy supports a lot of types, (such as integers, floats, chars etc.) there is no support for walkers as a type.[4] It is possible to use generic Python objects in NumPy arrays, but this will mainly make NumPy array comparable to native arrays.

The approach with native Python arrays is quite straightforward and easy to implement. It is, however, quite inefficient as well. There are two main reasons for this. First, looping is known to be an inefficient construct in Python. With a native Python array, the loops over walkers have to be done in Python. Second, native Python arrays are slower than, e.g., NumPy arrays, because native arrays are written for a much more general use than just numerics. The question is how we can use the most of the C++ walker code as-is while avoiding explicit for-loops in Python.

One solution is to implement an array class (lets call it WalkerArray) in Python which is wrapper class to a C++ class containing a C++ array of walkers and functionality to create and kill walkers. Even though this is a good approach in a serial implementation, we still have to convert these arrays to NumPy arrays to be able to send the walkers in MPI.

In our Python implementation, we keep the WalkerArray, but store all the walker data in a NumPy array. To do this we have modified the C++ Walker class so that it only uses pointers to an array for all arrays and variables that should be stored. This array is then provided by NumPy. Even though this approach taints the C++ implementation of the Walker class, we get the advantage that the Python DMC class only has to care about NumPy arrays, providing us with powerful tools for vectorizing the Python code.

4.4.1. WalkerArray

To implement the class WalkerArray we need to have some knowledge of the C++ Walker class. The Walker class contains some arrays storing all particle positions and all previous particle positions and variables to know if the walker should be removed or duplicated. In addition it stores the last computed local energy, quantum force and wave function to minimize the number of times we compute these quantities. In the C++ code this information is allocated and stored in each walker, making the creation of a walker rather costly. When we send walkers in MPI the information is collected from each walker and concatenated into one array before communication and inserted into the walkers after communication. Now, we want turn it the other way, i.e. we want to allocate and store all information from all walkers in one array (preferably a NumPy array) and let the walkers operate on pointers to this array. This way the time to initialize a walker is reduced dramatically and all information on walkers are readily available from Python. This change of view for the Walker class is realized by changing all variables that define the walker to references to the corresponding pointers to the NumPy array.

[4] To the best of my knowledge.

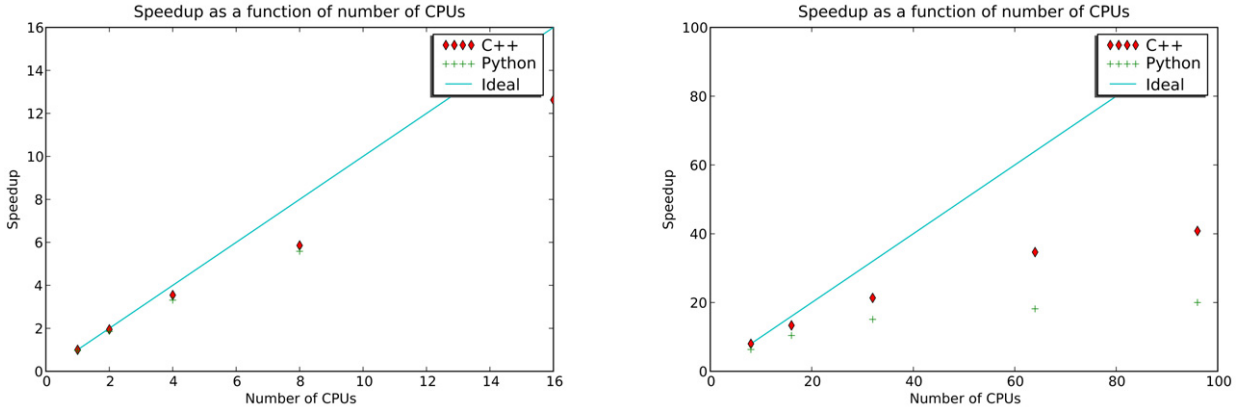


Fig. 3. Speedup of a simulation as a function of the number of CPUs used. In the left figure the serial run took about 30 minutes and was run with an initial 480 walkers moved in 3500 time steps. In the right figure the serial run took about 4 hours 30 minutes and was run with an initial 4800 walkers moved in 1750 time steps.

5. Python vs. C++

Now we know how to implement a Python version of a Monte Carlo solver. We then need to know if MontePython is efficient enough. We can assume that the Python implementation will never be faster that the corresponding C++ code, as Python will always have some degree of overhead just to access the C++ code. The question is how big this overhead may be. In Fig. 3 we have plotted the speedup of a Monte Carlo simulation as a function of the number of CPUs.[5] To the left we have done a relatively light simulation with 20 particles in 3 dimensions using 480 walkers. The walkers are spread out evenly and communicated from the master node to the slave nodes and back again 3500 times so that all walkers are sent 7000 times. The size of one walker is in this case 1.2 kB which means that for, e.g., 4 CPUs the size of each message is about 144 kB. We can see that for this message size the overhead of using Python is almost none.

To the right in Fig. 3 we have increased the number of walkers to 4800. However, we are also using a much higher number of CPUs. The message size for, e.g., 64 CPUs is only 9 kB. As we explicitly loop over the number of CPUs when we send walkers to and from the master, the overhead increases dramatically when compared to the time to send one message. The send and receive methods should therefore be vectorized with scatter and gather routines. Unfortunately we do not have uniform message sizes, making generic scatter and gather routines unusable. We will therefore have to write these routines ourselves.

Python has similar speed to C++, but the curve flattens out much faster as we increase the number of CPUs. As Monte Carlo simulations are known to have perfect speedup, we cannot be satisfied with the parallel algorithms of either the C++ version or the Python version.

6. Optimizing MontePython

One of the points in using Python in scientific programming is that you can implement new and improved algorithms efficiently. We have seen that distribution of the random walkers over the compute nodes leads to a bottleneck due to communication when the number of CPUs grows large. This bottleneck is evident both for the C++ implementation and the Python implementation. In this section we will improve the algorithm for load balancing of the walker in two ways. First, we will improve on the way the walkers are killed and reproduced. Second, we will improve on the load balancing itself by optimizing for heterogeneous clusters of CPUs.

6.1. Distributing walkers in parallel

The C++ Diffusion Monte Carlo application was originally written in serial and then ported to parallel using MPI. In the serial version we used an algorithm where, in order to kill a walker, we moved the last walker in the sequence onto the walker that was to be killed and decreased the number of walkers with one. To reproduce a walker, we copied the walker to the end of the sequence. This algorithm is optimal and widely used in serial Diffusion Monte Carlo. When we parallelized the code, we kept this serial algorithm by gathering all the walkers to the master node, let the master node do the killing and reproducing in serial, and then spread the walkers evenly among the slave nodes again. This way the load was always balanced, and the master had full control of the walkers at all times. The main problem is that, apart from memory issues as the master needs to store a lot of walkers, the

[5] Speedup is the time of a serial simulation divided by the walltime of the simulation.

serial work load for the master increases fast when we increase the number of CPUs in the calculation. This problem is very clear in Fig. 3, where the speedup is quite poor already for 32 CPUs, both for the C++ version and the Python version.

Again, the algorithm is best explained through the source code. First we let the slave nodes individually move their walkers and kill and reproduce their local walkers, then the function `DMC.load_balancing()` balances the load:

```
1   def load_balancing(self):
        self.t1 = time.time()
3       w_numbers = self.pypar.gather(Numeric.array([len(self.w_block)]),
                                      self.master_rank)
5       tmp_w_numbers = copy.deepcopy(w_numbers)
        w_numbers = self.pypar.broadcast(tmp_w_numbers,
7                                        self.master_rank)
9       self.no_of_walkers = Numeric.sum(w_numbers)

11      self.__find_opt_w_p_node()

13      self.first_balance = False
        balanced = Numeric.array(self.loc_walkers)
15
        difference = w_numbers-balanced
17
        diff_sort = Numeric.argsort(difference)
19      prev_i_min = diff_sort[0]

21      while sum(abs(difference))!=0:
            diff_sort = Numeric.argsort(difference)
23          i_max = diff_sort[-1]
            i_min = diff_sort[0]
25
            if i_min == prev_i_min:
27              i_min = diff_sort[1]

29          if self.myrank==i_max:
                self.pypar.send(self.w_block[balanced[i_max]:],i_min)
31              args = [balanced[i_max],
                        self.particles,
33                      self.dimensions,
                        self.w_block[0:balanced[i_max]]]
35              self.w_block = WalkerArray.WalkerArray(*args)
            elif self.myrank==i_min:
37              recv_buff = self.pypar.receive(i_max)
                args = [len(self.w_block)+difference[i_max],
39                      self.particles,
                        self.dimensions,
41                      Numeric.concatenate((self.w_block[:],recv_buff))]
                self.w_block = WalkerArray.WalkerArray(*args)
43          difference[i_min]+=difference[i_max]
            difference[i_max]=0
45          prev_i_min = i_min
```

This function deserves some explanation. From line 4 to 9 we update the current walker distribution and total number of walkers. In line 11 we determine the optimal distribution of walkers. to be explained in Section 6.2. At this point we know the actual distribution of walkers and the optimal distribution of walkers. The idea is then to find the length of the difference between the optimal and actual distribution and move walkers among nodes until the length, or the sum of the absolute value of the differences is zero, see line 21. This is realized in lines 29–44 by moving the excess walkers from the node with maximum difference to the walker with minimum difference recursively. A problem with this procedure is that the same node can have a minimum difference in subsequent cycles of the while-loop.[6] This leads to unnecessary waiting in the program. The remedy is seen in line 26 where we

[6] E.g., if a node with minimum difference needs 4 walkers and the second minimum is 1, while the maximum difference is 2, the minimum node is still the same after the first cycle. Then the message of the next cycle will have to wait till the first message is sent.

take the second minimum node if the minimum node is the same node as in the previous cycle. Of course this problem may just be transferred to the second minimum node, but this is much less likely to happen.

It should be noted that this optimization does not preserve the result from the non-optimized code, in the sense that we will not get an identical output in the end. This is due to the fact that the random sequences are distributed per node and not per walker, meaning that each walker will get a different series of random numbers depending on which node it is sent to. The output will, however, be within the error range of the non-optimized code.

6.2. Heterogeneous clusters

Most new high performance clusters are more or less homogeneous, in the sense that the computation nodes have identical specifications with respect to CPU, RAM, network and storage. However, as a cluster usually expands in time, due to more funds and need for more resources, it is very likely that it will become a heterogeneous cluster. Also, with the trend of multiple cores and CPUs per computation nodes, combined with the fact that there is more than one user per cluster, different nodes will have different (and possibly too high) load and therefore different computational speed. This means that even if a cluster is homogeneous on paper, it will act like a heterogeneous cluster in practice. If we want to gain the optimal performance from a cluster, we need to take into account this heterogeneity.

In the function `__find_opt_w_p_node()` we use the time from the DMC class is initialized to the function is called to determine how the optimal distribution of walkers at every time step. The function itself goes as follows:

```
1   def __find_opt_w_p_node(self):
      self.t1 = time.time()
3     timings = self.pypar.gather(Numeric.array([abs(self.t1-self.t0)]),
                                  self.master_rank)
5     tmp_timings = copy.deepcopy(timings)
      timings = self.pypar.broadcast(tmp_timings,
7                                    self.master_rank)

9     C = self.no_of_walkers/sum(1./timings)

11    tmp_loc_walkers = C/timings

13    self.loc_walkers = self.NumericFloat2IntList(tmp_loc_walkers)
      remainders = tmp_loc_walkers-self.loc_walkers
15
      while sum(self.loc_walkers) < self.no_of_walkers:
17       maxarg = Numeric.argmax(remainders)
         self.loc_walkers[maxarg] += 1
19       remainders[maxarg] = 0

21    if self.master and self.first_balance:
         print timings
23       print self.loc_walkers

25    return self.loc_walkers
```

To understand this function we just need some simple linear algebra. Say that we have set of walkers $[x_1, x_2, \ldots x_N]$ spread in an optimal way over N nodes. On node i the time to move one walker is given by a_i yielding a set $[a_1, a_2, \ldots a_N]$. By *optimal distribution* we mean a distribution of walkers were each node finishes the work assigned to it between synchronizations at the same time C, i.e. $a_i x_i = C$. In addition we know the total number of walkers, T. We know that

$$\sum x_i = \sum \frac{C}{a_i}, \tag{19}$$

so that the problem reduces to finding C. However, as $T = \sum x_i$, we have

$$C = \frac{T}{\sum 1/a_i}. \tag{20}$$

Here Eq. (20) corresponds to line 9 of `__find_opt_w_p_node()`. The rest of the function is merely taking care of the fact that x_i are integers while a_i and C are real numbers.

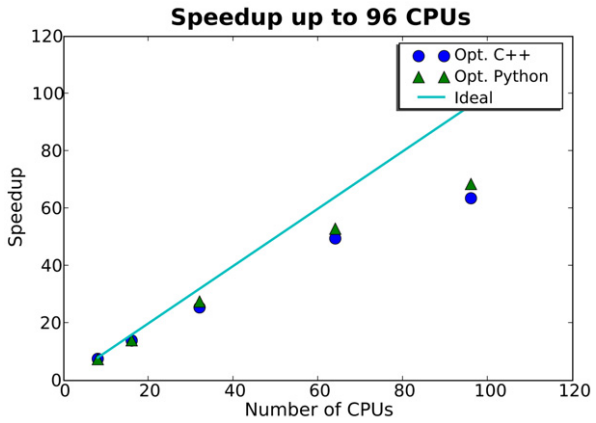


Fig. 4. The figure shows the speedup for the simulation as a function of the number of CPUs used. The serial run took about 209 minutes for C++, 225 minutes for Python and was run with an initial 4800 walkers moved in 1750 time steps.

6.3. Optimized results

Fig. 4 shows the speedup of the improved walker distribution compared to the same improvement in C++. We see that the speedup is actually better for Python for higher numbers of CPUs. C++ version. The main reason for this is that the serial version of C++ is faster than that of Python, with 209 minutes compared to 225 minutes. When increasing the number of CPUs the Python version gets closer to the C++ version in walltime, and already for 32 CPUs the walltimes are inseparable, with differences of less than 1%. Due to storing the data in a NumPy array, Python does not need to allocate memory for every moved walker separately every time a block of walkers is moved. It should be noted that we again see a slope in the speedup curve when going to large numbers of CPUs. This is again due to the small number of walkers on each node when having a constant global number of walkers. This effect is now independent on whether we use Python or C++.

The simple syntax in Python and the use of NumPy arrays to store walkers also allow us to concentrate our effort directly on the optimization of the algorithm instead of dealing with, e.g., how to send, receive and concatenate a slice of walkers. As a simple comparison, the author used approximately two working days to implement the optimization in Python. Due to some problems with segmentation faults in the timer function, it took the author about seven working days to implement the C++ version. This could be just as much due to lack of programming skills from the author as problems with C++, but it is quite clear that debugging is more efficient in Python than C++.

7. Visualizing with Python

We mentioned in the introduction that integration of simulation and visualization is an important feature of Matlab, Maple and others. This feature is maybe even more powerful in Python. In Figs. 5 and 6 we have used pyVTK and Mayavi [18] to plot the particle density, which is an output of diffusion Monte Carlo. pyVTK [19] is a Python interface to the Visualization ToolKit (VTK), while Mayavi, which is built on pyVTK, is a scriptable graphic interface for 3D visualization.

A signature of a Bose–Einstein condensate is that it is irrotational. If we try to rotate the condensate, it will compensate by setting up quantum vortices along the rotational axis. Vortices is therefore crucial to the study of Bose–Einstein condensates. We need only small modifications to the Hamiltonian (Eq. (3)) and trial wave function (Eq. (4)) to consider a single vortex along the z-axis in our system [7].

Figs. 5 and 6 shows the change in the ground state when inserting the vortex. The repulsive nature of the vortex pushes the particles away from the z-axis, decreasing the maximum density when compared to the ground state.

8. Conclusion

We have implemented a Monte Carlo solver using three different approaches; pure C++, a Python implementation, and an efficient, vectorized Python implementation. Furthermore we have compared the vectorized Python implementation with a corresponding C++ implementation and shown that the overhead of using Python is small for sufficiently large problems. In fact, with only two rather simple functions we were able to introduce an improved parallel algorithm for walker distribution, making the difference between Python and C++ close to negligible. In addition, we have shown that Python can be used directly as a visualization tool for rendering three dimensionally scientific visualizations. We conclude that Python can serve as a powerful tool in scientific programming.

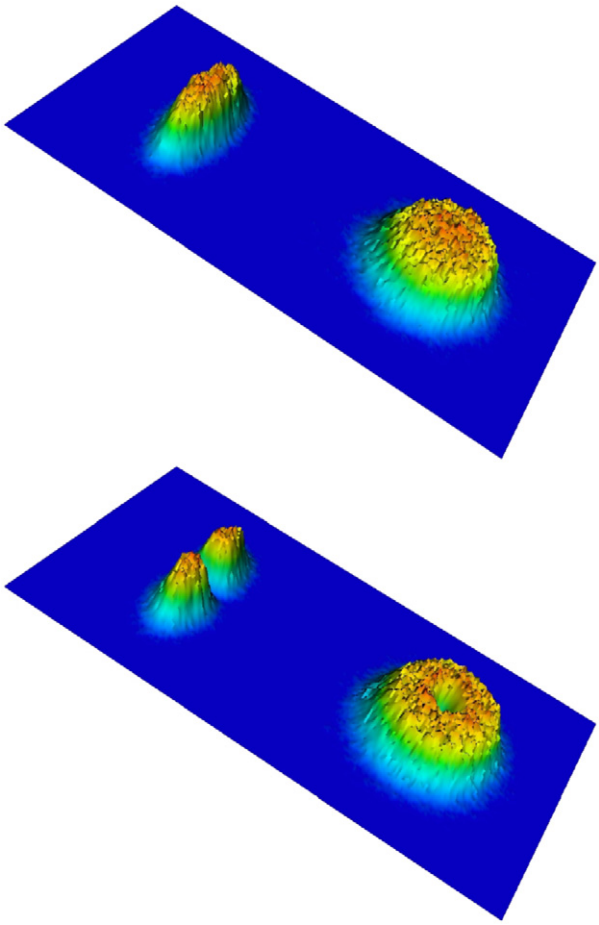


Fig. 5. The figures show where particles are detected. Plotted are the expectation values in two spatial dimensions, the yz-plane to the left and the xy-plane to the right. The topmost figure corresponds to the ground state, while the bottom figure corresponds to a state with one vortex in the center of the trap.

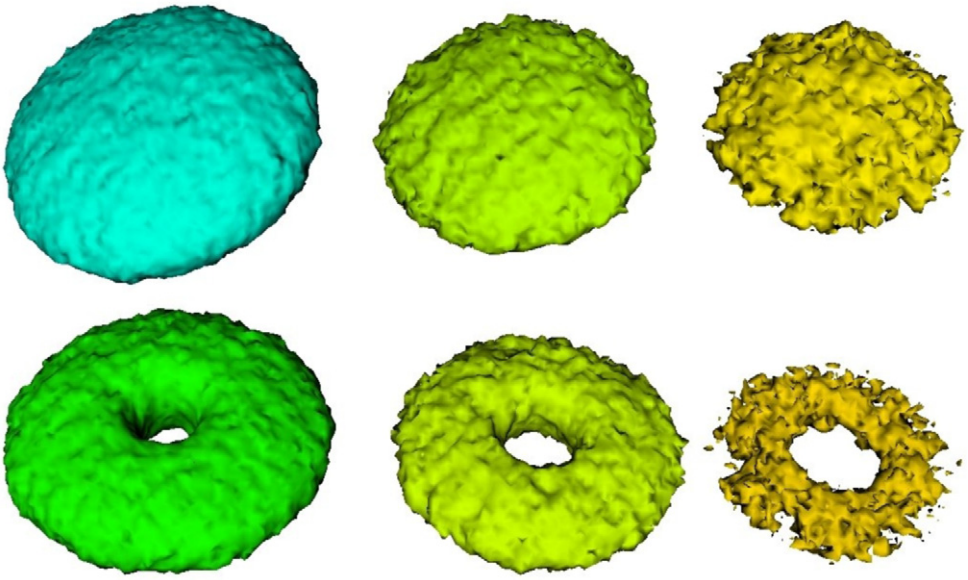


Fig. 6. The figures show where particles are detected. Plotted are the expectation values for finding, from left to right, 1, 2, 3 and 4 particles. The topmost figure corresponds to the ground state, while the bottom figure corresponds to a state with one vortex in the center of the trap.

References

[1] G. van Rossum, et al., The Python Programming Language, 1991.

[2] P. Peterson, E. Jones, T. Oliphant, et al., SciPy: Open Source Scientific Tools for Python, 2001.
[3] D. Beazley, et al., SWIG: Simplified Wrapper and Interface Generator, 1995.
[4] J.L. DuBois, H.R. Glyde, Natural orbitals and bec in traps, a diffusion Monte Carlo analysis, Phys. Rev. A 68 (2003).
[5] A. Harju, Variational Monte Carlo for interacting electrons in quantum dots, J. Low Temperature Phys. 140 (2005) 181–210.
[6] J.R. Anderson, M.R. Ensher, M.R. Matthews, C.E. Wieman, E.A. Cornel, Observation of Bose–Einstein condensation in a dilute atomic vapor, Science 269 (1995) 198.
[7] J.K. Nilsen, J. Mur-Petit, M. Guilleumas, M. Hjorth-Jensen, A. Polls, Vortices in atomic Bose–Einstein condensates in the large gas parameter region, Phys. Rev. A 71 (2005).
[8] J.L. DuBois, H.R. Glyde, Bose–Einstein condensation in trapped bosons: A variational Monte Carlo analysis, Phys. Rev. A 63 (2001).
[9] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.M. Teller, E. Teller, Equations of state calculations by fast computing machines, J. Chem. Phys. 21 (1953) 1087.
[10] B.L. Hammond, W.A. Lester Jr., P.J. Reynolds, Monte Carlo Methods in Ab Initio Quantum Chemistry, World Scientific, 1994.
[11] R. Guardiola, Monte Carlo methods in quantum many-body theories, in: J. Navarro, A. Polls (Eds.), Microscopic Quantum Many-Body Theories and their Applications, in: Lecture Notes in Physics, Springer, Verlag, 1997, pp. 269–336.
[12] P.J. Reynolds, D.M. Ceperley, B.J. Alder, W.A. Lester Jr, J. Chem. Phys. 77 (1982) 5593.
[13] A. Sarsa, J. Boronat, J. Casulleras, Quadratic diffusion Monte Carlo and pure estimators for atoms, Phys. Rev. A (2001).
[14] W.H. Press, S.A. Teukolsky, W.T. Vetterlin, B.P. Flannery, Numerical Recipes in C, second ed., Cambridge University Press, 2002.
[15] D.E. Knuth, The Art of Computer Programming, second ed., Addison-Wesley Publishing Company, 1981.
[16] M. Mascagni, A. Srinivasan, Sprng: A scalable library for pseudorandom number generation, ACM Trans. Math. Software 26 (2000) 436–461.
[17] A. Srinivasan, M. Mascagni, D. Ceperley, Testing parallel random number generators, Parallel Comput. 29 (2003) 69–94.
[18] P. Ramachandran, Mayavi: A free tool for CFD data visualization, in: 4th Annual CFD Symposium, Aeronautical Society of India, 2001.
[19] P. Peterson, PyVTK: Tools for Mmanipulating VTK Files in Python, 2001.

A.3 Simplifying Parallelization of Scientific Codes by a Function-Centric Approach in Python

Article submitted to Comp. Science & Discovery.

The article "Simplifying Parallelization of Scientific Codes by a Function-Centric Approach in Python" was the product of a cross-science collaboration between two computer scientists (Xing Cai and Hans Petter Langtangen), a social scientist (Bjørn Høyland) and a physicist (me).

The main idea of the article was to present a "toolbox" of generic methods for parallelizing existing scientific applications, exemplified by applications from the three sciences.

My main contributions to the article was the source code of the Monte Carlo examples and the text in Sections 3.2 and 4.2 and Appendix B.

Simplifying Parallelization of Scientific Codes by a Function-Centric Approach in Python

Jon K. Nilsen[1,2], Xing Cai[3,4], Bjørn Høyland[5] and Hans Petter Langtangen[3,4]

[1] USIT, P.O. Box 1059 Blindern, N-0316 Oslo, Norway
[2] Department of Physics, P.O. Box 1048 Blindern, University of Oslo, N-0316 Oslo, Norway
[3] Center for Biomedical Computing, Simula Research Laboratory, P.O. Box 134, N-1325 Lysaker, Norway
[4] Department of Informatics, P.O. Box 1080 Blindern, University of Oslo, N-0316 Oslo, Norway
[5] Department of Political Science, P.O. Box 1097 Blindern, University of Oslo, N-0317 Oslo, Norway

E-mail: `j.k.nilsen@usit.uio.no`, `xingca@simula.no`, `bjorn.hoyland@stv.uio.no`, `hpl@simula.no`

Abstract.

The purpose of this paper is to show how existing scientific software can be parallelized using a separate thin layer of Python code where all parallel communication is implemented. We provide specific examples on such layers of code, and these examples may act as templates for parallelizing a wide set of serial scientific codes. The use of Python for parallelization is motivated by the fact that the language is well suited for reusing existing serial codes programmed in other languages. The extreme flexibility of Python with regard to handling functions makes it very easy to wrap up decomposed computational tasks of a serial scientific application as Python functions. Many parallelization-specific components can be implemented as generic Python functions, which may take as input those functions that perform concrete computational tasks. The overall programming effort needed by this parallelization approach is rather limited, and the resulting parallel Python scripts have a compact and clean structure. The usefulness of the parallelization approach is exemplified by three different classes of applications in natural and social sciences.

Submitted to: *Computational Science & Discovery*

1. Introduction

Due to limited computing power of standard serial computers, parallel computing has become indispensable for investigating complex problems in all fields of science. A frequently encountered question is how to transform an existing serial scientific code into a new form that is executable on a parallel computing platform. Although portable parallel programming standards, such as MPI and OpenMP, have greatly simplified the programming work, the task of parallelization may still be quite complicated for domain scientists. This is because inserting MPI calls or OpenMP directives directly into an existing serial code often requires extensive code rewrite as well as detailed knowledge of and experience with parallel programming.

The hope for non-specialists in parallel computing is that many scientific applications possess high-level parallelism. That is, the entire computational work can be decomposed into a set of individual (and often collaborative) computational tasks, each of coarse grain, and can be performed by an existing piece of serial code. Depending on the specific application, the decomposition can be achieved by identifying a set of different parameter combinations, or (fully or almost) independent computations, or different data groups, or different geometric subdomains. For a given type of decomposition, the parallelization induced programming components, such as work partitioning, domain partitioning, communication, load balancing, and global administration, are often generic and independent of specific applications. These generic components can thus be implemented as reusable parallelization libraries once and for all. This is what we exemplify in the present paper.

It is clear that a user-friendly parallelization approach relies on at least two factors: (1) The existing serial code should be extensively reused; (2) The programming effort by the end user must be limited. To achieve these goals we suggest to use Python to wrap up pieces of existing serial code (possibly written in other languages), and implement the parallelization tasks in separate and generic Python functions.

Python [1] is an extremely expressive and flexible programming language at its core. The language has been extended with numerous numerical and visualization modules such as NumPy [2] and SciPy [3]. The two requirements of a user-friendly parallelization mentioned above are actually well met by Python. First of all, Python is good at inter-operating with other languages, especially Fortran, C, and C++, which are heavily used in scientific codes. Using wrapper tools such as F2PY [4], it is easy to wrap up an existing piece of code in Fortran and C and provide it with a Pythonic appearance.

Moreover, among its many strong features, Python is extremely flexible with handling functions. Python functions accept both positional arguments and keyword arguments. The syntax of a variable set of positional and keyword arguments (known as "`(*args,**kwargs)`" to Python programmers) allows writing libraries routines that work with any type of user-defined functions. That is, the syntax makes it possible to call a Python function without revealing the exact number of arguments.

It is also straightforward to pass functions as input arguments to a Python function

and/or return a function as output. A callable class object in Python can be used as if it were a stand-alone function. Such a construction, or alternatively a closure (known from functional programming), can be used to create functions that carry a state represented through an arbitrarily complex data structure. The result is that one can express the flow of a scientific code as a Python program containing a set of calls to user-defined Python functions. These user-defined functions can be ordinary functions or classes that wrap pieces of the underlying scientific code. This is what we call a *function-centric* representation of the scientific code. With such a function-centric approach, we can build a general framework in Python for almost automatic parallelization of the program flow in the original scientific code. Later examples will convey this idea in detail.

Performance of the resulting parallel application will closely follow the performance of the serial application, because the overhead of the parallelization layer in Python is just due to a small piece of extra code, as we assume the main computational work to take place in the Python functions that call up pieces of the original scientific code. In the parallelization layer, good performance can be ensured by using efficient array modules in Python (such as `numpy` [2]) together with light-weight MPI wrappers (such as `pypar` [5]). For examples of writing efficient Python code segments for some standard serial and parallel numerical computations, we refer the reader to Cai et al. [6].

Related Work. In C++, generic programming via templates and object-oriented programming has been applied to parallelizing serial scientific codes. Two examples can be found in [7] and [8], where the former uses C++ class hierarchies to enable easy implementation of additive Schwarz preconditioners, and the latter uses C++ templates extensively to parallelize finite element codes. Many scientific computing frameworks have also adopted advanced programming to incorporate parallelism behind the scene. In these frameworks (see, e.g., [9, 10, 11, 12, 13]) the users can write parallel applications in a style quite similar to serial programming, without being exposed to many parallelizing details. Likewise are frameworks that are specially designed to allow coupling of different serial and parallel components, such as Cactus [14] and MpCCI [15]. The Python programming language, however, has not been widely used to parallelize existing serial codes. The Star-P system [16] provides the user with a programming environment where most of the parallelism is kept behind the scene. Hinsen [17] has combined Python with BSP to enable high-level parallel programming. In addition, quite a number of Python MPI wrappers exist, such as `pyMPI` [18], `pypar` [5], `MYMPI` [19], `mpi4py` [20, 21], and `Scientific.MPI` [22]. Efforts in incorporating parallelism via language extensions of Python can be found in [23, 24, 25].

The contribution of the present paper is to show by examples that a function-centric approach using Python may ease the task of parallel scientific programming. This result is primarily due to Python's flexibility in function handling and function arguments. As a result, generic tasks that arise in connection with parallelization can often be programmed as a collection of simple and widely applicable Python functions,

which are ready to be used by non-specialists to parallelize their existing serial codes.

This paper contains three examples with different algorithmic structures. A wide range of problems in science can be attacked by extending and adapting the program code in these examples. Moreover, readers whose problems are not covered by the examples will hopefully from these examples understand how we solve programming problems by identifying the principal, often simplified, underlying algorithmic structure; then creating generic code to reflect the structure; and finally applying the generic code to a specific, detailed case. Our approach is much inspired by the success of mathematics in problem solving, i.e., detecting the problem's principal structure and devising a generic solution makes complicated problems tractable. With Python as tool, we demonstrate how this strategy carries over to parallelization of scientific codes.

The remainder of the paper is organized as follows. We give in Section 2 a simple but motivating example, explaining the principles of splitting a problem into a set of function calls that can easily be parallelized. Generic parallelization of three common types of real scientific applications are then demonstrated in Section 3. Afterwards, Section 4 reports the computational efficiency of the suggested parallelization approach applied to specific cases in the three classes of scientific problems. Some concluding remarks are given in Section 5.

2. A Motivating Simple Example

2.1. Serial Version

Suppose we want to carry out a parameter analysis that involves a large number of evaluations of a multi-variable mathematical function $f(a_1, \ldots, a_q)$. The Python implementation of f may use p positional arguments and k keyword arguments such that the total $p + k$ arguments contain at least the variables $a_1, \ldots, a_q$ (i.e., $q \leq p + k$). As a very simple example, consider the parabola $f(x, a, b, c) = ax^2 + bx + c$ with the following Python implementation ($q = 4, p = 1, k = 3$):

```
def func(x, a=0, b=0, c=1):
    return a*x**2+b*x+c
```

Suppose we want to evaluate `func` for a particular set of input parameters chosen from a large search space, where x, a, b, and c vary in specified intervals. The complete problem can be decomposed into three main steps: (1) initialize a set of arguments to `func`; (2) evaluate `func` for each entry in the set of arguments; (3) process the set of function return values from all the `func` calls.

Step (1) calls a user-defined function `initialize` which returns a list of 2-tuples, where each 2-tuple holds the positional and keyword arguments (as a tuple and a dictionary) for a specific call to `func`. Step (2) iterates over the list from step (1) and feed the positional and keyword arguments into `func`. The returned value (tuple) is stored in a result list. Finally, step (3) processes the result list in a user-defined function `finalize` which takes this list as input argument.

A generic Python function that implements the three-step parameter analysis can be as follows:

```
def solve_problem(initialize, func, finalize):
    input_args = initialize()
    output = [func(*args, **kwargs) for args, kwargs in input_args]
    finalize(output)
```

Note that the use of *list comprehension* in the above code has given a very compact implementation of the `for`-loop for going through all the evaluations of `func`. The `initialize`, `func`, and `finalize` functions are passed to `solve_problem` as input arguments. These three user-defined functions are independent of `solve_problem`.

As an example, assume that `x` is a set of n uniformly distributed coordinates in $[0, L]$, and we vary a and b in $[-1, 1]$ each with m values, while c is fixed at the value 5. For each combination of a and b, we call `func` with the vector `x` as a positional argument and the a, b, c values as keyword arguments, and store the evaluation results of `func` in a list named `output`. The objective of the computations is to extract the a and b values for which `func` gives a negative value for one or several of the coordinates $x \in [0, L]$. For this very simple example, the concrete implementation of the `initialize` and `finalize` functions can be put inside a class named `Parabola` as follows:

```
class Parabola:
    def __init__(self, m, n, L):
        self.m, self.n, self.L = m, n, L

    def initialize(self):
        x = numpy.linspace(0, self.L, self.n)
        a_values = numpy.linspace(-1, 1, self.m)
        b_values = numpy.linspace(-1, 1, self.m)
        c = 5

        self.input_args = []
        for a in a_values:
            for b in b_values:
                func_args = ([x], {'a': a, 'b': b, 'c': c})
                self.input_args.append(func_args)
        return self.input_args

    def func(self, x, a=0, b=0, c=1):
        return a*x**2+b*x+c

    def finalize(self, output_list):
        self.ab = []
        for input, result in zip(self.input_args, output_list):
            if min(result) < 0:
                self.ab.append((input[0][1], input[0][2]))
```

Now, to find the combinations of a and b values that make $ax^2 + bx + c < 9$, we can write the following two lines of code (assuming $m = 100$, $n = 50$, and $L = 10$):

```
problem = Parabola(100, 50, 10)
solve_problem(problem.initialize, problem.func, problem.finalize)
```

Note that the desired combinations of a and b values will be stored in the list `problem.ab`. Also note that we have placed `func` inside class `Parabola`, to have all

pieces of the problem in one place, but having `func` as stand-alone function or a class method is a matter of taste.

Despite the great mathematical simplicity of this example, the structure of the `solve_problem` function is directly applicable to a wide range of much more advanced problems. Although `initialize` and `finalize` are Python functions with very simple arguments (none and a list, respectively), this is not a limitation of their applicability. For example, the `initialize` step in our simple example needs values for m, n, and L, the a and b interval and so on, which can not be specified in the generic `solve_problem` function. To overcome this limitation, the information of m, n, and L can be hard-coded (not recommended), or transferred to `initialize` through global variables (not recommended in general) or carried with `initialize` as a state, either as class attributes or as a surrounding scope in a closure. We have chosen the class approach, i.e., class attributes store user-dependent data structures such that the `initialize` and `finalize` methods can have the simple input argument structure demanded by the generic `solve_problem` function. Alternatively, a closure as follows can be used instead of a class (this construct requires some knowledge of Python's scoping rules):

```
def initialize_wrapper(m, n, L):
    def initialize(self):
        x = numpy.linspace(0, L, n)
        a_values = numpy.linspace(-1, 1, m)
        ...
        return input_args
    return initialize
```

Now, the returned `initialize` function will carry with it the values of m, n, and L in the surrounding scope. The choice between the class approach and the closure approach, or using global variables in a straightforward global `initialize` function, is up to the programmer. The important point here is that `initialize` must often do a lot, and the input information to `initialize` must be handled by some Python construction. Similar comments apply to `finalize`.

2.2. Parallel Version

Let us say that we want to utilize several processors to share the work of all the `func` evaluations, i.e., the `for`-loop in the generic `solve_problem` function. This can clearly be achieved by a task-parallel approach, where each evaluation of `func` is an independent task. The main idea of parallelization is to split up the `for`-loop into a set of shorter `for`-loops, each assigned to a different processor. In other words, we need to split up the `input_args` list into a set of sub-lists for the different processors. Note that this partitioning work is generic, independent of both the `func` function and the actual arguments in the `input_args` list. Assuming homogeneous processors and that all the function evaluations are equally expensive, we can divide the `input_args` list into `num_procs` (number of processors) sub-lists of equal length. In case `input_args` is not divisible by `num_procs`, we adjust the length of some sub-lists by 1:

```
def simple_partitioning(length, num_procs):
```

```
    sublengths = [length/num_procs]*num_procs
    for i in range(length % num_procs):  # treatment of remainder
        sublengths[i] += 1
    return sublengths

def get_subproblem_input_args(input_args, my_rank, num_procs):
    sub_ns = simple_partitioning(len(input_args), num_procs)
    my_offset = sum(sub_ns[:my_rank])
    my_input_args = input_args[my_offset:my_offset+sub_ns[my_rank]]
    return my_input_args
```

Using the above generic `get_subproblem_input_args` function, each processor gets its portion of the global `input_args` list, and a shorter `for`-loop can be executed there. Note that the syntax of Python lists and `numpy` arrays has made the function very compact.

The next step of parallelization is to collect the function evaluation results from all the processors into a single global `output` list. Finally, we let `finalize(output)` run only on the master processor (assuming that this work does not require parallelization). For the purpose of collecting outputs from all the processors, the following generic Python function can be used:

```
def collect_subproblem_output_args(my_output_args, my_rank,
                                   num_procs, send_func, recv_func):
    if my_rank == 0:    # master process?
        output_args = my_output_args
        for i in range(1, num_procs):
            output_args += recv_func(i)
        return output_args
    else:
        send_func(my_output_args, 0)
        return None
```

The last two input arguments to the above function deserve some attention. Both `send_func` and `recv_func` are functions themselves. In the case of using the `pypar` wrapper of MPI commands, we may simply pass `pypar.send` as the `send_func` input argument and `pypar.receive` as `recv_func`. Moreover, switching to another MPI module is transparent with regard to the generic function named `collect_subproblem_output_args`. It should also be noted that most Python MPI modules are considerably more user-friendly than the original MPI commands in C/Fortran. This is because (1) the use of keyword arguments greatly simplifies the syntax, and (2) any picklable (marshalable) Python data type can be communicated directly.

Now that we have implemented the generic functions `get_subproblem_input_args` and `collect_subproblem_output_args`, we can write a minimalistic parallel solver as follows:

```
def parallel_solve_problem(initialize, func, finalize,
                           my_rank, num_procs, send, recv):
    input_args = initialize()
    my_input_args = get_subproblem_input_args(input_args,
                                              my_rank, num_procs)
    my_output = [func(*args, **kwargs) \
                 for args, kwargs in my_input_args]
```

```
output = collect_subproblem_output_args(my_output, my_rank,
                                        num_procs, send, recv)
if my_rank == 0:
    finalize(output)
```

We remark that the above function is generic in the sense that it is independent of the actual implementation of `initialize`, `func`, and `finalize`, as well as the Python MPI module being used. All problems that can be composed from independent function calls can (at least in principle) be parallelized by the shown small pieces of Python code.

As a specific example of using this parallel solver, we may address the problem of evaluating the parabolic function (`func` and class `Parabola`) for a large number of parameters. Using the `pypar` MPI module and having the problem-dependent code in a module named `Parabola` and the general function-centric tools in a module named `function_centric`, the program becomes as follows:

```
from Parabola import func, Parabola
from function_centric import parallel_solve_problem

problem = Parabola(m=100, n=50, L=10)
import pypar
my_rank = pypar.rank()
num_procs = pypar.size()
parallel_solve_problem(problem.initialize,
                       func,
                       problem.finalize,
                       my_rank, num_procs,
                       pypar.send, pypar.receive)
pypar.finalize()
```

To the reader, it should be obvious from this generic example how to parallelize other independent function calls by the described function-centric approach.

3. Function-Centric Parallelization

We have shown how to parallelize a serial program that is decomposable into three parts: `initialize`, calls to `func` (i.e., a set of independent tasks), and `finalize`. In this section, we describe how the function-centric parallelization is helpful for three important classes of scientific applications: Markov chain Monte Carlo simulations, dynamic population Monte Carlo simulations, and solution of partial differential equations. We use Python to program a set of simple and generic parallelization functions.

3.1. Parallel Markov chain Monte Carlo Simulations

The standard Markov chain Monte Carlo algorithms are embarrassingly parallel and have exactly the *same* algorithmic structure as the example of parameter analysis in Section 2. This means that the functions `initialize`, `func`, and `finalize` can easily be adapted to Monte Carlo problems. More specifically, the `initialize` function prepares the set of random samples and other input parameters. Some parametric model is

computed by the `func` function, whereas `finalize` collects the data returned from all the `func` calls and prepares for further statistical analysis.

Function-centric parallelization of Markov chain Monte Carlo applications closely follows the example in Section 2. We can reuse the three generic functions named `get_subproblem_input_args`, `collect_subproblem_output_args`, and `parallel_solve_problem`, assuming that all the `func` evaluations are equally costly and all the processors are equally powerful so there is no need for more sophisticated load balancing.

In Section 4.1, we will look at a real-life Markov chain problem from political science (Appendix Appendix A gives its mathematical description).

3.2. Population Monte Carlo with Dynamic Load Balancing

A more advanced branch of Monte Carlo algorithms is population Monte Carlo, see [26]. Here, a group of walkers, also called the *population*, is used to represent a high-dimensional vector and the computation is carried out by a random walk in the state space. During the computation some of these walkers may be duplicated or deleted according to some acceptance/rejection criteria, i.e., the population is dynamic in time. Population Monte Carlo algorithms have been proven useful in a number of fields, spanning from polymer science to statistical sciences, statistical physics, and quantum physics.

Unlike the examples so far, where the computational tasks were totally independent and of static size, population Monte Carlo algorithms may be viewed as an iteration in time where we repeatedly do some work on a dynamic population, including moving the walkers of the population and adjusting the population size, which in a parallel context calls for dynamic load balancing.

3.2.1. Serial Implementation A serial implementation of the time integration function can be as follows:

```
def time_integration(initialize, do_timestep, finalize):
    walkers, timesteps = initialize()
    output = []
    for step in range(timesteps):
        old_walkers_len = len(walkers)
        output.append(do_timestep(walkers))
        walkers.finalize_timestep(old_walkers_len, len(walkers))
    finalize(output)
```

The input arguments to the generic `time_integration` function are three functions: `initialize`, `do_timestep`, and `finalize`. This resembles the three-step structure discussed in Section 2. The `do_timestep` function can have a unified implementation for all the variants of population Monte Carlo algorithms. The other two input functions are typically programmed as methods of a class that implements a particular algorithm (such as diffusion Monte Carlo in Section 4.2). Here, the `initialize` method sets up a population object `walkers` (to be explained below) and determines the number of

time steps the walkers are to be propagated. The `finalize` method can, e.g., store the output for later analysis.

The purpose of the `do_timestep` function is to implement the work for one time step, including propagating the walkers and adjusting the population. An implementation that is applicable for all population Monte Carlo algorithms may have the following form:

```
def do_timestep(walkers):
    walkers.move()
    for walker in range(len(walkers)):
        if walkers.get_marker(walker) == 0:
            walkers.delete(walker)
        elif walkers.get_marker(walker) > 1:
            walkers.append(walker, walkers.get_marker(walker)-1)
    return walkers.sample_observables()
```

The above implementation of `time_integration` and `do_timestep` assumes that `walkers` is an object of a class, say with name `Walkers`, that has a certain number of methods. Of course, the flexibility of Python allows that the concrete implementation of class `Walkers` be made afterwards, unlike C++ and Java that require class `Walkers` be written before implementing `time_integration` and `do_timestep`. Here, we expect class `Walkers` to provide a generic implementation of a group of walkers, with supporting methods for manipulating the population. The most important methods of class `Walkers` are as follows:

- `move()` carries out the work of moving each walker of the population randomly according to some rule or distribution function.
- `get_marker(walker)` returns the number of copies belonging to a walker with index `walker`, where 0 means the walker should be deleted, 2 or more means that clones should be created.
- `append(walker, nchilds)` and `delete(walker)` carry out the actual cloning and removal of a walker with index `walker`.
- `sample_observables()` returns the observables at a given time step, e.g., an estimate of the system energy.
- `finalize_timestep(old_size, new_size)` does some internal book keeping at the end of each time step, such as adjusting some internal variables. It takes as input the total number of walkers before and after the walker population has been adjusted by the `do_timestep` function.
- `__len__` is one of Python's special class methods and is in our case meant to return the number of walkers. A call `len(walkers)` yields the same result as `walkers.__len__()`.

For a real application, such as the diffusion Monte Carlo algorithm (see Section 4.2 and Appendix Appendix B), the concrete implementation of the methods should reflect the desired numerical algorithm. For example, the `move` method of diffusion Monte Carlo uses diffusion and branching as the rule to randomly move each walker, and the `finalize_timestep` method adjusts the branching ratio.

3.2.2. Parallelization Parallelism in population Monte Carlo algorithms arises naturally from dividing the walkers among the processors. Therefore, a parallel version of the `time_integration` function may be as follows:

```
def parallel_time_integration(initialize, do_timestep, finalize,
                              my_rank, num_procs, send, recv, all_gather):
    my_walkers, timesteps = initialize(my_rank, num_procs)
    old_walkers_len = sum(all_gather(numpy.array([len(my_walkers)])))
    my_output = []
    for step in range(timesteps):
        # do what is required at this time step and measure CPU time
        t_start = time.time()
        results = do_timestep(my_walkers)
        my_output.append(results)
        task_time = time.time() - t_start

        # redistribute walkers and get walker size per process
        num_walkers_per_proc = dynamic_load_balancing(\
            my_walkers, task_time, my_rank, num_procs,\
            send, recv, all_gather)

        # finalize task for this time step
        new_walkers_len = sum(num_walkers_per_proc)
        my_walkers.finalize_timestep(old_walkers_len, new_walkers_len)
        old_walkers_len = new_walkers_len
    my_output = collect_subproblem_output_args(my_output, my_rank,
                                               num_procs, send, recv)
    if my_rank == 0:
        finalize(my_output)
```

In comparison with its serial counterpart, the `parallel_time_integration` function has a few noticeable changes. First, the input arguments have been extended with five new arguments. The two integers `my_rank` and `num_procs` are, as before, meant for identifying the individual processors and finding the total number of processors. The other three new input arguments are MPI communication wrapper functions: `send`, `recv`, and `all_gather`, which can be provided by any of the Python wrapper modules of MPI. The only exception is that `pypar` does not directly provide the `all_gather` function, but we can easily program it as follows:

```
def all_gather (input_array):
      array_gathered_tmp = pypar.gather (input_array, 0)
      array_gathered = pypar.broadcast (array_gathered_tmp, 0)
      return array_gathered
```

Second, we note that the `initialize` function is slightly different from the serial case, now accepting `my_rank` and `num_procs` as input. This is because initial division of the walkers is assumed to be carried out here, giving rise to `my_walkers` on each processor. Third, a new function `dynamic_load_balancing` is called during each time step. This function will be explained below in detail. Fourth, unlike that the serial counterpart could simply pass the size of its walkers to `finalize_timestep`, the parallel implementation needs to collect the global population size before calling `finalize_timestep`. We remark that each local population knows its own size, but not the global population size. For this purpose, the `dynamic_load_balancing` function returns the individual local population sizes as a `numpy` array. Last, the

`collect_subproblem_output_args` function from Section 2.2 is used to assemble all the individual results onto the master processor before calling the `finalize` function.

As mentioned before, parallelization of population Monte Carlo algorithms has to take into account that the total number of walkers changes with time. Dynamic redistribution of the walkers is therefore needed to avoid work load imbalance. The generic `dynamic_load_balancing` function is designed for this purpose, where we evaluate the amount of work for each processor and, if the work distribution is too skew, we move the excess walkers from a busy processor to a less busy one. The function first checks the distribution of local population sizes. If the difference between the smallest number of walkers and the largest number of walkers exceeds some predefined threshold, `dynamic_load_balancing` finds a better population distribution and redistributes the walkers:

```
def dynamic_load_balancing(walkers, task_time, my_rank, num_procs, \
                           send, recv, all_gather):
    walkers_per_proc = all_gather(numpy.array([len(walkers)]))
    if imbalance_rate(walkers_per_proc) > walkers.threshold_factor:
        timing_list = all_gather(numpy.array([task_time]))
        rebalanced_work = find_optimal_workload(timing_list,
                                                walkers_per_proc)
        walkers = redistribute_work(walkers,
                                    walkers_per_proc,
                                    rebalanced_work,
                                    my_rank, num_procs, send, recv)
    return walkers_per_proc
```

Two helper functions `find_optimal_workload` and `redistribute_work` are used in the above implementation. Here, `find_optimal_workload` finds the optimal distribution of work, based on how much time each local population has used. The `redistribute_work` function carries out the re-shuffling of walkers. A straightforward (but not optimal) implementation of these functions goes as follows:

```
def find_optimal_workload(timing_list, current_work_per_proc):
    total_work = sum(current_work_per_proc)
    C = total_work/sum(1./timing_list)
    tmp_rebalanced_work = C/timing_list
    rebalanced_work = numpy.array(tmp_rebalanced_work.tolist(),'i')
    remainders = tmp_rebalanced_work-rebalanced_work
    while sum(rebalanced_work) < total_work:
        maxarg = numpy.argmax(remainders)
        rebalanced_work[maxarg] += 1
        remainders[maxarg] = 0
    return rebalanced_work

def redistribute_work(my_walkers, work_per_proc, rebalanced_work,
                      my_rank, num_procs, send, recv):
    difference = work_per_proc-rebalanced_work
    diff_sort = numpy.argsort(difference)
    prev_rank_min = diff_sort[0]
    while sum(abs(difference)) != 0:
        diff_sort = numpy.argsort(difference)
        rank_max = diff_sort[-1]
        rank_min = diff_sort[0]
        if rank_min == prev_rank_min and rank_max != diff_sort[1]:
            rank_min = diff_sort[1]
```

```
        if my_rank==rank_max:
            send(my_walkers.cut_slice(rebalanced_work[my_rank]),\
                                      int(rank_min))
        elif my_rank==rank_min:
            my_walkers.paste_slice(recv(int(rank_max)))
        difference[rank_min] += difference[rank_max]
        difference[rank_max] = 0
        prev_rank_min = rank_min
    return my_walkers
```

Careful readers will notice that two particular methods, `my_walkers.cut_slice` and `my_walkers.paste_slices`, provide the capability of migrating the work load between processors in the `redistribute_work` function. These two methods have to be programmed in class `Walkers`, like the other needed methods described earlier: `move`, `get_marker`, `append`, `delete`, and so on. The `cut_slice` method takes away excess work from a local population and the `paste_slice` method inserts additional work into a local population. Note that the input argument to the `cut_slice` method is an index threshold meaning that local walkers with indices larger than that are to be taken away. The returned slice from `cut_slice` is a picklable Python object that can be sent and received through MPI calls.

The generic `redistribute_work` function deserves a few more words. Among its input arguments is the ideal work distribution, `rebalanced_work`, which is calculated by `find_optimal_workload`. The `redistribute_work` function first calculates the difference between the current distribution, `work_per_proc`, and the ideal distribution. It then iteratively moves walkers from the processor with the most work to the processor with the least work until the difference is evened out.

This load balancing scheme is in fact independent of population Monte Carlo algorithms. As long as you have an algorithm repeatedly doing a task over time and where the amount of work in the task varies over time, this scheme can be reused. The only requirement is that an application-specific implementation of class `Walkers`, in terms of method names and functionality, should match with `dynamic_load_balancing` and `redistribute_work`. It should be noted that the given implementation of the latter function is not optimal.

The algorithm of diffusion Monte Carlo, described in Appendix Appendix B, is a typical example of a population Monte Carlo algorithm. The implementation is described in Section 4.2 and Appendix Appendix B.

3.3. Parallel Additive Schwarz Iterations

From the perspective of communication between processors, parallelization of the Monte Carlo algorithms is relatively easy. Parallel Markov chain Monte Carlo algorithms only require communication in the very beginning and end, whereas parallel population Monte Carlo algorithms only require communication at the end of each time step. Actually, our function-centric approach to parallelization can allow more frequent communication. To show the versatility of function-centric parallelization, we apply

it to an implicit method for solving partial differential equations (PDEs) where communication is frequent between processors.

More specifically, many PDEs can be solved by an iterative process called *domain decomposition*. The idea is to divide the global domain, in which the PDEs are to be solved, into n overlapping subdomains. The PDEs can then be solved in parallel on the n subdomains. However, the correct boundary condition at the *internal* subdomain boundaries are not known, thus leading to an iterative approach where one applies boundary conditions from the last iteration, solves for the n subdomain problems again, and repeats the process until convergence of the subdomain solutions (see e.g. [27, 28]). This algorithm is commonly called additive Schwarz iteration and can successfully be applied to many important classes of PDEs [29, 30, 31]. The great advantage of the algorithm, especially from a software point of view, is that the PDE solver for the global problem can be reused for each subdomain problem. Some additional code is needed for communicating the solutions at the internal boundaries between the subdomains. This code can be implemented in a generic fashion in Python, as we explain later.

Let us first explain the additive Schwarz algorithm for solving PDEs in more detail. We consider some stationary PDE defined on a global domain Ω:

$$\mathcal{L}(u) = f, \quad x \in \Omega, \tag{1}$$

subject to some boundary condition involving u and/or its derivatives. Dividing Ω into a set of overlapping subdomains $\{\Omega_s\}_{s=1}^{P}$, we have the restriction of (1) onto Ω_s, for all s, as

$$\mathcal{L}(u) = f, \quad x \in \Omega_s. \tag{2}$$

The additive Schwarz method finds the global solution u by an iterative process that generates a series of approximations u_0, u_1, u_2 and so on. During iteration k, each subdomain computes an improved local solution $u_{s,k}$ by locally solving (2) for $u = u_{s,k}$ with $u_{s,k} = u_{k-1}$ as (an artificial) boundary condition on Ω_s's non-physical internal boundary that borders with neighboring subdomains. All the subdomains can *concurrently* carry out the local solution of (2) within iteration k, thus giving rise to parallelism. At the end of iteration k, neighboring subdomains exchange the latest local solutions in the overlapping regions to (logically) form the global field u_k. The subdomain problems (2) are of the same type as the global problem (1), which implies the possibility of reusing an existing serial code that was originally implemented for (1). The additional code for exchange of local solutions among neighbors can be implemented by generic communication operations, independently of specific PDEs.

A generic implementation of parallel additive Schwarz iteration algorithm can be realized as the following Python function:

```
def additive_Schwarz_iterations(subdomain_solve, communicate,
    set_BC, max_iter, threshold, solution,
    convergence_test=simple_convergence_test):
    iter = 0;  not_converged = True   # init

    while not_converged and iter < max_iter:
```

```
iter += 1
solution_prev = solution.copy()
set_BC(solution)
solution = subdomain_solve()
communicate(solution)
not_converged = not convergence_test(\
                solution, solution_prev, threshold)
```

In the above function, `max_iter` represents the maximum number of additive Schwarz iterations allowed, and `subdomain_solve` is a function that solves the subdomain problem of form (2) and returns an object `solution`, which is typically a `numpy` array containing the latest subdomain solution $u_{s,k}$ on a processor (subdomain). However, `solution` may well be a more complex object, say holding a collection of scalar fields over computational grids, provided that (i) the object has a `copy` method, (ii) `convergence_test` and `communicate` can work with this object type, and (iii) `subdomain_solve` returns such an object. This flexibility in choosing `solution` reflects the major dynamic power of Python and provides yet another illustration of the generality of the examples in this paper.

Given an existing serial code, for example in a language like Fortran or C/C++, the `subdomain_solve` function is easily defined by wrapping up an appropriate piece of the serial code as a Python class (since `subdomain_solve` does not take any arguments, the function needs a state with data structures, conveniently implemented as class attributes as explained in Section 2.1).

The `communicate` argument is a function for exchanging the latest local solutions among the subdomains. After the call, the `solution` object is updated with recently computed values from the neighboring subdomains, and contents of `solution` have been sent to the neighbors. The `communicate` function is problem independent and can be provided by some library. In our implementation, the implementation is entirely in Python to take advantage of easy programming of parallel communication in Python. The `set_BC` argument is a function for setting boundary conditions on a subdomain's internal boundary. This function depends on the actual serial code and is naturally implemented as part of the class that provides the `subdomain_solve` function.

The `convergence_test` function is assumed to perform an appropriate convergence test. The default generic implementation can test

$$\max_{1\leq s\leq P} \frac{\|u_{s,k} - u_{s,k-1}\|^2}{\|u_{s,k}\|^2}$$

against a prescribed threshold value. An implementation reads

```
def simple_convergence_test(solution, solution_prev, threshold=1E-3):
    diff = solution - solution_prev
    loc_rel_change = vdot(diff,diff)/vdot(solution,solution)
    glob_rel_change = all_reduce(loc_rel_change, MAX)
    return glob_rel_change < threshold
```

We remark that `all_reduce` is a wrapper of the MPI `MPI_Allreduce` command and `vdot` computes the inner product of two `numpy` arrays.

Unlike the three-component structure described in Sections 3.1 and 3.2, the main ingredients for parallel additive Schwarz iterations are the functions of `subdomain_solve`, `communicate`, `set_BC`, and `convergence_test`. In other words, it is not natural to divide the work of solving a PDE into `initialize`, `func`, and `finalize`. Nevertheless, function-centric parallelization is also here user-friendly and gives a straightforward implementation of `additive_Schwarz_iterations` as above. The `convergence_test` function shown above is clearly generic, and so is the `communicate` function in the sense that it does not depend on the PDE. Both functions can be reused for different PDEs. The other two functions are PDE dependent, however, `subdomain_solve` normally wraps an existing serial code, while the implementation of `set_BC` is typically very simple.

4. Applications and Numerical Experiments

In this section we will address three real research projects involving the three classes of algorithms covered in Section 3. The projects have utilized our function-centric approach to parallelizing existing codes. That is, we had some software in Fortran, C++, and R performing the basic computations needed in the projects. The serial software was wrapped in Python, adapted to components such as `initialize`, `func`, `do_timestep`, `finalize`, `subdomain_solve`, `communicate`, `set_BC`. Parallelization was then carried out as explained in previous sections. An important issue to be reported is the parallel efficiency obtained by performing the parallelization in a Python layer that is separate from the underlying serial scientific codes.

The Python enabled parallel codes have been tested on a Linux cluster of 3.4 GHz Itanium2 processors, which are interconnected through 1Gbits ethernet. The purpose is to show that the function-centric parallelization approach is easy to use and that satisfactory parallel performance is achievable.

4.1. Parallel Markov Chain Monte Carlo Simulations

The first case is from political science and concerns estimating legislators' ideal points by the Markov chain Monte Carlo (MCMC) method. For a detailed description of the mathematical problem and the numerical method, we refer the reader to Appendix Appendix A. This application fits into the setup in Section 3.1. The statistical engine is provided by the PSCL library [32] in R [33], for which there exists a Python wrapper.

To use the function-centric parallelization described in Section 3.1, we have written a Python class named `PIPE`. In addition to the constructor of the class (i.e., the `__init__` method), there are three methods as follows:

- `initialize` sets up the functionality of the PSCL library through the Python wrapper of R (named `rpy`), and prepares the input argument list needed for `func`.

- `func` carries out the computation of each task by invoking appropriate functions available through `rpy` (in short, `func` is a Python wrapper to the R function `ideal` from the PSCL library).
- `finalize` summarizes the output and generates an array in R format.

The resulting parallel Python program is now as short as

```
from function_centric import parallel_solve_problem
import pypar
my_rank = pypar.rank()
num_procs = pypar.size()

from pypipe import PIPE
problem = PIPE("EP1.RData", "rcvs", "NULL", "NULL")
parallel_solve_problem(problem.initialize, problem.func, problem.finalize,
                       my_rank, num_procs, pypar.send, pypar.receive)
pypar.finalize()
```

The practical importance of a parallel MCMC code is that large and computationally intensive simulations are now easily doable. More specifically, data from the European Parliament between 1979 and 2004 [34] are used for simulation. During the five year legislative terms, the European Parliament expanded the size of the membership as well as the number of votes taken. (This trend has continued since 2004.) It is hence increasingly computationally intensive to estimate the ideal point model without reducing the length of the Markov chain. We examined the parallel performance by comparing the computing time for each of the five legislatures, running the parallelized code on 32 CPUs. The results are reported in Table 1. When comparing the results, the reader should note that we have not made any attempts to optimize the `ideal` code (called by our `func` function) for the purpose of parallelization. This makes it straightforward to switch to new versions of the `ideal` function. We ran 100,000 MCMC iterations. The parallel efficiency was about 90%.

Table 1. Speedup results associated with voting analysis.

Legislature	Votes	Members	1 CPU	32 CPUs	Efficiency
1979 - 1984	810	548	287m 32.560s	10m 13.318s	87.91%
1984 - 1989	1853	637	783m 59.059s	26m 58.702s	91.06%
1989 - 1994	2475	597	1006m 59.258s	33m 26.140s	94.11%
1994 - 1999	3603	721	1905m 0.930s	66m 0.068s	90.20%
1999 - 2004	5639	696	2898m 45.224s	102m 7.786s	88.70%

4.2. Parallel Diffusion Monte Carlo Simulations

As an example of population Monte Carlo methods, we will now look at parallel Diffusion Monte Carlo (DMC) computations (see Appendix Appendix B for a detailed numerical description), which is used here to simulate Bose-Einstein condensation. We recall from Section 3.2 that dynamic load balancing is needed in connection with the parallelization,

and can be provided by the generic `dynamic_load_balancing` function. To utilize the parallel time integration function `parallel_time_integration` from Section 3.2, we need to program a parallel version of the `initialize` function. The `do_timestep` function from Section 3.2 can be used as is.

```
def initialize(my_rank, num_procs):
    nwalkers = 1000
    nspacedim = 3
    stepsize = 0.1
    timesteps = 200
    walkers_per_proc = simple_partitioning(nwalkers, num_procs)
    my_walkers = Walkers(walkers_per_proc[my_rank], nspacedim, stepsize)
    my_walkers.threshold_factor = 1.1
    return my_walkers, timesteps
```

This `initialize` function is quite similar to its serial counter part. As noted in Section 3.2, it takes as input `my_rank` and `num_procs`. The `simple_partitioning` function is called to partition the walker population. A `my_walkers` object is assigned to each processor, and a threshold factor is prescribed to determine when load balancing is needed.

Together with the `parallel_time_integration` function from Section 3.2, the above `initialize` function is the minimum programming effort needed to parallelize a serial population Monte Carlo code. For the particular case of our parallel Diffusion Monte Carlo implementation, we also need to know the global number of walkers in every timestep to be able to estimate its observables globally. Moreover, the load balancing scheme needs the time usage of each processor during each time step.

A class with name `Walkers` needs to be implemented to match with the implementations of `parallel_time_integration`, `dynamic_load_balancing`, and the above `initialize` function. The essential work is to provide a set of methods with already decided names (see Section 3.2), such as `move`, `append`, `delete`, `finalize_timestep`, `cut_slice`, and `paste_slice`. A concrete example of the `Walkers` class is described with more details in Appendix Appendix B.

We report in Table 2 the timing results of a number of parallel DMC computations. The parallel efficiency was about 85%. We increased the total number of walkers when more processors were used in the simulation, such that the number of walkers assigned to each processor remained as 200. Such a use of parallel computers for DMC simulations mimics the everlasting wish of quantum physicists to do larger computations as soon as more computing resource becomes available. Note that in this scaled scalability test, good speedup performance is indicated by an almost constant time usage independent of the number of processors.

4.3. Parallel Boussinesq Simulations

Simulating the propagation of ocean waves is the target of the our third and final concrete case. The reader is referred to Appendix Appendix C for the mathematical model and the numerical method. The involved equations can be solved in parallel by the additive Schwarz algorithm of Section 3.3.

Table 2. Timing results of the parallel DMC simulations where each processor is constantly assigned with 200 walkers, all moved in 5000 time steps.

CPUs	Time	Efficiency
1	37m10.389s	N/A
5	42m32.359s	87.39%
10	42m00.734s	88.48%
20	42m29.945s	87.47%
30	42m33.895s	87.33%
40	43m30.092s	85.45%
50	43m39.159s	85.16%

Our starting point for parallelization is a 25 years old legacy Fortran 77 code consisting of a set of subroutines. More specifically, the most important subroutines are `KONTIT` and `BERIT`, which target the two semi-discretized equations (C.3) and (C.4) of the mathematical model (see Appendix Appendix C). These two Fortran 77 subroutines contain intricate algorithms with nested layers of do-loops, which are considered to be very difficult to parallelize by directly inserting MPI calls in the Fortran code. Performing the parallelization outside the Fortran code is therefore much more convenient. Using the proposed framework in the present paper the parallelization is a technically quite straightforward task.

The subdomain solver consists of calls to the subroutines `KONTIT` and `BERIT`. The implementation of the Python function `subdomain_solve` (see Section 3.3) requires a Python interface to `KONTIT` and `BERIT`, which can easily be produced by the F2PY software. Since a subdomain solver needs to set artificial boundary conditions at non-physical boundaries, we have programmed two light-weight wrapper subroutines in Fortran, `WKONTIT` and `WBERIT`, which handles the boundary conditions before invoking `KONTIT` and `BERIT`. We then apply F2PY to make `WKONTIT` and `WBERIT` callable from Python. Since the Fortran subroutines have lots of input data in long argument lists and `subdomain_solve` takes no arguments, we have created a class where the Fortran input variables are stored as class attributes:

```
import f77  # extension module for the Fortran code

class SubdomainSolver:
    def __init__(self, ...):
        # set input arguments to the Fortran subroutines as class attributes
        # (nbit,F,YW,H,QY,WRK,dx,dy,dt,kit,ik,gg,alpha,eps,
        # lower_x_neigh,upper_x_neigh,lower_y_neigh,upper_y_neigh)

    def continuity(self):
        self.Y, self.nbit = f77.WKONTIT(\
          self.F, self.Y, self.YW, self.H, self.QY, self.WRK,
          self.dx, self.dy, self.dt, self.kit, self.ik,
          self.gg, self.alpha, self.eps, self.nbit,
          self.lower_x_neigh, self.upper_x_neigh,
          self.lower_y_neigh, self.upper_y_neigh)
```

```
    def bernoulli(self):
        # similar to the continuity method

sd = SubdomainSolver(...)
subdomain_solve1 = sd.continuity
subdomain_solve2 = sd.bernoulli
```

Note that since there are two PDEs (C.3) and (C.4), we have created two functions: `subdomain_solve1` and `subdomain_solve2`. The main computation of the resulting parallel program is in the following while loop:

```
t = 0
while t < t_stop:
    t = t+dt
    additive_Schwarz_iterations(subdomain_solve1, communicate,
                                set_BC, 10, 1E-3, sd.Y)
    additive_Schwarz_iterations(subdomain_solve2, communicate,
                                set_BC, 10, 1E-3, sd.F)
```

The `additive_Schwarz_iterations` function from Section 3.3 can be placed in a reusable module. The `communicate` function is borrowed from a Python library for mesh partitioning and inter-subdomain communication. The `set_BC` function actually does not do anything for this particular application.

Speedup results are reported in Table 3, for which the global solution mesh was fixed at 1000×1000, and the number of time steps was 40. The results show that we can handle a quite complicated mathematical problem in a black-box Fortran code with our suggested simple framework and obtain a remarkable good speedup, with just a trivial extension of the Fortran code.

Table 3. The speedup results of the Python enabled parallel Boussinesq simulations.

CPUs	Time	Speedup	Efficiency
1	166.66s	N/A	N/A
2	83.61s	1.99	99.67%
4	44.45s	3.75	93.73%
8	20.16s	8.27	103.33%
16	11.43s	14.58	91.13%

5. Conclusion

We have shown how serial scientific codes written in various common languages, including Fortran, C, C++, and Python, can be parallelized in a separate, small software unit written in Python. The advantage of such an approach is twofold. First, the existing, often complicated, scientific high-performance code remains (almost) unchanged. Second, the parallel algorithm and its inter-processor communication are conveniently implemented in high-level Python code.

This approach to parallelization has been implemented in a software framework where the programmer needs to implement a few Python functions for carrying out the

key steps in the solution approach. For example, our first application involves doing a set of independent tasks in parallel, where a small Python framework deals with the parallelism and demands the user to supply three functions: `initialize` for preparing input data to the mathematical model, `func` for calling up the serial scientific code, and `finalize` for processing the computational results. Some more functions must be supplied in more complicated problems where the algorithm evolves in time, with a need for dynamic load balancing and more parallel communication.

Our simple software frameworks outlined in this paper are applicable to many different scientific areas, and we have described some common classes of problems: parameter investigation of a mathematical model, standard Monte Carlo simulation, Monte Carlo simulation with need for dynamic load balancing, and numerical solution of partial differential equations. In each of these cases, we have outlined fairly detailed Python code such that most technical details of the parallel implementations are documented. This may ease the migration of the ideas to new classes of problems beyond the scope of this paper.

In particular, the shown frameworks have been used to parallelize three real scientific problems taken from our research. The problems concern Markov Chain Monte Carlo models for voting behavior in political science, Diffusion Monte Carlo methods for Bose-Einstein condensation in quantum mechanics, and additive Schwarz and finite difference methods for simulating ocean waves by a system of partial differential equations. The results of our investigations of the parallel efficiency are very encouraging: In all these real science problems, parallelizing serial codes in the proposed Python framework gives almost optimal speedup results, showing that there arises no significant loss due to using Python and performing the parallelization "outside" the serial codes.

As a conclusion, we believe that the ideas and code samples from this paper can simplify parallelization of serial codes greatly, without significant loss of computational efficiency. This is good news for scientists who are non-experts in parallel programming but want to parallelize their serial codes with as small efforts as possible.

Appendix A. Voting in Legislatures

In the spatial model of politics, both actors' preferences over policies (ideal points) and policy alternatives are arranged geometrically in a low-dimensional Euclidean space. An actor receives the highest possible utility if a policy is located at her ideal point; she gains or loses utility as the policy moves towards or away from her ideal point [35]. We adopt the Bayesian approach proposed by Clinton, Jackman and Rivers [36]. Assume there are n legislators who vote on m proposals. On each vote $j = 1, \ldots, m$, legislator $i = 1, \ldots, n$ chooses between a "Yea" position ζ_j and a "Nay" position ψ_j located in the policy-space $\mathbb{R}^d$, where d is the number of dimensions. Then, we have $y_{ij} = 1$ if legislator i votes "Yea" on roll call j, and $y_{ij} = 0$ if she votes "Nay". The model assumes quadratic utility functions. The ideal point of legislator i is $x_i \in \mathbb{R}$, while η_{ij} and υ_{ij} are stochastic elements whose distribution is jointly normal. The variance of the stochastic elements is $(\eta_{ij} - \upsilon_{ij}) = \sigma_j^2$. Denote the Euclidean norm by $\| \cdot \|$, utility maximising implies that legislator i votes "Yea" on vote j if

$$U_i(\zeta_j) = -\|x_i - \zeta_j\|^2 + \eta_{ij} > U_i(\psi_j) = -\|x_i - \psi_j\|^2 + \upsilon_{ij} \tag{A.1}$$

and "Nay" otherwise. Clinton, Jackman and Rivers [36] show that the model can be understood as a hierarchical probit model:

$$P(y_{ij} = 1) = \Phi(\beta_j' \mathbf{x}_i - \alpha_j), \tag{A.2}$$

where β_j=$2(\zeta_j - \psi_j)/\sigma_j$, α_j=$(\zeta_j'\zeta_j - \psi_j'\psi_j)/\sigma_j$, $\Phi(\cdot)$ is the standard normal function, β_j is the midpoint between the "Yea" and "Nay" positions on proposal j, and $\mathbf{x}_i$ is the legislator's ideal point. The direction of α_j indicates the location of the status quo relative to the proposal. If α_j is positive, the new proposal is located higher on the dimension than the status quo. If α_j is negative, the new proposal is located lower on the dimension than the status quo.

The MCMC Algorithm. In the Markov Chain Monte Carlo (MCMC) algorithm for the statistical analysis of voting behavior [36], the difference between utilities of the alternatives on the jth vote for the ith legislator is given by $y_{ij}^* = \beta_j \mathbf{x}_i - \alpha_j + \epsilon_{ij}$, where β_j and α_j are model parameters, $\mathbf{x}_i$ is a vector of regression coefficients and ϵ_{ij} are standard normal errors. If we know β_j and α_j, $\mathbf{x}_i$ can be imputed from the regression of $y_{ij}^* + \alpha_j$ on β_j using the m votes of legislator i and vice versa. If we know $\mathbf{x}_i$, we can use the votes of the n legislators on roll call j to find β_j and α_j. Given $\mathbf{x}_i$, β_j and α_j (either from priors or from the previous iteration), we can find y_{ij}^* by drawing ϵ_{ij} randomly from a normal distribution subject to the constraints implied by the actual votes, i.e., if $y_{ij} = 0$, $y_{ij}^* < 0$ and if $y_{ij} = 1$, $y_{ij}^* > 0$.

The goal is to compute the joint posterior density for all model parameters β_j and α_j, $j = 1, \ldots, m$ and all coefficient vectors x_i, $i = 1, \ldots, n$. The MCMC algorithm forms a Markov chain to explore as much as possible of this joint density, i.e., letting t index an MCMC iteration,

(i) find $y_{ij}^{*,t}$ from $\mathbf{x}_i^{t-1}$, β_j^{t-1} and α_j^{t-1},

(ii) sample β_j^t and α_j^t using $\mathbf{x}_i^{t-1}$ and $y_{ij}^{*,t}$,
(iii) find $\mathbf{x}_i^t$ from β_j^t, α_j^t and $y_{ij}^{*,t}$.

This process must then be repeated until convergence, i.e., that the samples have moved away from the priors to the neighborhood of the posterior mode before samples are drawn.

Clinton, Jackman and Rivers [36, p. 369] find that the computation time is increasing in nmT, where n is the number of legislators, m is the number of votes and T is the number of MCMC iterations. Although they argue that very long runs are normally not necessary, they nevertheless advise long runs to ensure that the MCMC algorithm has converged. It is increasingly time-consuming to estimate the model on on a standard desktop computer as the size of the legislature and the number of votes increase.

Appendix B. Bose-Einstein Condensation

The famous experiment of Anderson et al. [37] was about cooling 4×10^6 ^{87}Rb down to temperatures in the order of $100\,nK$ for observing Bose-Einstein condensation in the dilute gas. To model this fascinating experiment in the framework of Quantum Monte Carlo, so that numerical simulations can be extended beyond the physical experiments, we may use the governing Schrödinger equation:

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{R},t) = H\Psi(\mathbf{R},t). \tag{B.1}$$

The most important parts of the mathematical model are a Hamiltonian H and a wave function Ψ, see [38]. The Hamiltonian for N trapped interacting atoms is given by

$$H = -\frac{\hbar^2}{2m}\sum_{i=1}^{N}\nabla_i^2 + \sum_{i=1}^{N}V_{ext}(\mathbf{r}_i) + \sum_{i<j}^{N}V_{int}(|\mathbf{r}_i - \mathbf{r}_j|). \tag{B.2}$$

The external potential V_{ext} corresponds to the trap used to confine the ^{87}Rb atoms, and was in the experiment in the order of $\mathbf{r}^2$. The two-body interaction $V_{int}(|\mathbf{r}_i - \mathbf{r}_j|)$ can be easily described by a hard-core potential of radius a in a dilute gas. We have however, for the sake of simplicity, neglected these interactions in our example implementation of class `Walkers`.

The Method of Diffusion Monte Carlo. In the Diffusion Monte Carlo (DMC) method [39], the Schrödinger equation is solved in imaginary time,

$$-\frac{\partial d\psi(\mathbf{R},t)}{\partial t} = [H - E]\psi(\mathbf{R},t). \tag{B.3}$$

The formal solution of (B.3) is

$$\psi(\mathbf{R},t) = e^{-[H-E]t}\psi(\mathbf{R},0), \tag{B.4}$$

where $e^{[-(H-E)t]}$ is called the *Green's function*, and E is a convenient energy shift.

The wave function $\psi(\mathbf{R}, t)$ in DMC is represented by a set of random vectors $\{R_1, R_2, \ldots, R_M\}$, in such a form that the time evolution of the wave function is actually represented by the evolution of a set of walkers. This feature gives rise to task parallelism. The wave function is positive definite everywhere, as it happens with the ground state of a bosonic system, so it may be considered as a probability distribution function.

The DMC method involves Monte Carlo integration of the Green's function by every walker. The time evolution is done in small time steps τ, using the following approximate form of the Green's function:

$$e^{-[H-E]t} = \prod_{i=1}^{n} e^{-[H-E]\tau}, \tag{B.5}$$

where $\tau = t/n$. Assume that an arbitrary starting state can be expanded in the basis of stationary,

$$\psi(\mathbf{R}, 0) = \sum_{\nu} C_\nu \phi_\nu(\mathbf{R}), \tag{B.6}$$

we have

$$\psi(\mathbf{R}, t) = \sum_{\nu} e^{-[E_\nu - E]t} C_\nu \phi_\nu(\mathbf{R}), \tag{B.7}$$

in such a way that the lowest energy components will have the largest amplitudes after a long elapsed time, and in the limit of $t \to \infty$ the most important amplitude will correspond to the ground state (if $C_0 \neq 0$)‡.

The Green's function is approximated by splitting it up in a diffusional part,

$$G_D = (4\pi D\tau)^{-3N/2} \exp\{-(\mathbf{R}' - \mathbf{R})^2/4D\tau\}, \tag{B.8}$$

which has the form of a Gaussian and a branching part,

$$G_B = \exp\{-((V(\mathbf{R}) + V(\mathbf{R}'))/2 - E_T)\tau\}. \tag{B.9}$$

While diffusion is taken care of by a Gaussian random distribution, the branching is simulated by creation and destruction of walkers with a probability G_B. The idea of DMC computation is quite simple; once we have found an appropriate approximation of the short-time Green's function and determined a starting state, the computation consists in representing the starting state by a collection of walkers and letting them independently evolve in time. That is, we keep updating the walker population, until a large enough time when all other states than the ground state are negligible.

‡ This can easily be seen by replacing E with the ground state energy E_0 in (B.7). As E_0 is the lowest energy, we will get $\lim_{t\to\infty} \sum_\nu \exp[-(E_\nu - E_0)t]\phi_\nu = C_0\phi_0$.

Algorithm 1 Diffusion Monte Carlo

for `step` **in range(** `0`, `timesteps` **):**
  for i **in range(** `0`, N_{walkers} **):**
    Diffusion;
      propose move $\mathbf{R}' = \mathbf{R} + \xi$
    Branching;
      calculate replication factor n:
        $n = int(\exp\{-((V(\mathbf{R}) + V(\mathbf{R}'))/2 - E_T)\tau\})$
      if $n = 0$**:**
        mark walker as dying
      if $n > 0$**:**
        mark walker to make $n - 1$ clones
    Remove dead walkers and make new clones;
    Update walker population N_{walkers} and adjust trial energy;
    Sample contributions to observable.

The Implementation. In Algorithm 1 we summarize the DMC algorithm corresponding to (B.8)-(B.9). In the algorithm ξ is a Gaussian with zero mean and a variance of $2D\tau$ corresponding to (B.8). The deleting and cloning of walkers are, as mentioned in Section 3.2, performed by the `do_timestep` function, repeated here for clarity:

```
def do_timestep(walkers):
    walkers.move()
    for walker in range(len(walkers)):
        if walkers.get_marker(walker) == 0:
            walkers.delete(walker)
        elif walkers.get_marker(walker) > 1:
            walkers.append(walker, walkers.get_marker(walker)-1)
    return walkers.sample_observables()
```

The main computational work of the DMC algorithm at each time step is implemented in the `move` function inside class `Walkers`, together with a helper function `branching`:

```
class Walkers:
    ...

    def branching(self, new_positions):
        old_potential = potential(self.positions)
        new_potential = potential(new_positions)
        branch = numpy.exp(-(0.5 * (old_potential + new_potential)
                         - self.adjust_branching) * self.stepsize)
        self.markers = numpy.array(branch+
                       numpy.random.uniform(0,1,branch.shape), 'i')

    def move(self):
        displacements = numpy.random.normal(0, 2*self.stepsize,
                                            self.positions.shape)
        new_positions = self.positions+displacements
        self.branching(new_positions)
```

```
            self.positions = new_positions
        ...
```

The `move` function first generates a set of Gaussian (normal) distributed random numbers, corresponding to (B.8). Next, it calls the `branching` function, which calculates a potential $V(\mathbf{r}) = \mathbf{r}^2$ for the old and the new positions§. These potentials are used to calculate G_B following (B.9) and create an integer array `self.markers` with its average value equal to G_B (stored in the `branch` variable). This array is of the same length as the number of walkers (stored in `self.positions`) and marks the walkers as dying or clone-able.

It is worth noticing that if the new potential of a walker is much higher than that in the previous time step (i.e., the walker is far from the center of the trap), the value of `branch` will be close to 0 and the walker will be deleted. However, if the new potential is much lower (i.e. closer to the center of the trap), `branch` will be greater than 1 and the walker will be cloned. As long as the two-body interaction is ignored, the walkers will only be encouraged to move towards the center of the trap, thus yielding a lower energy than seen in real experiments.

Appendix C. Ocean Wave Propagation

The following two PDEs, normally termed as the Boussinesq water wave equations [40], can be used to model wave propagation:

$$\frac{\partial \eta}{\partial t} + \nabla \cdot \left((H + \alpha\eta)\nabla\phi + \epsilon H \left(\frac{1}{6}\frac{\partial \eta}{\partial t} - \frac{1}{3}\nabla H \cdot \nabla\phi \right) \nabla H \right) = 0, \qquad \text{(C.1)}$$

$$\frac{\partial \phi}{\partial t} + \frac{\alpha}{2}\nabla\phi \cdot \nabla\phi + \eta - \frac{\epsilon}{2} H \nabla \cdot \left(H \nabla \frac{\partial \phi}{\partial t} \right) + \frac{\epsilon}{6} H^2 \nabla^2 \frac{\partial \phi}{\partial t} = 0. \qquad \text{(C.2)}$$

The primary unknowns of (C.1)-(C.2) are the water surface elevation $\eta(x, y, t)$ and the depth-averaged velocity potential $\phi(x, y, t)$. The symbol H denotes the water depth as a function of (x, y). The advantage of the above Boussinesq wave model, in comparison with the standard shallow water equations, is its capability of modeling waves that are weakly dispersive ($\epsilon > 0$) and/or weakly nonlinear ($\alpha > 0$), see [41]. Therefore, the Boussinesq water wave equations are particularly adequate for simulating ocean wave propagation over long distances and large water depths.

Discretization of the Boussinesq water wave equations (C.1)-(C.2) normally starts with a temporal discretization as follows:

$$\frac{\eta^\ell - \eta^{\ell-1}}{\Delta t} + \nabla \cdot \left(\left(H + \alpha \frac{\eta^{\ell-1} + \eta^\ell}{2} \right) \nabla\phi^{\ell-1} + \epsilon H \left(\frac{1}{6}\frac{\eta^\ell - \eta^{\ell-1}}{\Delta t} - \frac{1}{3}\nabla H \cdot \nabla\phi^{\ell-1} \right) \nabla H \right) = 0, \qquad \text{(C.3)}$$

§ In a more optimized implementation, the old potential would have been stored from the previous move and not calculated every time.

$$\frac{\phi^\ell - \phi^{\ell-1}}{\Delta t} + \frac{\alpha}{2}\nabla\phi^{\ell-1}\cdot\nabla\phi^{\ell-1} + \eta^\ell - \\ \frac{\epsilon}{2}H\nabla\cdot\left(H\nabla\left(\frac{\phi^\ell - \phi^{\ell-1}}{\Delta t}\right)\right) + \frac{\epsilon}{6}H^2\nabla^2\left(\frac{\phi^\ell - \phi^{\ell-1}}{\Delta t}\right) = 0, \tag{C.4}$$

where we use ℓ to denote the time level, and Δt denotes the time step size. The basic idea of computation at each time step is to first compute η^ℓ based on $\eta^{\ell-1}$ and $\phi^{\ell-1}$ from the previous time step, and then compute ϕ^ℓ using the new η^ℓ and the old $\phi^{\ell-1}$. To carry out the actual numerical computation, we need a spatial discretization of (C.3)-(C.4), using e.g. finite differences or finite elements, so we end up with two systems of linear equations that need to be solved during each time step.

References

[1] Guido van Rossum et al. The Python programming language, 1991–. `http://www.python.org/`.

[2] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, and T. Oliphant. Numerical Python. Technical report, Lawrence Livermore National Lab., CA, 2001. `http://www.pfdubois.com/numpy/numpy.pdf`.

[3] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. `http://www.scipy.org/`, 2001–.

[4] F2PY software package. `http://cens.ioc.ee/projects/f2py2e`.

[5] Pypar – parallel programming with Python. `http://sourceforge.net/projects/pypar`, 2007.

[6] X. Cai, H. P. Langtangen, and H. Moe. On the performance of the Python programming language for serial and parallel scientific computations. *Scientific Programming*, 13(1):31–56, 2005.

[7] X. Cai and H. P. Langtangen. Developing parallel object-oriented simulation codes in Diffpack. In *Proceedings of the Fifth World Congress on Computational Mechanics*, Vienna University of Technology, 2002. ISBN 3-9501554-0-6.

[8] F. Cirak and J. C. Cummings. Generic programming techniques for parallelizing and extending procedural finite element programs. *Engineering with Computers*, 24:1–16, 2008.

[9] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2001.

[10] O. Lawlor, S. Chakravorty, T. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L. Kalé. ParFUM: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22(3):215–235, 2006.

[11] J. R. Stewart and H. C. Edwards. A framework approach for developing parallel adaptive multiphysics applications. *Finite Elements in Analysis and Design*, 40:1599–1617, 2004.

[12] The Trilinos project. `http://trilinos.sandia.gov/`, 2008.

[13] UG home page. `http://sit.iwr.uni-heidelberg.de/~ug/`, 2007.

[14] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, , and J. Shalf. The Cactus framework and toolkit: Design and applications. In J. M. L. M. Palma et al., editor, *Proceedings of VECPAR 2002*, volume 2565 of *Lectures Notes in Computer Science*, pages 197–227. Springer Verlag, 2003.

[15] MpCCI 3.0. `http://www.mpcci.de/`, 2008.

[16] Star-P Overview.
`http://www.interactivesupercomputing.com/products/`, 2008.

[17] K. Hinsen. Parallel scripting with Python. *Computing in Science & Engineering*, 9(6):82–89, 2007.

[18] pyMPI: Putting the py in MPI.
`http://pympi.sourceforge.net/`, 2008.

[19] MYMPI webpage.
`http://peloton.sdsc.edu/~tkaiser/mympi/`, 2008.

[20] L. Dalcín, R. Paz, and M. Storti. MPI for Python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115, 2005.

[21] L. Dalcín, R. Paz, M. Storti, and J. D'Elía. MPI for Python: Performance improvements and MPI-2 extensions. *Journal of Parallel and Distributed Computing*, 68(5):655–662, 2008.

[22] ScientificPython webpage.
`http://dirac.cnrs-orleans.fr/plone/software/scientificpython/`, 2007.

[23] G. D. Benson and A. S. Fedosov. Python-based distributed programming with Trickle. In H. R. Arabnia, editor, *Proceedings of PDPTA'07*, pages 30–36. CSREA Press, 2007.

[24] G. Olson. Introduction to concurrent programming with Stackless Python.
`http://members.verizon.net/olsongt/stackless/why_stackless.html`, 2006.

[25] C. E. Rasmussen, M. J. Sottile, J. Nieplocha, R. W. Numrich, and E. Jones. Co-array Python: A parallel extension to the Python language. In M. Danelutto, D. Laforenza, and M. Vanneschi, editors, *Proceedings of Euro-Par 2004*, Lectures Notes in Computer Science, pages 632–637. Spriner Verlag, 2004.

[26] Yukito IBA. Population Monte Carlo algorithms. *Transactions of the Japanese Society for Artificial Intelligence*, 16:279–286, 2001.

[27] T. F. Chan and T. P. Mathew. Domain decomposition algorithms. In *Acta Numerica 1994*, pages 61–143. Cambridge University Press, 1994.

[28] B. F. Smith, P. E. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

[29] X. Cai, G. K. Pedersen, and H. P. Langtangen. A parallel multi-subdomain strategy for solving Boussinesq water wave equations. *Advances in Water Resources*, 28(3):215–233, 2005.

[30] X. Cai and H. P. Langtangen. Parallelizing PDE solvers using the Python programming language. In A. M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, pages 295–325. Springer-Verlag, 2006.

[31] H. P. Langtangen and X. Cai. A software framework for easy parallelization of PDE solvers. In C. B. Jensen, T. Kvamsdal, H. I. Andersson, B. Pettersen, A. Ecer, J. Periaux, N. Satofuka, and P. Fox, editors, *Parallel Computational Fluid Dynamics*. North-Holland, 2001.

[32] Simon Jackman. PSCL: classes and methods for R developed in the Political Science Computational Laboratory, Stanford University. Technical report, Department of Political Science, Standford University, 2006.

[33] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. ISBN 3-900051-07-0.

[34] Simon Hix, Abdul Noury, and Gerard Roland. Power to the parties: cohesion and competition in the European Parliament, 1979-2001. *British Journal of Political Science*, 35(2):209–234, 2005.

[35] Melvin J. Hinich and Michael C. Munger. *Analytical Politics*. Cambridge University Press, 1997.

[36] Joshua Clinton, Simon Jackman, and Doug Rivers. The statistical analysis of roll call data. *American Political Science Review*, 98(4):355–370, 2004.

[37] J.R. Anderson, M.R. Ensher, M.R. Matthews, C.E. Wieman, and E.A. Cornel. Observation of Bose-Einstein condensation in a dilute atomic vapor. *Science*, 269:198, 1995.

[38] J. K. Nilsen, J. Mur-Petit, M. Guilleumas, M. Hjorth-Jensen, and A. Polls. Vortices in atomic Bose-Einstein condensates in the large gas parameter region. *Phys. Rev. A*, 71, 2005.

[39] R. Guardiola. Monte Carlo methods in quantum many-body theories. In J. Navarro and A. Polls, editors, *Microscopic Quantum Many-Body Theories and Their Applications*, volume 510 of *Lecture Notes in Physics*, pages 269–336. Springer Verlag, 1998.

[40] D. M. Wu and T. Y. Wu. Three-dimensional nonlinear long waves due to moving surface pressure. *Proc. 14th Symp. Naval Hydrodyn.*, pages 103–129, 1982.

[41] G. Pedersen and H. P. Langtangen. Dispersive effects on tsunamis. In *Proceedings of the International Conferance on Tsunamis, Paris, France*, pages 325–340, 1999.

A.4 ARC middleware: evolution towards standards-based interoperability

Article submitted to Journal of Physics, Conference Series.

The article "ARC middleware: evolution towards standards-based interoperability" is based on the efforts of the KnowARC development team. It presents an overview of ARC and the new components to be included in the production release of ARC.

My main contributions to this article has been to develop parts of the code described in Section 4.5, in particular the replicated A-Hash, the FUSE module and the ARC DMC.

ARC middleware: evolution towards standards-based interoperability

O Smirnova[1,2], D Cameron[1,3], P Dóbé[2,4], M Ellert[1,5], T Frågåt[3], M Grønager[1], D Johansson[1,6], J Jönemo[2,5], J Kleist[1,7], M Kočan[8], A Konstantinov[3,9], B Kónya[2], I Márton[4], S Möller[10], B Mohn[5], Zs Nagy[4], J K Nilsen[3], F Ould Saada[3], W Qiang[3], A Read[3], P Rosendahl[11], G Rőczei[4], M Savko[8], M Skou Andersen[11], P Stefán[4], F Szalai[4], A Taga[3], S Z Toor[12] and A Wäänänen[11]

[1] NDGF, Kastruplundsgade 22, DK-2770 Kastrup, Denmark
[2] Lund University, Experimental High Energy Physics, Institute of Physics, Box 118, SE-22100 Lund, Sweden
[3] University of Oslo, Dept. of Physics, P. O. Box 1048, Blindern, N-0316 Oslo, Norway
[4] NIIF/HUNGARNET, Victor Hugo 18-22, H-1132 Budapest, Hungary
[5] Uppsala University, Dept. of Physics and Astronomy, Div. of Nuclear and Particle Physics, Box 535, SE-75121 Uppsala, Sweden
[6] Linköping University, National Supercomputer Centre, SE-581 83 Linköping, Sweden
[7] Aalborg University, Dept. of Computer Science, Frederik Bajersvej 7E, DK-9220 Aalborg Ø, Denmark
[8] Pavol Jozef Šafárik University, Faculty of Science, Jesenná 5, SK-04000 Košice, Slovak Republic
[9] Vilnius University, Institute of Material Science and Applied Research, Saulėtekio al. 9, Vilnius 2040, Lithuania
[10] University of Lübeck, Inst. Of Neuro- and Bioinformatics, Ratzeburger Allee 160, D-23538 Lübeck, Germany
[11] University of Copenhagen, NBI, Blegdamsvej 17, DK-2100 Copenhagen Ø, Denmark
[12] Uppsala University, Dept. of Information Technology, Box 337, SE-75105 Uppsala, Sweden

E-mail: oxana.smirnova@hep.lu.se

Abstract. The Advanced Resource Connector (ARC) middleware introduced by NorduGrid is one of the leading Grid solutions used by scientists worldwide. Its simplicity, reliability and portability, matched by unparalleled efficiency, make it attractive for large-scale facilities like the Nordic DataGrid Facility (NDGF) and its Tier1 center, and also for smaller scale projects. Being well-proven in daily production use by a wide variety of sciences, ARC of today is still largely based on conventional Grid technologies and custom interfaces introduced chiefly by Globus a decade ago. In order to guarantee sustainability, true cross-system portability, standards-compliance based interoperability, the ARC community undertakes a massive effort of introducing Web Service (WS) based components into the middleware. With support from the EU KnowARC project, new components were introduced and the existing key ARC components got extended with WS technology based standard-compliant interfaces following a service-oriented architecture. Such components include the hosting environment framework, the resource-coupled execution service, the re-engineered client library, the self-healing storage solution and the peer-to-peer information system, to name a few. Gradual introduction of these new services and client tools into the production middleware releases is carried out together with NDGF and thus ensures a smooth transition to the next generation Grid middleware. Standard interfaces and modularity of the new component design are essential for ARC contributions to the planned Universal Middleware Distribution of the European Grid Initiative.

1. Introduction

When Grid middleware development started a decade ago, the only guidance was the vision of computing power as a distributed pervasive resource, a commodity easily available for customers [1]. This matched very well the requirements of the High Energy Physics community that needed to rely on a network of computing centers in order to supply computing and storage power to the LHC [2]. A number of start-up Grid middleware projects chose the Globus Toolkit [3] as a reference implementation, which effectively promoted several Globus solutions to *de facto* standards. However, none of them were formalized as a standard specification, neither did they cover all the aspects of Grid computing. Furthermore, the Globus Toolkit [3] deliberately did not offer a complete, turnkey solution but a collection of plumbing blocks which resulted in the need of numerous middleware-specific additions. As a result, Grid middleware solutions quickly diverged, to the point of being incompatible with each other. The ARC middleware [4] is one such solution that originally started being heavily based on the Globus Toolkit libraries and API, only to later develop proprietary approaches in order to meet performance and operational requirements.

In recent years, a strong drive towards standardization emerged in the Grid community on all levels: the need of standards-based interoperability of existing production grid middlewares. The Open Grid Forum [5] transformed itself into an efficient standards development organization, delivering specifications that can be used as a basis for standard-conforming and thus interoperable Grid solutions.

The emergence of community-embraced Grid standards opened new possibilities for the Grid developers, and at the same time presented them with new challenges. ARC developers were among the first to start the transition towards the new, many times not yet fully matured standards. The main challenge on this path is two-fold: how to transform the middleware without undermining what has been achieved in terms of performance and usability and how to be able to continuously adapt to the changing specifications. In what follows, we will present a brief summary of the existing ARC features and illustrate its performance using the example of the Nordic DataGrid Facility [6], give an overview of the relevant standards, and proceed to the description of the new standards-based ARC components being developed by the EU KnowARC project [7].

2. ARC features and performance

ARC is designed and implemented as a reliable, efficient and easy-to handle middleware. It is optimized for serial data-intensive computational tasks, hence input and output data manipulation is considered an integral part of a computing service. It implements a clear service-based architecture with well defined though custom interfaces. The key concept is the absence of a single point of failure. Combined with a stateful implementation of services, this ensures high stability of the system.

ARC is deployed in production mode since summer 2002, when it became the first Grid middleware to contribute to LHC computing [8]. What started as a NorduGrid testbed consisting of several Linux clusters of modest size, grew into a unique distributed Tier1 center by NDGF, established in 2006. Figure 1 shows higher than average percentage of utilization of computing resources pledged for ATLAS production by the ARC-enabled NDGF as compared to most other Tier1 centers running on alternative middlewares. The only two centers exceeding NDGF's figure are those that acquired extra hardware to that originally pledged. High resource utilization number is a direct consequence of higher efficiency provided by ARC as compared to gLite [9] and VDT [10] middlewares deployed at other Tier1 centers.

In data-intensive HEP computing, the leading reasons of failures and inefficiencies are related to data unavailability for the task. ARC is optimized for this kind of high-throughput distributed computing because input and output data are handled consistently by the computing service

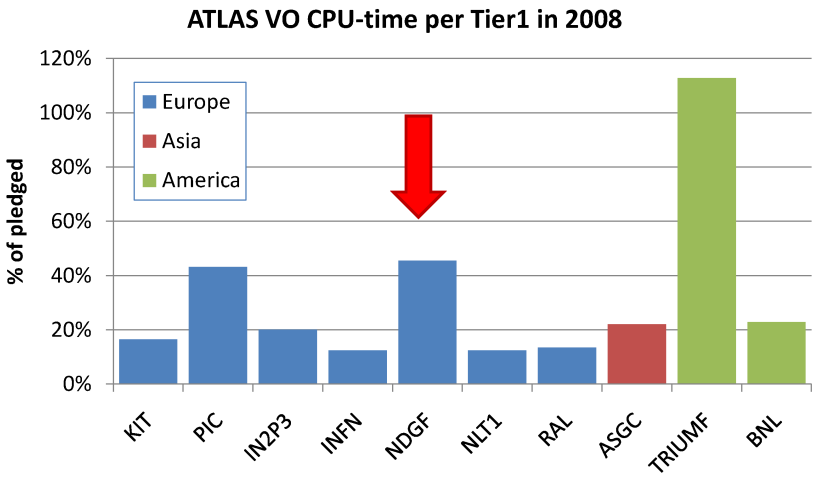


Figure 1. Utilization of computing resources pledged for ATLAS production by the involved Tier-1 centers in 2008. Arrow indicates NDGF. PIC and TRIUMF sites acquired more physical resources in course of the year than originally pledged.

plug-ins executed on the front-end. This dramatically increases CPU utilization, automatically allows for data caching, enables optimal bandwidth usage through configuration tuning, and minimizes the risk of accidental distributed denial of service attacks.

Another advantage of ARC is the clear separation between the local batch systems and the Grid: the clear interface allows integration of any batch system via plug-ins. ARC currently supports the seven most relevant batch systems plus a simple fork process launch.

The efficient and descriptive distributed information system implemented in ARC means uniform distribution of jobs among the clusters as well as clear descriptions and up-to-date status information of available resources. This allows to encapsulate resource discovery, matchmaking and brokering in the client, which in turn makes a centralized workload management system unnecessary and thus avoids a typical single point of failure. The distributed, multi-rooted and dynamic nature of the information system contributes to the overall redundancy of the Grid system and its scalability. The architecture of the information system is based on the decision to keep local information close to the place where it was generated. As a consequence, the ARC information system holds information of all the current tasks on the actual execution resource (cluster), an important feature of the system which contributes to the straightforward mobility of the users: it removes the necessary to stick to the same instance of the user interface or workload management system, as every user client will see the same information.

ARC comes with a light-weight and yet smart client component. It integrates all the necessary client functionality, including resource discovery and brokering. The tool is implemented on top of a powerful API, which facilitates development of application-specific clients, portals or even workload or workflow management systems, and a number of such exist.

The innovative code base is very portable. Globus libraries constitute the main external dependencies: this concerns mostly those pre-Web-Service packages that are related to the GSI [11] security layer and GridFTP [12] functionality. Due to its quite straightforward portability, ARC will soon be added to official Fedora and Debian repositories. The first step of adding the necessary Globus libraries there is already accomplished through the joint effort of KnowARC and NDGF projects.

Another key aspect of ARC is its non-intrusiveness with respect to the underlying resources:

it deploys as a comparatively thin layer, with software installation only on the front-end. It can thus co-exist with any other middleware or setup. Not only is ARC light-weight, it is also easy in maintenance and operations: it includes support for retries of transfer failures, transparent downtime handling, support for memory and CPU time limits, and all configuration is done through only one file. This results in very low Grid failure rate when compared to other middlewares.

3. Grid standards

As follows from the previous section, ARC efficiently implements many core Grid functionalities. Naturally, there are bugs to fix and more features to implement, but in general there is no pressing need for dramatic changes. That is, as long as ARC occupies its own niche and does not interact with other solutions.

However, one must always keep in mind that the key attractive point of the Grid concept is that of resource sharing. This is particularly important for the distributed collaborative HEP computing. Non-interoperable middleware solutions create an unnecessary overhead, forcing users to create higher-level application-specific tools.

In general, when different applications need to use different infrastructures, standards must be in place. Like in any other infrastructure, such standards must cover all possible aspects: from basic interfaces and schemas, to access control, to service level agreements and policies.

When HEP-driven projects all over the world started developing Grid solutions, there were no standards in place, only a couple of reference implementations. By now HEP computing makes use of three major middlewares, plus a large set of experiment-specific solutions bordering the line between application software and middleware. Absence of standards creates enormous overhead and at times leads to bottlenecks, especially when the documentation of proprietary interfaces and schemas is scarce.

It is practically impossible to abandon all the developed tools or to re-write them in a standard-conforming manner. ARC strategy is to add standard interfaces to existing services, thus preserving the architecture and the underlying functionality. When other middleware developers will follow the same path, the community will finally get the pervasive interoperable Grid.

As was mentioned earlier, Grid-specific standards are developed in the framework of OGF by a large number of working groups representing all middleware developers.

Some OGF standards are already implemented and used in production by ARC and other middlewares. Most notably, these are the ones pertaining to data transfer and management: the GSI-enabled file transfer protocol GridFTP and the Storage Resource Manager (SRM) interface [13]. This in practice creates the basis for the current LHC computing, allowing for seamless cross-infrastructure data movement.

Most relevant standards in the job description and execution domain are the Job Submission Description Language (JSDL) [14] and the Basic Execution Service (BES) interface [15]. Although being final specifications, they are not suitable for production environments and thus early implementations of these specifications are not widely deployed in HEP computing. One of the obstacles is the inherently generic nature of these standards, which means that real-life solutions must add numerous extensions. There is on-going work in the OGF's *PGI* working group driven by ARC, gLite and UNICORE [16] consortia to agree on a common set of extensions and modifications defined via a production profile. JSDL is already supported by ARC, and BES interface is available in the new components – both with the necessary ARC-specific extensions.

For the information and resource discovery domain, a new Glue2 [17] specification has been recently released. New components of ARC come compliant with this standard, and there is commitment from the gLite developers to move to Glue2 in near future as well.

Some standards are still sorely missing; this is particularly true for those related to access

control, accounting and user management. Middlewares deployed in HEP computing, including ARC, make use of *de facto* standards like GSI and VOMS [18], which effectively provides another solid basis for current interoperation. The work of OGF and its PGI group in particular will hopefully fill this gap soon.

4. New components of ARC

The ongoing development of the ARC middleware is driven by the following considerations:

- The ARC server-side components should exhibit standard-based, well-documented open interfaces. The custom interfaces are being replaced by community-embraced ones (see Sections 4.2, 4.4 and 4.5).
- In order to facilitate the rapid development of new components (both on the service and the client side) a modular and developer-friendly framework should be implemented (Sections 4.1 and 4.3). The new framework must offer easy extensibility.
- The ARC code base should carefully select and isolate 3rd party dependencies into optional and modular plugins. In particular, the dependency on the legacy Globus libraries should be as minimal as possible. A clean code base is particularly important due to our portability goal of ARC being available on MacOS and MS Windows platforms in addition to the already supported numerous Linux flavors.
- ARC should come with rich and self-sufficient services. The service development framework should facilitate easy creation of intelligent and powerful services such as the A-REX (Section 4.2) and the storage solution (Section 4.5).
- Last but not least the increased modularity and the introduction of new interfaces should not result in decreased performance or usability.

In the following sections, overview of key new ARC components will be given.

4.1. Hosting Environment Daemon

The cornerstone of the new approach is a light-weight Web Service container called the *Hosting Environment Daemon* (HED), which provides hosting environment for various services, as well as a number of modules to support flexible, interoperable, and efficient communication mechanism for building SOAP-based Web Services. The design of the HED is built around the idea of flexibility and modularity, such that the service developers can simply concentrate on the application level Web Service implementation by only using the core minimum amount of components. It also simplifies work on the middleware level: for example, it makes possible to implement another communication protocol or authentication mechanism. Meanwhile, a service administrator can easily configure and deploy the middleware and application-specific services satisfying a variety of requirements without having to know much about the implementation.

The architecture of HED is illustrated in Figure 2. The key components of it are the so-called *Message Chain Components* (MCC) which are in charge of implementing different protocol levels. For example, as shown in the message flow, the HTTP MCC will process a stream from the TLS MCC to parse the HTTP message and pass its body to the SOAP MCC, and also process the SOAP response from the SOAP MCC to generate the HTTP message for the TLS MCC.

The dotted line in Figure 2 shows an alternative path for the information propagation between MCCs. A service administrator can configure the MCCs according to the interoperability requirements with a counterpart. For instance, the configuration shown with the dotted line is compatible with the WSE (Web Services Enhancement for .NET) SOAP message mechanism. Another configuration could be SOAP over HTTPG (HTTP over GSI) which is needed to interoperate with services like the Storage Resource Manager (SRM). This approach ensures flexibility of HED in terms of protocols support.

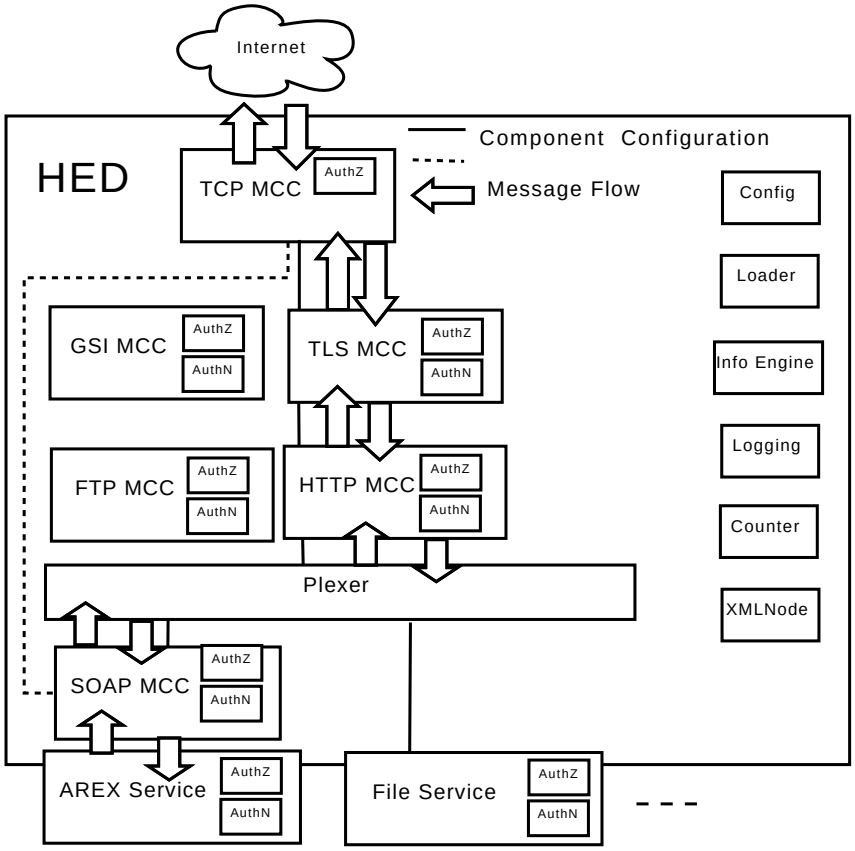


Figure 2. Example of the Hosting Environment Daemon deployed with job execution services

HED contains a flexible security framework for implementing and enforcing security-related functionality, such as authentication and authorization. Each security-related functionality can be implemented as a pluggable and configurable component (plug-in) called *SecHandler*. Each MCC or service is usually configured with two queues of SecHandler – one for the incoming messages and another for outgoing ones. In Figure 2, the "AuthZ" and "AuthN" sub-modules inside MCCs and services are examples of SecHandlers.

HED is thus sufficient for our purposes and does not require any other traditional Web Service hosting environment.

4.2. Execution service with a standard-compliant interface

The *ARC Resource-coupled EXecution service* (A-REX) is the next generation computing element of ARC offering WS-interfaces and advanced security solutions [19]. The powerful computing element implements job execution capability over a large variety of computational resources. A-REX is built around the Grid Manager component of the production ARC computing element. A-REX interprets standard job description (JSDL with NorduGrid extensions), offers OGF-compliant job execution and management interface (NorduGrid extended BES), features transparent, automatic and integrated data staging and caching capability, support for large number of batch systems, session directory management, comes with logging capability and support for *Runtime Environments*. A-REX offers Web Service-based local information interface serving Glue2 information. A-REX is also capable of working together with community approved security frameworks.

4.3. Interoperable client

The client tools for job and data management are based on libraries implemented in C++. These libraries are plugin based, and adding support for a new grid job execution service flavor or a new data access protocol can be done by developing a new plugin. The main libraries and the client tools can take advantage of the new functionality provided by the new plugin without having to be re-compiled or re-linked.

The client libraries have been wrapped using SWIG to create language bindings for Python and Java, thereby making it easy to integrate the grid client functionality provided by the libraries in application frameworks based on these languages.

The client libraries implement uniform handling of user and host credentials, computing resource discovery and information retrieval as well as matchmaking and brokering, job submission and input/output data handling.

Plugins for job handling for several Grid job execution services are already available, including the ARC Grid Manager, A-REX and the gLite CREAM service. Support for UNICORE execution services is currently being developed.

The client library supports several job description languages, including the extended resource description language (xRSL) [20] used by the ARC Grid Manager, the JDL used by e.g. CREAM, and the standard JSDL. The library is capable of converting between these descriptions as needed to interoperate with different services.

Plugins for data management exist for most data management protocols currently in use by different Grid projects, including HTTP (including HTTPS and HTTPg), GridFTP and SRM as well as support for several replica location catalogues such as the Globus RLS and the gLite LFC. There is also support for the new ARC self-healing storage solution.

Also the brokering used in the client is based on plugins and users can write their own brokering modules using either C++ or Python.

4.4. P2P information system backbone

The operation of Grid infrastructures is based on co-operation of services. The information system holds any Grid system together and provides a way for the other participants to find each other: the information system is the backbone of any Grid. Its special role requires high redundancy. To adhere to this requirement the information should be stored over the network in multiple places. The next generation information indexing backbone of ARC is being implemented as a P2P network of *ISIS* components.

ISIS as the building block of the P2P information cloud collects information from other registered services (and also from other ISIS services), stores these in its local database and then redistributes them towards its neighbors in the network in a reliable manner.

An ISIS instance stores the so-called Registration Entries submitted by the services. The maintained database of these entries can be queried by XQuery 1.0 and XPath 2.0 [21] expressions through a custom Web Service interface. The same interface is used to insert records into the ISIS database. The ISIS database itself is a soft-state database, which allows its entries to be up-to-date even in case of lost network connection.

While ARC services can register information to ISIS services and these records can be queried by clients, ISIS services cannot discover one another following this way. ISIS solves this issue by using P2P to build a cloud of information systems and to balance the load among them. To join the P2P cloud, an ISIS service only needs to know the address of some already connected ISIS and then it can connect the existing network by simply synchronizing its local database. The databases are kept synchronized by sending modification messages to other ISIS services in multiple paths.

ISIS is implemented as a service within the HED, thus all the WS related communications are performed by the hosting environment framework. Further advantages of using the HED

are the flexible and uniform configuration possibility, ready-to-use security framework and the built-in self-registration mechanism.

4.5. Self-healing storage solution

An ever increasing number of grid applications demand not only increased CPU power, but also vast amounts of storage space. Nowadays, single Grid jobs can easily produce gigabytes or even terabytes of data, thus ramping up the requirements of storage systems to the petabyte-scale and beyond.

A traditional Grid storage system provides its clients with access to data stored at remote storage systems. The architecture usually consists of an *indexing service*, indexing files from storage resources, a *replication service* for managing replica locations, and a (often centralized) *metadata catalog* imposing a global namespace on top of the resources. Data access can be handled either through a file transfer service or directly through the storage resource by querying the metadata catalog.

While this architecture has strongly improved the accessibility of physically distributed data, common Grid storage solutions typically have several limitations:

- The metadata catalog, which in most cases is centralized, can quickly become a bottleneck.
- In many systems, the storage resource is unaware of the state of its files, hampering consistency throughout the system.
- It is often taken for granted that the storage resources are dedicated, thus requirements like specific operating systems are imposed. This limit the amount of available hardware resources.
- Present day systems are often complex and sometimes difficult to deploy, their maintenance and operation requires considerable resources.

In order to improve on the above mentioned limitations the KnowARC project set forth to develop a truly distributed, self-healing and flexible Grid storage solution which has a replicated metadata catalog, state aware storage resources and an operating system agnostic implementation. The developed system consists of a set of SOAP based services residing within HED, which together provide a robust, scalable, and consistent data storage system. Data is managed in a hierarchical global namespace with files and subcollections grouped into collections[1]. A dedicated root collection serves as a reference point for accessing the global namespace, making it possible to reference the hierarchy using logical names. The global namespace is accessed in the same manner as in local filesystems.

Being based on a service-oriented architecture, the developed system consists of a set of services as shown in Figure 3. The services are as follows: The Bartender (B) provides the high-level interface to the user and offers the possibility to connect to third party storage solutions; the Librarian (L) handles the entire storage namespace, using the A-Hash (A-H) as a metadatabase; and the Shepherd (S) is the frontend for the physical storage node.

As is the case for all openly accessible web services, the security model is of crucial importance for the developed system. The security architecture of the storage system can be split into three parts; the inter-service authorization which ensures that only trusted services can get or modify data; the transfer-level authorization which ensures secure file transfers; and the high-level authorization which handles user's permissions to modify files and collections, e.g. read and write access to files and collections.

Three different client tools, serving different needs, have been developed for the storage system. A prototype command line tool offers the end user all the storage system functionality

[1] A concept very similar to files and directories in most common file systems.

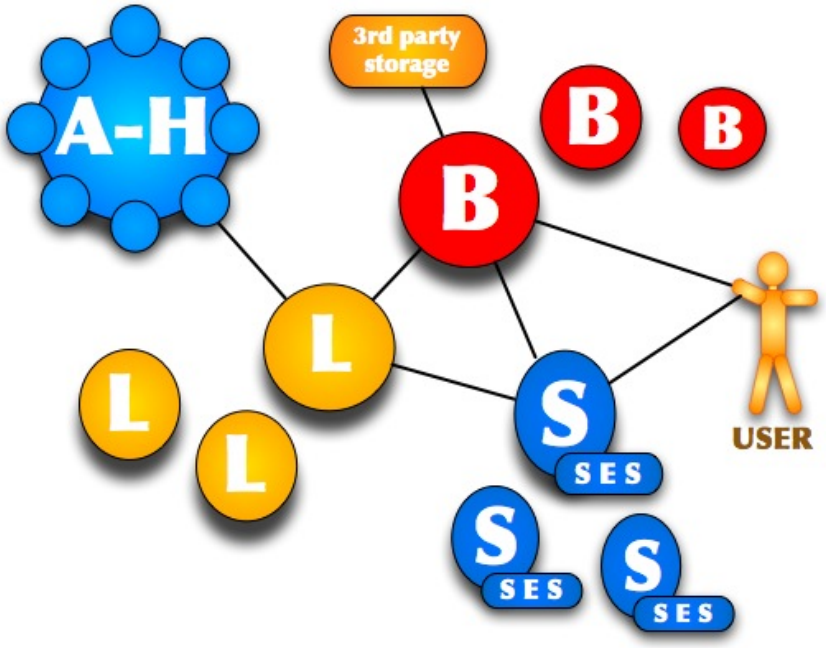


Figure 3. High-level overview of the ARC self-healing storage solution

from a terminal window. However, end users frequently interacting with the storage system may want to an even easier tool than the command line tool, and for this purpose a FUSE module has been developed. The FUSE module allows the user to mount the namespace of the system into her local filesystem, and use the storage system as if it was a local directory. Finally a storage system plugin (so-called *Data Management Component*) for the ARC data libraries has been developed. The plugin enables transparent, integrated access to the storage system from the ARC middleware thus making it possible to run Grid jobs which both down and uploads data from the developed storage system.

5. Conclusion: Towards a common middleware distribution

Thanks to the maturing community Grid standards, a growing number of middleware providers develop standard-compliant and thus interchangeable and complementary components. This is a welcoming sign for resource owners who would like to join large Grid infrastructures and yet retain the freedom of technology choice. The European Grid Initiative (EGI) [22] project strives to achieve exactly this: a pan-European Grid infrastructure for research communities. Such an environment has to rely on a set of agreed middleware tools and utilities that satisfy common criteria and meet common standards. This set currently goes by the name *Universal Middleware Distribution* (UMD) and will be based on middleware components provided by ARC, gLite and UNICORE consortia. Standard-based interoperability between the UMD components is the principal criterion in the selection process. There is a strong confidence that this middleware suite is achievable, not least because of the commitment of the core consortia to the OGF standardization process. With the re-designed and newly introduced components, ARC is well on track to offer its traditionally reliable and efficient Grid tools and services conforming to strictest interoperability and standardization requirements.

Acknowledgments

This work was supported in part by the Information Society and Technologies Activity of the European Commission through the work of the KnowARC project (Contract No.: 032691).

References

[1] Foster I and Kesselman C 1999 *The Grid: Blueprint for a New Computing Infrastructure* (Morgan Kaufmann)
[2] Aderholz M *et al.* 2000 Models of Networked Analysis at Regional Centres For LHC Experiments (MONARC), Phase 2 Report Tech. Rep. CERN-LCB-2000-001. KEK-2000-8 CERN Geneva
[3] Foster I and Kesselman C 1997 *International Journal of Supercomputer Applications* **11** 115–128 available at: http://www.globus.org
[4] Ellert M *et al.* 2007 *Future Gener. Comput. Syst.* **23** 219–240 ISSN 0167-739X
[5] Openl Grid Forum Web site URL `http://www.ogf.org/`
[6] Nordic DataGrid Facility Web site URL `http://www.ndgf.org`
[7] EU KnowARC project Web site URL `http://www.knowarc.eu`
[8] Eerola P *et al.* 2003 *Proc. of CHEP 2003, eConf C0303241:MOcT011*
[9] gLite, Lightweight Middleware for Grid Computing Web site URL `http://glite.web.cern.ch/glite/`
[10] Virtual Data Toolkit Web site URL `http://vdt.cs.wisc.edu/`
[11] Foster I *et al.* 1998 *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security* (ACM Press) pp 83–92 ISBN 1-58113-007-4
[12] Allcock W *et al.* 2002 *Parallel Comput.* **28** 749–771 ISSN 0167-8191
[13] A Sim, A Shoshani and others 2008 The Storage Resource Manager Interface (SRM) Specification v2.2 GFD-R-P.129 URL `http://www.ggf.org/documents/GFD.129.pdf`
[14] Anjomshoaa A *et al.* 2008 Job Submission Description Language (JSDL) Specification, Version 1.0 (first errata update) GFD-R.136 URL `http://www.gridforum.org/documents/GFD.136.pdf`
[15] Foster I *et al.* 2007 OGSA™ Basic Execution Service Version 1.0 GFD-R-P.108 URL `http://www.ogf.org/documents/GFD.108.pdf`
[16] UNICORE, Uniform Interface to Computing Resources Web site URL `http://www.unicore.eu`
[17] Andreozzi S *et al.* 2009 GLUE Specification v2.0 GFD-R-P.147 URL `http:/www.ogf.org/documents/GFD.147.pdf`
[18] Alfieri R *et al.* 2005 *Future Gener. Comput. Syst.* **21** 549–558 ISSN 0167-739X
[19] Konstantinov A *The ARC Computational Job Management Module - A-REX* NORDUGRID-TECH-14 URL `http://www.nordugrid.org/documents/a-rex.pdf`
[20] Smirnova O *Extended Resource Specification Language* The NorduGrid Collaboration NORDUGRID-MANUAL-4 URL `http://www.nordugrid.org/documents/xrsl.pdf`
[21] 2007 XQuery 1.0: An XML Query Language W3C Recommendation URL `http://www.w3.org/TR/xquery/`
[22] European Grid Initiative Web site URL `http://www.eu-egi.eu/`

A.5 Chelonia - A Self-healing Storage Cloud

Article to appear in the proceedings for Cracow '09 Grid Workshop, Oct. 12-14.

The article "Chelonia - A Self-healing Storage Cloud" is written by the KnowARC storage team. It gives an overview of the Chelonia architecture and presents preliminary test results of a Chelonia deployment distributed between three different countries.

The text was written by Salman Toor and me together. The tests where run by Salman Toor, Zsombor Nagy and me.

Chelonia – A Self-healing Storage Cloud

Jon K. Nilsen[1], Salman Toor[2], Zsombor Nagy[3] and Bjarte Mohn[2]

[1] University of Oslo, Norway
[2] Uppsala University, Sweden
[3] NIIF, Hungary

Abstract

The concept of storage clouds based on the Service-Oriented Architecture (SOA) provides a framework for managing storage resources in a consistent and reliable manner. Under the umbrella of the KnowARC project we have designed and implemented a resilient, reliable and self-healing storage system. In this paper, we will present an overview of Advanced Resource Connector's novel storage system, Chelonia, consisting of a set of services whic together provide a self-healing, grid-enabled storage cloud. We will also present some proof-of-concept test results from the deployment and the utilization of storage resources distributed across three different countries.

Keywords: Grid, Storage, Cloud

1 Introduction

An increasing number of scientific applications and experiments demand not only increased CPU power, but also vast amounts of storage space. Nowadays, we can easily find applications/experiments which produce terabytes of data. The availability of the requested storage space (data) is not only required for the duration of the application or experiment, but in many cases for years after the completion of the application/experiment. Such requirements push the development of distributed storage in the direction of a reliable, resilient and consistent data management system.

In the last couple of years the concept of storage clouds has gone from being completely unknown to being the subject of significant attention from end-users and developers alike [11]. A storage cloud addresses many of the above mentioned challenges, but from a user perspective it also hides the complexity of the machinery involved in making the system work. Such a framework requires well-defined roles for the involved parties (the Service Providers and Infrastructure Providers). It also requires these groups to have explicit understandings of and commitments to their roles. Apart from the conceptual agreements, there is a fundamental need for sustainable technology on which the framework can built.

Storage clouds aim to provide a unified view of the storage (regardless of where the files are physically stored) and a user-centric interface to the cloud. To achieve high quality of service, such as high availability, reliability and consistency, storage clouds need some sort of self-healing capability.

In the advent of the next generation of the Advanced Resource Connector (ARC) grid middleware [7] (new release due during spring 2010), we hereby

present Chelonia, a storage cloud from the research team behind the ARC middleware. The aim of Chelonia is to address high availability of data in a secure environment but also the consistency and reliability of the storage system itself. The architecture of Chelonia is close to the ideology of storage clouds, thus encouraging us not to develop yet another grid storage solution, but to address an even wider community and build a grid-aware storage solution within the paradigm of storage clouds. In this paper we will present how Chelonia provides the above identified capabilites of a storage cloud.

This paper is organized as follows: An overview of Chelonia and its architecture is given in Section 2, while the security of the storage system is elaborated in Section 3. Section 4 gives an overview of the use of Chelonia, while in Section 5 we give some early, proof-of-concept results. In section 6 we give some comparisons with related work in both the grid and cloud communities before making some concluding remarks in Section 7.

2 Chelonia Architecture

Over the years, different concepts have evolved to deal with the challenges of handling distributed storage resources. Whereas cluster storage solutions addressed the limitations of single machines and grid storage addressed the limitations of cluster storage, the now emerging concept of storage clouds addresses the shortcomings of grid storage. While grid storage has greatly improved the availability of geographically distributed data, actual data access can still be cumbersome and may require knowledge of where the data is physically located. In comparison, cloud storage typically exposes a limited set of features, hiding details like where the data is stored. However, whereas collaboration and resource sharing is an important aspect of the grid, clouds tend to handle security through isolation of users [10].

The storage cloud is an emerging paradigm and this shift from data grid to storage cloud is due to the years of research and development of grid tools in academia. The Chelonia storage cloud is designed to combine the best of these two paradigms for a truly distributed, self-healing, user-friendly storage solution, providing means for resource sharing and collaboration.

Chelonia consists of a set of SOAP-based services residing within the Hosting Environment Daemon (HED) [6]. Together, the services provide a self-healing, reliable, resilient and consistent data storage. Data is managed in a hierarchical global namespace with files and subcollections grouped into collections (a concept very similar to files and directories in most common file systems). A dedicated root collection serves as a reference point for accessing the namespace. The hierarchy can then be referenced using Logical Names. The global namespace is accessed in the same manner as in local filesystems. Being based on a service-oriented architecture, the Chelonia cloud consists of a set of services, the Bartender, Librarian, A-Hash and Shepherd as shown in Fig. 1. All the services are replicated to ensure high availability, scalability and to avoid single points of failure.

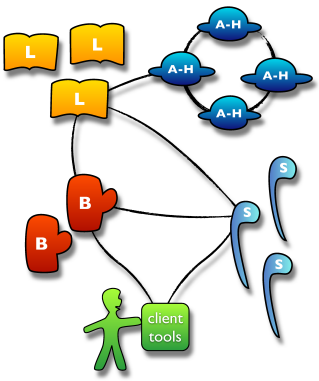


Fig. 1: Schematic of the Chelonia cloud architecture. The figure shows the main services of Chelonia; The Bartender (cup), the Librarian (book), the A-Hash (space ship) and the Shepherd (staff). The communication channels are depicted by black lines.

The Bartender, being the only service visible to the user, provides a high-level interface for the storage system. Clients connect to the Bartender to create and remove files, collections and mount-points using their Logical Names and the Bartender communicates with the Librarian and Shepherd services to execute the clients' requests. The Shepherd serves as a front-end to a storage node. The Shepherd service reports to the Librarian about the state of the stored data. The Librarian manages the hierarchy and metadata of files, collections and mount points. In addition, the Librarian handles the information about registered Shepherd services. The Librarian uses the A-Hash for consistently storing the entire state of the system, including file metadata and registration of Shepherds. Additionally the A-Hash itself, being replicated, stores information about other registered A-Hash services. It is therefore crucial for the system that the A-Hash is consistent. The A-Hash is built on the Oracle Berkeley DB High Availability [1] database. The replication is based on a single master, multiple clients framework where all clients can read from the database, while only the master is allowed to write to the database. In the event of a master not responding, a new master is automatically chosen between the remaining replicas.

3 Chelonia Security Model

As is the case for all openly accessible web services, the security model is of crucial importance for Chelonia. The security architecture of the storage can be split into three parts; the inter-service authorization; the transfer-level authorization; and the high-level authorization.

The **inter-service authorization** maintains the integrity of the internal communication between services. There are several communication paths between the services in the storage system. The Bartenders send requests to the Librarians and the Shepherds, the Shepherds communicate with the Librarians and the Librarians with the A-Hash. If any of these services is compromised or a new rogue service gets inserted in the system, the security of the entire system is compromised. To enable trust between the services, they need to know each other's Distinguished Names (DN's). The services nood to obtain a certificate from a trusted Certificate Authority, thus preventing rogue services from attacking a Chelonia cloud.

The **transfer-level authorization** handles the authorization in the cases of

uploading and downloading files. When a transfer is requested, the Shepherd will provide a one-time Transfer URL (TURL) to which the client can connect. In the current architecture, this TURL is world-accessible. This may not seem very secure at first. However, provided that the TURL has a very long, unguessable name, that it is transfered to the user in a secure way and that it can only be accessed once before it is deleted, the chance of being compromised is very low.

The **high-level authorization** considers the access policies for the files and collections in the system. These policies are stored in A-Hash, in the metadata of the corresponding file or collection, providing a fine-grained security in the system.

The communication with and within the storage system is realized through HTTPS with standard X.509 authentication.

4 Chelonia in Operation

Once in operation, the Chelonia storage cloud will be a pulsing system where heartbeats are periodically sent from each Shepherd to a Librarian together with information about replicas whose state changed since the last heartbeat. Heartbeats are stored in the A-Hash, thus making them visible to all Librarians in the system. If any of the Librarians notices that a Shepherd is late with its heartbeat, it will mark all the replicas in that Shepherd as offline.

In addition to the heartbeat, the Shepherds periodically check with the Librarians to see if there are sufficient replicas of the files in Chelonia and if the checksums of the replicas are correct. If a file is found to have too few replicas, the Shepherd informs a Bartender about this situation and together they ensure that a new replica is created at a different storage node. A file having too many replicas will also be automatically corrected by Chelonia as the first Shepherd to notice this will mark its replica(s) as unneeded and later delete it (them). Replicas with invalid checksums are marked as invalid, and as soon as possible replaced with a valid replica.

The Gateway is built into the Bartender service as an independent plug-in and provides means to mount other Chelonia clouds as well as external storage systems into a global namespace. The Gateway plug-ins are protocol-oriented in the sense that external storage managers which support a certain protocol will be handled using the Gateway plug-in based on that protocol. While excluding some of the features provided by accessing storage managers directly, this approach reduces the number of Gateway plug-ins required for different storage managers. The currently available Gateway plug-in is based on the GridFTP protocol[5].

In addition to the traditional command line tool, Chelonia comes with a FUSE module which provides a high-level access to the storage system. Filesystem in Userspace (FUSE) [2] provides a simple library and a kernel-userspace interface. Using FUSE and the ARC Python interface, the FUSE module allows users to mount the storage namespace into the local namespace, enabling the use of, e.g., graphical file browsers.

5 Testing and Discussion

Even though internal performance and scalability is important for a storage cloud, the main concerns of a user are the response time of the system, that files are always available and that files will survive server crashes. Before a storage system can be considered for deployment, thorough testing is required. We will in this section present a proof-of-concept test, focusing on the usability of a geographically distributed storage cloud. While extensive performance and scalability tests have been carried out they are beyond the scope of this paper.

In the test deployment the computer resources were geographically distributed with three machines provided by Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX), Sweden, five machines provided by the Center for Information Technology (USIT) at the University of Oslo, Norway, and one machine provided by National Information Infrastructure Development Institute (NIIF), Hungary. The actual services were then distributed as follows:

- Three A-Hashes, two at USIT and one at UPPMAX.
- Two Bartenders, one at UPPMAX and one at NIIF.
- Two Librarians, one at UPPMAX and one at USIT.
- Three Shepherds, one at each site.

While the Shepherds at UPPMAX and NIIF had ample storage space for our tests, the Shepherd at USIT was limited to 1GB. The actual test was then carried out in five steps:

1. 100 10 kB files from UPPMAX and 10 1 GB files from USIT were uploaded to the system, generating 1 file replica of each file. The time from requesting an upload to the transfer started (the system response time) and the distribution of files between the Shepherds were noted.
2. The number of replicas for each 10 kB file was increased to two. The file distribution was noted.
3. One Shepherd was stopped and the file distribution between the two remaining Shepherds was noted.
4. The stopped Shepherd was restarted. The file distribution was noted.
5. The 10 kB files were downloaded to USIT. The system response time was noted.

Fig. 2 shows the time taken by the system before starting the file transfers in the case of uploading and downloading 100 10 kB files. The files were uploaded at UPPMAX and downloaded at USIT. The difference between upload (dots) and download (stars) is due to the extra steps needed to upload a file. While in downloading the system only needs to look up a logical name and generate a transfer URL, a file upload requires the existence of the logical name to be checked, the file and its metadata to be registered in the system and a transfer URL to be generated. As the checksum of the file is part of the metadata, the client needs to calculate the checksum as part of the registration. Additionally, registering data to the A-Hash requires that all A-Hash replicas acknowledge the data before the master A-Hash can return, making writing to a slower process than reading from the A-Hash, again in favor of the download time. It should be mentioned that ideally the checksum should be calculated during the actual

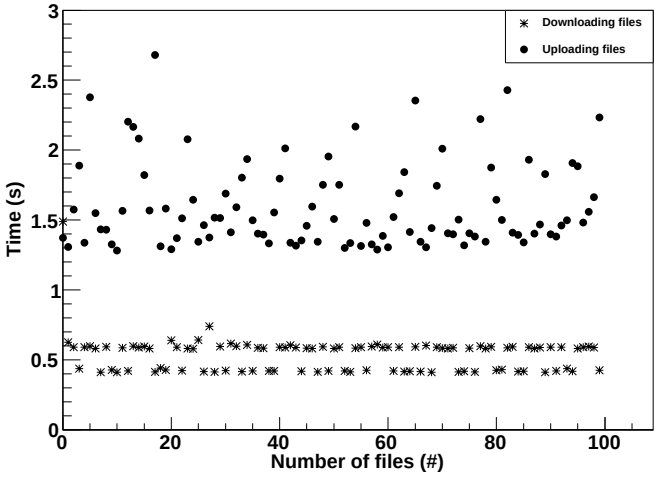


Fig. 2: Time from request to transfer is started. Dots show timing for upload, stars show timing for download. The difference between upload and download is due to extra time to register the file and calculate checksum on the client side in the case of file upload.

	UPPMAX	USIT	NIIF
10 kB files	37	36	27
1 GB files	3	0	7
Total	40	36	34

Tab. 1: Overall file distribution after uploading all 10 kB files and 1 GB files.

file transfer to avoid scanning the entire file twice. This was unfortunately not implemented in the client tool used in this test.

An important feature of Chelonia is that file replicas are distributed evenly between the Shepherds. Evidently, one Shepherd cannot have more than one replica of the same file, but additionally an even distribution of file locations can help avoiding one Shepherd becoming a hot-spot in the system. Thus, a balanced distribution will help in achieving high availability. Tab. 1 shows the distribution of the file replicas between Shepherds after all files were uploaded to the system. For the 10 kB files, where an ideal distribution would be between 33 and 34 files per Shepherd, the fluctuation is due to randomly choosing which Shepherd should receive a file. Randomly choosing Shepherds will lead to a uniform distribution of replicas when the number of replicas grows very large. Looking at the distribution of 1 GB files, we note that the USIT Shepherd has received 0 files. Since this Shepherd have less than 1 GB disk space available, it will not accept any transfer above this limit, and all 1GB files are directed to the two other Shepherds.

Tab. 2 shows the file distribution of 100 10 kB files having 2 replicas each.

	All online	One offline	All online
UPPMAX	71	N/A	50
USIT	58	100	69
NIIF	71	100	81
Total	200	200	200

Tab. 2: File distribution before, during and after the UPPMAX Shepherd was offline.

In the first column all three Shepherds are online, in the second column one Shepherd is offline, and in the third column all Shepherds are again online, corresponding to step 2, 3 and 4 of the test. Since all the files had two replicas, the system was able to heal it self by duplicating the replicas stored at the offline Shepherd, as can be seen in column two where the two online Shepherds had 100 replicas each. When the third Shepherd again came online the Shepherds discovered that there were more replicas than required, and removed extraneous replicas.

6 Related Work

When compared with typical grid distributed data management solutions, the closest resemblance with Chelonia is the combination of the storage element Disk Pool Manager (DPM) and the file catalog LCG (LHC (Large Hadron Collider) Computing Grid) File Catalog (LFC) [8]. By registering all files uploaded to different DPM's in LFC one can achieve a single uniform namespace similar to the namespace of Chelonia. However, where Chelonia has a strong coupling between the Bartenders, the Librarians and the Shepherds to maintain a consistent namespace, DPM and LFC has no coupling and registration and replication of files is handled on the client side.

While Chelonia is designed for geographically distributed users and data storage, Hadoop [3] with its file system HDFS is directed towards physically closely grouped clusters. HDFS builds on the master-slave architecture where a single NameNode works as a master and is responsible for the metadata whereas DataNodes are used to store the actual data. Though similar to Chelonia's metadata service, the NameNode cannot be replicated and may become a bottleneck in the system. Additionally, HDFS uses non-standard protocols for communication and security while Chelonia uses standard protocols like HTTP(S), GridFTP and X509.

In the cloud storage family, Amazon Simple Storage Service (S3) [4] is a storage solution promising unlimited storage and high availability. Amazon uses a two level namespace as opposed to the hierarchical namespace of Chelonia. In the security model of S3, users have to implicitly trust S3 entirely, where as in Chelonia users and services need to trust a common independent third party Certificate Authority. Additionally, S3 lacks fine-grained delegation and access control lists are limited to 100 principals, limiting the usability for larger scientific communities [9].

7 Conclusion

Chelonia, a cloud-like grid storage solution, has been described and the presented test of a storage cloud spanning three European countries shows that several key features of the Chelonia storage service are ready. The system runs in a geographically distributed environment, the stored files are accessible both during and after server outages, maintaining the required number of replicas, and files can be shared between sites situated in different countries. Even though more testing is required to have Chelonia ready for production, the test shows promise for Chelonia to be a robust, self-healing storage cloud suitable for scientific communities.

Acknowledgements. We wish to thank UPPMAX, NIIF and USIT for providing resources for running the storage tests. The work has been supported by the European Commission through the KnowARC project (contract nr. 032691) and by the Nordunet3 programme through the NGIn project.

References

1. Oracle Berkeley DB, `http://www.oracle.com/technology/products/berkeley-db/index.html`.
2. Filesystem in Userspace, `http://fuse.sourceforge.net/`.
3. Apache Hadoop, `http://hadoop.apache.org/`.
4. Amazon Simple Storage Service, `http://s3.amazonaws.com`
5. W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming and S. Tuecke: GridFTP: Protocol extensions to FTP for the Grid, *GWD-R (Recommendation)*, page 3, 2001.
6. D. Cameron, M. Ellert, J. Jönemo, A. Konstantinov, I. Marton, B. Mohn, J. K. Nilsen, M. Nordén, W. Qiang, G. Rőczei, F. Szalai and A. Wäänänen: The Hosting Environment of the Advanced Resource Connector middleware, *NorduGrid*, NORDUGRID-TECH-19, `http://www.nordugrid.org/documents/arc1-storage-documentation.pdf`.
7. M. Ellert and others: Advanced Resource Connector middleware for lightweight computational Grids, *Future Gener. Comput. Syst.*, **23**, 1, 219-240, 2007.
8. Akosh Frohner: Official Documentation for LFC and DPM, `https://twiki.cern.ch/twiki/bin/view/LCG/DataManagementDocumentation`.
9. M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel: Amazon S3 for science grids: a viable solution?, *DADC '08: Proceedings of the 2008 international workshop on Data-aware distributed computing*, pages 55-64, 2008. New York, NY, USA, ACM, 2008.
10. L. M. Vaquero, L. R. Merino, J. Caceres and M. Lindner: A break in the clouds: towards a cloud definition, *SIGCOMM Comput. Commun. Rev.*, **39**, 1, pages 50-55, 2009.
11. L. Wang, J. Tao, M. Kunze, A. C. Castellanos, D. Kramer and W. Karl: Scientific Cloud Computing: Early Definition and Experience, *HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 825-830, 2008. Washington, DC, USA, IEEE Computer Society, 2008.

A.6 Chelonia - distributed cloud storage

Article submitted to Journal of Parallel and Distributed Computing.

The article "Chelonia - distributed cloud storage" gives an extensive overview of Chelonia and presents the results of a series of performance and stability tests.

My main contributions to the article was the results in Section 5.4 and all of the text, written in collaboration with Salman Toor and Bjarte Mohn.

Performance and Stability of the Chelonia Storage Cloud

J. K. Nilsen[a,b], S. Toor[c], Zs. Nagy[d], B. Mohn[e], A. L. Read[a]

[a] *University of Oslo, Dept. of Physics, P. O. Box 1048, Blindern, N-0316 Oslo, Norway*
[b] *University of Oslo, Center for Information Technology, P. O. Box 1059, Blindern, N-0316 Oslo, Norway*
[c] *Dept. Information Technology, Div. of Scientific Computing Uppsala University, Box 256, SE-751 05 Uppsala, Sweden*
[d] *Institute of National Information and Infrastructure Development NIIF/HUNGARNET, Victor Hugo 18-22, H-1132 Budapest, Hungary*
[e] *Dept. of Physics and Astronomy, Div. of Nuclear and Particle Physics, Uppsala University, Box 535, SE-751 21 Uppsala, Sweden*

Abstract

In this paper we present the Chelonia storage cloud middleware. It was designed to fill the requirements gap between those of large, sophisticated scientific collaborations which have adopted the grid paradigm for their distributed storage needs, and of corporate business communities which are gravitating towards the cloud paradigm. The similarities to and differences between Chelonia and several well-known grid- and cloud-based storage solutions are commented. The design of Chelonia has been chosen to optimize high reliability and scalability of an integrated system of heterogeneous, geographically dispersed storage sites and the ability to easily expand the system dynamically. The architecture and implementation in term of web-services running inside the Advanced Resource Connector Hosting Environment Dameon (ARC HED) are described. We present results of tests in both local-area and wide-area networks that demonstrate the fault-tolerance, stability and scalability of Chelonia.

Key words: Data Grid, Cloud Storage, Middleware, Distributed Data Management

Email addresses: j.k.nilsen@usit.uio.no (J. K. Nilsen), salman.toor@it.uu.se (S. Toor), zsombor@niif.hu (Zs. Nagy), bjarte.mohn@fysast.uu.se (B. Mohn), a.l.read@fys.uio.no (A. L. Read)

1. Introduction

As computationally demanding research areas expand, the need for storage space for results and intermediate data increases. Currently running experiments in areas like high energy physics, atmospheric science and molecular dynamics already generate petabytes of data every year. The increasing number of international and even global scientific collaborations also contributes to the growing need to share (and protect) data efficiently and effortlessly.

While different research groups have different requirements for a storage system, a set of key characteristics can be identified. The storage system needs to be *reliable* to ensure data integrity. It needs to be *scalable* and capable to dynamically *expand* to future needs, and given that many research groups today use computational grids to process their data, it is highly favorable if the storage system is *grid-enabled*.

Lately, the concept of storage clouds has gone from being completely unknown to being the subject of significant attention from end-users and developers alike [23]. A storage cloud addresses many of the needs mentioned above, but from a user perspective it also hides the complexity of the machinery involved in making the system work. Such a framework requires well-defined roles for the involved parties (the Service Providers and Infrastructure Providers). It also requires these groups to have explicit understandings of and commitments to their roles. Apart from the conceptual agreements, there is a fundamental need for sustainable technology on which the framework can built.

In this paper we will present the design and performance of the Chelonia storage cloud [1, 21]. With Chelonia we aim at designing a system which fulfills the requirements of global research collaborations, but which also meets the specifications of a storage cloud, e.g., user-centric interfaces and transparency. With Chelonia it is possible to build anything from simple storage systems for sharing holiday pictures to large-scale storage systems for storing petabytes of scientific data.

This paper is organized as follows: First we give a brief introduction to distributed data storage solutions in Section 2, before we in Section 3 give an architectural overview of Chelonia and its services. Section 4 exemplifies important features of Chelonia, while Sections 5 and 6 present performance

and stability of Chelonia, respectively. A comparison with other storage solutions on the market is given in Section 7. Section 8 presents ongoing development work, before the conclusions are given in Section 9.

2. Distributed Data Storage Solutions

Over the years, different concepts have evolved to deal with the challenge of handling increasing volumes of data and the fact that the data tend to be generated and accessed over vast geographic regions. The largest storage solutions nowadays can be roughly divided between the data grids developed and used by scientific communities, and cloud storage arising from the needs of corporate business communities. While both concepts have the same goal of distributing large amounts of data across distributed storage facilities, their focuses are slightly different.

Data grid solutions focus on sharing data stored at several large storage facilities which are usually supported by public funding and run by different organizations. A grid storage system provides its clients with access to data stored at remote storage systems. A traditional architecture (see e.g. [15, 12]) typically consists of an *indexing service*, indexing files from storage resources, a *file transfer service* for transferring files, a *replication service* for managing replica locations, and a (often centralized) *metadata catalog* imposing a global namespace on top of the resources. While they enable the sharing of resources between large number of users, data grids are often considered to be rather cumbersome in use and maintenance. For example, there is no common method of establishing a global namespace and this is achieved only additional effort of the organization that cares for its own data.

Cloud storage focuses more on providing large amounts of storage to other organizations, and one cloud storage facility is usually run by a single organization. The main building blocks of a cloud are the services. The cloud *actors* access the services both to add resources and utilize resources. The services provides Quality of Service (QoS) guarantees through Service Level Agreements (SLA's). In clouds, storage is provided through the concept of Data as a Service (DaaS), which together with Infrastructure as a Service (IaaS), Hardware as a Service (HaaS) and Software as a Service (SaaS) can form a Platform as a Service (PaaS). Hence, by combining services, the service user can create a customized virtual platform. While this provides more flexibility for the cloud user and service providers than the grid concept, it limits the ability of sharing resources. When a user has set up a virtual com-

puting platform, this platform is typically limited to be used by this user. Data security is usually realized through isolating the virtual platform to be used only by the one user.

Storage clouds are an emerging paradigm, and even though the focus differs from the data grids, the paradigm has learned from the experience of the grid paradigm and improved on several features like usability and payment plans. However, arising from the needs of corporate business the storage clouds lack features like file-sharing and high-level tools needed by the scientific communities. The Chelonia storage cloud is designed to combine the best of two paradigms for a truly distributed, self-healing, flexible storage solution, with a replicated metadata catalog, easy-to-use storage resources and an operating system-agnostic implementation.

3. Architecture of Chelonia

Chelonia consists of a set of SOAP-based web services residing within the Hosting Environment Daemon (HED) [14] service container from the ARC grid middleware. Together, the services provide a self-healing, reliable, robust, scalable, resilient and consistent data storage system. Data is managed in a hierarchical global namespace with files and subcollections grouped into collections[1]. A dedicated root collection serves as a reference point for accessing the namespace. The hierarchy can then be referenced using Logical Names. The global namespace is accessed the same manner as in local filesystems.

Being based on a service-oriented architecture, the Chelonia storage cloud consists of a set of services as shown in Figure 1. The Bartender (cup) provides the high-level interface to the user; the Librarian (book) handles the entire storage namespace, using the A-Hash (space ship) as a metadatabase; and the Shepherd (staff) is the front-end for the physical storage node. Note that any of the services can be deployed in multiple instances for high availability and load balancing. Before going into the technical details of Chelonia itself, it may be beneficial to have a brief look at the middleware providing the communication layer of Chelonia.

[1] A concept very similar to files and directories in most common file systems.

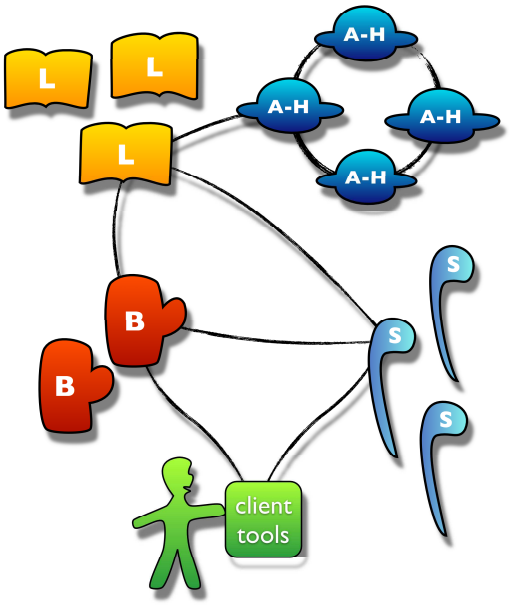


Figure 1: Schematic of Chelonia's architecture. The figure shows the main services of Chelonia; The Bartender (cup), the Librarian (book), the A-Hash (space ship) and the Shepherd (staff). The communication channels are depicted by black lines.

3.1. The Advanced Resource Connector

The Chelonia communication layer is provided by the next generation Advanced Resource Connector (ARC) Grid middleware, developed by NorduGrid [2] and the EU-supported KnowARC project [3]. The next generation ARC is based upon web-services which make frequent use of pluggable components for offering certain capabilities. The ARC services, including Chelonia (Section 3.2), run inside a container called the Hosting Environment Daemon (HED).

There are three basic kinds of plugable components for HED: Data Management Components (DMC's) are used to transfer the data using various protocols; Message Chain Components (MCC's) are responsible for the communication within services as well as between the clients and the services; and Policy Decision Components (PDC's) are responsible for the security model within the system. In Chelonia, the Shepherd uses DMC's to transfer files, all client-service and inter-service communication goes through the SOAP MCC and the Bartender uses the PDC to decide if a user has access

or not.

3.2. Core Services

The main components of Chelonia are the four services, the A-Hash, the Librarian, the Bartender and the Shepherd. They each have separate roles based on the distinct characteristics of a distributed storage system. When compared with traditional data grid solutions, the Librarian may be viewed as an indexing service and the Shepherd as the manager of the file transfer service. The replication service is provided by the Shepherd, Librarian and Bartender services acting together and the Librarian and A-Hash function as a metadata catalog. When compared with cloud storage solutions, Chelonia provides DaaS with the Bartender providing a well-defined API for easy-to-use access. Acting together, the services provide an easy-to-use, lightweight storage system without single points of failure.

3.2.1. The A-Hash

The A-Hash service is a database that stores objects which contain key-value pairs. In Chelonia, it is used to store the global namespace, all file metadata and information about itself and storage elements. Being such a central part of the storage system, the A-Hash needs to be consistent and fault-tolerant. The A-Hash is replicated using the Oracle Berkeley DB [4] (BDB), an open source database library wih a replication API. The replication is based on a single master, multiple clients framework where all clients can read from the database and only the master can write to the database. While a single master ensures that the database is consistent at all times, it raises the problem of having the master as a single point of failure. If the master is unavailable, the database cannot be updated, files and entries cannot be added to Chelonia and file replication will stop working. The possibility of the master failing cannot be completely avoided, so to ensure high availability means must be taken to find a new master if the first master becomes unavailable. BDB uses a variant of the Paxos algorithm [18] to elect a master amongst peer clients: Every database update is assigned an increasing number. In the event of a master going offline, the clients sends a request for election, and a new master is elected amongst the clients with the highest numbered database update.

3.2.2. The Librarian

The Librarian service manages the hierarchy and metadata of files and collections, handles the Logical Names and monitors the Shepherd services. The

Librarian service is stateless, instead it stores all the persistent information in the A-Hash. This makes it possible to deploy any number of independent Librarian services to provide high-availability and load-balancing. In this case all the Librarians should communicate with the same set of A-Hashes in order to use the same database of metadata. As only the master A-Hash can be written to and the Librarian cannot know *a priori* which A-Hash replica is the master, the Librarian needs to get this information from one of the A-Hashes. For this reason, the master A-Hash stores the list of all available A-Hashes, so that the information is replicated to all A-Hash replicas. As all A-Hash replicas are readable, the Librarian only needs to know about a few of the A-Hashes at start-up to be able to get this list. During run-time the Librarian holds a local copy of the A-Hash list and refreshes it both regularly and in the case of a failing connection.

3.2.3. The Shepherd

Each instance of the Shepherd service manages a particular storage node and provides a uniform interface for storing and accessing file replicas. On a storage node there must be at least one independent storage element service (with an interface such as HTTP(S), ByteIO, etc.) which performs the actual file transfer. A storage node then consists of a Shepherd service and a storage element service connected together. Storage element services can either be provided by ARC or by third-party services. For each kind of storage element service, a Shepherd backend module is needed to enable the Shepherd service to communicate with the storage element service, e.g., to initiate file uploads, downloads and removal, and to detect whether a file transfer was successful or not. Currently there are three Shepherd backends: One for the ARC native HTTP(S) server called Hopi; one for the Apache web server; and one for a service which implements a subset of the ByteIO interface. In addition to storing files and providing access to them, the Shepherd is responsible for checking if a file replica is valid and, if necessary, initiating replication of the file to other Shepherds.

3.2.4. The Bartender

The Bartender service provides a high-level interface of the storage system for the clients (other services or users). The clients can create and remove collections (directories), create, get and remove files, and move files and collections within the namespace using Logical Names. Access policies associated with files and collections are evaluated by the Bartender (using

the PDC plugin of HED) every time a user wants to access them. The Bartender communicates with the Librarian and Shepherd services to execute the client's requests. The file content itself does not go through the Bartender; file transfers are directly performed between the storage nodes and the clients.

The Bartender also supports so-called gateway modules which make it possible to communicate with third-party storage solutions, thus enabling the user to access multiple storage systems through a single Bartender client. These modules are protocol-oriented in the sense that external storage managers which support a certain protocol will be handled using the gateway module based on that protocol. While excluding some of the features provided by accessing storage managers directly, this approach reduces the number of gateway modules required for different storage managers. The currently available gateway module is based on the GridFTP protocol[9].

3.3. Security

As is the case for all openly accessible web services, the security model is of crucial importance for the Chelonia storage cloud. While the security of the communication with and within the storage system is realized through HTTPS with standard X.509 authentication, the authorization related security architecture of the storage can be split into three parts; the inter-service authorization; the transfer-level authorization; and the high-level authorization:

- The inter-service authorization maintains the integrity of the internal communication between services. There are several communication paths between the services in the storage system. The Bartenders send requests to the Librarians and the Shepherds, the Shepherds communicate with the Librarians and the Librarians with the A-Hash. If any of these services are compromised or a new rogue service gets inserted in the system, the security of the entire system is compromised. To enable trust between the services, they need to know each other's Distinguished Names (DN's). This way a rogue service would need to obtain a certificate with that exact DN from some trusted Certificate Authority (CA).

- The transfer-level authorization handles the authorization in the cases of uploading and downloading files. When a transfer is requested, the

Shepherd will provide a one-time Transfer URL (TURL) to which the client can connect. In the current architecture, this TURL is world-accessible. This may not seem very secure at first. However, provided that the TURL has a very long, unguessable name, that it is transfered to the user in a secure way and that it can only be accessed once before it is deleted, the chance of being compromised is fairly low.

- The high-level authorization considers the access policies for the files and collections in the system. These policies are stored in the A-Hash, in the metadata of the corresponding file or collection, providing a fine-grained security in the system.

3.4. Accessing Chelonia

Being the only part a user will (and should) see from a storage system, the client tools are an important part of the Chelonia storage cloud. In addition to a vanilla command-line interface, two ways of accessing Chelonia are supported.

3.4.1. FUSE Module

The FUSE module provides a high-level access to the storage system. Filesystem in Userspace (FUSE) [5] provides a simple library and a kernel-userspace interface. Using FUSE and the ARC Python interface, the FUSE module allows users to mount the storage namespace into the local namespace, enabling the use of graphical file browsers as shown in the screenshot in Figure 2.

3.4.2. Grid Job Access

To access data through the ARC middleware client tools, one needs to go through Data Manager Components (DMC's). These are protocol-specific plugins to the client tools. For example, to access data from a HTTPS service, the HTTP DMC will be used with a URL starting with `https://`, to access data from an SRM service, the SRM DMC will be used with a URL starting with `srm://`. Similarly, to access Chelonia, the ARC DMC will be used with a URL starting with `arc://`.

The ARC DMC allows grid jobs to access Chelonia directly. As long as A-REX, the job execution service of ARC, and ARC DMC are installed on a site, files can be both downloaded and uploaded by specifying the corresponding URL's in the job description. In this case, the Bartender URL needs to

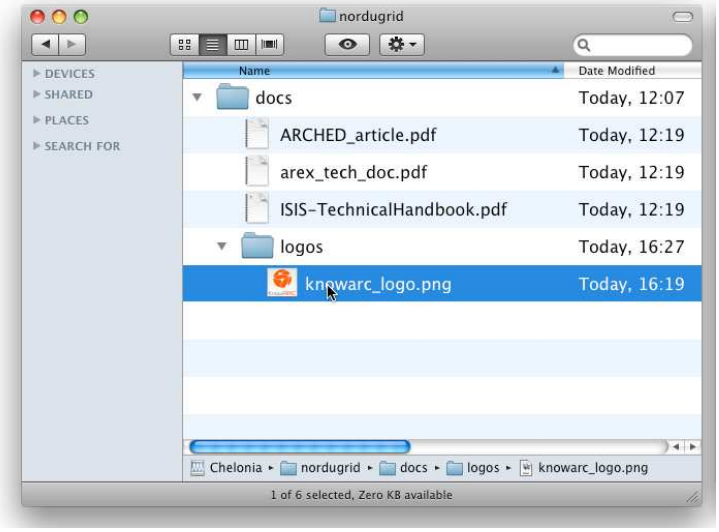


Figure 2: Screenshot of the Chelonia FUSE module in use. Through the FUSE module Chelonia offers users a drag and drop functionality to upload or download files to the storage cloud.

be embedded in the URL as a URL option. For example, if a job requires an input file `/user/me/input.dat`, the URL specified in the job description will be as follows (given that the file can be found by a Bartender with URL `https://storage/Bartender`):

```
arc:///user/me/input.dat?BartenderURL=https://storage/Bartender
```

4. Chelonia in Operation

Both the Chelonia storage system and its clients can be installed from binary packages (available for several different platforms) or after compiling the source packages. A fully operational storage cloud requires a minimum installation of one instance of every service described above. The Chelonia Administrator manual [19] gives detailed instructions on how to install, configure and run the services. In order for users to interact with Chelonia, several user tools are provided. These are documented both in the Chelonia user manual [20] and Linux man pages and directly through command line calls.

For the user, transfering files to and from Chelonia are simple operations. For example, if a user wants to upload a file `orange.jpg` to Chelonia, he/she can use, e.g., the Chelonia CLI. Assuming that the URL of one or more Bartenders and the required number of file replicas are written in a configuration file, the user gives the command

```
chelonia put orange.jpg /user/me/orange.jpg
```

Note that there is no need for the user to know where files are physically stored or will be stored in Chelonia.

Under the hood of Chelonia, the Bartender receiving the request from the CLI contacts a Librarian to create an entry in the Chelonia namespace. If the Librarian confirms the new entry, the Bartender then contacts a Shepherd to get a transfer URL which is returned to the CLI. When the CLI has uploaded the file, the Shepherd queries the Librarian to find out how many replicas are needed and, if needed, initiates a file transfer to another Shepherd. In the case of downloading a file, the Bartender gets the locations of the file from a Librarian, chooses one of them and contacts the corresponding Shepherd for a transfer URL.

When in operation, the Chelonia storage cloud is a pulsing system where heartbeats are periodically sent from each Shepherd to a Librarian together with information about replicas whose state changed since the last heartbeat. Heartbeats are stored in the A-Hash, thus making them visible to all Librarians in the system. If any of the Librarians notices that a Shepherd is late with its heartbeat, it will mark all the replicas in that Shepherd as offline.

In addition to the heartbeat, the Shepherds periodically check with the Librarians to see if there are sufficient replicas of the files in Chelonia and if the checksums of the replicas are correct. If a file is found to have too few replicas, the Shepherd informs a Bartender about this situation and together they ensure that a new replica is created at a different storage node. A file having too many replicas will also be automatically corrected by Chelonia - the first Shepherd to notice this will mark its replica(s) as unneeded and later delete it (them). Replicas with invalid checksums are marked as invalid, and as soon as possible replaced with a valid replica.

With the Chelonia gateway module, a user can mount external storage systems into the Chelonia namespace. For example, if a Chelonia user has access to a set of files stored in dCache [16] (see Section 7) under `/fruits` he

can add a mount point (say `/my/dCache`) to easily access these data through the Chelonia namespace, i.e., using standard Chelonia commands like

```
chelonia get /my/dCache/fruits/apple.jpg
```

More technically, when the Bartender looks up the entry `/my/dCache` which is a mount point to dCache, it will use the corresponding gateway module to generate an external URL which the client tool will use to contact dCache directly. This way, Chelonia can include third-party storage namespaces in its global namespace by simply storing a single entry.

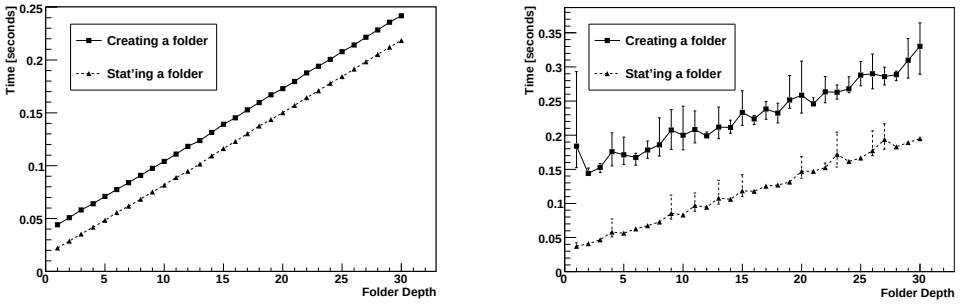
5. System Performance

5.1. Adding and Querying the Status of Files

In a hierarchical file system files are stored in levels of collections and sub-collections. The time to add or get a file depends mainly on two factors; the number of entries in the collection where the file is inserted, and the number of parent collections to the collection where the file is inserted (the depth of the collection). Based on these two factors we have run two different tests:

- **Depth test** tests the performance when creating many levels of sub-collections. The test adds a number of sub-collections to a collection, measures the time to add and stat the sub-collections, then adds a number of sub-sub-collections to one of the sub-collections and so forth. To query a collection at a given level means that all the collections at the lower levels needs to be queried first. In Chelonia, each query causes a message to be sent through TCP. Hence, it is expected that time will increase linearly as the level of collections increases. As every message is of equal size, this test ideally depends only on network latency.
- **Width test** tests the performance when adding many entries (collections) to one collection. The test is carried out by adding a given number of entries to a collection and measuring the time to add each entry and the time to stat the created entry. When adding an entry to a collection, the system needs to check first if the entry exists. In Chelonia, this means that the list of entries in the collection needs to be transferred through TCP. It is therefore expected that the time to add an entry will increase linearly as this list increases and ideally the time will depend only on the bandwidth of the network.

Both tests were run in two types of environments: In the Local Area Network (LAN) setup four computers were connected to the same switch. A centralized A-Hash service, a Librarian, a Bartender, and the client were each run on a separate computer. In the Wide Area Network (WAN) setup, the client and Bartender were located in Uppsala, Sweden, while the Librarian and a replicated A-Hash consisting of three replicas were located in Oslo, Norway.



(a) Services running on LAN

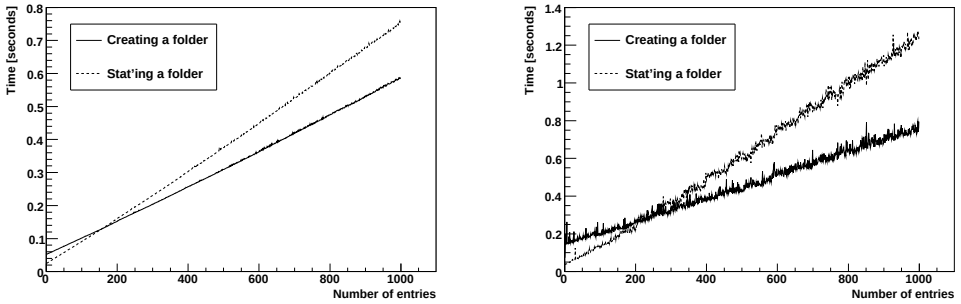
(b) Services running on WAN

Figure 3: Time to add an entry to a collection (continuous line) and time to get status of a collection (dashed line) given the hierarchical depth of the collection.

The test result for the depth test is shown in Figures 3(a) (LAN) and 3(b) (WAN). The continuous lines show the time to create an entry, while the dashed lines show the time to get the status of the collection. All the plots are averages of five samples, with the error bars representing the minimum and maximum values. The LAN test shows a near-perfect linear behaviour, with the error bars too small to be seen. As mentioned earlier, since the packet size for each message is constant in this test, the main bottleneck (apart from Chelonia itself) is the network latency. Since all computers in the LAN test are connected to the same switch we can assume that the latency is near constant. Hence, the LAN test shows that in a very simple network scenario Chelonia works as expected, with the network being the major bottleneck. Notice also that creating an entry consistently takes 0.021 s longer than getting the status of the collection, again corresponding to the extra message needed to create an entry.

In the WAN test, the complexity is a bit increased, as in addition to sending messages over WAN, the A-Hash is now replicated. The time still increases linearily, albeit with more fluctuation due to the WAN environment.

Creating an entry at the first level in the hierarchy now takes 0.11 s longer than getting the status of the collection, corresponding to the fact that the entry needs to be replicated three times. However, at higher levels, getting the status over WAN is actually faster than getting the status over LAN. The effect of the replicated A-Hash will be discussed in more detail in Section 5.4.



(a) Services running on LAN

(b) Services running on WAN

Figure 4: Time to add an entry to a collection (continuous line) and time to get status of a collection (dashed line) given the number of entries already in the collection.

The test results for the width test are shown in Figures 4(a) (LAN) and 4(b) (WAN). The continous lines show the time to add entries to a collection and the dotted lines show the time to get the status of a collection containing the given number of entries. The operations were repeated five times, with the plots showing the averages of these five samples. As expected, the time increase linearily with increasing number of entries. The results of the WAN test fluctuate more than those of the LAN test, which is to be expected; in the LAN test all services are on computers connected to the same switch, while in the WAN test, the services are distributed between two different countries. However, the WAN test shows similar linearity, albeit with a slightly higher response time. It is worth noticing that for the width test, in contrast to the depth test, we see no benefit of using a replicated A-Hash. This is due to the fact that the bandwith is the limiting factor in this test.

An interesting feature of both tests is that while creating an entry takes more time when there are only few entries in the collection, creating new entries is actually faster than stating the collection when the collection has many entries. This is due to the fact that getting the status of a collection requires the metadata of the collection (and hence the list of entries in the collection) to be transfered first from the A-Hash to the Librarian, and second

from the Librarian to the Bartender and last from the Bartender to the client. When creating an entry, neither the Bartender nor the client needs this list of entries, so that less data is transfered between services. However, for fewer entries, creating new entries is more expensive since the Bartender needs to query the Librarian twice, first to check if it is allowed to add the entry and second to actually add it.

5.2. File Replication

The concept of automatic file replication in Chelonia was presented in Section 4. In this section we will demonstrate both how Chelonia works with different file states to ensure that a file always has the requested number of valid (ok) replicas and how Chelonia distributes the replicas in the system in order to ensure maximum fault tolerance of the system.

The test system consists of one Bartender, one Librarian, two A-Hashes (one client and one master) and five Shepherds. All services are deployed within the same LAN. As a a starting point 10 files of 114 MB are uploaded to the system and for each file 4 replicas are requested. Thus the initial setup of the test system contains 40 replicas with an initial distribution of file replicas as shown in Table 1.

Shepherd	Initial	Final
S1	9	9
S2	8	7
S3	8	8
S4	7	9
S5	8	7
Total	40	40

Table 1: Initial and final load distribution of 40 files on 5 Shepherds

The first phase of the file replication test was to kill one of the Shepherd services, S3, of the test system. Chelonia soon recognized the loss of this service (no heartbeat received within one cycle) and started compensating for the lost replicas. File replicas in Chelonia have states ALIVE, OFFLINE, THIRDWHEEL or CREATING which are recorded in the A-Hash. Initially the test system had 40 ALIVE replicas (10 files with 4 replicas each), but when the Librarian did not get the S3 heartbeat it changed the state of the 8 replicas stored in S3 to OFFLINE.

At this point a number of files stored in our Chelonia setup had too few ALIVE replicas. As explained above, the Shepherds check periodically that files with replicas stored on its storage element have the correct number of ALIVE replicas. Thus, in the next cycle the S1, S2, S4 and S5 Shepherds started to create new file replicas which initially appeared in the system with the state CREATING.

Figure 5 gives a graphical overview of the number of replicas and the replica states in the test system every 15 seconds. At 15 s (00:15) all 40 file replicas were ALIVE as explained above, but soon thereafter S3 was turned off and the other Shepherds started creating new replicas. Queried at 30 s (00:30) the system contained 32 ALIVE, 8 OFFLINE and 1 CREATING file replica.

While the system worked on compensating for the loss of S3, the second phase of the replication test was initiated by turning the S3 shepherd online again. The reappearence of the S3 Shepherd can be seen in Figure 5 at 90s (01:30) as a significant increase in the number of ALIVE replicas. In fact there were now too many ALIVE replicas in the system and at 105 s (01:45) 2 replicas were marked as THIRDWHEEL (the Chelonia state for redundant replicas). THIRDWHEEL replicas are removed from Chelonia as soon as possible, and during the next 45 s the system removed all such replicas. A query of the system at 165 s (02:45) shows that the system once again contained 40 ALIVE replicas. The final distribution of replicas between Shepherds is given in Table 1.

5.3. Multi-User Performance

While any distributed storage solution must be robust in terms of multi-user performance, it is particularly important for a grid-enabled storage cloud like Chelonia where hundreds of grid jobs and interactive users are likely to interact with the storage system in parallel. To analyze the performance of Chelonia in such environments we have studied the response time of the system while increasing the number of simultaneous users (multi-client).

Due to limited hardware resources, multiple clients for the tests were simulated by running multiple threads from three different computers. Each client thread creates 50 collections sequentially and tests were done for an increasing number of simultaneous clients. For each test the minimum, average and maximum time used by the client was recorded. Figure 5.3 shows the system response times for up to 100 simultaneous clients using the above-

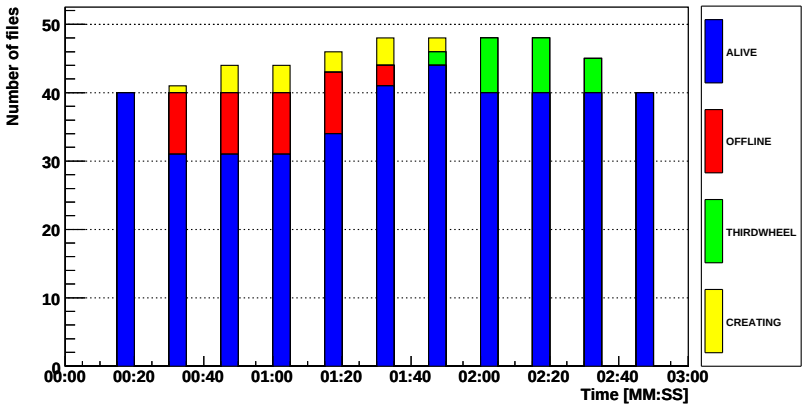


Figure 5: Number of replicas and their corresponding states when querying the test system every 15 seconds. Shepherd S3 is turned offline between 00:15 and 00:30 and back online between 01:15 and 01:30.

mentioned testbed deployment. The shown test was run in a LAN environment with one centralized A-Hash, one Librarian and one Bartender.

The test results show that the response time of the system increases linearly with an increasing number of simultaneous clients. From 40 clients and onwards the difference between the fastest client and slowest client starts to become sizeable. When running 50 clients or more in parallel, it was occasionally observed that a client's request failed due to the heavy load of the system. When this happened, the request was retried until successfully completed, as shown by the slightly fluctuating slope of the mean curve. The same linearity was seen in the corresponding WAN test (not shown), albeit with a factor two higher average time, consistent with the results observed in the depth and width tests in Section 5.1.

As can be noted in Figure 5.3, for more than 30 clients, the maximum times increase approximately linearly while the minimum times are close to constant. The reason for this is a limitation on the number of concurrent threads in the Hosting Environment Daemon (HED). If the number of concurrent requests to HED reaches the threshold limit, the requests are queued so that only a given number of requests are processed at the same time. In the test each client used only one connection for creating all 50 collections. Hence, the fastest request was one that had not been queued so that when

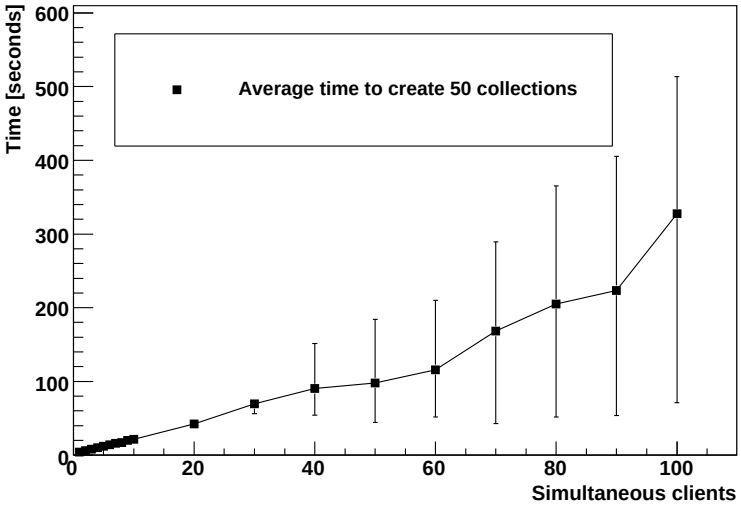


Figure 6: Average (square), minimum (lower bar) and maximum (bar) system response time as a function of the number of simultaneous clients of the system. Each client creates 50 collections sequentially.

the number of requests was above the threshold, the minimum timing did not depend on the total number of clients. On the other hand, the slowest request was queued behind several other requests so that the maximum time increased with the number of simultaneous clients.

5.4. Centralized and Replicated A-Hash

As the A-Hash stores all metadata about files, file locations and shepherds, it is important that the A-Hash is fault tolerant and able to survive even fatal hardware failures. While in theory replicating the A-Hash provides these features, the replication adds complexity to the A-Hash in that all data need to be replicated to all A-Hash instances. Additionally, in the event of a failing A-Hash instance, the Librarians need to seamlessly find and connect to other A-Hashes.

To test the fault tolerance and performance overhead of the replicated A-Hash in a controlled environment, four tests have been set up with services and a client on different computers in the same LAN:

1. **Centralized**: One client contacting a centralized A-Hash was set up as a benchmark, as this is the simplest possible scenario.

2. **Replicated, stable**: One client contacting three A-Hash instances (one A-Hash master, two A-Hash clients) randomly. All the A-Hashes were running during the entire test.
3. **Replicated, unstable clients**: Same setup as in point 2, but with a random A-Hash client restarted every 60 seconds.
4. **Replicated, unstable master**: Same setup as in point 2, but with the master A-Hash restarted every 60 seconds.

While setups 1 and 2 test the differences in having a centralized A-Hash and a replicated A-Hash, setups 3 and 4 tests how the system responds to an unstable environment. In all four setups the system has services available for reading at all times. However, in setup 3 one may need to reestablish connection with an A-Hash client and in setup 4 the system is not available for writing during the election of a new master. During the test the client computer constantly and repeatedly contacted the A-Hash for either writing or reading for 10 minutes. During write tests the client computer reads the newly written entry to ensure it is correctly written.

Reading			
	Minimum (s)	*Average (s)*	*Maximum (s)*
Centralized	0.003399	0.003780	0.013441
Replicated, stable	0.003453	0.003738	0.013261
Replicated, unstable clients	0.003412	0.003754	0.289535
Replicated, unstable master	0.003402	0.003763	1.971131

Writing			
	Minimum (s)	*Average (s)*	*Maximum (s)*
Centralized	0.003828	0.004260	0.014459
Replicated, stable	0.016866	0.033902	1.057602
Replicated, unstable clients	0.016434	0.034239	1.131142
Replicated, unstable master	0.016293	0.044868	60.902862

Table 2: Timings for reading from and writing to a centralized A-Hash compared with a stable replicated A-Hash, a replicated A-Hash where clients are restarted and a replicated A-Hash where the master is restarted.

Table 2 shows timings of reading from and writing to the A-Hash for the four setups. As can be seen, for reading, all four setups have approximately the same performance. Somewhat surprisingly, the replicated setups actually perform better than the centralized setup, even though only one client computer was used for reading. While reading from more A-Hash instances is an

advantage for load balancing in a multi-client scenario, one client computer can only read from one A-Hash instance at a time. Thus, the only difference between the centralized and replicated setups in terms of reading is how entries are looked up internally in each A-Hash instance, where the centralized A-Hash uses a simple Python dictionary, while the replicated A-Hash uses the Berkeley DB which has more advanced handling of cache and memory.

Looking at the write performance in Table 2, there is a more notable difference between the centralized and the replicated setups. On average, writing to the replicated A-Hash takes almost 10 times as long as writing to the centralized A-Hash. The reason for this is that the master A-Hash will not acknowledge that the entry is written until all the A-Hash instances have confirmed that they have written the entry. While this is rather time-consuming, it is more important that the A-Hash is consistent than fast. It is however worth noticing when implementing services using the A-Hash, that reading from a replicated A-Hash is much faster than writing to it.

6. System Stability

While optimal system performance may be good for the day-to-day user experience, the long-term stability of the storage system is an absolute requirement. It does not help to have a response time of a few milliseconds under optimal conditions if the services need to be frequently restarted due to memory leaks or if the servers become unresponsive due to heavy load.

To test the system stability, a Chelonia deployment was run continuously over a week's time. During the test a client regularly interacted with the system, uploading and deleting files and listing collections. The deployment consisted of one Bartender, one Librarian, three A-Hashes and two storage nodes, each consisting of a Shepherd and a Hopi service. All the services and the client ran on separate servers in a LAN environment.

Figure 7 shows the overall memory utilization for seven of the services for the entire run time (top), the A-Hashes and the Librarian in the first 24 hours (bottom left) and the Librarian, the Bartender and one of the Shepherds in the last 37 hours (bottom right). The memory usage was measured by reading the memory usage of each service process in 5 seconds intervals using the Linux `ps` command.

The most crucial part of Chelonia, when it comes to handling server failures, is the replicated A-Hash. If the A-Hash becomes unavailable, the entire system is unavailable. If a client A-Hash goes down, the Librarians

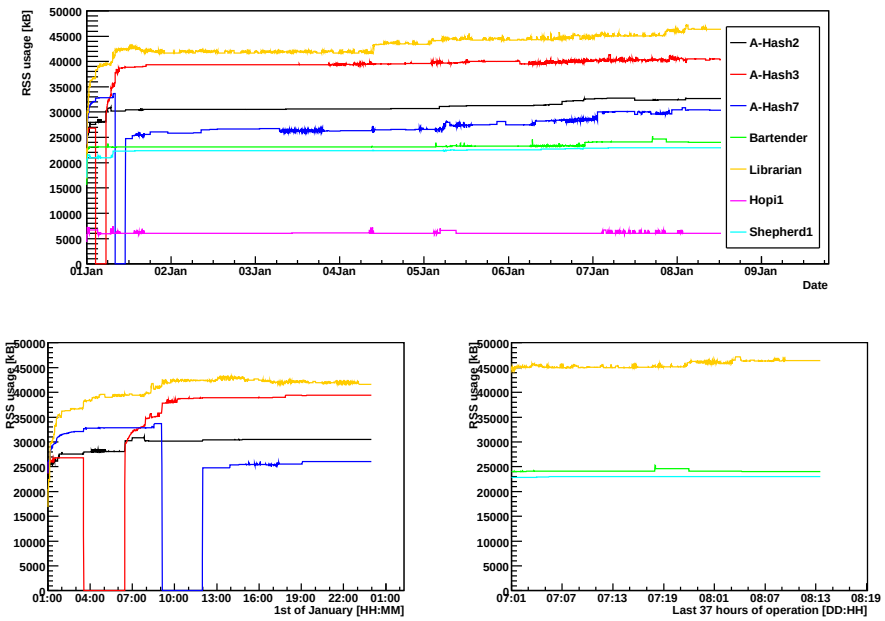


Figure 7: Resident memory utilization of the Chelonia services during an 8 day run.

may need to find a new client. If the master A-Hash goes down, the entire system will be unavailable until a new master is elected. The bottom left of Figure 7 shows the memory consumption of the three A-Hashes, A-Hash2, A-Hash3 and A-Hash7, and the Librarian during the first 24 hours of the stability test. When the test started A-Hash3 (red line) was elected as A-Hash master, and the Librarian started using A-Hash7 (blue line) for read operations. After 2.5 hours, A-Hash3 was stopped (seen by the sudden drop of the red line), thus forcing the two remaining A-Hashes to elect a new master between them. While not visible on the figure, the A-Hash, and hence Chelonia, was unavailable for a 10 seconds period during the election, which incidentally was won by A-Hash2 (black line). After three additional hours, A-Hash3 was restarted, thus causing an increase of memory usage for the master A-Hash as it needed to update A-Hash3 with the latest changes in the database. Eight hours into the test, the same restart procedure was carried out on A-Hash7 which was connected to the Librarian. This time there was no noticeable change in performance. However A-Hash3 increased memory usage when the Librarian connected to it.

The bottom right of Figure 7 shows the memory usage of the Bartender, the Librarian and one of the shepherds in the last 37 hours of the test. Perhaps most noteworthy is that the memory usage is very stable. The main reason for this is due to the way Python, the programming language of Chelonia, allocates memory. As memory allocation is an expensive procedure, Python tends to allocate slightly more memory than needed and avoids releasing the already acquired memory. As a result, the memory utilization gets evened out after a period of time even though the usage of the system varies. During the run-time of the test, files of different sizes were periodically uploaded to and deleted from the system. The slight jump in memory usage for the Bartender (green line) was during an extraordinary upload of a set of large files. This jump was followed by an increase in memory for the Librarian when the files were starting to be replicated between the Shepherds, thus causing extra requests to the Librarian.

Figure 8 shows the CPU load for six of the services. While the load on the A-Hashes, the Bartender and the Librarian are all below 2.5% of the CPU, Shepherd-1 (bottom left) and Shepherd-8 (bottom right) use around 10% and 20%, respectively. While the difference between the Shepherds is due to the fact that Shepherd-1 was run on a server with twice the number of CPU's as the server of Shepherd-8, the difference between the Shepherds and the other services is due to the usage pattern during the test. To confirm that

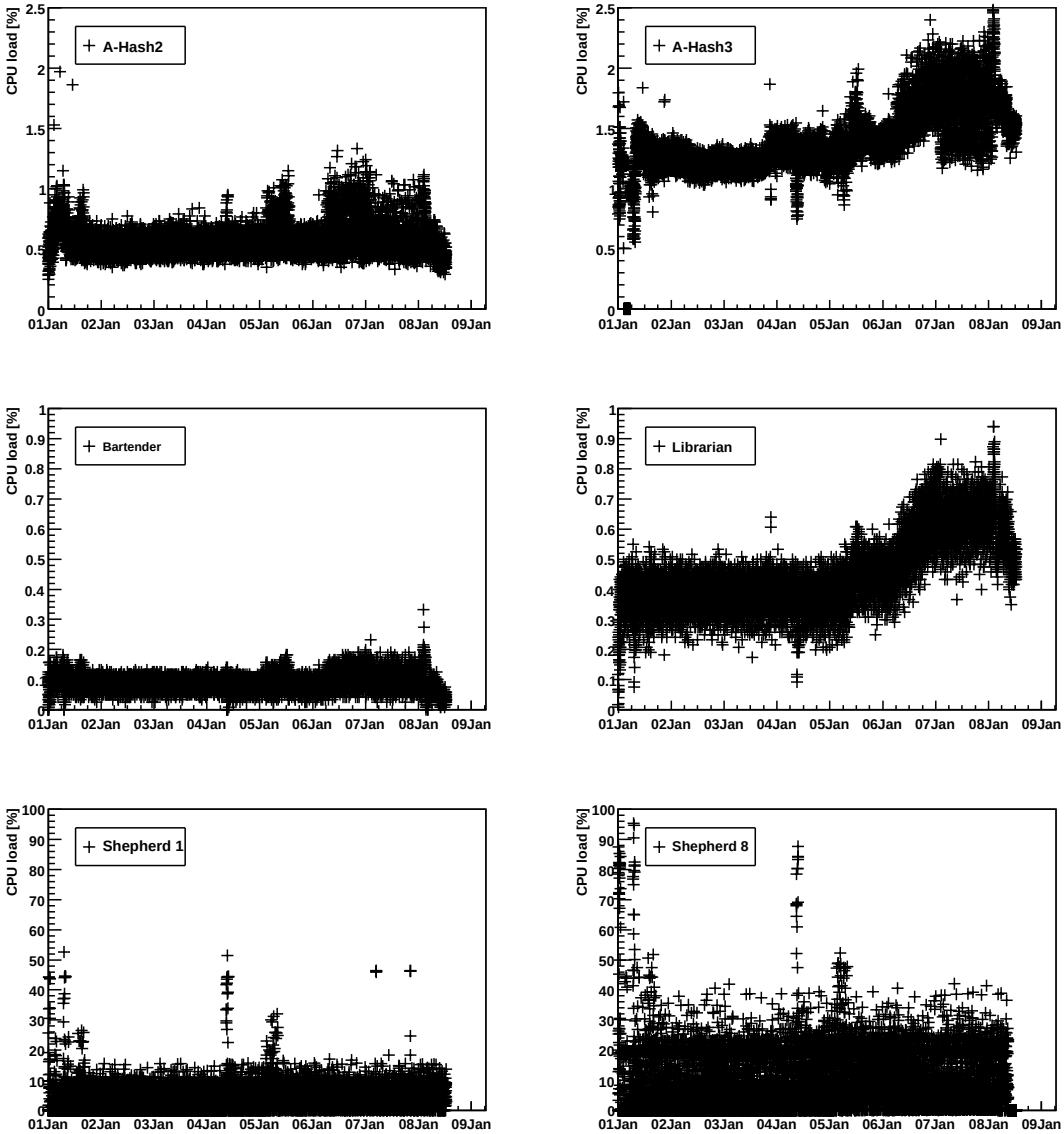


Figure 8: CPU load of the Chelonia services during an 8 day run. Each point is an average of the CPU usage of the previous 60 seconds.

the files stored on the storage node are healthy, the Shepherd calculates a checksum for each file, first when the file is received and later periodically. In periods where no new files are uploaded, the Shepherds use almost no CPU as already stored files don't need frequent checksum calculations. However, when files are frequently uploaded, deleted and re-uploaded, as was the case during the test, the number of checksum calculations, and hence the CPU load, increases significantly. This can particularly be seen on January 1 and 4 when a set of extra large files were uploaded, causing spikes in the CPU load of the two Shepherds. Note also that the spikes occurs at the same time for both Shepherd, as should be expected in a load balanced system.

7. Related Work

There are a number of grid and cloud storage solutions on the market, focused on different storage needs. While direct performance comparison with Chelonia is beyond the scope of this paper, some similarities and differences between Chelonia and related storage solutions are worth mentioning.

In the cloud storage family, Amazon Simple Storage Service (S3) [6] promises unlimited storage and high availability. Amazon uses a two-level namespace as opposed to the hierarchical namespace of Chelonia. In the security model of S3, users have to implicitly trust S3 entirely, whereas in Chelonia users and services need to trust a common independent third party Certificate Authority. Additionally, S3 lacks fine-grained delegation and access control lists are limited to 100 principals, limiting the usability for larger scientific communities [22].

While Chelonia is designed for geographically distributed users and data storage, Hadoop [7] with its file system HDFS is directed towards physically closely-grouped clusters. HDFS builds on the master-slave architecture where a single NameNode works as a master and is responsible for the metadata whereas DataNodes are used to store the actual data. Though similar to Chelonia's metadata service, the NameNode cannot be replicated and may become a bottleneck in the system. Additionally, HDFS uses non-standard protocols for communication and security while Chelonia uses standard protocols like HTTP(S), GridFTP and X509.

When compared to typical grid distributed data management solutions, the closest resemblance with Chelonia is the combination of the storage el-

ement Disk Pool Manager (DPM) and the file catalog LCG [2] File Catalog (LFC) [17]. By registering all files uploaded to different DPM's in LFC one can achieve a single uniform namespace similar to the namespace of Chelonia. However, where Chelonia has a strong coupling between the Bartenders, Librarians and Shepherds to maintain a consistent namespace, DPM and LFC have no coupling such that registration and replication of files is handled on the client side. If a file is removed or altered in DPM, this may not be reflected in LFC. In Chelonia, a change of a file has to be registered through the Bartender and propagated to the Librarian before it is uploaded to the Shepherd.

dCache [16] differs from Chelonia in that dCache has a centralized set of core services while Chelonia is distributed by design. dCache is a service-oriented storage system which combines heterogeneous storage elements to collect several hundreds of terabytes in a single namespace. Originally designed to work on a local area network, dCache has proven to be useful also in a grid environment, with the Nordic Data Grid Facility (NDGF) dCache installation [10] as the largest example. There, the core components, such as the metadata catalogue, indexing service and protocol doors are run in a centralized manner, while the storage pools are distributed. Chelonia, designed to have multiple instances of all services running in a grid environment, will not need a centralized set of core services. Additionally, dCache is relatively difficult to deploy and integrate with new applications. Being a more lightweight and flexible storage solution, Chelonia aims more towards new, less demanding, user groups which are generally less familiar with grid solutions.

Scalla [11] differs from Chelonia in that Scalla is designed for use on centralized clusters, while Chelonia is designed for a distributed environment. Scalla is a widely used software suite consisting of an xrootd server for data access and an olbd server for building scalable xrootd clusters. Originally developed for use with the physics analysis tool ROOT [13] , xrootd offers data access both through the specialized xroot protocol and through other third-party protocols. The combination of the xrootd and olbd components offers a cluster storage system designed for low latency, high bandwidth environments. In contrast, Chelonia is optimized for reliability, consistency and scalability at some cost of latency and is more suitable for the grid environment where wide area network latency can be expected to be high.

[2] LHC (Large Hadron Collider) Computing Grid

Unlike Chelonia, iRODS [8] does not provide any storage itself but is more an interface to other, third-party storage systems. Based on the client-server model, iRODS provides a flexible data grid management system. It allows uniform access to heterogeneous storage resources over a wide area network. Its functionality, with a uniform namespace for several Data Grid Managers and file systems, is quite similar to the functionality offered by our gateway module. However, iRODS uses a database system for maintaining the attributes and states of data and operations. This is not needed with Chelonia's gateway modules.

8. Future Work

In addition to the continous process of improvements and code-hardening (based on user feedback) there are plans to add some new features.

The security of the current one-time URL based file transfers could be improved by adding to the URL a signed hash of the IP and the DN of the user. In this way the file transfer service could do additional authorization, allowing the file transfer only for the same user with the same IP.

Because of the highly modular architecture of both Chelonia and the ARC HED hosting environment, the means of communication between the services could be changed with a small effort. This would enable less secure but more efficient protocols to replace HTTPS/SOAP when Chelonia is deployed inside a firewall. This modularity also allows additional interfaces to Chelonia to be implemented easily. For example, an implementation of the WebDAV protocol would make the system accesible to standard clients built into the mainstream operating systems.

Another possible direction for enhancing the functionality of Chelonia is to add handling of SQL databases. In addition to files, the system could store database objects and use databases as storage nodes to store them. SQL databases allow running extensive queries to get the desired information. In a distributed environment, high availability and consistency is often ensured by the replication of data. Access to multiple copies of the data in the system also allows queries to be run in parallel. Consistent, multiple copies of the data also provdies a simple, transparent platform for scalable access to the same data to a large number of distributed clients.

9. Conclusions

Chelonia is a cloud-like storage solution with grid capabilities. While its core services resembles those of a traditional data grid, the single-entry interface and the capabilities resemble those of storage clouds.

An important part of developing a distributed storage system is proper testing of the system, both in terms of performance and stability. The presented tests are designed to give an understanding of how Chelonia behaves in a real-life environment while at the same time controlling the environment enough to get interpretable results. The tests have shown that Chelonia can handle both deep and wide hierarchies as expected, both in LAN environments and in WAN environments. The system has shown self-healing capabilities, both in terms of individual service stops and in terms of file availability. Multiple clients have accessed the system simultaneously with reasonable performance results and, even more importantly, Chelonia has been heavily used for more than a week with stable performance even with vital services being shut down during the test.

10. Acknowledgements

We wish to thank Mattias Ellert for vital comments and proof reading. Additionally, we would like to thank UPPMAX, NIIF and USIT for providing resources for running the storage tests.

The work has been supported by the European Commission through the KnowARC project (contract nr. 032691) and by the Nordunet3 programme through the NGIn project.

References

[1] `http://www.nordugrid.org/chelonia/`. Chelonia Web page.

[2] `http//:www.nordugrid.org/`. NorduGrid Collaboration.

[3] `http://www.knowarc.eu/`. EU KnowARC project.

[4] `http://www.oracle.com/technology/products/berkeley-db/index.html`. Oracle Berkeley DB.

[5] `http://fuse.sourceforge.net/`. Filesystem in Userspace.

[6] `http://s3.amazonaws.com`. Amazon Simple Storage Service.

[7] `http://hadoop.apache.org/`. Apache Hadoop.

[8] R. Moore A. Rajasekar, M. Wan. Event Processing in Policy Oriented Data Grids. In *Proceedings of Intelligent Event Processing AAAI Spring Symposium*, pages 61–66. AAAI, March 2009.

[9] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Tuecke, Status Of This Memo, L. Liming, and S. Tuecke. GridFTP: Protocol extensions to FTP for the Grid. *GWD-R (Recommendation)*, page 3, 2001.

[10] G Behrmann, P Fuhrmann, M Gronager, and J Kleist. A distributed storage system with dcache. *Journal of Physics: Conference Series*, 119(6):062014 (10pp), 2008.

[11] Chuck Boeheim, Andy Hanushevsky, David Leith, Randy Melen, Richard Mount, Teela Pulliam, and Bill Weeks. Scalla: Scalable Cluster Architecture for Low Latency Access Using xrootd and olbd Servers. Technical report, Stanford Linear Accelerator Center, 2006.

[12] M Branco, D Cameron, B Gaidioz, V Garonne, B Koblitz, M Lassnig, R Rocha, P Salgado, and T Wenaus. Managing atlas data on a petabyte-scale with dq2. *Journal of Physics: Conference Series*, 119(6):062017 (9pp), 2008.

[13] Rene Brun and Fons Rademakers. ROOT – An object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 389(1-2):81–86, April 1997.

[14] D. Cameron, M. Ellert, J. Jönemo, A. Konstantinov, I. Marton, B. Mohn, J. K. Nilsen, M. Nordén, W. Qiang, G. Rőczei, F. Szalai, and A. Wäänänen. *The Hosting Environment of the Advanced Resource Connector middleware*. NorduGrid. NORDUGRID-TECH-19.

[15] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23(3):187 – 200, 2000.

[16] M de Riese, P Fuhrmann, T Mkrtchyan, M Ernst, A Kulyavtsev, V Podstavkov, M Radicke, N Sharma, D Litvintsev, T Perelmutov, and T Hesselroth. *dCache Book*.

[17] Akosh Frohner. `https://twiki.cern.ch/twiki/bin/view/LCG/DataManagementDocumentation`. Official Documentation for LFC and DPM.

[18] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[19] Zs. Nagy, J. K. Nilsen, and S. Toor. *Chelonia Administrator's Manual*. NorduGrid. NORDUGRID-MANUAL-10.

[20] Zs. Nagy, J. K. Nilsen, and S. Toor. *Chelonia User's Manual*. NorduGrid. NORDUGRID-MANUAL-14.

[21] Zs. Nagy, J. K. Nilsen, S. Toor, and B. Mohn. Chelonia – A Self-healing Storage Cloud. To appear in the proceedings of the Cracow Grid Workshop 2009.

[22] Mayur R. Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon S3 for science grids: a viable solution? In *DADC '08: Proceedings of the 2008 international workshop on Data-aware distributed computing*, pages 55–64, New York, NY, USA, 2008. ACM.

[23] Lizhe Wang, Jie Tao, Marcel Kunze, Alvaro Canales Castellanos, David Kramer, and Wolfgang Karl. Scientific cloud computing: Early definition and experience. In *HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 825–830, Washington, DC, USA, 2008. IEEE Computer Society.

A.7 Parallel Monte Carlo simulations on an ARC-enabled computing grid

Article submitted to Computer Physics Communication.

The article "Parallel Monte Carlo simulations on an ARC-enabled computing grid" is an "in-house" production with co-EPF'er Bjørn H. Samset.

The main idea of the article was to present a viable framework for running scientific applications in parallel on the grid.

My contributions were the text in Sections 3.2 and 4 and the performance results. The source code was implemented in collaboration with Bjørn H. Samset.

Parallel Monte Carlo simulations on an ARC–enabled computing grid

Jon K. Nilsen[a], Bjørn H. Samset[a]

[a] *Department of Physics, University of Oslo*

Abstract

Grid computing opens new possibilities for running heavy Monte Carlo simulations of physical systems in parallel. This paper presents GaMPI, a system for running an MPI-based random walker simulation on grid resources. Integrating the ARC middleware and the new storage system Chelonia with the Ganga grid job submission and control system, we show that MPI jobs can be run on a world–wide computing grid with good performance and promising scaling properties. Results for a diffusion Monte Carlo simulation run on three heterogeneous computing clusters in three countries are compared to standard running on a single MPI cluster. We scale to larger, more CPU–intensive system sizes and show that GaMPI can run also these cases efficiently.

Key words: Grid, parallelization, high throughput computing

1. Introduction

Monte Carlo (MC) simulations of physical processes are usually too heavy for a single CPU, or even a multi–core machine, to process in a reasonable time. The natural extension from a single machine is to run the simulations in parallel on a cluster of nodes, with one or more cores each, and have the processes communicate via the Message Passing Interface (MPI).

With the recent improvements in grid computing technology, especially in the areas of storage solutions and user–space job configuration tools, it is interesting to run massively parallel MC simulation on a large computing grid. The main obstacles to doing this have been the lack of efficient communication between parallel threads, which on a cluster would be handled via MPI, and also the complexity of configuring a grid–based MC solution.

In this paper we present a diffusion Monte Carlo (DMC) simulation, running on NorduGrid resources, via the ARC [1] grid middleware. MPI–like inter–process communication is handled through the new Chelonia grid storage system [2, 3], while the jobs are configured and submitted via the Ganga grid user tool [4]. We first describe the DMC simulation itself in Section 2, including a baseline result achieved on a single multi–core computing node. Next, in Section 3, we describe the various components developed or used to run the code on the grid, and how we integrated them to form the system called GaMPI. Finally, in Section 5, we show three series of performance tests where we run various configurations of the DMC simulation on the grid, employing ARC–enabled computing clusters located in several countries.

2. DMC simulations and baseline result

Diffusion Monte Carlo (DMC) has a wide range of application, for example studies of Bose-Einstein condensates of dilute atomic gases (bosonic systems) [5] and studies of so-called quantum dots (fermionic systems) [6], consisting of electrons confined between layers in semi-conductors. In DMC a group of walkers is used to represent a high-dimensional vector and the computation is carried out by having each walker do a random walk in the phase-space. During the computation, some of these walkers may be duplicated or deleted according to a branching factor, i.e., the set of walkers is dynamic in time. A DMC simulation may be regarded as a statistical experiment where each walker represent one sample of the experiment, and the accuracy of the simulation increases with the number of walkers.

DMC is in its nature quite trivially parallelized. There are no interactions between walkers and the order in which the walkers walk is not significant, making it trivial to split the total set of walkers into sub-sets of walkers. The main obstacle to parallelize the DMC simulation is that the branching factor depends on the total number of walkers, and this number needs to be synchronized after each step in the walk. For more information about the DMC algorithm, see [7, 8]. DMC has most commonly been parallelized with MPI to run on a single cluster of machines. While this is a limiting parallelization technique when compared to a Grid parallelization, it serves as a good baseline for the results shown in Section 5.

Figure 1 shows an example DMC simulation, which is based on 30k walkers over 200 timesteps. Each walker consists of 100 particles interacting with each other in three dimensions, so the full simulation handles 3 million par-

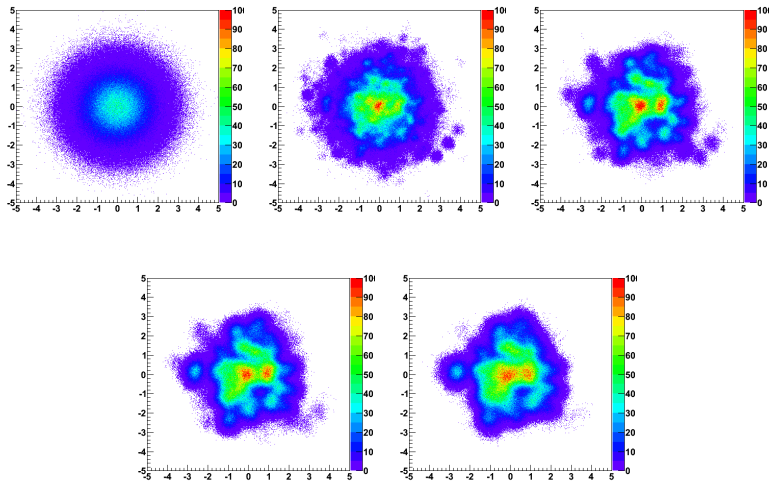


Figure 1: DMC simulation of 30k random walkers, over 200 timesteps. From left to right the figures show timesteps 1, 50, 100, 150 and 200 respectively. The round outliers seen in the middle three panels correspond to single walkers that each contain 100 interacting particles. The evolution from an initial state with a gaussian density to a closer packed system is apparent.

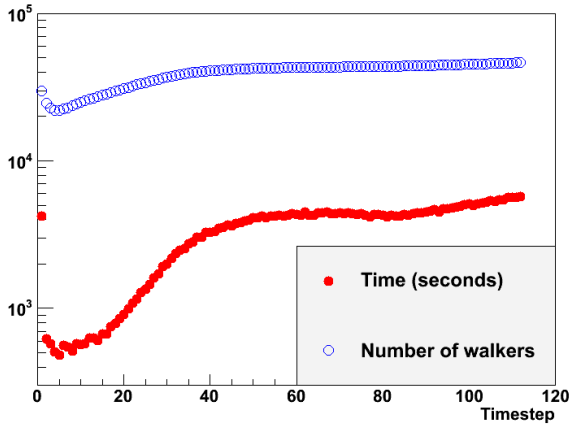


Figure 2: Number of random walkers per timestep in our baseline DMC simulation (open circles), and the time in seconds taken by each timestep (closed circles). The transient behaviour has stabilized by timestep 30.

ticles. Figure 2 shows the evolution with timestep of the number of random walkers in the simulation (open circles) and the time in seconds per timestep (closed circles) when run on an MPI–enabled cluster utilizing 16 CPUs. As expected, we see both that after an initial instability the number of walkers increases smoothly with timestep, and that the simulation time is well correlated with the number of walkers.

This simulation will be compared to GaMPI performance in Section 5. Two observations will be of importance here. First we note that even for 30k walkers on a powerful system, the simulation took an average of more than 10 minutes per timestep for a total simulation time of more than 30 hours. As Figure 2 also shows, adding more walkers to the simulation will increase the time significantly. To quantify this, a test run with 120k walkers on 16 CPUs gave an average of approximately one hour per timestep. Secondly, we see that after a relaxation time of approximately 30 timesteps, our particular system reaches smooth and predictable behaviour. For our performance tests, we can therefore restrict the simulations to 50 timesteps without loss of generality. We therefore aim to use GaMPI to run more walkers in a comparable time, run each timestep more efficiently, or both.

3. Adapting the DMC simulations for use on a computing grid

Having established baseline results for a given DMC simulation on a standard single–cluster implementation, we have constructed a system — called GaMPI — for running the same simulation on an international computing grid. This section describes the elements we have integrated, namely the grid itself (NorduGrid), its middleware (ARC), a leading edge grid storage system (Chelonia), and an extendible grid job definition toolkit (Ganga).

3.1. ARC and NorduGrid

The Advanced Resource Connector (ARC) is an open source software solution enabling production-quality computational and data grids, maintained and developed by the NorduGrid collaboration [1]. ARC software is currently deployed across more than 70 sites in 13 countries, and across a variety of architectures and system flavours. ARC provides all fundamental Grid services such as an information system, resource discovery and monitoring, job submission and management, brokering and data– and resource management.

For the present work, we take NorduGrid to mean a general, widely distributed computing grid that forms the basis of our further tests. In principle all ARC–connected resources were available to us, though for practical reasons the present paper restricts testing to three clusters in Norway, Sweden and the Ukraine.

The clusters used for the test were *Titan* at the University of Oslo, Norway, *Ritsem* at the University of Umeå, Sweden, and *Bitp* at the Bogolyubov Institute for Theoretical Physics in the Ukraine. They ran ARC over various linux flavours, and made available 4000, 500 and 100 CPUs respectively. The clusters also ran different internal file systems, making the test sites very heterogeneous.

All clusters were running normally during the GaMPI tests, and the jobs to be described below were submitted as a normal part of the grid job quotas of the sites.

In analogy with a standard MPI system, ARC provides access to a distributed cluster with a large number of CPUs.

3.2. Chelonia

Chelonia [3, 9] is one of the new components of the web-service based ARC. It is a light-weight, self-healing storage solution with a global hierarchical name-space. And, vital to our use case, Chelonia is accessible from the Grid using the ARC data tools and Python API.

Chelonia consists of a cataloging service (the Librarian), a metadata store (the A-Hash), a storage element front-end (the Shepherd) and a front-end coordination service (the Bartender). The functionality used in this paper are the `putFile`, `getFile` and `delFile` methods to upload, download and delete files from Chelonia using their logical names, `list` to get the content of a file collection and `move` and `unlink` to create and remove hard links to files.

In the MPI system analogy, Chelonia then plays the role of the MPI protocol, allowing the parallel processes to communicate when needed.

3.3. Ganga

The final link in the chain is the Ganga job definition and submission toolkit [4]. It is a python–based, highly flexible frontend that allows grid users to work with processes on their local machine, on a batch system or on several grid flavours from within the same interface. A "job" in Ganga is a python object with information on what application to run, what input

data is needed and where it can be found, what output the job will produce and where to place it, where to run the job (locally, batch system or grid), etc. A job configuration that has been tested on a local machine can then be reused for grid running by changing a single data member of the job object.

One of the grid flavours that Ganga can interact with is ARC. Ganga was therefore ideally suited to be configured as the main controller in the GaMPI setup to be described below.

In the final piece of the MPI analogy, Ganga is the glue core process that sends and controls MPI jobs on the cluster.

4. GaMPI description

This section describes GaMPI in more detail. In brief, it is a system for running the DMC simulation described in Section 2 on a computing grid, and to test the resulting performance. The technical implementation we created for this is sketched in Figure 3, and uses the components described in the previous section. NorduGrid clusters are used for moving the walkers, Ganga for submitting and monitoring jobs, Chelonia as storage pool from which the Grid jobs could get new walkers and upload moved walkers.

In somewhat more detail, the workflow of GaMPI can be described as follows:

1. A master job is running inside Ganga, which creates, submits and monitors the grid jobs. After initializing the submission of the grid jobs (the slaves) the master leaves the management of the grid jobs to Ganga.
2. The master splits the required number of walkers into blocks. The number of blocks is independent of the number of slaves. The walker blocks are uploaded, named with the prefix **walker_block**, to a *walker pool* in Chelonia. Additionally, the master uploads an empty file named **do_timestep**.
3. The slaves check if the file **do_timestep** exists and, if it does, try to download a block of walkers, chosen randomly from the files with prefix **walker_block**. When all the walkers in the block are moved and diffusion and branching is done on the walker block, the slave uploads the walker block with the prefix **moved_walker_block**.
4. The main variables of interest after a timestep are the number of walkers (which is dynamic due to the branching part of DMC) and the ground state energy after moving the walkers. Chelonia supports arbitrary metadata for a file, enabling these two variables to be appended

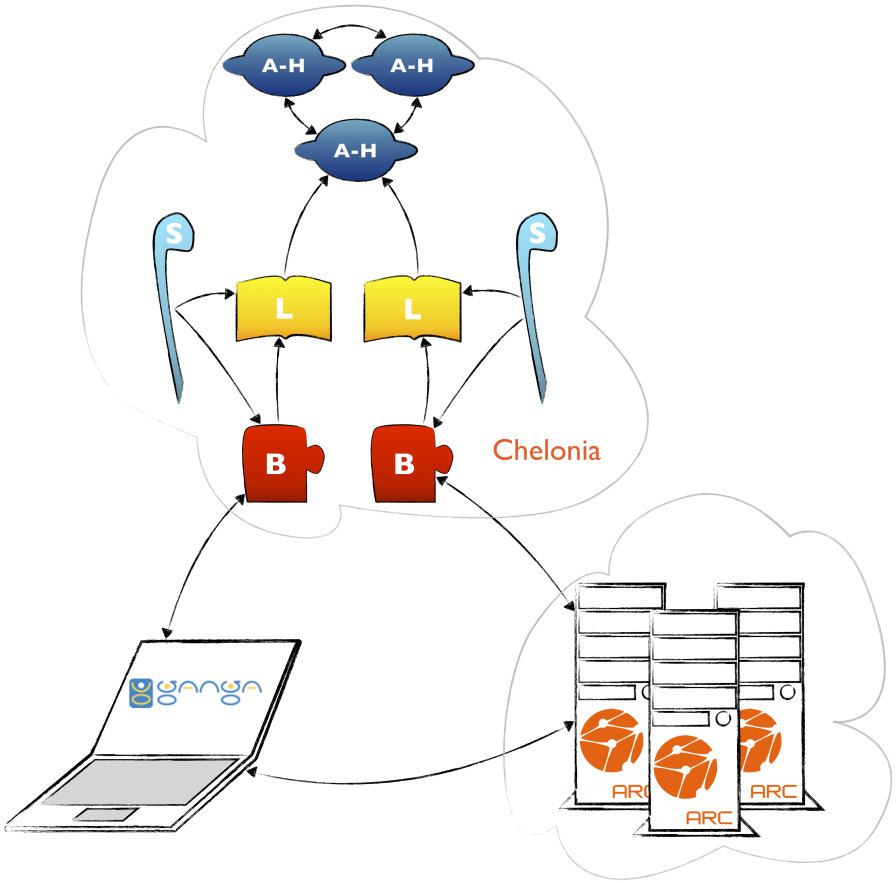


Figure 3: The communication flow between Ganga (laptop), Chelonia (top cloud) and ARC (bottom cloud), and between the services internally in Chelonia. Communication between Ganga and the grid is carried out by exchanging files through the Chelonia cloud. Internally, Chelonia is set up with two sets of Bartender-Librarian-Shepherd and a ring of three A-Hash replicas

to the metadata of the walker block file. This way, the master can simply do a **stat** on the walker block files to calculate the global energy and number of walkers, without actually downloading all the walkers.

5. The master monitors the walker pool, and as soon as there are only moved walker blocks in the walker pool, the master removes the file named **do_timestep**, thus telling the slaves that they can poll the walker pool less frequently.
6. The master then checks the metadata of the files to get the ground state energy and the number of walkers, thus avoiding a full download of all walker blocks. Each timestep involves calculating a trial energy, which is a best estimate of the ground state energy. The best estimate of the ground state energy is the mean of the energy per walker. For every time-step the master acquires this mean from the walker blocks and refreshes the energy in the metadata. When the energy is updated, the master renames the walker blocks with prefix **walker_block** and again uploads the **do_timestep** file.
7. After repeating steps (1) to (5) for the required number of time steps, the master uploads the file **stop**. As soon as the slaves see the **stop** file, they end their work and exit.

To avoid that more than one slave works on the same walker block, the slaves will try to rename the walker block to have the prefix **.walker_block** before downloading it. In Chelonia, renaming is an atomic operation and it is not allowed to overwrite an already existing file. Hence, only the first slave to rename the walker block will get the walkers, thus limiting data transfer and avoiding multiple slaves working on the same task. It should be mentioned that a way to further reduce the amount of network load could be to introduce caching on the slaves where each slave maintained a local walker pool, and tried to reserve these files first. However, GaMPI being in a proof-of-concept state at the moment, caching is not yet implemented and the gain remains to be seen.

After a number of time-steps, the walker blocks may be unbalanced. The master, knowing the number of walkers per walker block, checks after each time-step if any walker block is more than 50% above or below the mean of the block-sizes. If so, the master downloads the walker blocks and iterates through them, taking as much from the largest walker block as needed for the smallest walker block to be at the mean of walker blocks. To avoid superfluous file transfers the iteration stops as soon as no walker blocks are

above or below 10% of the mean.

Figure 3 shows the communication flow between the components of GaMPI. Here, Ganga and the grid jobs communicate by exchanging files through Chelonia. Chelonia itself is set up with two sets of Bartender-Librarian-Shepherd and a ring of three A-Hash replicas. In this setup, any of the two Bartenders can be used and will yield the same result on queries, and an uploaded file can end up on either of the two Shepherds. However, each of the Shepherds contacts only one Bartender and one Librarian. The reason for such a setup is that a Shepherd will reuse the connection to the first Bartender/Librarian it successfully contacts. With only two of each service, there is a great chance of both Shepherds constantly communicating with the same Librarian and/or Bartender. Thus, to balance the load between the services, each Shepherd gets assigned one Librarian and Bartender. This means that if one Librarian or one Bartender stops, one of the Shepherds will stop. While this is not optimal in a production setup, it may make sense in a simple setup where one set of Shepherd, Librarian and Bartender is run on the same machine, as is the case for the results shown below.

5. DMC and performance results on the grid

This section presents performance results on running the DMC simulation described in Section 2 via the GaMPI framework described above. Three major test series were run, and are described in turn below. The first establishes a baseline result on an MPI cluster and then shows that the same performance can be achieved using our GaMPI setup on the grid. The second test studies the scaling of GaMPI performance with the number of grid jobs run in parallel, and the third studies the scaling to more complex simulations by increasing the number of random walkers.

5.1. From cluster to grid

For the first test we ran three identical simulations, each with 30k walkers, 50 timesteps and 16 CPUs or grid jobs. The jobs were sent (a) to a single MPI–enabled cluster, (b) to the same cluster but using the GaMPI setup, and (c) to a set of three clusters in three different countries.

The results are presented in Table 1. Our performance metric is the average walltime per timestep that the simulations used, in addition to the maximum and minimum time taken by a single timestep. The max and min values indicate both variations in actual computing time due to a varying

Time per iteration for 30000 walkers, 16 CPUs				
Case	Description	*Minimum (s)*	*Average (s)*	*Maximum (s)*
a	MPI, single cluster	735	790	838
b	GaMPI, single cluster	545	778	1191
c	GaMPI, three clusters	738	885	1157

Table 1: Timings per iteration (time-step) for running 50 iteration of diffusion Monte Carlo using regular MPI on a single cluster, using GaMPI on the same cluster, and running GaMPI with grid jobs distributed between three clusters in three different countries. In all runs 16 CPUs were used in parallel. Note that the timings do not take into account the number of walkers in each time-step. Values are for timesteps 30 through 50.

number of walkers per block, and in the load on the system used in each case. To capture the realistic variation between relatively similar timesteps, we calculate the average and find max/min values for timesteps 30 through 50 only. Figure 2 illustrates that after the initial 30 timesteps the system has reached a relatively stable, only slowly expanding state.

Table 1 shows that for our baseline test (a) on a normal MPI cluster, the average time per step was somewhat in excess of 10 minutes. Tests (b) and (c) show an approximately identical behaviour, with comparable average times but with larger variations between timesteps. This means that there is no net performance drop in going from a standard MPI system to a grid– and Chelonia–storage–based implementation, but that GaMPI is more vulnerable to e.g. the load on the clusters or the network connections. Test (c) proves that the simulations can also be run over a wide area network without a major performance hit. For DMC–style MC calculations, a very significant fraction of the process time is spent doing pure calculations, meaning that changing the interprocess communication protocol should not significantly impact the absolute performance. This is consistent with the results in this section.

5.2. Scaling with number of CPUs

The second test series involved scaling up the number of CPUs, or in our case parallel grid jobs, used. The baseline, called case (d) in Table 2, is similar to case (c) of the first test, with 16 parallel processes on three computing clusters connected via ARC and Chelonia. However, to tax the systems we here increased the number of random walkers to 60k. Cases (e)

Time per iteration for 60000 walkers				
Case	Processes	*Minimum (s)*	*Average (s)*	*Maximum (s)*
d	16	1388	1656	1993
e	32	785	1073	1454
f	60	692	802	1060

Table 2: Timings per iteration (time-step) for running 50 iteration of diffusion Monte Carlo using GaMPI with 16, 32 and 60 distributed grid jobs. The initial number of walkers is kept constant. Values are for timesteps 30 through 50.

and (f) then employed 32 and 60 parallel processes respectively.[1]

Results are shown in Table 2, where we again give average time per timestep (fourth column), as well as maximum and minimum times for a single step. The results are also illustrated in Figure 4, where the average times are shown as red circles and the max and min times are indicated by the red hatched band.

While the variations between individual steps are still large, as expected on heterogeneous systems running steps with a varying number of walkers, there is a systematic decrease in average times with the number of processes. This comes from each process having to process fewer walkers per timestep. The total simulation time, $T = t_{Average} \cdot N_{steps}$, was reduced from 19 to 10 hours when going from 16 to 60 parallel processes.

5.3. Scaling the system size

For the third test we performed a scaling from small to large systems, in our case meaning changing the number of random walkers in the simulation. To keep our performance metric meaningful, we also varied the number of processes but kept the ratio $N_{walkers}/N_{processes}$ approximately constant.[2]

Both the test cases and results are presented in Table 3. Timing results are also shown as blue triangles and a blue hatched band in Figure 4. Our largest system (i), 120k walkers, was simulated through 50 timesteps in 17 hours, using 60 parallel processes. The smallest system (g), corresponding

[1]Using 60 and not 64 CPUs for case (f) comes from a known limit in Chelonia. All components of Chelonia are designed to be replicated to ensure scaling, but since this would introduce additional uncertainties we opted to reduce the number of parallel threads to 60 and rather stay with a single storage instance.

[2]The deviation from constant comes from the limitation to 60 processes discussed in a previous footnote.

Mean time for increasing number of walkers and CPUs				
Case	Description	*Minimum (s)*	*Average (s)*	*Maximum (s)*
g	16 CPUs, 30k walkers	738	885	1157
h	32 CPUs, 60k walkers	785	1073	1454
i	60 CPUs, 120k walkers	1245	1387	1705

Table 3: Timings per iteration (time-step) for running 50 iteration of diffusion Monte Carlo using GaMPI with 16, 32 and 60 distributed grid jobs. The ratio of initial walkers and CPUs is kept constant. Values are for timesteps 30 — 50.

to the single–cluster MPI baseline result above, took 9 hours but processed only 30k walkers.

Figure 5 gives a graphic illustration of going from 30k (top row) to 120k (bottom row) walkers. The left plots show the gaussian initial state of the DMC system, the middle plots show the 25th timestep and the right plots show the walker positions after the 50th timestep. In all cases it is even visually clear that the increased number of walkers in the simulation will lead to a better determination of the ground state energy, which is the goal of the exercise. See ref. [7] for more detail.

6. Conclusions

A diffusion Monte Carlo simulation has been successfully run on a world-wide computing grid, using a combination of the ARC grid middleware, Chelonia storage and the Ganga grid job control interface (GaMPI).

We have shown not only that it is technically feasible to run a large set of complex MC simulations that require inter–process communication in this environment, but also that there is no appreciable performance loss associated with going from MPI communications to a grid–based storage, for a system that can be properly modularized. When scaling to larger numbers of parallel CPUs we see a smooth decrease in the time spent per timestep for a given simulation. Also we see no unexpected performance issues when scaling to more complex simulations, beyond what is expected from the increased need for inter-process communication.

The goal of this exercise was to run a DMC simulation on the grid, either going beyond the number of random walkers feasible on a 16 CPU cluster in a reasonable timeframe, or making each timestep of the simulation more efficient. We have shown that both can be done. Having proven the concept

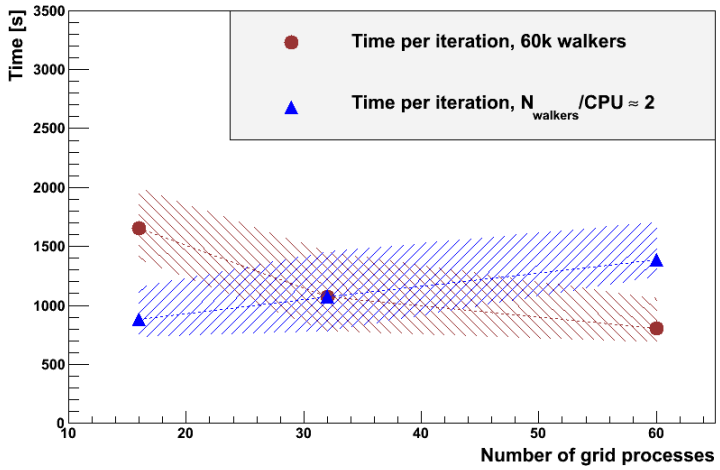


Figure 4: Average time for timesteps 30 through 50 for the data series shown in tables 2 (cases d, e, f, circles) and 3 (cases g, h, i, triangles), versus the number of grid processes. Maximum and minimum times are indicated by the hashed bands.

of running MPI–type simulations on a grid, it is our intention to go on to use the GaMPI setup for studies of more realistic physical systems.

References

[1] Ellert, M. et al., Future Generation Computer Systems **23** (2007) 219 .

[2] Nilsen, J. K., Toor, S., Nagy, Z., Mohn, B., and Read, A. L., Submitted to Future Generation Computing Systems (2010).

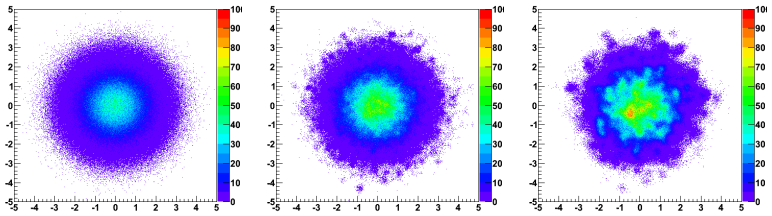
[3] Nagy, Z., Nilsen, J. K., Toor, S., and Mohn, B., (2010), To appear in the proceedings of the Cracow Grid Workshop 2009.

[4] Moscicki, J. et al., Comp. Phys. Comm. **180** (2009) 2303.

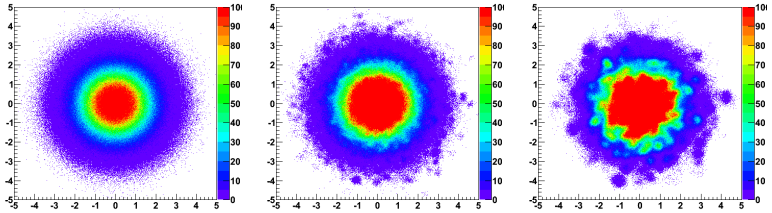
[5] DuBois, J. and Glyde, H., Phys. Rev. A **68** (2003).

[6] Harju, A., Journal of Low Temperature Physics **140** (2005) 181.

[7] Nilsen, J. K., Comp. Phys. Comm. **177** (2007) 799.



(a) 30k, step 1 (b) 30k, step 25 (c) 30k, step 50



(d) 120k, step 1 (e) 120k, step 25 (f) 120k, step 50

Figure 5: Time evolution of the random walker DMC system, for 30k walkers (top row) and 120k walkers (bottom row).

[8] Nilsen, J. K., Cai, X., Hoyland, B., and Langtangen, H. P., Submitted to Computational Science & Discovery (2010).

[9] `http://www.nordugrid.org/chelonia/`, Chelonia Web page.