Parallel Algorithms for Semiclassical Quantum Dynamics

Manfred Liebmann * David Sattlegger*

May 22, 2015

* Institute for Mathematics and Scientific Computing, University of Graz,
Heinrichstraße 36, 8010 Graz, Austria
manfred.liebmann@uni-graz.at

* Fakultät für Mathematik, Technische Universität München,
Boltzmannstr. 3, 85747 Garching bei München, Germany
david.sattlegger@tum.de

We investigate aspects of the practical parallel implementation of the Herman–Kluk and Egorov approaches to semiclassical quantum dynamics on modern multicore and manycore hardware architectures. We discuss an automatic code generator implemented in the computer algebra system Mathematica for computational kernel of the semiclassical methods for quantum systems with tens or even hundreds of degrees of freedom. Furthermore we investigate an explicit vectorization library to speed up the computation on modern CPUs with vector processing capability. Finally, we compare an implementation of our methods on graphics processing units (GPUs) that gives a significant performance advantage over even explicitly vectorized CPU code. The numerical experiments were conducted on a variety of of high dimensional model problems, in particular for the Henon–Heiles potential.

1 Introduction

We investigate parallel algorithms for the Herman–Kluk and the Egorov approach to semiclassical quantum dynamics in high dimensional systems. Both methods are based on the time evolution of classical point mechanics with typically millions of particles which allows a straight forward parallelization. However, there are two areas that make the practical implementation challenging.

The first is the formulation of the time stepping scheme itself. We use a higher order method based on the Størmer–Verlet scheme. While it is simple to write down the time stepping scheme for low dimensional systems, it becomes very tedious to formulate the dynamics of system with tens or hundreds of dimensions. For this reason we developed an automated code generator in the computer algebra system Mathematica that is capable of generating the complete source code for C++ classes. This approach is advantageous because we can use the built-in symbolic differentiation and simplification methods of Mathematica.

The second area is the challenge to adapt the code for the various parallelization approaches and technologies. Today we have an ever expanding range of technologies to speed up parallel programs. On the one side we now have multicore processors with up to 18 cores that include vector processing capabilities with vector widths of up to 16 single precision and 8 double precision IEEE 754 floating point numbers. On the other side we have accelerators, foremost highly capable graphics processing units in the form of Nvidia Tesla series of processors, but recently also the Intel Xeon Phi manycore architecture made an entry to the scene. This multitude of technologies provides a formidable challenge for the software design for the semiclassical methods. We tackled this problem on one hand with a high level vector processing friendly structure of arrays (SoA) data layout for all core data structures to run the Verlet integrator and on the other hand with an explicit vectorization library that overloads all needed arithmetic operators and functions. This allows for the reuse of the automatically generated integration routines with only a change from the build in data types float and double to their associated vector types that are handled by the vectorization library. This approach makes it possible to adapt the code nearly effortlessly to support the SSE, AVX, and AVX512 vector units. For the graphics processors the structure of arrays data layout also enables a straight forward implementation of CUDA kernels for parallel processing using the compiler supported single instruction multiple threads (SIMT) approach. With the SIMT approach there is no need of an explicit vectorization of the code on the graphics processor in contrast to the traditional vector processing enabled CPUs. To compare the different hardware architectures with respect to their performance on the semiclassical algorithms we implemented a Henon–Heiles system for up to 512 dimensions.

The outline of the paper is as follows. First we introduce the theoretical background

of the Herman–Kluk and the Egorov approaches to semiclassical quantum dynamics. Chapter 3 discusses the implementation of these algorithms. The subsequent chapter describes in details the use of a code generator within Mathematica. Chapter 5 gives an overview of the explicit vectorization library to support the vector extensions on modern CPUs. After that we briefly elaborate on the CUDA implementation of the computational kernels of the integrator. Chapter 7 collects various benchmark results for the different architectures and compares the results with previous results form the literature.

2 Some Methods in Semiclassical Quantum Dynamics

A model for the motion of the nuclei in a molecule originates from the famous Born–Oppenheimer approximation. The resulting effective dynamics is governed by the time-dependent semiclassical Schrödinger equation

$$\mathrm{i}\,\varepsilon\,\partial_t\psi(t,x) = -\frac{\varepsilon^2}{2}\Delta\,\psi(t,x) + V(x)\,\psi(t,x), \qquad \psi(0,\cdot) = \psi_0. \tag{1}$$

The small parameter $\varepsilon > 0$ reflects the square root of the mass ratio between electrons and nuclei in a molecule and typically ranges between 10^{-3} and 10^{-2}. The linear operator $\mathrm{H}^\varepsilon = -\frac{\varepsilon^2}{2}\Delta + V$ on the right-hand side of the equation is called Hamiltonian. It is supposed to be self-adjoint on $\mathrm{L}^2(\mathbb{R}^d)$ and its potential $V : \mathbb{R}^d \to \mathbb{R}$ to be a smooth function of sub-quadratic growth. By means of the spectral theorem H^ε gives rise to a well-defined unitary group of operators

$$\mathrm{U}_t^\varepsilon = \mathrm{e}^{-\mathrm{i}\mathrm{H}^\varepsilon t/\varepsilon} \tag{2}$$

for all times $t \in \mathbb{R}$. This provides existence and uniqueness of the solution

$$\psi(t,\cdot) = \mathrm{U}_t^\varepsilon\,\psi_0 \tag{3}$$

to (1) for all square integrable initial data $\psi_0 \in \mathrm{L}^2(\mathbb{R}^d)$. Typical solutions are wavepackets with a width of order $\sqrt{\varepsilon}$, wavelength of order ε, and an envelope moving at velocity of order one. For small ε, grid-based numerical methods need a very fine resolution and thus become expensive even in one and computationally infeasible in higher dimensions. In this situation, semiclassical methods come into play. They approximate the quantum evolution by solving the corresponding classical Hamiltonian systems.

2.1 Classical mechanics

We shall briefly review the Hamiltonian formulation of classical mechanics in order to introduce notations for the objects we need later on. Let us consider a classical physical system having d degrees of freedom. The dynamics of the system may be described by

a smooth function on phase space $\mathbb{R}^{2d}$. We call this function Hamiltonian function and denote it by $h : \mathbb{R}^{2d} \to \mathbb{R}$. It defines the equations of motion

$$\dot{z} = \mathcal{J}\nabla h(z), \qquad z(0) = z_0 \tag{4}$$

which depend on the underlying standard symplectic form on phase space $\mathbb{R}^{2d}$ represented by the matrix

$$\mathcal{J} = \begin{pmatrix} 0 & \mathbf{I} \\ -\mathbf{I} & 0 \end{pmatrix} \in \mathbb{R}^{2d\times 2d}.$$

For suitable Hamiltonian functions the above system of ordinary differential equations have a unique solution. This allows us to define the classical flow

$$\Phi^t : \mathbb{R}^{2d} \to \mathbb{R}^{2d}, \quad z_0 \mapsto \begin{pmatrix} X^t(z_0) \\ \Xi^t(z_0) \end{pmatrix}$$

such that $z(t) = \Phi^t(z)$ satisfies (4) with initial datum $z(0) = z_0$.

2.2 Weyl quantization

Quantization is a mathematical setting for the Bohr correspondence principle between the classical and the quantum world. It associates a self-adjoint operator on $\mathrm{L}^2(\mathbb{R}^d)$ (quantum observable) to every real-valued function on the phase space (classical observable). A natural way of doing so is called Weyl quantization and was first introduced in [Wey27]. For Schwartz functions $a \in \mathcal{S}(\mathbb{R}^{2d})$ and $\psi \in \mathcal{S}(\mathbb{R}^d)$ it is given by the absolutely convergent integral

$$\left(\mathrm{op}^\varepsilon(a)\psi\right)(x) = (2\pi\varepsilon)^{-d} \int_{\mathbb{R}^{2d}} h\left(\tfrac{1}{2}(x+y), \xi\right) \mathrm{e}^{\frac{\mathrm{i}}{\varepsilon}\xi(x-y)} \psi(y)\, dy\, d\xi. \tag{5}$$

2.3 The Herman–Kluk propagator

Typical ansatz functions in semiclassical quantum dynamics have the correct localization both in space and frequency as for example the Gaussian wavepacket

$$g_z^\varepsilon(x) = (\pi\varepsilon)^{-d/4} \exp(-\tfrac{1}{2\varepsilon}|x-q|^2 + \tfrac{\mathrm{i}}{\varepsilon} p \cdot (x-q)), \qquad x \in \mathbb{R}^d \tag{6}$$

do. They are parametrised by the phase space point $z = (q, p) \in \mathbb{R}^{2d}$. By means of these wavepackets any square integrable function $\psi \in \mathrm{L}^2(\mathbb{R}^d)$ can be decomposed as

$$\psi = (2\pi\varepsilon)^{-d} \int_{\mathbb{R}^{2d}} \langle g_z^\varepsilon, \psi \rangle\, g_z^\varepsilon dz.$$

The precise meaning of this integral is given by the inversion formula of the Fourier–Bros–Iagolnitzer transform, see [Mar02] for details. From this we get the formal equation

$$\mathrm{U}_t^\varepsilon \psi_0 = (2\pi\varepsilon)^{-d} \int_{\mathbb{R}^{2d}} \langle g_z^\varepsilon, \psi_0 \rangle \, (\mathrm{U}_t^\varepsilon g_z^\varepsilon) \, dz$$

that motivates to approximate the unitary propagator by continuously superimposing approximately propagated Gaussian wavepackets. In the chemical literature such methods are known as Initial Value Representations, cf. [TW04]. The propagator introduced by Herman and Kluk in [HK84] can be seen in this context. It is defined by

$$\mathcal{I}_t^\varepsilon : \mathrm{L}^2(\mathbb{R}^d) \to \mathrm{L}^2(\mathbb{R}^d), \quad \psi \mapsto (2\pi\varepsilon)^{-d} \int_{\mathbb{R}^{2d}} u(t,z) \mathrm{e}^{\frac{\mathrm{i}}{\varepsilon} S(t,z)} \, \langle g_z^\varepsilon, \psi \rangle \, g^\varepsilon_{\Phi^t(z)} dz. \tag{7}$$

The factor

$$u(t,z) := \sqrt{2^{-d} \det \left(\partial_q X^t(z) + \partial_p \Xi^t(z) + \mathrm{i}(\partial_q \Xi^t(z) - \partial_p X^t(z)) \right)}$$

is often called Herman–Kluk prefactor. It is computed from the Jacobian matrix of the flow,

$$(D\Phi^t)(z) = \begin{pmatrix} \partial_q X^t(z) & \partial_p X^t(z) \\ \partial_q \Xi^t(z) & \partial_p \Xi^t(z) \end{pmatrix} \in \mathbb{R}^{2d \times 2d}. \tag{8}$$

The phase factor $\mathrm{e}^{\frac{\mathrm{i}}{\varepsilon} S(t,z)}$ is derived from the classical action integral

$$S(t,z) = \int_0^t \left(\tfrac{d}{d\tau} X^\tau(z) \cdot \Xi^\tau(z) - h(\Phi^\tau(z)) \right) d\tau. \tag{9}$$

In [RS07] it is shown that the Herman–Kluk propagator is a bounded operator on $\mathrm{L}^2(\mathbb{R}^d)$. Furthermore, it is shown in [SR09, Theorem 2] that $\mathcal{I}_t^\varepsilon$ approximates the unitary propagator (2) in the sense that for every $T > 0$ there exists $C > 0$ such that for all $\varepsilon > 0$

$$\sup_{t \in [0,T]} \| \mathcal{I}_t^\varepsilon - \mathrm{U}_t^\varepsilon \| \leq C\varepsilon.$$

This provides us with an approach to calculate the full solution to potentially high dimensional quantum systems up to an error of order of ε by only computing solutions to the classical flow of (4) and its Jacobian (8) as well as the corresponding action (9). We describe a numerical scheme including a rigorous analysis on the discretization errors in [LS15].

2.4 Egorov's Theorem

Another approach that calculates expectation values of quantum mechanical observables instead of the full solution to the Schrödingier equation (1) takes advantage of Egorov's Theorem. First introduced in [Ego69] the theorem states that

$$\langle \mathrm{op}^\varepsilon(a) \psi(t,\cdot), \psi(t,\cdot) \rangle = \langle \mathrm{op}^\varepsilon(a \circ \Phi^t) \psi_0, \psi_0 \rangle + \mathcal{O}(\varepsilon^2) \tag{10}$$

for the solution ψ of the semiclassical Schrödinger equation (1) and a general Weyl-quantized operator $\mathrm{op}^\varepsilon(a)$. This means that we can calculate the expectation values up to an error of order ε^2 by only knowing the dynamics of the corresponding classical equations of motion. For a proof of this result we refer to [BR02]. A numerical procedure for the actual computation is developed in [LR10]. The key idea is to rephrase (10) in terms of the Wigner function $W(\psi_0) : \mathbb{R}^d \to \mathbb{R}$ corresponding to ψ_0 which is defined by

$$W(\psi_0)(z) = (2\pi\varepsilon)^{-d} \int_{\mathbb{R}^d} \mathrm{e}^{\frac{\mathrm{i}}{\varepsilon} p\cdot y}\, \overline{\psi_0\,(q + y/2)}\; \psi_0\,(q - y/2)\;\, dy$$

for every $z = (q, p) \in \mathbb{R}^{2d}$. Approximation (10) may then be restated as

$$\langle \mathrm{op}^\varepsilon(a)\psi(t,\cdot), \psi(t,\cdot)\rangle = \int_{\mathbb{R}^{2d}} (a \circ \Phi^t)(z)\, W(\psi_0)(z)\, dz + \mathcal{O}(\varepsilon^2).$$

This serves as a basis for numerical computations as we shall elaborate in the following chapter.

3 Discretization

So far we have seen the theoretic properties of the Herman–Kluk (HK) propagator and the method based on Egorov's Theorem. The algorithms we will apply for actual computations are based on the ones proposed in [LS15] and [LR10] respectively. In both cases we have to evaluate high-dimensional integrals over classical phase space, see (7) and (10). For a system of one or two degrees of freedom one could still consider grid based quadrature methods. Keep in mind however that phase space has twice that dimension and grid based approaches are no longer practical for $d > 3$. We shall therefore turn to Monte Carlo and quasi-Monte Carlo quadrature. They are grid free methods that permit the evaluation of high dimensional integrals. In addition, their shortcoming of having a low order of accuracy is of little consequence since the total error is already dominated by the asymptotic error.

For the Egorov method we have to apply these quadrature rules on (10) and get

$$\int_{\mathbb{R}^{2d}} (a \circ \Phi^t)(z)\, W(\psi_0)(z)\, dz \approx \frac{1}{M} \sum_{m=1}^{M} (a \circ \Phi^t)(z_m). \tag{11}$$

Hence we only need to calculate the classical flow Φ^t for all points $z_1, \ldots, z_M$ which are sampled from the Wigner function of the initial wave function $W(\psi_0)$. The HK propagator (7) on the other hand can be reinterpreted as a weighted integral over phase space

$$\mathcal{I}_t^\varepsilon \psi_0 = \int_{\mathbb{R}^{2d}} r_0(z)\, u(t,z)\, \mathrm{e}^{\frac{\mathrm{i}}{\varepsilon} S(t,z)}\, g^\varepsilon_{\Phi^t(z)}\, d\mu_0(z)$$

where r_0 is a smooth function and μ_0 a probability density that both depend solely on a suitable initial wave function ψ_0. We will use the above quadrature methods to approximate it and get

$$\mathcal{I}_t^\varepsilon \psi_0 \approx \frac{1}{M} \sum_{m=1}^{M} r_0(z_m)\, u(t, z_m)\, \mathrm{e}^{\frac{\mathrm{i}}{\varepsilon} S(t,z_m)}\, g^\varepsilon_{\Phi^t(z_m)} \tag{12}$$

where the points $z_1, \ldots, z_M$ which are sampled from μ_0. Thus, in addition to the flow Φ^t, we need to compute the classical action S and the HK prefactor u. In order to do so we need the solution to the variational equation corresponding to (4), i.e.

$$\dot{W}(t) = \mathcal{J}\, \nabla^2 h(\Phi^t)\, W(t), \quad W(0) = \mathbf{I} \tag{13}$$

where $W(t) = D_z \Phi^t$ is the derivative of the flow with respect to the initial values and $\nabla^2 h$ is the Hessian of the Hamiltonian function. The classical action S may be computed directly from the flow.

We then solve Equations (4) and (13) simultaneously by a single numerical integrator. In order to preserve the underlying symplectic structure we use a composition method based on the Størmer–Verlet scheme which is symplectic and symmetric, cf. [HLW06]. The order of the scheme is controlled by using a composition strategy with composition constants taken from [KL97]. If we assume a separable system of the form $h(q, p) = T(p) + V(q)$ the resulting method is an explicit one, which makes our calculations even more efficient.

4 Automatic Code Generation

It is quite simple to write down the Størmer–Verlet time stepping scheme for low dimensional systems. For high dimensional system and complicated potentials it is already a tedious task to calculate the gradient and the Hessian of the potential. Implementing the time stepping scheme is even worse since we additionally have to multiply the Hessian with the matrix W in order to calculate the right hand side of (13). Therefore we developed an automated code generator in the computer algebra system Mathematica. The main idea is to find a way to create code of very high complexity by only a couple of well-placed commands. In addition we implemented the generator in such a way that the creation of the code can be done by the same commands for arbitrary dimensions. As a first step we choose the dimension d and initialise all the variables we need. First, the $2d$ components of the vector $z = (q, p) \in \mathbb{R}^{2d}$ are initialized by

```
q = Array[Symbol[StringJoin["q",ToString[#]]]&,d];
p = Array[Symbol[StringJoin["p",ToString[#]]]&,d];
```

and then the $4d^2$ entries of the matrices $\partial_q X^t, \partial_p X^t, \partial_q \Xi, \partial_p \Xi^t \in \mathbb{R}^{d \times d}$ by

```
Xq =Array[Symbol[StringJoin["Xq",ToString[#1],"x",ToString[#2]]]&,{d,d}];
Xp =Array[Symbol[StringJoin["Xp",ToString[#1],"x",ToString[#2]]]&,{d,d}];
Xiq =Array[Symbol[StringJoin["Xiq",ToString[#1],"x",ToString[#2]]]&,{d,d}];
Xip =Array[Symbol[StringJoin["Xip",ToString[#1],"x",ToString[#2]]]&,{d,d}];
```

We will illustrate the further course of action using the example of a Henon–Heiles type potential which shall be given by

$$V(x) = \frac{1}{2}\sum_{k=1}^{d} x_k^2 + \sigma \sum_{k=1}^{d-1}\left(x_k x_{k+1}^2 - \frac{1}{3}x_k^3\right) + \beta \sum_{k=1}^{d-1}\left(x_k^2 + x_{k+1}^2\right)^2 \tag{14}$$

where $\sigma > 0$ and $\beta > 0$ are two positive constants controlling the strength of interaction and confinement respectively. We may just enter the formula of the potential and Mathematica analytically computes the gradient and the Hessian and simplifies the resulting expressions.

```
V = 1/2 Sum[q[[k]]^2, {k, 1, d}] +
    sigma Sum[q[[k]] q[[k + 1]]^2 - q[[k]]^3/3, {k, 1, d - 1}] +
    beta Sum[(q[[k]]^2 + q[[k + 1]]^2)^2, {k, 1, d - 1}] // FullSimplify;
GradV = D[V, {q}] // FullSimplify;
HessV = D[V, {q, 2}] // FullSimplify;
```

Furthermore we compute the matrix-matrix multiplication of the Hessian and W symbolically and again simplify the results.

```
HessVXq = HessV.Xq //FullSimplify;
HessVXp = HessV.Xp //FullSimplify;
```

With a series of elaborate string operations the source code of the time stepping scheme is also produced automatically. Let us just give an example of the string manipulation needed for one step in p, $\partial_q \Xi$, and $\partial_p \Xi$ of our time stepping scheme.

```
ToString[ Array[ StringJoin[
    ToString[p[[#1]]], "+=", ToString[CForm[dt N[GradV[[#1]]]
    //. {Power[x_,2] -> HoldForm[x*x], Power[x_,3] -> HoldForm[x*x*x]}]],
    ";"] &, {d, 1}]] <>

 ToString[ Array[ StringJoin[
     ToString[Xiq[[#1,#2]]], "+=", ToString[CForm[dt N[HessVXq[[#1,#2]]]]
       //. {Power[x_, 2] -> HoldForm[x*x]}], ";"] &, {d, d}]] <>

 ToString[ Array[ StringJoin[
     ToString[Xip[[#1,#2]]], "+=", ToString[CForm[dt N[HessVXp[[#1,#2]]]]
       //. {Power[x_, 2] -> HoldForm[x*x]}], ";"] &, {d, d}]];
```

The code for the variables q, $\partial_q X$, and $\partial_p X$ is constructed in an analogous manner. Note that only a couple of commands created the whole time stepping scheme. For $d = 12$, for example, we already produce more than 1000 lines of code that way. With a

sequence of further string operations our Mathematica program is capable of generating the complete header files for the C++ classes ready for compilation. This includes the code for the evolution of (11) and (12).

5 Vectorization

Modern CPUs and accelerators like the Intel Xeon Phi processor include vector extensions, SSE, AVX, and AVX512, in their instruction set to speed up floating point computations. Although these vector operations offer high theoretical floating point instruction throughput on vectors of up to 16 single precision and 8 double precision floating point numbers, it is tedious to write code for the vector extensions using intrinsics directly. To make the vector processing capabilities more easily accessible for our simulations without the need to maintain separate versions of the simulation code, we implemented an explicit vectorization library in C++ , which hides the underlying complexities. This vectorization library defines classes for the vector data types `double2`, `double4`, and `double8` and provides all elementary arithmetic operators $+, -, *, /$ by overloading the standard built-in operators. Likewise all common functions, e.g. `exp`, `sqrt`, `sin`, `cos`, are provided for the vector data types by the library. A specific design goal was to keep the syntax for the instantiation of the vector data types as simple as possible and thus close to the built-in `double` data type. Consequently we avoided a heavy C++ template approach in contrast to Vc [KL12], a C++ library for explicit vectorization. A few member functions that encapsulate the vector extension intrinsics in the C++ class for the `double2` data type are given below.

```
class double2
{
    __m128d v;
    double2& operator*=(const double2& rhs)
    {
        v = _mm_mul_pd(v, rhs.v);
        return *this;
    }
    double2& sqrt()
    {
        v = _mm_sqrt_pd(v);
        return *this;
    }
};
```

The arithmetic operators and functions are not only overloaded to act directly on the instances of the vector data types as member functions, for example `v.sqrt()` or `v *= 2.0`, but are also provided as globally defined operators to allow a more natural use of the operators similar to the built in types. For example the expression `v = sqrt(x) * y + 2.0 * z` is valid code for all vector data types. The code listing shows a typical implementation of the operators and functions in C++.

```
inline double2 operator*(double2 lhs, const double2& rhs)
{
    lhs *= rhs;
    return lhs;
}

inline double2 sqrt(double2 lhs)
{
    lhs.sqrt();
    return lhs;
}
```

The strategy to implement the same syntax for arithmetic operations for vector data types as for built-in data types makes it possible to reuse the automatically generated code. The only thing that has to be adapted is the definition of the data type. This can be easily implemented by using the C++ template mechanism. Section 7.2 contains a performance comparison of our time stepping kernel using the `double` data type as well as the vectorized versions with `double2` and `double4`, cf. Table 3.

6 GPU Implementation

While it is necessary to explicitly vectorize the simulation code to gain performance from the vector units on a traditional CPU, this is no longer required on graphics processing units (GPUs). Modern GPUs implement a single instruction multiple thread (SIMT) model as a parallelization paradigm. The vectorization is performed by the compiler using essentially the same code as it would for in the sequential case. To make this model work efficiently, it is necessary to adapt the data layout of the simulation code to allow coalesced memory access. Often this is the most important aspect of optimizing code for graphics processors. Rearranging the data structures for vector access is also required for efficient vectorized CPU code. A design for efficient memory access for vector data types is thus generally a good investment. The computational kernels for the GPU accelerated simulation code are compiled from the same automatically generated code base as the CPU version. The only modification done for the GPU kernel is typically the replacement of the outer for loop iterating over the sample points with the computation of a unique thread id. For kernels that calculate expectation values a GPU specific accumulation routine is also required to perform a reduction on the calculated values.

7 Numerical Experiments and Performance

In this section we will present numerical experiments. First we take a look at the GPU implementation of Egorov's method. After that we test the performance of our vectorization library by means of the Herman–Kluk method.

7.1 GPU Implementation of the Egorov method

Following [KL14] we consider a confined Henon–Heiles potential of the form

$$V(x) = \frac{1}{2}\sum_{k=1}^{d} x_k^2 + 1.8436 \sum_{k=1}^{d-1} \left(x_k x_{k+1}^2 - \frac{1}{3} x_k^3 \right) + 0.4 \sum_{k=1}^{d-1} \left(x_k^2 + x_{k+1}^2 \right)^2 .$$

The semiclassical parameter is chosen to be $\varepsilon = 0.0037$ and to initial wave function a Gaussian

$$\psi_0(x) = (\pi\varepsilon)^{-d/4} \exp\left(-\frac{1}{2\varepsilon} |x - q|^2 + \frac{\mathrm{i}}{\varepsilon} p \cdot (x - q) \right) \tag{15}$$

centered at position $q_k = 0.1215$ and momentum $p_k = 0$ for all $k \in \{1, \ldots, d\}$. Figure 1 shows the time-evolution of the potential energy for 16 up to 512 dimensions. We used 2^{18} Monte Carlo points and our composition Størmer–Verlet method of order 8 with a time step of 1 femptosecond to transport them up to a final time of 100 fs for each individual calculation.

Figure 1: Evolution of the expectation values of the potential energy for various dimensions.

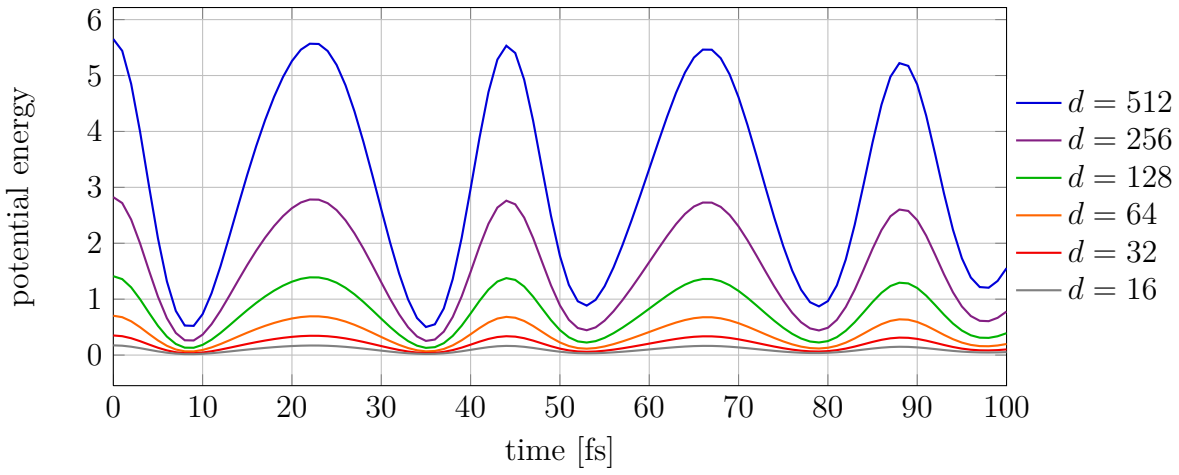


Table 1 lists the respective execution times. For the 32 dimensional problem the GPU code on a Nvidia Tesla K20 is over 140 times faster compared to a single CPU core and about 9 times faster compared to a dual socket eight core Intel Xeon E5-2650 @ 2.00GHz compute node. For higher dimensional systems the GPU performance advantage drops to a factor 3 compared to the dual socket compute node. The reason for the performance drop is the exhaustion of the 255 registers per thread and the increased register spilling to the stack that resides in main memory and is cached only be a small L1 cache.

Table 1: Execution times for the Egorov method for the Henon–Heiles potential.

dim	2 × Intel Xeon E5-2650	1 × Nvidia Tesla K20	speedup
16	4.260 s	0.641 s	6.646
32	10.386 s	1.180 s	8.802
64	24.793 s	4.412 s	5.619
128	109.393 s	29.266 s	3.738
256	219.183 s	71.980 s	3.045
512	454.614 s	155.915 s	2.916

Another interesting aspect is the quick growth in code complexity due to fact that it was automatically generated. This is reflected in the compilation times needed for the Henon–Heiles simulation. For CPU and GPU code we used the GNU compiler `g++` (version 4.8.2) and the Nvidia's nvcc compiler (version 7.0) respectively. The compilation times are collected in Table 2.

Table 2: Compilation times for Nvidia nvcc 7.0 and GNU g++ 4.8.2

dim	nvcc	g++
16	9.037 s	2.113 s
32	9.916 s	3.181 s
64	11.527 s	5.887 s
128	20.539 s	20.193 s
256	35.137 s	71.920 s
512	133.548 s	352.200 s

7.2 Vectorization of the Herman–Kluk method

In order to test the performance of our vectorization library we take a look at another model that has been studied in mathematics [LR10] [FGL09] as well as chemistry [NM02]. Consider again a Henon–Heiles potential

$$V(x) = \frac{1}{2}\sum_{k=1}^{6} x_k^2 + \sigma \sum_{k=1}^{5} \left(x_k x_{k+1}^2 - \frac{1}{3} x_k^3 \right) + \beta \sum_{k=1}^{5} \left(x_k^2 + x_{k+1}^2 \right)^2 , \tag{16}$$

this time in six dimensions with parameters $\sigma = \frac{1}{\sqrt{80}}$ and $\beta = \frac{\sigma^2}{16}$. The semiclassical parameter is chosen to be $\varepsilon = 0.01$ and the initial wave function shall again be a Gaussian of the form (15) with $q_k = 2$ and $p_k = 0$ for all $k \in \{1, \ldots, d\}$. We use our Herman–Kluk method with 2^{19} quasi - Monte Carlo points. For time-evolution we used a composition Størmer–Verlet method of order 8 for 200 time steps of 0.01.

We conducted this experiment for several different setups where we varied the number of MPI processes from 1 to 16 and used different vector data types. The benchmark results are shown in Table 3. Note that the algorithm scales perfectly with the number of MPI processes. The data type `double2` gives an almost optimal speedup of a factor two and `double4` is about three times faster than the non-vectorized code.

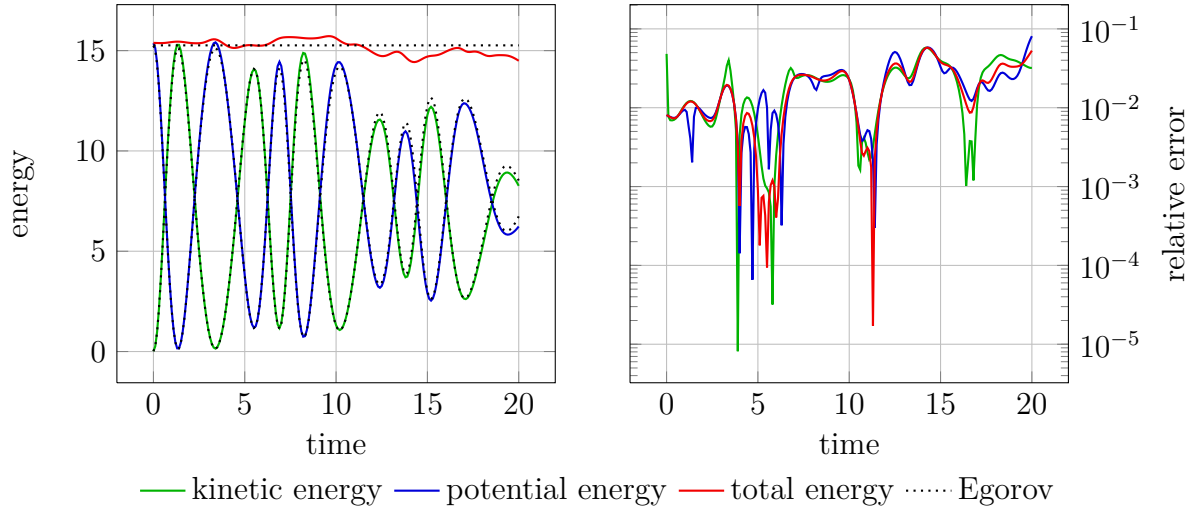
Table 3: Mephisto cluster: 1 compute node with 2 × Intel Xeon CPU E5-2650 @ 2.00GHz

no. of MPI threads	**data type `double` (64 bit)**	**speedup**
1	881.14 s	1.00
2	441.31 s	2.00
4	221.23 s	3.98
8	111.65 s	7.89
16	54.78 s	16.09
no. of MPI threads	**data type `double2` (128 bit)**	**speedup**
1	449.78 s	1.96
2	226.34 s	3.89
4	113.61 s	7.76
8	57.38 s	15.36
16	28.05 s	31.41
no. of MPI threads	**data type `double4` (256 bit)**	**speedup**
1	289.34 s	3.05
2	145.16 s	6.07
4	72.58 s	12.14
8	36.25 s	24.31
16	18.00 s	48.95

For reasons of consistency we compare the results provided by the Herman–Kluk method with those of the Egorov method. Figure 2 shows the evolution of the expectation of kinetic, potential, and total energy and the relativ error between the two

methods.

Figure 2: Evolution of energy expectation values computed by the Herman–Kluk propagator (left) and the error to the solution computed by the Egorov method (right).



8 Conclusions

We showed that the Herman–Kluk method and the Egorov method are very well suited for modern multicore and manycore computer architectures and provide a unique approach to solve very high dimensional problems in semiclassical quantum dynamics.

Acknowledgements

The authors gratefully acknowledge the support from the International Research Training Group IGDK1754, funded by the DFG and FWF. This work is also supported by the TUM Graduate School at Technische Universität München.

References

[BR02] A. Bouzouina and D. Robert. Uniform semiclassical estimates for the propagation of quantum observables. *Duke Math. J.*, 111(2):223–252, 2002.

[Ego69] Yu. V. Egorov. The canonical transformations of pseudodifferential operators. *Uspekhi Mat. Nauk*, 24(5 (149)):235–236, 1969.

[FGL09] E. Faou, V. Gradinaru, and Ch. Lubich. Computing semiclassical quantum dynamics with Hagedorn wavepackets. *SIAM J. Sci. Comput.*, 31(4):3027–3041, 2009.

[HK84] M.F. Herman and E. Kluk. A semiclasical justification for the use of non-spreading wavepackets in dynamics calculations. *Chem. Phys.*, 91(1):27–34, 1984.

[HLW06] E. Hairer, Ch. Lubich, and G. Wanner. *Geometric numerical integration. Structure-preserving algorithms for ordinary differential equations*. Number 31 in Springer Series in Computational Mathematics. Springer, Berlin, second edition edition, 2006.

[KL97] W. Kahan and R.-C. Li. Composition constants for raising the orders of unconventional schemes for ordinary differential equations. *Math. Comp.*, 66(219):1089–1099, 1997.

[KL12] M. Kretz and V. Lindenstruth. Vc: A c++ library for explicit vectorization. *Softw. Pract. Exper.*, 42(11):1409–1430, November 2012.

[KL14] J. Keller and C. Lasser. Quasi-classical description of molecular dynamics based on egorov's theorem. *The Journal of Chemical Physics*, 141(5), 2014.

[LR10] C. Lasser and S. Röblitz. Computing expectation values for molecular quantum dynamics. *SIAM J. Sci. Comput.*, 32(3):1465–1483, 2010.

[LS15] C. Lasser and D. Sattlegger. Discretizing the herman–kluk propagator. *to appear*, 2015.

[Mar02] A. Martinez. *An Introduction to Semiclassical and Microlocal Analysis*. Universitext. Springer, New York, 2002.

[NM02] M. Nest and H.-D. Meyer. Benchmark calculations on high-dimensional henon–heiles potentials with the multi-configuration time dependent hartree (mctdh) method. *The Journal of Chemical Physics*, 117(23):10499–10505, 2002.

[RS07] V. Rousse and T. Swart. Global l^2-boundedness theorems for semiclassical fourier integral operators with complex phase. *ArXiv e-prints*, 2007.

[SR09] T. Swart and V. Rousse. A mathematical justification for the herman-kluk propagator. *Comm. Math. Phys.*, 286:725–750, 2009.

[TW04] M. Thoss and H. Wang. Semiclassical description of molecular dynamics based on initial-value representation methods. *Annu. Rev. Phys. Chem.*, 55:299–332, 2004.

[Wey27] H. Weyl. Quantenmechanik und gruppentheorie. *Zeitschrift für Physik*, 46(1-2):1–46, 1927.