# Accelerating the B-spline evaluation in Quantum Monte Carlo

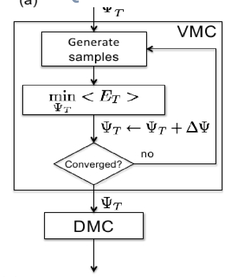
**Ye Luo, Anouar Benali, Vitali Morozov | Leadership Computing Facility at Argonne National Laboratory**


## Abstract

Thanks to fast evolving high performance computers, first principle simulations of realistic systems with large numbers of atoms nowadays become affordable for better understanding new materials. Quantum Monte Carlo (QMC) is one of the most accurate and scalable methods for these simulations. When solving the many-body Schrödinger equation in QMC, the wavefunction is usually represented in a planewave basis, which comes at high computational cost O($N^3$) with the number of electrons N. In our software package QMCPACK, the cost of evaluating the wavefunction at a given electronic configuration has been reduced to a quadratic scaling O($N^2$) by representing it with B-splines, real space localized cubic splines centered on a regular grid. Nevertheless, the evaluation of B-splines still takes over 20% of the total application time. We recently improved the algorithm by fully taking advantage of the vectorization and optimizing the memory access on BG/Q. About 3-fold speedup was achieved in the subroutines calculating multiple B-splines. Threading capability is also added to the new algorithm to maximize the single node performance. According to the specifications of the upcoming HPC systems (long vector units, more integrated cores, higher memory bandwidth), all the methods used to design the new algorithm make it ready to efficiently exploit the new features of these systems.

## Quantum Monte Carlo basics

(a)




### QMC Workflow

- Starting with an initial trial wavefunction
- Optimizing the trial wavefunction using VMC
- Performing DMC with the improved wavefunction

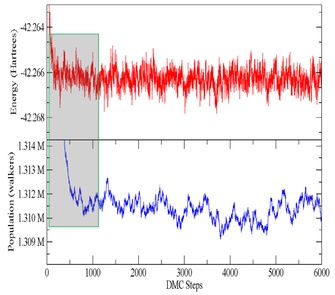
### QMC Algorithm: Diffusion Monte Carlo Perspective

**for** generation $= 1 \cdots M$ **do**
  **for** walker $= 1 \cdots N_w$ **do**
    let $\mathbf{R} = \{\mathbf{r}_1 \ldots \mathbf{r}_N\}$
    **for** particle $i = 1 \cdots N$ **do**
      set $\mathbf{r}_i' = \mathbf{r}_i + \delta$
      let $\mathbf{R}' = \{\mathbf{r}_1 \ldots \mathbf{r}_i' \ldots \mathbf{r}_N\}$
      **ratio** $\rho = \Psi_T(\mathbf{R}')/\Psi_T(\mathbf{R})$
      **if** $\mathbf{r} \rightarrow \mathbf{r}'$ is accepted **then**
        update state of a walker
      **end if**
    **end for**{particle}
    Compute $E_L = \hat{H}\Psi_T(\mathbf{R})/\Psi_T(\mathbf{R})$
    Reweight and branch walkers
    Update $E_T$
  **end for**{walker}
**end for**{generation}

Walker, basic units of parallelism

Single electron move

Communication




The data during the equilibration (grey region) are discarded.

## Algorithms

In both VMC and DMC stages, all the values of single particle orbitals (SPOs) are updated for each single electron move step. For solids, all the SPOs are represented by molecular orbitals computed with cubic B-spline interpolation. The feature and detail of the algorithms used in the benchmark are provided here.

- Phase 0: the original algorithm has 4 nested loops (x,y,z,spline), used as a baseline.
- Phase 1: the innermost spline and z loops are interchanged. Loop z is completely unrolled. a(x)·b(y) is computed only once. Cross-platform
- Phase 2: The spline loop is unrolled 4 times and vectorization is applied. Fused multiply-add (IBM QPX 4 SIMD lanes) instructions are used.
- Phase 3: Add the data pre-fetch built-in functions to speed up accessing the spline coefficients table.
- Ref Algo.1: Skip Phase 1 and apply all techs of Phase 2,3 on the spline loop. Only for Eval_X_V functions.
- Ref Algo.2: Rearrange the loop orders to (spline,x,y,z ) and unloop y,z loops. Destroy data locality. Only for Eval_X_VGH functions.

**Phase 0**

```
for x =0, 1, 2, 3 do
  for y = 0, 1, 2, 3 do
    for z = 0 , 1, 2, 3 do
      for spline = 1, ..., N_spline do
        abc = a(x)·b(y)·c(z)
        val(spline) += coef(x0+x, y0+y, z0+z, spline)·abc
      end for {spline}
    end for {z}
  end for {y}
end for {x}
```

**Phase 1**

```
ab = a(x)·b(y)
for spline = 1, ..., N_spline do
  val(spline) += ab ·
        [c(0) · coef(x0+x, y0+y, z0, spline) +
         c(1) · coef(x0+x, y0+y, z0+1, spline) +
         c(2) · coef(x0+x, y0+y, z0+2, spline) +
         c(3) · coef(x0+x, y0+y, z0+3, spline)]
end for {spline}
```

**Phase 2**
**Phase 3**

```
ab = a(x)·b(y)
for spline = 1, 5, 9, ..., N_spline do
  Prefetch coef_vec
  val_vec(spline) += ab ·
        [c(0) · coef_vec(x0+x, y0+y, z0, spline) +
         c(1) · coef_vec(x0+x, y0+y, z0+1, spline) +
         c(2) · coef_vec(x0+x, y0+y, z0+2, spline) +
         c(3) · coef_vec(x0+x, y0+y, z0+3, spline)]
end for {spline}
remainder
```

## Comments

Vectorization is very helpful to accelerate computational intensive subroutines but it requires a good algorithm as a starting point. In our path to the best algorithm, we adopt a better algorithm in Phase 1, take advantage of the vector unit in Phase 2 and maximize the memory efficiency in Phase 3. Both Phase 1 and Phase 2 are transferable to future architectures with long vector units.

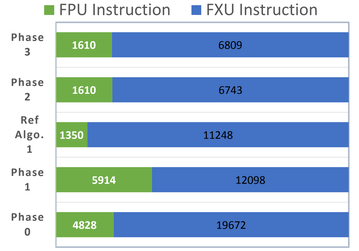
## Benchmarks

Several benchmark tests on BG/Q from different perspective reveal how the code speeds up with a better algorithm and vectorization.

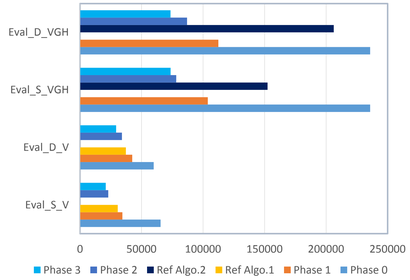
Test case: Eval_S_V subroutine which evaluates only the values of the SPOs.

- The phase 1 significantly reduces the number of FXU instructions and balances FPU and FXU instruction ratio (ideal ->1)
- The vectorization in phase 2 reduces both FPU and FXU instructions. FPU instruction is cut by 5914/1610=3.67 (ideal = 4 on BG/Q vector unit). Vector load/store also save FXU instructions.
- The data pre-fetch added in Phase 3 further helps this memory-bound routine (see the execution cycles).
- Ref Algo. 1 is also vectorized but significantly slower than Phase 3 due to the missing optimization applied in Phase 1.

**Eval_S_V instruction mix**




**Kernel execution cycles (smaller is better)**




Test cases: Eval_D_V is the double precision version of Eval_S_V.
The Eval_S/D_VGH compute not only the value but also the gradients and hessian of the orbitals and they are more compute-bound. A similar algorithm based on the same ideas is applied to achieve significant speed-up.

In all the test cases, Phase 3 is always the best algorithm, 3X faster than Phase 0.

## Threading

In QMCPACK, walkers are distributed among threads within one node. For extremely large simulations, the number of walkers per node is limited by the memory capacity. If this number is smaller than the number of cores on a single node, nested threading on the spline evaluation routines can be enabled to fully utilize the compute power on the node.
The threading is added on the Phase 3 algorithm. The innermost loop over all splines is divided by the number of threads. Parallel region is created at the outmost loop to minimize OpenMP overhead. The strong scaling result is shown on the right based on a test case with 4000 splines executed on BG/Q.