PHY515: High Performance Computing 1

Final Project

Parallel computing in any order imaginary time propagation method for solving the Schrödinger equation

Hsuan-Hao Fan

Department of Physics, University at Buffalo SUNY Buffalo NY 14260

(Dated: December 26, 2015)

CONTENTS

I. INTRODUCTION

Density-functional method is broadly used to solve quantum chemistry problems and solid state physics [1]. It is a method to solve the Schrödinger equation on a large three dimensional mesh with grid points equal or greater than 10^6. Due to such a large mesh, conventional matrix methods (matrix-vector multiplication) would be very slow [2]. Instead, we adopt matrix-free method. That is, we consider the Hamiltonian as an operator acting directly to eigenstates, not a matrix-vector product. We use the so-called imaginary time diffusion algorithm [2] introduced in the following. The calculations in this work are per-

formed by using the program package limerec [3], which solves the Schrödinger equations on a real-space grid using a diffusion algorithm.

The lowest n states of the one-body time-independent Schrödinger equation are given by

$$\hat{H}\psi_j(\mathbf{r}) = E_j\psi_j(\mathbf{r}) \tag{1}$$

where the Hamiltonian $\hat{H}$ is written as

$$\hat{H} = -\frac{\hbar^2}{2m}\nabla^2 + V(\mathbf{r}) \equiv T + V. \tag{2}$$

Here T is the kinetic energy and V is the effective potential energy. We have known that the solutions to time-dependent Schrödinger equation can be evolved by the time evolution operator $U(t_0, t) = \exp(-i\hat{H}(t - t_0)/\hbar)$ from the initial state $\psi_j(\mathbf{r}, t_0)$ at time $t_0 = 0$, which are solutions to Eq. (1). For convenience, we set $\hbar = 1$ throughout this report. By using imaginary time $it = \epsilon$, where ϵ is real and positive, the evolution operator becomes

$$\mathscr{T}(\epsilon) \equiv e^{-\epsilon H}. \tag{3}$$

The lowest n solutions of an eigenvalue problem (1) can be obtained by applying the evolution operator $\mathscr{T}(\epsilon)$ repeatedly on the set of states $\psi_j(\mathbf{r})$ and orthogonalizing the states after every time step. Note that one has to orthogonalize the states after every step or one will always get only ground state. We illustrate this in Appendix A in more details.

II. DIFFUSION ALGORITHM

As we introduced in last section, we can obtain the lowest n eigenstates of the one-body Schrödinger equation (1) by repeatedly applying the imaginary time evolution operator

$$\mathscr{T}(\epsilon) = e^{-\epsilon(T+V)} \tag{4}$$

to a set of trial functions $\{\psi_j(\mathbf{r})\}$, which are orthogonalized after each time step ϵ. That is, repeatedly applying

$$\psi_i^{(k+1)} \equiv \mathscr{T}(\epsilon)\psi_i^{(k)} \tag{5}$$

and orthonormalizing makes the state functions evolve toward, and in turn converge to the lowest n eigenfunctions of the Hamiltonian H. This can be seen in the source code, *imstep.f90*.

A. Operator factorization

Since the evolution operator cannot be calculated exactly for a Hamiltonian, one must use approximate forms. One can use the so-called Trotter's product rule (The derivation is shown in Appendix B) to factorize the imaginary time evolution operator like

$$\mathscr{T}(\epsilon) = e^{-\epsilon(T+V)} = \prod_{i=1}^{M} e^{-a_i \epsilon V} e^{-b_i \epsilon T} \tag{6}$$

with coefficients $\{a_i, b_i\}$ determined by the required order of accuracy [4, 5]. As shown in Appendix B, this approximation holds only if the time step ϵ is small enough. Also, a_i and b_i must be positive, otherwise some of the $e^{-a_i \epsilon T}$ and/or $e^{-b_i \epsilon V}$ will be unbound and the method diverges.

The most common used approximation is the second-order factorization

$$\mathscr{T}^{(2)}(\epsilon) \equiv e^{-\frac{1}{2}\epsilon V} e^{-\epsilon T} e^{-\frac{1}{2}\epsilon V} = \mathscr{T}(\epsilon) + \mathcal{O}(\epsilon^3). \tag{7}$$

One can easily derive this from Eq. (B15):

$$e^{-\frac{1}{2}\epsilon T} e^{-\frac{1}{2}\epsilon V} = e^{-\frac{1}{2}\epsilon(T+V)} + \frac{\epsilon^2}{8}(-1)^2[T,V] + \mathcal{O}(\epsilon^3) \tag{8}$$

$$e^{-\frac{1}{2}\epsilon V} e^{-\frac{1}{2}\epsilon T} = e^{-\frac{1}{2}\epsilon(T+V)} + \frac{\epsilon^2}{8}(-1)^2[V,T] + \mathcal{O}(\epsilon^3) \tag{9}$$

Since $[V,T] = -[T,V]$, after the multiplication of above two equations, we obtain

$$e^{-\frac{\epsilon}{2}V} e^{-\frac{\epsilon}{2}T} e^{-\frac{\epsilon}{2}V} = e^{-\epsilon(T+V)} + \frac{\epsilon^2}{8}(-1)^2[T,V] + \frac{\epsilon^2}{8}(-1)^2[V,T] + \mathcal{O}(\epsilon^3). \tag{10}$$

The second term is cancelled out by third term. Then we obtain Eq. (7). This second-order factorization is useful, and it has been employed in applications of time-dependent density functional theory (DFT), or quantum Monte Carlo (QMC) calculations. Note that all these approximations holds only if the time step ϵ is very small. That means we need many iterations to get convergent results. However, the imaginary time evolution converges faster when the time step is large. Since we desire to converge to the eigenstates of H reasonably quickly, it is desirable to use approximations of the evolution operator that are both stable and accurate for large time steps. The most obvious way would be to use the order of factorizations higher than 2. In Refs. [2, 6], an expansion is indeed possible and takes the form

$$e^{-\epsilon(T+V)} = \sum_{k=1}^{n} c_k \mathscr{T}_2^k\left(\frac{\epsilon}{k}\right) + \mathcal{O}(\epsilon^{2n+1}) \equiv \mathscr{T}_n(\epsilon) + \mathcal{O}(\epsilon^{2n+1}), \tag{11}$$

where the coefficients c_k are given in closed form for any n:

$$c_i = \prod_{j=1(\neq i)}^{n} \frac{k_i^2}{k_i^2 - k_j^2} \tag{12}$$

with $\{k_1, k_2, \ldots, k_n\} = \{1, 2, \ldots, n\}$. One can use the relation (11) to obtain any order factorization. Ref. [2] lists the factorization up to 12th order.

B. Orthogonalization

For convenience, we suppress in this section spin-subscripts. It is understood that the orthogonization is carried out either independently for spin-up and spin-down electrons, or only in the spin-symmetric or spin-polarized case.

EK : One needs the independent spins only if one has different electron numbers for spin-up and spin-down.

As we have discussed above, an orthonormal set of trial vectors $\{\psi_j^{(k)}(\mathbf{r}), j = 1, \ldots, n\}$, where k is the iteration number, is needed after each propagation step. Since the evolution operator in imaginary time is not unitary (i.e., it does not preserve the normalization nor the orthogonality), it is then necessary to orthonormalize the resulting functions after each propagation step. That is,

$$\mathscr{T}^\dagger(\epsilon) = e^{-\epsilon H^\dagger} = e^{-\epsilon H} \tag{13}$$

since ϵ is positive and real; H is Hermitian operator. Thus we obtain

$$\mathscr{T}^\dagger(\epsilon)\mathscr{T}(\epsilon) = e^{-2\epsilon H} \neq \mathbb{I} \Rightarrow \langle\psi_j|\mathscr{T}^\dagger(\epsilon)\mathscr{T}(\epsilon)|\psi_i\rangle \neq \langle\psi_j|\psi_i\rangle = \delta_{ij}. \tag{14}$$

Due to this reason, we define a new set of vectors, not orthonormal, as

$$\Psi_j^{k+1}(\mathbf{r}) \equiv \mathscr{T}(\epsilon)\psi_j^{(k)}(\mathbf{r}) \tag{15}$$

for any approximation of the operator $\mathscr{T}(\epsilon)$.

III. RESULTS

A. Profiling

In order to improve the serial codes, we use profiling tool, **gprof**, to see the performance of the program. Take C_{60} as an example in this report. Each carbon (C) atom has 4 valence

electrons so there are 240 valence electrons with 120 spin-up and 120 spin-down. That means we need to consider 120 states ($n = 120$) in (1). The results of gprof is shown in the directory *results/gprof/gprof_C60.txt*. By using **gprof2dot** software [7], we can visualize the call graph as shown in Figure 1.

In order to understand the result of gprof, we introduce the whole program briefly at first. The main program is *limerec.f90*. In this whole program, there are two main tasks. First one is to solve the effective Schrödinger equation (1) in three dimensional by using diffusion algorithm introduced in Sec. II. In this work, we use mesh grid size, $(2 \times 48)^3$. This part of program is shown in *imstep.f90*. Once the first step has been achieved, the resulting orbitals are used to update the density, where a new potential is calculated, and the process is repeated until self-consistency is achieved. The second part is shown in *update.f90* which calls the module *response.f90*. As shown in Figure 1, *imstep.f90* costs 95.88% of total elapsed time. Thus in this report, we focus on improving *imstep.f90* at the beginning. However, in some cases the situation is opposite. That is, the density-update part (*update.f90*) takes longer time in whole program so that we show the performance in the second part.

Inside *imstep.f90*, there are two main subroutines used: prop20 and ortho. The former is relevant to applying the evolution operator $\mathscr{T}(\epsilon)$ on each state $\psi_j(\mathbf{r})$. This is the part we can apply parallel computing since the evolution operator $\mathscr{T}(\epsilon) \equiv e^{-\epsilon(T+V)}$ repeatedly applies on each kth time step approximation $\{\psi_j^k(\mathbf{r})\}$ independently to give the set of states $\{\phi_j(\mathbf{r})\}$ $(1 \leq j \leq n)$,

$$\phi_j^{(k+1)} \equiv \mathscr{T}(\epsilon)\psi_j^k. \tag{16}$$

We call this step as **the propagation step**. After this step, we orthogonalize the states every step. This is relevant to the subroutine ortho in *ortho.f90*. Since we need all states in this step, this part of program must run in serial. We name this step as **orthogonalization step**.

B. Parallel Computing

1. Propagation Step

As we mentioned before, the propagation step can be parallelized efficiently by simply distributing the states ψ_j across different processors. My advisor and his collaborators have

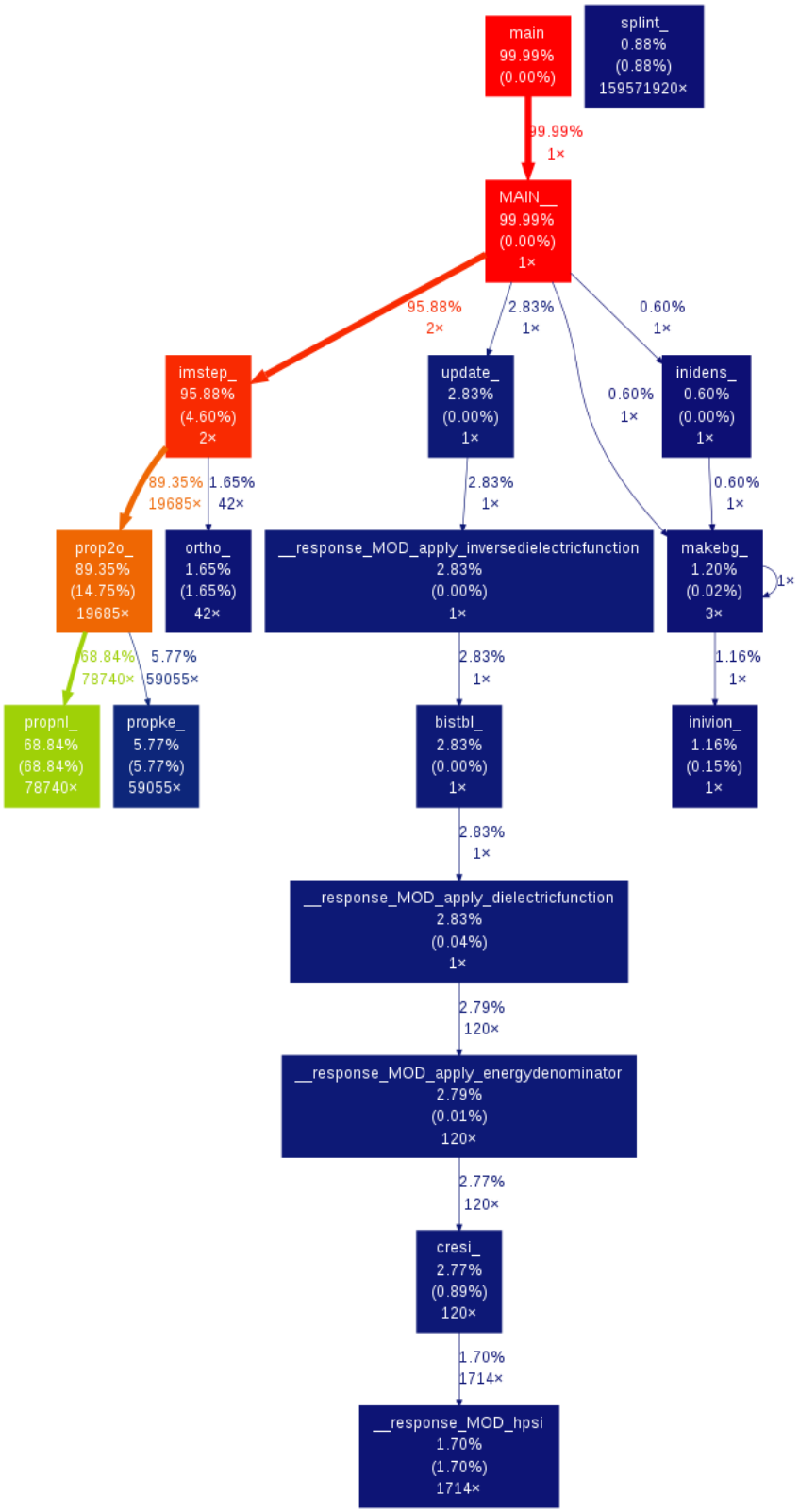

FIG. 1. Call graph from gprof in C_{60} case.

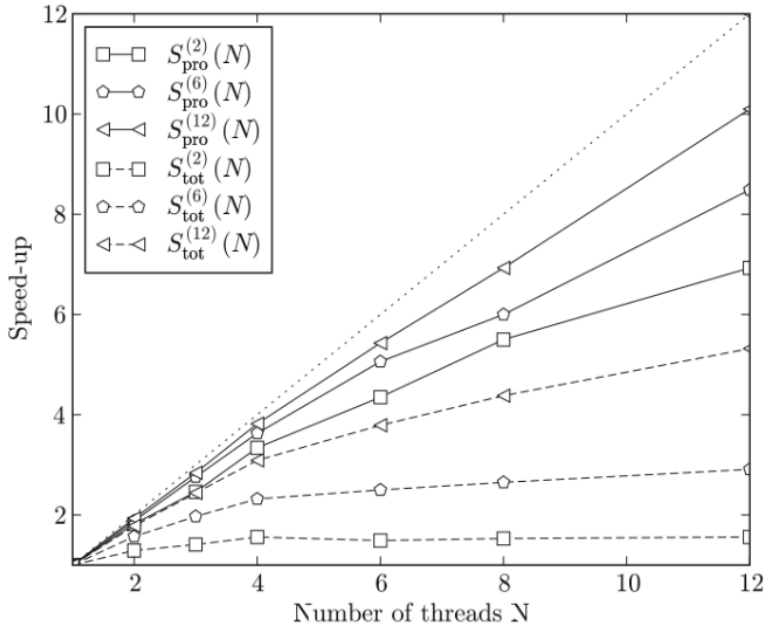

FIG. 2. The total speed-up time factor S_{tot} (solid lines) and the propagation only (without orthogonalization) time speed-up factor S_{pro}, as a function of parallel threads, for the 2nd-, 6th-, and 12th-order algorithm (filled squares, circles, and trangles, respectively). Also shown is the 'ideal' speed-up factor (dotted line). These results are generated by using OpenMP. [From Ref. [2]]

done this by using OpenMP [2]. However, OpenMP can only be run in shared memory computers. In contrast, MPI can run on either shared or distributed memory architectures. Also, MPI can be used on a wider range of problems than OpenMP, and each process has its own local variables. Thus it is meaningful to rewrite the code in MPI and compare the results with what OpenMP obtained.

We define the time consuming in propagation step as T_{pro} and the time consuming in orthogonalization step as T_{ort}. Note that orthogonalization step is run in serial mode. Moreover, the time T_{tot} for one iteration step on a machine with N processors is

$$T_{tot}(N) = T_{pro}(N) + T_{ort} \tag{17}$$

and the speed-up for the propagation only and the total time step including orthogonalization for the jth-order algorithm is

$$S^{(j)}_{pro/tot}(N) = \frac{T_{pro/tot}(1)}{T_{pro/tot}(N)}. \tag{18}$$

As shown in Figure 2, the effectiveness of the any order imaginary time propagation method for solving the Schrödinger equation is demonstrated by computing 120 eigenstates of a model potential for C_{60} molecule to very high precisions. In that work, the quantum well model of C_{60} is used, written in the form of

$$V(\mathbf{r}) = -\sum_i \frac{V_0}{\cosh^2(|\mathbf{r} - \mathbf{R}_i|/d)}, \tag{19}$$

where $\mathbf{R}_i$ are the locations of the carbon atoms in the C_{60} cage. The strength V_0 was chosen 1 in units of $\hbar^2/2m$ and the width of the troughs $d = 0.05$ a.u. From Figure 2, the higher order algorithm, the better performance in parallel computing. In this report, we use any order algorithm, not limited to 12th order. Also, we use realistic potential (Troullier-Martins pseudopotetials), not a model potential.

In order to compare MPI with OpenMP, we use the shared-memory machine with 40 cores in our group. The information about the information of CPU is shown in Apeendix D. The total memory of this machine is

```
MemTotal:       230999156 kB.
```

We show the performance by using different number of processors (N) with OpenMP and MPI in Table I. As $N = 1$, $T_{pro,omp}$ and $T_{ort,omp}$ are results by turning off OpenMP and running in serial mode. In contrast, $T_{pro,mpi}$ and $T_{ort,omp}$ are results by setting number of processor equal to 1 in MPI. They are consistent. Note that one cannot use OpenMP by setting number of processor equal to 1, or one will get much slower performance. We do not show that result here. One can access those original data files in *results/mpi* and *results/omp* for MPI and OpenMP separately. We calculate T_{pro} from the results in the end of data file. That is, one can find the result in the end of *.txt file as follows:

```
Timing: T_tot = 1588.41 T_eva = 1247.36 T_ort = 34.19 T_upd =   340.25
```

where T_eva = $T_{pro} + T_{ort}$ and T_ort = T_{ort}. Hence, we can calculate T_{pro} from T_eva by given T_ort.

Since program run in serial in the orthogonalization step, $T_{ort,omp}$ and $T_{ort,mpi}$ should be the same and independent of N. In Table I , the results from OpenMP are as we expected; however, the results from MPI are not in that trend. In MPI, as we increase N, $T_{ort,mpi}$ also increases a lot. This is due to how many memories one processor can access. In

N	$T_{pro,omp}$ (s)	$T_{ort,omp}$ (s)	$T_{pro,mpi}$ (s)	$T_{ort,mpi}$ (s)
1	1208.67	34.05	1213.17	34.19
2	658.75	34.50	646.70	41.15
4	414.75	34.25	429.23	56.00
8	274.50	32.75	302.50	85.12
14	218.00	34.00	250.23	145.80
16	194.00	34.75	255.54	163.20
18	185.00	34.25	252.34	184.04
32	176.50	31.75	321.45	323.15
40	184.25	33.75	313.14	400.29

TABLE I. N is the number of processors we use, $T_{pro,omp}$ ($T_{pro,mpi}$) is the elapsed time in propagation step by using OpenMP (MPI), and $T_{ort,omp}$ ($T_{ort,mpi}$) is the elapsed time in orthogonalization step.

OpenMP, it is a shared-memory model so all processor can access same amount of memory. In MPI, however, it is a distributed-memory model so one processor can access the amount of memory relevant to N. To be more specific, we assume total memory is M so one processor can access M/N memory as we use N processors. This explains that the orthogonalization part depends on memory mainly. One can refer to *ortho.f90*. In this procedure, it involves matrix-matrix product and the size of matrix is very large. That means we need more memory to run this part. That is why when we increase N in MPI, $T_{ort,mpi}$ increases.

As for the propagation step, we ask each processor to send $EVEC$, the eigenvector ψ_j,

N	$T_{pro,mpi}$ (s)	$T_{ort,mpi}$ (s)
40 processors	1547.82	5151.56
40 nodes	1670.37	1175.25

TABLE II. The first (second) row is from *mpi_40.out* (*mpi_40_12.out*) in the directory *results/mpi/CCR*. The first row is the result by using 40 processors at CCR; the second row is the result by using 40 nodes, where only one processor work and other 11 processors do nothing.

for the corresponding states assigned to the processor to the root processor. After that, we ask the root processor to broadcast EVEC to all processors in order to let each processor to execute orthogonalization step. In this way, each processor can be synchronized in principle. From Table I, as $N = 2$ MPI runs faster than OpenMP. However, as $N \geq 4$, MPI runs more slowly as we increase N. That is due to the communications between processors. In order to study the effect of the communications, we measure the time consuming for two parts: sending EVEC to root processor and broadcasting EVEC to all processors from root processor. One can refer to this result from *mpi_40_study_comm.txt* in the director *results/mpi*. We list the case $N = 40$ as follows:

```
Total sending EVEC to root processor is    4.9712603092193604      (s)
Total broadcast EVEC costs    72.895803689956665       (s)
```

From these two results, even though these two values deducted from $T_{pro,mpi} = 313.14$, the resulting value is 235.28 seconds which is still larger than $T_{pro,omp}$. The only possible reason left is the amount of memory each processor can access. In order to examine this effect, we take the advantage of nodes at CCR since SLURM can achieve this goal. First one is to use 40 processors by using MPI; the other one is to use 40 nodes and let only one processor to do calculation. The former each processor can only access 3000 MB memory; the later one can access whole memories in a node $3000 \times 12 = 36000$ MB. The results are listed in Table II. The first row is the result of former case; the second row is the result of the later case. In T_{ort}, one can see significantly difference. This is consistent with what we expected before. In orthogonalization step, we needs more memory to run that part. However, in T_{pro} we cannot see any improvement when we let one processor access more amount of memories. The reason is that the communication between nodes takes more time than the communication between processors in a node. But comparing the result running on 40 processors at CCR with the result running on 40 processors at the shared-memory machine in our group, one can find that the memory plays an important role in these calculation. Next, we plot the speed-up (S_{pro}) versus number of processors (N) as shown in Figure 3. Note that in our work, we fix the size of problem. In other words, we use strong scaling. From our above discussions, one can see the performance of MPI in propagation step is not better than the performance of OpenMP. One can also compare the result by using OpenMP in Figures 2 and 3.

After discussing with lecturer in HPC, we realize that in serial part, one has to only add

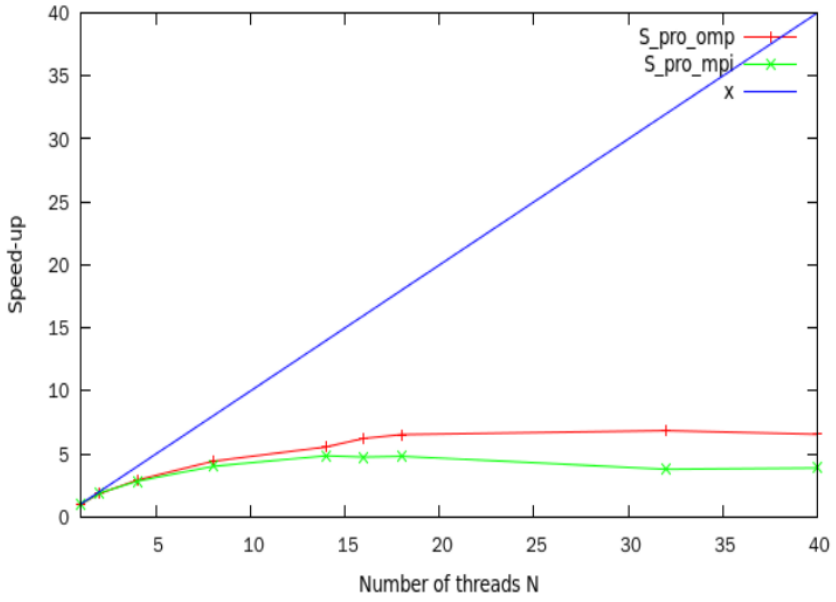

FIG. 3. For C_{60} case, speed-up in propagation step for OpenMP (red line) and MPI (green line). The blue line is the ideal speed-up.

N	$T_{pro,mpi,old}$ (s)	$T_{ort,mpi,old}$ (s)	$T_{pro,mpi,new}$ (s)	$T_{ort,mpi,new}$ (s)
1	1213.17	34.19	1238.75	33.75
2	646.70	41.15	630.87	37.75
4	429.23	56.00	408.50	40.50
8	302.50	85.12	277.12	45.50
14	250.23	145.80	220.62	61.50
16	255.54	163.20	213.50	68.50
18	252.34	184.04	227.37	85.88
32	321.45	323.15	204.00	99.50
40	313.14	400.29	202.12	265.50

TABLE III. N is the number of processors we use, $T_{pro,mpi,old}$ ($T_{pro,mpi,new}$) is the elapsed time in propagation step by using old-version MPI (new-version MPI), and $T_{ort,mpi,old}$ ($T_{ort,mpi,new}$) is the elapsed time in orthogonalization step. The results for old version mpi are same as those in Table I.

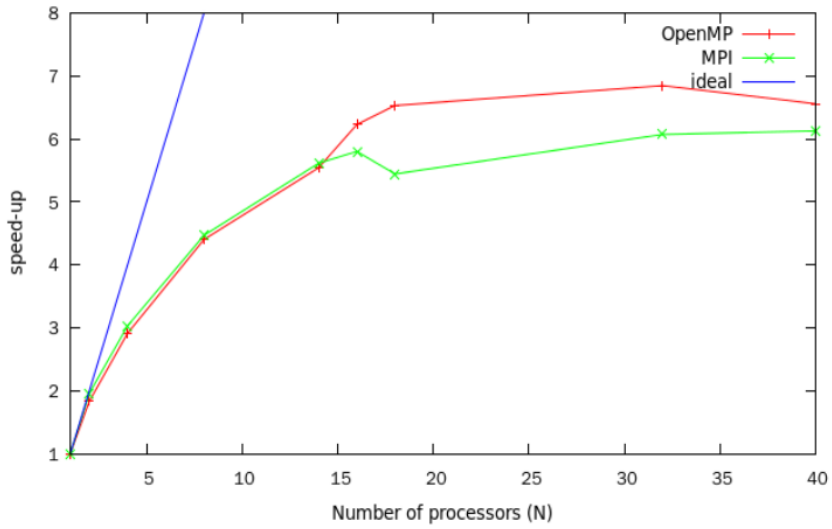

FIG. 4. For C_{60} case, speed-up in propagation step for OpenMP (red line) and MPI (green line). The blue line is the ideal speed-up.

an additional statement to ask only one processor to work on that part so the processor can access whole memory, not including the amount assigned to other processor. Note that in MPI each processor has its own local variables so once we assigned memories to local variables for each processor and those amount of memory cannot be accessed by other processor. This is how distributed memory program works. Therefore, we try to deallocate or not assign those large arrays to other processors if those arrays only are needed by root processor.

We list the results in Table III. When we use number of processors (N) less than 4, there are no difference between new version and old version. However, when we increase $N \geq 4$, one can see the improvement. Although we improve a lot in orthogonalization part, MPI still does not perform as well as OpenMP in this part. As we mentioned before, once if we assign any global arrays to other processors without deallocation, they cost amount of memory. As one increases N, those global arrays or variables will cost amount of memories proportional to N. In shared-memory machine, MPI is hard to avoid this problem. Normally, in clusters how many memories we can access depends on N. Take the clusters at CCR as an example. One node has 12 cores. Total memory in a node is 36000 MB. If we need to use more than 12 cores, then we need more nodes to achieve this goal. That means we can also access

N	$T_{pro,omp}$ (s)	$T_{ort,omp}$ (s)	$T_{pro,mpi}$ (s)	$T_{ort,mpi}$ (s)
1	2667.87	115.25	2689.37	115.88
2	1479.50	113.25	1403.74	125.38
4	926.50	114.50	921.75	133.25
8	589.50	115.00	624.88	150.62
14	462.25	117.00	603.37	189.75
16	451.50	112.75	590.88	217.62
18	455.25	110.75	580.00	260.50
32	440.50	109.25	523.76	357.12
40	438.75	112.50	531.25	690.75

TABLE IV. For Mg_{30}, N is the number of processors we use, $T_{pro,omp}$ ($T_{pro,mpi}$) is the elapsed time in propagation step by using OpenMP (MPI), and $T_{ort,omp}$ ($T_{ort,mpi}$) is the elapsed time in orthogonalization step.

more memories. Another solution is to use parallel version matrix-matrix product offered by ScaLAPACK, which supports MPI[8].

In order to compare the improved results with those obtained by OpenMP, we plot the speed-up versus number of processors (N) as shown in Figure 4. As $N < 15$, MPI performs as well as OpenMP in propagation step. In contrast, as $N > 15$, MPI does not perform as well as OpenMP. One reason is due to communication. The other reason is due to memory. Take $N = 40$ as an example. The total communication in propagation step is

```
Total communication time in propagation step is    14.563982486724854      sec
```

This is not that much. As for memory factor, it reflects on the performance of orthogonalization step. In orthogonalization step, only one processor execute the serial code so it can access whole free memories. However, in our program we need to store a lot of global variables. Since in MPI each processor has its own variables, the need of memory increases as N increases.

In order to see whether the performance is similar in other types of clusters. We show the results of Mg_{30}, where Mg atom has 2 valence electrons. Mg_{30} has 30 valence electrons with spin-up and spin-down respectively. We list the results in Table IV. In Mg_{30}, the

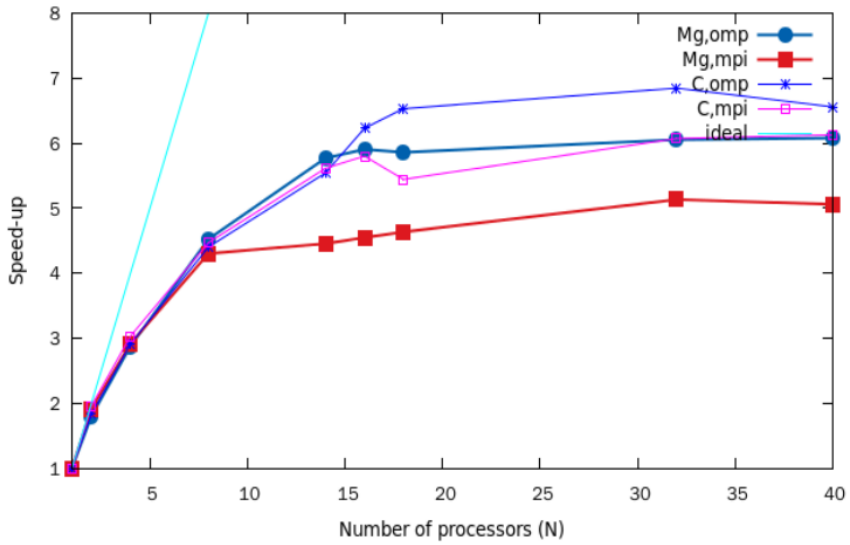

FIG. 5. To compare Mg_{30} and C_{60}, speed-up in propagation step for OpenMP and MPI.

orthogonalization step takes more time in the C_{60} case. It is strange... In Mg_{30}, we have less states but it takes more time. Does it make sense? Since in Mg_{30} it needs more iteration in propagation step. It is clear that MPI does not perform as well as OpenMP as $N > 4$. In Figure 5, we show the speed-up in Mg_{30} and C_{60} clusters. As $N \leq 8$, the performance are the same in each case. However, as $N > 8$, the performance in Mg_{30} is not as good as it in C_{60}. This is not what you expect. Is it because of the difference between local (Mg_{30}) and non-local (C_{60}) pseudopotential?

2. Orthogonalization step

3. Update density step

In this section, we show the performance in updating algorithm. We also compare C_{60} and Mg_{30} clusters. We list results in Table V. As $N > 14(16)$, OpenMP performs badly in C_{60} (Mg_{30})as N increases due to overhead. Although processors in OpenMP do not need to communicate with each other, OpenMP takes time to partition jobs and data as we start the parallel loop. In contrast, MPI performs better than OpenMP in updating density step. Although processors need to communicate with each other, it may not longer than that

N	[C_{60}] $T_{upd,omp}$ (s)	[C_{60}] $T_{upd,mpi}$ (s)	[Mg_{30}] $T_{upd,omp}$ (s)	[Mg_{30}] $T_{upd,mpi}$ (s)
1	331.73	332.00	628.12	634.38
2	215.50	200.75	434.50	402.38
4	140.00	128.88	279.00	236.00
8	104.00	96.00	182.75	145.50
14	91.00	94.62	154.00	134.62
16	109.00	88.12	153.75	131.88
18	97.25	89.75	164.25	133.38
32	130.75	91.50	233.00	138.38
40	142.50	84.75	230.50	148.12

TABLE V. N is the number of processors we use, $T_{upd,omp}$ ($T_{upd,mpi}$) is the elapsed time in update density step by using OpenMP (MPI).

OpenMP forks jobs to each processor. Another reason is that each processor in MPI has its own local variables. We know that FFTW takes a lot of time. Each processor in MPI has its own planner to execute FFT; in contrast, each processor in OpenMP share the same planner to execute. I'm not sure whether that is the main reason.

Appendix A: Fundamental principle of diffusion algorithm

Real space diffusion algorithm is an effective way to solve Schrödinger equations. In this appendix, we derive a fundamental principle in this algorithm. The imaginary-time evolution operator $\mathscr{T}(\beta) \equiv e^{-\beta H}$ projects out the ground state $|\psi_0\rangle$ from *any* initial state that is not orthogonal to the ground state. That is, given an arbitrary $|\phi_0^{(0)}\rangle$ that satisfies $\langle\phi_0^{(0)}|\psi_0\rangle \neq 0$, we have

$$\lim_{\beta\to\infty} \exp[-\beta(H - E_0)]|\phi_0^{(0)}\rangle \propto |\psi_0\rangle. \tag{A1}$$

Here β is real and positive, $|\rangle$ and $\langle|$ are ket and bra, known as Dirac notation, and E_0 is the ground-state energy. Also, the components of $|\psi_j\rangle$ in the $\{|\mathbf{r}\rangle\}$ basis are denoted by

$$\langle\mathbf{r}|\psi_j\rangle = \psi_j(\mathbf{r}), \tag{A2}$$

which is known as the *wave function* for the state vector $|\psi_j\rangle$, and $\langle\phi_0^{(0)}|\psi_0\rangle$ is a scalar product.

To prove this property, one can expand $|\phi_0^{(0)}\rangle$ by eigenstates $|\psi_n\rangle$:

$$|\phi_0^{(0)}\rangle = \sum_n c_n|\psi_n\rangle, \tag{A3}$$

where $c_n = \langle\psi_n|\phi^{(0)}\rangle$ and $|\psi_0\rangle$ is the ground state. Hence,

$$\begin{aligned}\lim_{\beta\to\infty}\exp(-\beta(H-E_0))|\phi_0^{(0)}\rangle &= \lim_{\beta\to\infty}\exp(-\beta(H-E_0))\left(\sum_n c_n|\psi_n\rangle\right)\\ &= \sum_n \lim_{\beta\to\infty}\exp(-\beta(E_n-E_0))c_n|\psi_n\rangle. \end{aligned}\tag{A4}$$

Here we have used the fact that $H|\psi_n\rangle = E_n|\psi_n\rangle$. Since β is real and positive, as $\beta\to\infty$ only ground state survives. That is,

$$\lim_{\beta\to\infty}\exp(-\beta(H-E_0))|\phi_0^{(0)}\rangle = c_0|\psi_0\rangle \propto |\psi_0\rangle. \tag{A5}$$

That is why we require $\langle\phi_0^{(0)}|\psi_0\rangle \neq 0$, or $c_0 \neq 0$.

According to this property, we can apply the evolution operator $\mathscr{T}(\epsilon)$ many times on the initial given state $|\phi_0^{(0)}\rangle$,

$$[\mathscr{T}(\epsilon)]^N|\phi_0^{(0)}\rangle = e^{-N\epsilon H}|\phi_0^{(0)}\rangle \tag{A6}$$

$$\Rightarrow \lim_{N\to\infty} e^{-N\epsilon H}|\phi_0^{(0)}\rangle \propto |\psi_0\rangle. \tag{A7}$$

In the end, we obtain the convergent ground state $|\psi_0\rangle$. Then we can repeat the above procedures to obtain higher state by requiring the initial state orthogonal to the states we obtain. For example, we require the first excited state orthogonal to the ground state $\langle\phi_1^{(0)}|\psi_0\rangle = 0$ so the expansion of $|\phi_1^{(0)}\rangle$ by eigenstates becomes

$$|\phi_1^{(0)}\rangle = \sum_m c_m|\psi_m\rangle, \tag{A8}$$

where $m \neq 0$. Thus

$$\begin{aligned}\lim_{\beta\to\infty}\exp(-\beta(H-E_1))|\phi_1^{(0)}\rangle &= \lim_{\beta\to\infty}\exp(-\beta(H-E_1))\left(\sum_m c_m|\psi_m\rangle\right)\\ &= \sum_m \lim_{\beta\to\infty}\exp(-\beta(E_m-E_1))c_m|\psi_m\rangle \propto |\psi_1\rangle. \end{aligned}\tag{A9}$$

Here we also require $\langle\phi_1^{(0)}|\psi_1\rangle \neq 0$ so only the first excited state $|\psi_1\rangle$ survives as we take the limit. Note that if $\langle\phi_1^{(0)}|\psi_0\rangle \neq 0$, then we again obtain the ground state $|\psi_0\rangle$. That is why we orthogonalize each state after each time step.

Appendix B: Trotter's Product Rule

In this appendix, we prove Trotter's product rule:

$$\lim_{N\to\infty}\left\{\left(\exp\left[-\frac{\lambda}{N}(\hat{T}+\hat{V})\right]\right)^N - \left(\exp\left[-\frac{\lambda}{N}\hat{T}\right]\exp\left[-\frac{\lambda}{N}\hat{V}\right]\right)^N\right\} = 0. \tag{B1}$$

First, we show that the two operator functions

$$\hat{F}(\alpha) = e^{-\alpha(\hat{T}+\hat{V})} \text{ and } \hat{G}(\alpha) = e^{-\alpha\hat{T}}e^{-\alpha\hat{V}} \text{ with } \alpha = \frac{\lambda}{N} \tag{B2}$$

differ only by commutation terms, which vanish in the limit $N \to \infty$.

The operator function $\hat{G}(\alpha)$ is given by

$$\begin{aligned}\hat{G}(\alpha) &= \sum_{n=0}^{\infty}\frac{(-\alpha)^n}{n!}\left(\hat{T}\right)^n\sum_{m=0}^{\infty}\frac{(-\alpha)^m}{m!}\left(\hat{V}\right)^m \\ &= \left[\mathbb{I}+(-\alpha)\hat{T}+\frac{(-\alpha)^2}{2!}\hat{T}^2+\frac{(-\alpha)^3}{3!}\hat{T}^3+\ldots\right] \\ &\quad\times\left[\mathbb{I}+(-\alpha)\hat{V}+\frac{(-\alpha)^2}{2!}\hat{V}^2+\frac{(-\alpha)^3}{3!}\hat{V}^3+\ldots\right] \\ &= \mathbb{I}+\alpha(-\hat{T}-\hat{V})+\frac{\alpha^2}{2!}(\hat{T}^2+\hat{V}^2+2\hat{T}\hat{V})+\ldots\end{aligned} \tag{B3}$$

It can be defined by its Taylor series,

$$\hat{G}(\alpha) = \sum_{n=0}^{\infty}\frac{(\alpha)^n}{n!}\left(\frac{d^n\hat{G}(\beta)}{d\beta^n}\right)\Bigg|_{\beta=0}, \tag{B4}$$

where

$n = 0$:

$$\hat{G}(\beta)|_{\beta=0} = \mathbb{I}; \tag{B5}$$

$n = 1$:

$$\begin{aligned}\frac{d\hat{G}}{d\beta} &= (-)\hat{T}\hat{G}(\beta)+(-)e^{-\beta\hat{T}}\hat{V}e^{-\beta\hat{V}} \\ &= (-)\hat{T}\hat{G}(\beta)+(-)e^{-\beta\hat{T}}\hat{V}e^{\beta\hat{T}}e^{-\beta\hat{T}}e^{-\beta\hat{V}} \\ &= (-)\hat{T}\hat{G}(\beta)+(-)\left(\hat{V}+\sum_{m=1}^{\infty}\frac{(-\beta)^m}{m!}[\hat{T},\hat{V}]_{(m)}\right)\hat{G}(\beta) \\ &= (-)(\hat{T}+\hat{V})\hat{G}(\beta)+(-)\sum_{m=1}^{\infty}\frac{(-\beta)^m}{m!}[\hat{T},\hat{V}]_{(m)}\hat{G}(\beta),\end{aligned} \tag{B6}$$

where $[\hat{T},\hat{V}]_{(0)} = \hat{V}, [\hat{T},\hat{V}]_{(1)} = [\hat{T},\hat{V}], [\hat{T},\hat{V}]_{(2)} = [\hat{T},[\hat{T},\hat{V}]], \ldots$, and $[\hat{T},\hat{V}] = \hat{T}\hat{V} - \hat{V}\hat{T}$. Here we have used the **Baker-Hausdorff lemma**,

$$e^{\alpha\hat{A}}\hat{B}e^{-\alpha\hat{A}} = \sum_{n=0}^{\infty} \frac{(\alpha)^n}{n!}[\hat{A},\hat{B}]_{(n)}. \tag{B7}$$

Hence,

$$\left.\frac{d\hat{G}}{d\beta}\right|_{\beta=0} = (-)(\hat{T}+\hat{V}); \tag{B8}$$

$n = 2$:

$$\frac{d^2\hat{G}}{d\beta^2} = \left((-)(\hat{T}+\hat{V}) + (-)\sum_{m=1}^{\infty}\frac{(-\beta)^m}{m!}[\hat{T},\hat{V}]_{(m)}\right)\frac{d\hat{G}}{d\beta} + (-1)\sum_{m=1}^{\infty}\frac{(-1)^m\beta^{m-1}}{(m-1)!}[\hat{T},\hat{V}]_{(m)}\hat{G}(\beta), \tag{B9}$$

$$\left.\frac{d^2\hat{G}}{d\beta^2}\right|_{\beta=0} = (-1)^2(\hat{T}+\hat{V})^2 + (-1)^2[\hat{T},\hat{V}]. \tag{B10}$$

$n = 3$:

$$\begin{aligned}\frac{d^3\hat{G}}{d\beta^3} = &\left((-1)\sum_{m=1}^{\infty}\frac{(-1)^m\beta^{m-1}}{(m-1)!}[\hat{T},\hat{V}]_{(m)}\right)\frac{d\hat{G}}{d\beta} \\ &+\left((-1)(\hat{T}+\hat{V}) + (-1)\sum_{m=1}^{\infty}\frac{(-\beta)^m}{m!}[\hat{T},\hat{V}]_{(m)}\right)\frac{d^2G}{d\beta^2} \\ &+(-1)\sum_{m=1}^{\infty}\frac{(-1)^m\beta^{m-1}}{(m-1)!}[\hat{T},\hat{V}]_{(m)}\frac{d\hat{G}}{d\beta} \\ &+(-1)\sum_{m=2}^{\infty}\frac{(-1)^m\beta^{m-2}}{(m-2)!}[\hat{T},\hat{V}]_{(m)}\hat{G},\end{aligned} \tag{B11}$$

$$\left.\frac{d^3\hat{G}}{d\beta^3}\right|_{\beta=0} = (-1)^3(\hat{T}+\hat{V})^3 + 3(-1)^3[\hat{T},\hat{V}](\hat{T}+\hat{V}) + (-1)[\hat{T},[\hat{T},\hat{V}]] \tag{B12}$$

Similarly, all higher derivatives can be determined. Then one gets

$$\left.\frac{d^n\hat{G}}{d\beta^n}\right|_{\beta=0} = (-1)^n(\hat{T}+\hat{V})^n + \{\text{commutator terms}\}. \tag{B13}$$

Inserting this into the Taylor expansion (B4), one obtains

$$\hat{G}(\alpha) = \sum_{n=0}^{\infty}\frac{(\alpha)^n}{n!}(-1)^n(\hat{T}+\hat{V})^n + \frac{\alpha^2}{2!}(-1)^2[\hat{T},\hat{V}] + \mathcal{O}(\alpha^3) \tag{B14}$$

$$= \hat{F}(\alpha) + \frac{\alpha^2}{2!}(-1)^2[\hat{T},\hat{V}] + \mathcal{O}(\alpha^3). \tag{B15}$$

Hence we find

$$[\hat{F}(\alpha)]^N - [\hat{G}(\alpha)]^N = \mathcal{O}(\alpha^2), \tag{B16}$$

i.e. the above difference is proportional to leading order $\alpha^2 = \lambda^2/N^2$. In the limit $N \to \infty$ the right-hand side of (B16) vanishes, so we prove the validity of Trotter's formula (B1).

Appendix C: Block-partitioned Algorithm

In this section, we introduce the scalability of matrix-matrix multiplication or matrix-vector multiplication in different partition algorithm in MPI. One can also refer to this website.

1. Matrix-Vector Multiplication

First of all, we consider the problem of multiplying a dense $n \times n$ matrix A with an $n \times 1$ vector x to yield the $n \times 1$ resulting vector y. A serial algorithm for this problem is in the following,

```
1.   procedure MAT_VECT ( A, x, y)
2.   begin
3.      for i := 0 to n - 1 do
4.      begin
5.         y[i]:=0;
6.         for j := 0 to n - 1 do
7.             y[i] := y[i] + A[i, j] * x[j];
8.      endfor;
9.   end MAT_VECT
```

The sequential algorithm requires n^2 multiplications and additions. Assuming that a multiplication or addition (one floating operation) takes time γ, the sequential run time (W) is

$$W = 2n^2\gamma. \tag{C1}$$

At least three distinct parallel formulations of matrix-vector multiplication are possible: rowwise 1-D, columnwise 1-D, or a 2-D partitioning.

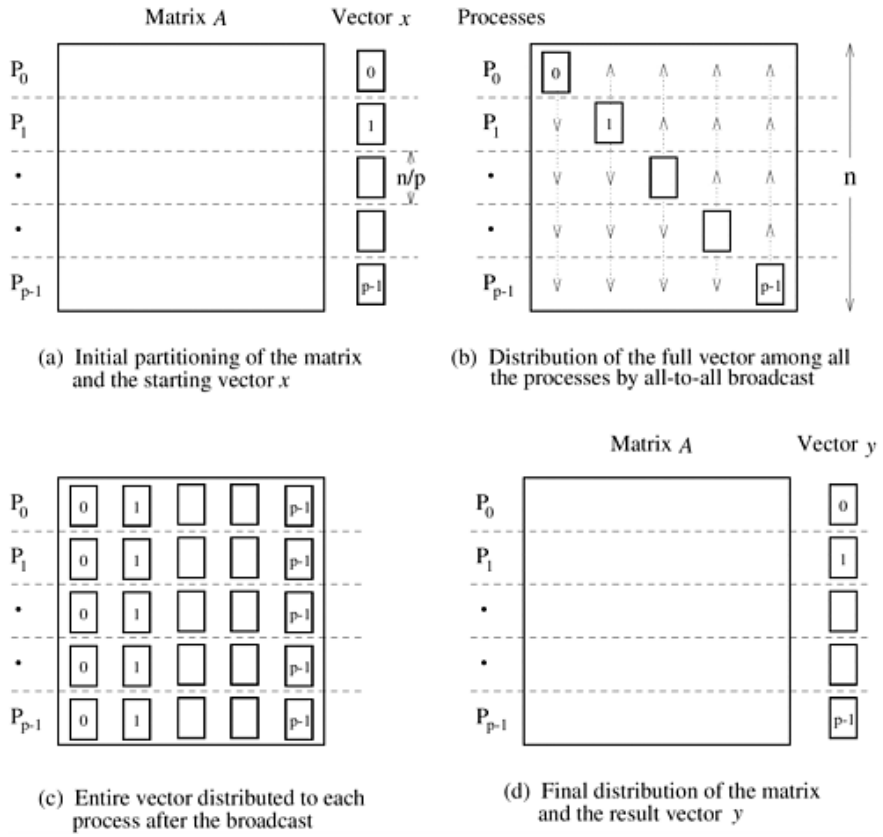

FIG. 6. The total speed-up time factor S_{tot} (solid lines) and the propagation only (without orthogonalization) time speed-up factor S_{pro}, as a function of parallel threads, for the 2nd-, 6th-, and 12th-order algorithm (filled squares, circles, and trangles, respectively). Also shown is the 'ideal' speed-up factor (dotted line). These results are generated by using OpenMP. [From Ref. website]

Appendix D: CPU information

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                40
On-line CPU(s) list:   0-39
Thread(s) per core:    2
Core(s) per socket:    10
Socket(s):             2
NUMA node(s):          2
```

Vendor ID:	GenuineIntel
CPU family:	6
Model:	63
Stepping:	2
CPU MHz:	2301.000
BogoMIPS:	4601.32
Virtualization:	VT–x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	25600K
NUMA node0 CPU(s):	0−9,20−29
NUMA node1 CPU(s):	10−19,30−39

---

[1] T. L. Beck, Rev. Mod. Phys. **72**, 1041 (2000).

[2] S. A. Chin, S. Janecek, and E. Krotscheck, Chem. Phys. Lett. **470**, 342 (2009).

[3] http://www.limerec.net.

[4] M. Creutz and A. Gocksch, Phys. Rev. Lett. **63**, 9 (1989).

[5] M. Suzuki, Phys. Lett. A **146**, 319 (1990).

[6] S. Chin, Celest. Mech. Dyn. Astron. **106**, 391 (2010).

[7] https://github.com/jrfonseca/gprof2dot.

[8] http://www.netlib.org/scalapack/scalapack_home.html.