Parallelized MP2 calculation for molecular system

Hieu Dinh[1], Artur Lyssenko[1], Olga Borodina[2], and Mouza Almualla[2]

[1]Harvard Department of Chemistry and Chemical Biology
[2]Harvard Department of Astronomy

May 4, 2023

Abstract

This proposal focuses on a parallelized implementation of RI-MP2 using both shared and distributed memory parallelism studying the model system C_3H_8 (propane). Our serial code is optimized using BLAS DGEMM subroutines. OpenMP is used for shared memory parallelization for different compute kernels and MPI is used for distributed memory parallelization for the last kernel. Strong scaling analysis shows a moderate speedup for shared memory parallelization (up to 15x for 32 cores) likely owing to poor exploitation of cache locality. Weak scaling analysis for non-blocking MPI implementation reveals a high efficiency (around 90%) for large numbers of processors indicating a good compute/transfer overlap. Resources are requested for 13 node hours to study C_{60} and $C_{34}H_{32}FeN_4O_4$ (heme) which are much larger in size compared to our model system. Future work will involve integrating OpenMP with MPI as well as extending our framework to periodic boundary condition RI-MP2 in order to study solid state systems.

1 Background and Significance

Solving the Schrodinger equation for systems with more than one electron is a formidable task due to the electron-electron interaction. To obtain a rough estimation for the ground state energy, the electron-electron interaction term is often treated using a mean-field approach with a single electronic configuration (Hartree-Fock calculation). There are many approaches, such as CCSD, CI, and MP2, to calculate the remaining electron correlation starting from the mean-field solution. Each method has its own advantage, depending on the type of interactions (static vs dynamic). In computational chemistry, we want to calculate the electronic ground state energy of molecules with high accuracy to model the quantum mechanical behavior of molecules. This would aid experimental research in understanding reaction mechanisms and the design of molecules. Furthermore, the ground state energy calculation can serve as a metric/input to other data-driven methods (e.g: using machine learning to predict the properties of molecules).

In this project, we aim to parallelize the calculation for MP2 energy correction starting from the mean-field solution. MP2 (Møller–Plesset) energy is derived from perturbation theory in quantum mechanics. Given the orbital energy and the two-electron integrals in the Molecular Orbitals basis, we have the expression

$$E_{MP2} = \sum_{ij} \sum_{ab} \frac{(ia|jb)\left[2\,(ia|jb) - (ib|ja)\right]}{\epsilon_i + \epsilon_j - \epsilon_a - \epsilon_b} \tag{1}$$

where ϵ is an orbital energy (discussed later) and $(ia|jb)$ is a four electron integral

$$(ia|jb) = \int d\mathbf{r_1} d\mathbf{r_2} \psi_i(\mathbf{r_1})^* \psi_a(\mathbf{r_1}) r_{12}^{-1} \psi_j(\mathbf{r_2})^* \psi_b(\mathbf{r_2}) \tag{2}$$

The electron correlation can be calculated using auxiliary basis functions to reduce the run time requirement to transform from Atomic Orbitals (AO) basis to Molecular Orbitals (MO) basis. This is known as the resolution of identity method (RI).

$$(ia|jb)^{RI} = \sum_{P} B_{ia}^{P} B_{jb}^{P} \tag{3}$$

The **B** matrices are constructed from the coefficient matrices (**C**) obtained from solving the self consistent Hartree Fock (HF) equations.

$$\mathbf{FC} = \epsilon \mathbf{SC} \tag{4}$$

Where **F** is the Fock matrix, **C** is the (AO) coefficient matrix, ϵ is an orbital energy matrix, and **S** is the overlap matrix for wavefunctions. These coefficient matrices are then contracted with the three center integrals between the auxiliary basis and AOs to transform to the MO basis .

$$(pq|P) = \sum_{\mu\nu} (\mu\nu|P)\, C_{\mu p} C_{\nu q} \tag{5}$$

This is further contracted with the overlap matrix $\mathbf{S^{-1/2}} = (L|P)$ of the auxiliary basis (not to be confused with the other overlap matrix) to orthogonalize the B matrix. From here on out $(L|P)$ will be plainly referred to as the overlap or **S**.

$$B_{ia}^{P} = \sum_{L} (ia|L)\, (L|P) \tag{6}$$

This completes the recipe for computing the RI-MP2 energy.

An implementation of RI-MP2 is presented by Katouda et. al. [3]. In this work the RI-MP2 algorithm was parallelized using MPI and OpenMP allowing for computation of massive systems. Although our code has some key differences and is not as integrated in the MPI / OpenMP formalism it allowed us to compare our work to some published results.

Our work is directly applicable to many quantum chemistry systems of interest. Due to the increase in computing power through HPC more complicated, and thus interesting, systems can be studied through parallelization. Our implementation opens the door for Q-Chem (a quantum chemistry code specializing in computation of molecular properties) to extend the scale of its simulations. Another application of this could be the generalization of our code to periodic boundary conditions. This would allow for the computation of properties of solids which due to the massive problem sizes involved benefit significantly from parallelism. Periodic boundary condition RI-MP2 work has been developed along with methods to reduce the scaling of the method with respect to basis functions. This work highlights that parallelization is even more natural within the periodic case as different k-space points can be distributed over workers[4]. Thus, extending our algorithm to periodic boundary conditions within Q-Chem with support for both OpenMP and MPI would allow for the computation of much larger systems of interest.

2 Scientific Goals and Objectives

As mentioned in the previous section parallelization opens doors to study much more interesting and complex systems. Our objective with this project is to investigated the benefit parallelized code provides in computing MP2 energies. The model system we study is propane (C_3H_8) which has 202 basis functions, 13 electrons, and 483 auxiliary basis functions. This system has a modest memory requirement of 157

MB. However, more complicated systems such as C_{60}, which has 1800 basis functions and 4860 auxiliary basis functions and a memory requirement of 110 GB, need better distributed memory parallelism as well as shared memory parallelism. It is thus necessary to explore parallelization to study chemically interesting systems such as metal-ligand complexes where number of basis functions and auxiliary basis functions is in the thousands. An example of such a calculation is the Heme center which is responsible for carrying oxygen throughout the body. The binding of oxygen to Heme and its derivatives have been studied using other methods, specifically another variant of second-order perturbation theory [2]. It would be interesting to study this system using our developed code.
We first optimize our serial code using CBLAS package and later provide justification of our plot using both strong and weak scaling analysis for OpenMP and MPI implementations respectively.

3 Algorithms and Code Parallelization

We obtain the coefficient matrix, three-center integral matrix, orbital energies, and overlap matrix from the Q-Chem software. The relevant parameters are defined as follows: N_{AUX} number of auxiliary basis functions, N_{BASIS} number of electrons, and N_{ELECS} number of electrons. The dimension of our matrices is as follows: **coefficient matrix (C)** $= N_{\text{BASIS}} \times N_{\text{BASIS}}$, **three-center integral matrix** $(\mu\nu|P) = N_{\text{BASIS}} \times N_{\text{BASIS}} \times N_{\text{AUX}}$, **overlap matrix** $= N_{\text{AUX}} \times N_{\text{AUX}}$, and **orbital energy matrix** $= N_{\text{BASIS}}$. We then load this matrix in our program and store it linearly in memory, keeping the slowest changing index (usually N_{AUX}) as our first index. This step is performed in all of our implementations. For the code development phase, we only use small molecule C_3H_8 as our benchmarking test and do not aim to optimize the I/O. In real application, there would be no I/O since all the integrals will be calculated on the fly and not be saved to any files. Thus, for subsequent benchmarks, we only record the time required to run each compute kernel given that we have access to the input data.
Throughout the proceeding subsections, we are the main developers of the code.

Naive Implementation

We first explore a naive implementation of our algorithm. Step 0 is the loading step described above. Step 1 follows equation 5 in which we contract the coefficient matrices with the three-center integral matrix sequentially (Algorithm 1).

Algorithm 1: Basis transformation (MO1 and MO2)

for $P < N_{AUX}$ **do**
  $(p\nu|P) \leftarrow \text{DGEMM}(C^{\dagger}, (\mu\nu|P))$;
end
for $P < N_{AUX}$ **do**
  $(pq|P) \leftarrow \text{DGEMM}(C, (\mu\nu|P))$;
end

The second step of our code involves formation of the B_{pq}^{P} matrix through orthongonalization with the overlap matrix as seen in Equation 6. The pseudocode is in Algorithm 2.

Algorithm 2: Orthogonalization

$B_{pq}^{P} \leftarrow \text{DGEMM}\left((pq|P), (P|Q)\right)$ with ColMajor order **OR**;
for $p < N_{BASIS}$ **do**
  $B_{pq}^{P} \leftarrow \text{DGEMM}\left((pq|P), (P|Q)\right)$ with RowMajor order;
end

The third step of our algorithm involves the MP2 energy calculation as seen in Equation 1 (see Algorithm 3).

Algorithm 3: MP2 Energy Calculation

```
for i < N_ELECS do
    for j < N_ELECS do
        (ia|jb) ← DGEMM (B^P_ia, B^P_jb) over the auxiliary basis dimension;
        for a < N_BASIS do
            for b < N_BASIS do
                Δ ← MO_energy[i] + MO_energy[j] − MO_energy[a] − MO_energy[b];
                E(MP2) ← 1/Δ ((ia|jb)(ia|jb) − (ia|jb)(ib|ja))) ;
            end
        end
    end
end
```

In all of these cases we use manual for loops. The slowest changing index (P or Q) is kept on the outside for Algorithm 1, making sure to *exploit cache locality* as much as possible and lower conflict misses. For our MP2 energy calculation we compute $(ia|jb)$ and $(ib|ja)$ in the inner most loop by looping over P, with the outside loops being over i, j, a, and b. This allows us to avoid storing the four electron integrals in memory while paying the price of capacity misses. We do not use a blocking implementation in our naive design.

BLAS Implementation

We optimize our serial code using BLAS DGEMM subroutines. For Algorithm 1, this involves keeping the P index fixed and multiplying through each matrix slice. For Algorithm 2, the operation could be performed using a singular BLAS call which maximizes operational intensity. We also implemented a version of this algorithm that is easier to parallelize using OpenMP in which the $(pq|P)$ data layout was reordered to make p the slowest-changing index and P the fastest-changing one. This allowed us to parallelize the loop over p. For Algorithm 3, the B^P_{pq} matrix was reshaped to keep only the occupied/virtual block (indices N_{BASIS} - N_{ELECS}), making the occupied block our slowest changing index and the virtual block the fastest changing one, as that is what is needed for our calculations of the energy. We then contract over P using BLAS to obtain a $(ip|jq)$ tensor. This allows us to do elementwise matrix multiplication to get the desired $(ia|jb)$ and $(ib|ja)$ elements for our energy calculation. Whenever possible, we attempted to perform DGEMM over the largest dimension to keep our operational intensity as high as possible.

OpenMP Implementation

Our OpenMP implementation built upon our BLAS implementation by parallelizing the for loops over which BLAS was operating. For Algorithm 1 this involved single parallel for pragmas. For Algorithm 2 we employed a parallel for as well as a collapse(3) clause to aid in reshaping our $(pq|P)$ tensor. A single parallel for was then used on our reshaped tensor to compute B^P_{pq}. A similar approach was taken for Algorithm 3 using a parallel for and collapse clause to resize our B matrix. We then used a parallel for and collapse while calculating $(ia|jb)$ as well as a reduction for the final MP2 energy.
This approach allowed us to further increase our operational intensity by using shared memory parallelism. Each parallel collapse clause is taken into careful consideration. The work-sharing benefit from collapsing the reshaping loops allows for a speedup which is trivial when compared to other kernels within our code.

The collapse in Algorithm 3 is justified as the collapse is over a small number of iterations ($N_{\text{ELECS}} \times N_{\text{ELECS}}$) and thus offers an increased iteration space. Furthermore, we set the thread affinity with the environment variable OMP_PROC_BIND = True. Thread affinity will boost the performance of our code since we use omp pragma to parallelize data layout reshuffling and resizing.
The OpenMP code presented here can be easily integrated into the MPI code later in case we need multi nodes for gigantic molecular systems.

MPI Implementation

We identify the rank-three tensor as the memory bottleneck of our MP2 energy calculation since it scales cubically with the problem size. Thus, calculating MP2 energy for large molecules or solid systems requires the distribution of the rank three tensor B_{ia}^{P} to different ranks. The first step and the second step (DGEMM) can be done independently on each rank, assuming that ranks in MPI World have access to **C**, **MO_energy**, and **overlap** matrix. Going from step 1 to step 2 requires some data transposition that can be achieved with specific MPI_Isend and MPI_Irecv. After step 3, each rank has some parts of rank three tensor in MO basis. Here we focus on implementing the non-blocking MPI procedure for step 3 to obtain the MP2 energy correction. We implement a non-blocking MPI procedure to get the partial MP2 energy on each rank and use MPI_Reduce to get the final MP2 energy on the root rank (Algorithm 4). To hide the communication latency with useful computation, we allocate extra memory Brecv_ to store the incoming message. The number of MPI communications is also minimized to (num_procs - 1) since the last communication returns the same matrix to B.

Algorithm 4: Non-blocking MPI Implementation

```
Initialize 1D Cartesian Grid;
for k = 1; k < NumRanks do
    MPI_Isend(B) to the RIGHT neighbor;
    MPI_Irecv(Brecv_) from the LEFT neighbor ;
    MP2 Energy Calculation kernel (3) + Update index ;
    MPI_Waitall(2);
    Swap B and Brecv_ ;
end
MP2 Energy Calculation kernel (3);
MPI_Reduce(&MP2_partial, &MP2_energy, root rank);
```

Vectorization with ISPC

After BLAS implementation, we realized that the two inner loops in the MP2 energy calculation are element-wise operations on the rank four tensor $(ia|jb)$; thus, it can be vectorized with ISPC to exploit the data level parallelism of the hardware and further push the performance of the compute kernel. The vectorized code can be found in the Vectorized kernel folder. However, since the bulk of the computation comes from the DGEMM to obtain the rank four-tensor, we do not observe significant speedup for the whole kernel and decide to not optimize/vectorize this vectorized subkernel further.

Validation, Verification

The numerical accuracy of our model can be compared to literature values obtained from the quantum chemistry literature. Since we are free to choose the inputs for our system a relatively inexpensive calculation can be performed using a propane dimer. This can then be compared with the results of

this paper taking care to choose our basis consistent with the reference [5]. A similar process can be done for other molecules and geometries of interest for which reference literature values exist if further confirmation is desired [1].

4 Performance Benchmarks and Scaling Analysis

Node-level Performance

To better understand the node-level performance of our compute kernel, we measured the runtime for each compute kernel (1, 2, 3) and characterize the performance of the DGEMM from BLAS and naive for loop using roofline analysis. For subsequent computation, we use C_3H_8 molecule with $N_{\text{ELECS}} = 13$, $N_{\text{BASIS}} = 202$, $N_{\text{AUX}} = 483$ as our test case.
We calculated the double precision peak arithmetic for Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz as follow

$$\pi = f \times n_c \times l_s \times \varphi = 2.1 \times 10^9 \times 32 \times \frac{256}{64} \times 4 \times \frac{1}{10^9} = 1075.2 \text{ Gflop /s}, \tag{7}$$

and we construct the roofline ceiling for performance analysis using $\beta = 76.8$GB/s (Figure 1). Most of our computation will be based on DGEMM kernel, which has the operational intensity that scales linearly as the dimension of the matrix. In step 3, the element-wise operation would have operational intensity much lower than the operational intensity from the DGEMM; hence, we can safely ignore the operational intensity from that subkernel.
To get the operational intensity and the performance of each kernel after time measurement, we count the number of floating point operations for each compute kernel.

$$F_{\text{MO1, 2}} = 2 \times N_{\text{BASIS}}^3 \times N_{\text{AUX}} \tag{8}$$

$$F_{\text{Orthog}} = 2 \times N_{\text{AUX}}^3 \times N_{\text{BASIS}} \tag{9}$$

$$F_{\text{MP2}} = 2 \times (N_{\text{BASIS}} - N_{\text{ELECS}})^2 \times N_{\text{AUX}} + (N_{\text{BASIS}} - N_{\text{ELECS}})^2 \times 11) \tag{10}$$

Using the above information, we get the following expression for the operational intensity of our DGEMM kernels

$$I_{\text{MO1, 2}} = \frac{F_{\text{MO1, 2}}}{8 \cdot 4 \cdot N_{\text{BASIS}}^2 N_{\text{AUX}}} = \frac{1}{16} N_{\text{BASIS}} \tag{11}$$

$$I_{\text{Orthog}} = \frac{F_{\text{Orthog}}}{8 \cdot 4 N_{\text{AUX}}^2 N_{\text{BASIS}}} = \frac{1}{16} N_{\text{AUX}} \tag{12}$$

$$I_{\text{MP2}} = \frac{F_{\text{MP2}}}{3 N_{\text{BASIS}}^2} = \frac{1}{16} N_{\text{AUX}} \tag{13}$$

To determine the performance (Gflop/s) for each kernel, we measure the runtime for each of them (Table 1) and use the number of floating point operations from above. We observe significant improvement when using the highly optimized subroutines from BLAS (Figure 1). The bad performance of our naive kernel can be attributed to the bad exploitation of the cache locality due to the large size of the matrix. The OpenMP helps push the performance of the compute kernel further by dividing the work between cores (Figure 1), and we will analyze this further in the next section.

Step	for-loops	CBLAS	OpenMP
MO1	20.99	1.42	0.09
MO2	28.67	1.45	0.13
Orthogonalization	50.54	3.29	0.28
MP2	34.04	1.18	0.07

Table 1: Wall-clock time in seconds for each kernel and different implementations.

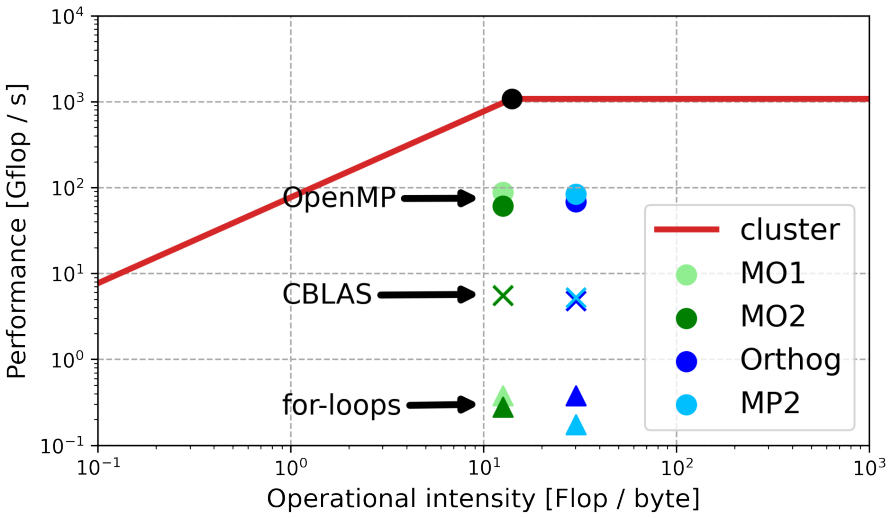

Figure 1: Roofline analysis for each compute kernel in the MP2 energy calculation

Strong scaling Analysis for shared memory parallelization

For our OpenMP implementation, a strong-scaling analysis is sufficient in representing the speed-up of our program as we scale up in cores. We show in Figure 2 the speed-up as a function of the number of threads for each kernel in the program and for the total kernel in the program. We observe that the scaling for MP2 kernel is the best, while the scaling for MO2 kernel is the bottleneck of the whole program. On 32 cores, we observe a speed-up of up to almost 20× in the MP2 calculation, and of ∼15× in the MO1 and orthogonalization kernels.

Interestingly, the MO2 calculation (shown as a green solid line) does not scale as well as the other kernels. As shown in Figure 1, this performance gap between MO1 and MO2 is apparent in our OpenMP implementation, but not the serial OpenBLAS implementation; this is likely because the speed-up with OpenMP reveals the limitation of the MO2 calculation due to the data array structure. More specifically, the matrix multiplication in MO2 does not exploit cache locality as well as that in MO1 (despite having the same operational intensity), and would require a reshuffling of the data layout. Although this reshuffling could be done, it is not necessary since the orthogonalization step is still the bottleneck when running on 32 threads, as can be seen in the last column of Table 2, which shows the wall-clock time for each kernel for varying number of threads.

Step	1	2	4	8	16	32
MO1	1.41	0.77	0.42	0.25	0.15	0.09
MO2	1.45	0.81	0.46	0.29	0.20	0.13
Orthogonalization	4.16	2.16	1.20	0.68	0.37	0.28
MP2	1.31	0.58	0.34	0.20	0.11	0.07

Table 2: Wall-clock time in seconds for each kernel for different number of cores.

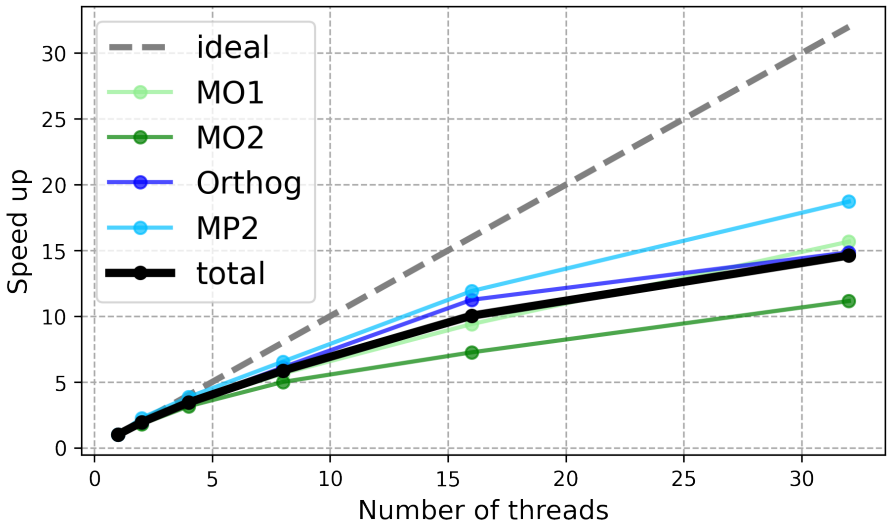

Figure 2: Strong scaling analysis shown for the different kernels in our program as solid lines. Ideal scaling is shown as a grey dashed line.

Weak scaling analysis for distributed memory parallelization

The weak scaling analysis for the non-blocking MPI implementation (Algorithm 4) can be found in Figure 3 and Table 3. In these tests, we fix the dimension of the problem variables N (Number of occupied orbitals in the rank), N_{ELECS}, N_{BASIS}, and N_{AUX} for each processor. Specifically, for small test (3), we have N = 15, $N_{\mathrm{ELECS}} = 30$, $N_{BASIS} = 100$, $N_{\mathrm{AUX}} = 400$, and for large test (3), we have N = 30, $N_{\mathrm{ELECS}} = 60$, $N_{BASIS} = 200$, $N_{\mathrm{AUX}} = 800$. For all the benchmarking, we are mapping each processor to one core in the Broadwell node. We only collected the runtime for large test case up to 8 processors due to the memory constraint on a single Broadwell node (This arises from the fact that we need extra data buffer to store matrices). We collected the time that each processor execute the work and notice no difference, indicating a perfect work balance for weak scaling analysis. The runtime recorded in Table 3 needs to be rescaled with the number of processors because as the number of processor increases, each rank is effectively doing more work due to the outermost loop in Algorithm 4. The weak scaling efficiency is calculated as follow

$$E_p = \frac{T(1) \times p}{T(p)}, \text{ where T(p) is the runtime with p processors} \tag{14}$$

We see that the weak efficiency of our non-blocking MPI implementation scales well to a large number of processors (Figure 3). The weak scaling efficiency is very close to 1 for a few processors and is still

around 0.9 for 32 processors, indicating that we have been able to hide the communication between MPI ranks with useful work. However, at the end of the whole compute/transfer overlap procedure, we still have to perform a collective reduction call with MPI. This collective procedure can potentially hurt our weak efficiency.

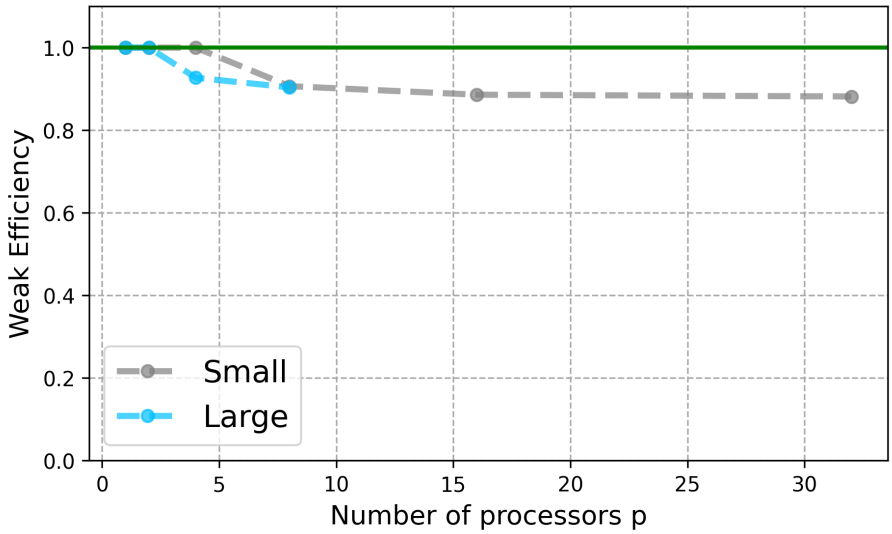

Figure 3: Weak scaling plot for non-blocking MPI implementation. Ideal scaling is the green line.

Number of processors	1	2	4	8	16	32
Time for small test	0.17	0.34	0.68	1.50	3.07	6.17
Time for large test	5.37	10.56	23.17	47.54	_	_

Table 3: Wall-clock time in seconds for different number of MPI ranks. The time should be normalized by the number of processors for weak scaling analysis.

5 Resource Justification

In this section, we give the estimation of the memory requirement (not accounting for extra memory buffer for MPI implementation) and the node hour for doing MP2 calculation on one node. Given that the rank-three tensor is the memory bottleneck of our calculation and the data in the rank-three tensor B_{ia}^{P} is double precision, the memory requirement is calculated as follows

$$\text{Memory} = 8 \times N_{\text{BASIS}}^2 \times N_{\text{AUX}}. \tag{15}$$

Time (in seconds) to calculate the is calculated using

$$\text{Time} = (3.11 \times 10^{-10}\ \text{s}) \times N_{\text{BASIS}}^2 \times N_{\text{AUX}}^2 \tag{16}$$

which comes from orthogonalization which is the bottleneck of our code. In order to calculate the prefactor, the C_3H_8 time to complete on 32 cores, N_{BASIS}, and N_{AUX} are plugged into 16. This provides an upper

bound for our calculation time. Node hours are calculated for one node for heme and C_{60} while both are using 32 cores.

	C_3H_8	C_{60}	Heme
N_{BASIS}	202	1800	632
N_{AUX}	483	4860	1618
N_{ELEC}	13	180	161
Memory	157 MB	110 GB	5.17 GB
Total node hours	2.96 s	6.61 hours	0.0903 hours

Table 4: Justification of the resource request

We propose a request for 13 node hours. This is necessary to cover the test case of C_{60} and the heme center. For bigger molecule or solid system, we would need to go further than a signel node and the non-blocking mPI implementation above is required. This add an upper bound on the memory that we can use since we need to allocate buffer for receiving and sending messages.

References

[1] Alexander G. Donchev et al. "Quantum chemical benchmark databases of gold-standard dimer interaction energies". en. In: *Scientific Data* 8.1 (Feb. 2021). Number: 1 Publisher: Nature Publishing Group, p. 55. ISSN: 2052-4463. DOI: 10.1038/s41597-021-00833-x. URL: https://www.nature.com/articles/s41597-021-00833-x.

[2] Kasper P. Jensen, Björn O. Roos, and Ulf Ryde. "O2-binding to heme: electronic structure and spectrum of oxyheme, studied by multiconfigurational methods". In: *Journal of Inorganic Biochemistry* 99.1 (2005), pp. 45–54. ISSN: 0162-0134. DOI: https://doi.org/10.1016/j.jinorgbio.2004.11.008. URL: https://www.sciencedirect.com/science/article/pii/S0162013404003691.

[3] Michio Katouda et al. "Massively parallel algorithm and implementation of RI-MP2 energy calculation for peta-scale many-core supercomputers". en. In: *Journal of Computational Chemistry* 37.30 (2016). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/jcc.24491, pp. 2623–2633. ISSN: 1096-987X. DOI: 10.1002/jcc.24491. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.24491.

[4] Tobias Schäfer, Benjamin Ramberger, and Georg Kresse. "Quartic scaling MP2 for solids: A highly parallelized algorithm in the plane wave basis". In: *The Journal of Chemical Physics* 146.10 (Mar. 2017). arXiv:1611.06797 [physics], p. 104101. ISSN: 0021-9606, 1089-7690. DOI: 10.1063/1.4976937. URL: http://arxiv.org/abs/1611.06797.

[5] Seiji Tsuzuki et al. "Estimated MP2 and CCSD(T) interaction energies of n-alkane dimers at the basis set limit: comparison of the methods of Helgaker et al. and Feller". eng. In: *The Journal of Chemical Physics* 124.11 (Mar. 2006), p. 114304. ISSN: 0021-9606. DOI: 10.1063/1.2178795.