FAST SEMISTOCHASTIC HEAT-BATH CONFIGURATION INTERACTION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Junhao Li

August 2019

FAST SEMISTOCHASTIC HEAT-BATH CONFIGURATION INTERACTION

Junhao Li, Ph.D.

Cornell University 2019

In this thesis, I present my work on the fast semistochastic heatbath configuration interaction (Fast SHCI), which is an efficient algorithm for doing essentially exact electronic structure calculations within a finite basis.

There are Hamiltonians for which the entire Hilbert space is enormous, but the important part of Hilbert space is of manageable size, say 10^{12}. Quantum chemistry Hamiltonians, for reasonably small systems, have this property. For such Hamiltonians, selected configuration interaction plus perturbation theory (SCI+PT) methods can be useful. The most important part of the Hilbert space is treated variationally, and the resulting energy is improved by using perturbation theory. Fast SHCI is more than an order of magnitude faster than other SCI+PT algorithms, and also much faster than other essentially exact algorithms for many chemical systems. This thesis provides an in-depth description of the Fast SHCI algorithm and its implementation. I use SHCI to compute the electronic structure of several chemical systems and the homogeneous electron gas. Some of these calculations are more accurate than those achieved by other high-order quantum chemistry methods. Others treat systems larger than those that can be treated by other equally accurate methods.

My implementation of SHCI uses a modular design, which not only makes the library highly extensible but also contributes several generic distributed computing building blocks to the open-source community. In this thesis, I also describe my design and implementation of these generic components.

Finally, I also provide a brief discussion of the usability of general software engineering best practices for the development of medium-scale scientific software packages with lessons learned from designing, developing, and leading the development of our SHCI package. Medium-scale scientific software packages are common in scientific research where a small group of researchers works on the same code base. Due to the differences in the requirements, some best practices that are common in the industry need to be adjusted to be useful for these projects.

BIOGRAPHICAL SKETCH

Junhao Li was born and grew up in Shanghai, China. From a young age, he had a strong interest in science and engineering and enjoyed disassembling all kinds of home appliances.

Junhao attended Shanghai Jiao Tong University from 2009, graduating with a B.S. in physics and a B.S.E in computer science in 2013. While an undergraduate, he did research in various fields, including semiconductor fabrication, photovoltaics, finite element analysis of the electromagnetic field, density functional theory, and social networks.

In the fall of 2013, Junhao came to Cornell University to pursue a Ph.D. in physics. During his Ph.D., he mainly worked with professor Cyrus Umrigar on the development and application of highly accurate quantum chemistry methods. He developed the fast heatbath configuration interaction method, which is more than an order of magnitude faster than other methods in its category, and achieved significantly higher accuracy than other well-developed methods, such as DMRG and FCIQMC, on many systems. He also did research in molecular dynamics and defects with professor James Sethna.

After obtaining a master's degree at Cornell, Junhao spent a summer as an intern at Google in California in 2016. He then returned to Ithaca, brought back the softeware engineering skills he learned from Google, and developed a highly efficient and extensible quantum chemistry package, Arrow, as well as several open-sourced generic high-performance computing libraries.

This document is dedicated to all Cornell graduate students.

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor Professor Cyrus Umrigar. His commitment to physics research is exemplary and inspiring. He led me into the field of quantum chemistry, which is an indispensable foundation of highly accurate quantum simulations. All my knowledge of quantum Monte Carlo are inherited from him, and my work on heatbath configuration interaction is impossible without his guidance and support. He taught me valuable research skills and helped me go through the process of publishing my first first-author paper in a top academic journal. I am also extremely grateful to him for always being supportive of the decisions I made, the ideas I wanted to try, and being tolerant and direct when I made mistakes. He made my six years of Ph.D. life a truly rewarding experience.

I would also like to thank Professor James Sethna. It was a pleasure to work with Jim on defects and molecular dynamics. His enthusiasm and optimism inspire me whenever I face challenging problems. I also learned a lot from his tremendous insights into physics and data analysis.

I also thank all the graduate students I worked with, especially Adam Holmes, Matt Otten, and Matt Bierbaum. They helped me get started in my research, patiently answered all my questions, and gave me valuable advice on my projects. I cannot imagine how much longer it would have taken me to get started and how many detours I might have taken without their help.

I would like to give special thanks to my parents for giving me birth and raising me. They gave me a warm family as I grew up and consistently supported me as I pursued my degree at Cornell. Thanks also to my elementary school, middle school, high school, and undergraduate teachers and friends back in China for all the invaluable lessons I learned from them, the wonderful memo-

ries I have of the time I spent with them, and all the help and encouragement I received from them.

Finally, I acknowledge the financial support from Cornell University physics department, National Science Foundation and the Air Force Office of Scientific Research, and the computing resources support from Pittsburg Computing Center, the Argonne National Lab, NERSC, and Google Cloud. Thank you, and I hope you are proud.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A major problem in the electronic structure theory is solving the many-body Schrödinger's equation accurately and efficiently since there is usually a trade-off between accuracy and efficiency. Density functional theory (DFT) [81, 36, 60] is the most popular method, and it is very efficient, but it uses approximate density functionals, and thus it is often not sufficiently accurate. Coupled cluster with single, double, and perturbative triple excitations CCSD(T) [85] is often considered as the gold standard of quantum chemistry. It is very accurate for single reference systems where a single determinant in the many-body wavefunction has a large amplitude, such as many organic molecules, but not for multireference or strongly-correlated systems. Density matrix renormalization group (DMRG) [109, 110, 22, 23, 92, 80, 90, 48] and full configuration interaction quantum Monte Carlo (FCIQMC) [16, 28, 82, 15, 53] are systematically improvable and give nearly exact solutions for small systems, but rapidly get expensive with the number of electrons and the size of the basis set.

Fast heatbath configuration interaction (SHCI) [52, 93, 54, 97, 77, 25, 65] is another systematically improvable method capable of providing essentially exact energies for small chemical systems. In common with FCIQMC, the computational cost of the method scales exponentially in the number of electrons, but SHCI is much faster than FCIQMC. The comparison with DMRG is more involved. SHCI is much faster than DMRG for small, moderately correlated systems. SHCI scales exponentially with system size with a prefactor that is typically small, but the prefactor grows with the strength of the correlation. DMRG scales exponentially with the $(D-1)/D$-th power of the system size (where D

is the system dimension) with a prefactor that is typically larger, but the prefactor is not very sensitive to the strength of the correlation. Hence it is usually the method of choice for 1- and 2-dimensional systems, and also for some 3-dimensional systems.

SHCI is an example of selected configuration interaction plus perturbation theory (SCI+PT) methods [55, 17, 40, 27, 50, 18, 58, 29, 39, 89, 44, 68, 105]. SCI+PT methods have two stages. In the first stage, a variational wavefunction is constructed iteratively, starting from a determinant that is expected to have a significant amplitude in the final wavefunction, e.g., the Hartree-Fock determinant. Each iteration of the variational stage has three steps: the selection of important determinants, construction of the Hamiltonian matrix, and iterative diagonalization of the Hamiltonian matrix. In the second stage, 2^{nd}-order perturbation theory is used to improve upon the variational energy.

The Fast SHCI algorithm has greatly improved the efficiency of both stages of SCI+PT calculations. First, it greatly speeds up the determinant selection and Hamiltonian construction by using the heat-bath selection criterion and the fast Hamiltonian construction algorithm, and, second, it drastically reduces the central processing unit (CPU) cost as well as the memory cost of performing the perturbation step by using a 3-step perturbation algorithm. With fast SHCI, we can employ two orders of magnitude larger variational wavefunctions than other SCI+PT methods and can incorporate the entire Hilbert space in the perturbative correction calculation.

I apply fast SHCI to the homogeneous electron gas (HEG). HEG is one of the most fundamental models in condensed-matter physics and is often used for benchmarking new methods. With a tunable parameter controlling the den-

sity of the electrons, HEG provides a wide range of interacting fermion systems from weakly correlated to strongly correlated. HEG is also the foundation of density functional theory (DFT). In this thesis, I focus on the HEG systems in the mid to high-density region. In this region, the basis set incompleteness error is the dominant error for most quantum chemistry methods. SHCI works well with large basis sets and thus naturally solves the basis set incompleteness problem. In our calculation, we use up to 4×10^4 orbitals, which is over an order of magnitude more than the number of orbitals used in previous calculations with other methods, such as FCIQMC.

I also apply fast SHCI to the chromium dimer system. The chromium dimer is a challenging strongly-correlated system that has been used as a benchmark molecule for a variety of methods [91, 62, 84, 71, 106, 49]. I calculate both the energy at the Cr_2 equilibrium geometry and its entire potential energy curve. For the equilibrium geometry, with 2 billion variational determinants, we achieve extremely high accuracy, which beats the accuracy of well-developed methods, such as the density matrix renormalization group (DMRG). For the potential energy curve, we include in our Hilbert space up to 28 active electrons with a cc-pVDZ basis. This Hilbert space is several orders of magnitude larger than the Hilbert space used in other systematically improvable methods.

One of the core components of our SHCI software is a generic cluster computing library, which consists of a high-performance implementation of MapReduce and several parallel data containers. MapReduce [33, 34] provides users a high-level abstraction for defining their parallel computation and takes care of the intricate low-level execution steps internally. Each MapReduce operation consists of two phases: a map phase where each input is mapped to a set of inter-

mediate key/value pairs, and a reduce phase where the pairs with the same key are reduced to a single key/value pair according to a user-specified reduce function. Many algorithms are expressible with this model, including the algorithm for finding important determinants and evaluating the perturbative correction in our SHCI method. MapReduce implementations can automatically allocate resources and execute these algorithms in parallel. Most industrial implementations focus on handling big data rather than achieving higher performance. Our library focuses on high performance, which is achieved by introducing three major improvements to the MapReduce algorithm: eager reduction, fast serialization, and special treatment for a small fixed key range. Our library achieves an order of magnitude faster performance than the popular cluster computing library, Spark, on several compute-intensive MapReduce tasks.

The use of software engineering techniques is essential to the successful development and maintenance of the SHCI software. Software engineering studies the management of the software development process in a scientific and systematic way. It helps to make the software development faster, easier, and the software is more robust and extensible. Software engineering is especially important to scientific software development because if the software has poor quality, the results may be unreliable, which can lead to wrong conclusions and mislead future research. There are many software engineering practices in the industry. I examine the most common ones and discuss their applicability in the setting of scientific software development.

CHAPTER 2

FAST HEAT-BATH CONFIGURATION INTERACTION

This chapter[1] presents in detail our fast semistochastic heat-bath configuration interaction (SHCI) method for solving the many-body Schrödinger equation. We identify and eliminate computational bottlenecks in both the variational and perturbative steps of the SHCI algorithm. We also describe the parallelization and the key data structures in our implementation, such as the distributed hash table.

2.1 Introduction

The choice of quantum chemistry methods requires a trade-off between accuracy and efficiency. Density functional theory (DFT) [81, 36, 60] methods with approximate density functionals are popular and efficient, but are often not sufficiently accurate. Coupled cluster with single, double, and perturbative triple excitations CCSD(T) [85] is very accurate for single reference systems, but not for strongly-correlated systems, such as systems with stretched bonds. Density matrix renormalization group (DMRG) [109, 110, 22, 23, 92, 80, 90, 48] and full configuration interaction quantum Monte Carlo (FCIQMC) [16, 28, 82, 15, 53] are systematically improvable but rapidly get expensive with the number of electrons and the size of the basis set.

The recently developed semistochastic heat-bath configuration interaction (SHCI) [52, 93, 54, 97, 77, 25] is another systematically improvable method capable of providing essentially exact energies for small systems. In common with

[1] This chapter was published in Ref. [66]

FCIQMC, the computational cost of the method scales exponentially in the number of electrons but with a much smaller exponent than in full configuration interation (FCI). However, SHCI is much faster than FCIQMC. The comparison with DMRG is more involved. While SHCI is much faster than DMRG for small moderately correlated systems, the ratio of costs changes in DMRG's favor as the system size increases and as the correlation strength increases, because the methods have different scaling with these parameters. In particular SHCI scales exponentially with system size with a prefactor that is typically small, but which grows with the strength of the correlation. DMRG scales exponentially with the $(D-1)/D$-th power of the system size (where D is the system dimension) with a prefactor that is typically larger, but is not very sensitive to the strength of the correlation.

SHCI is an example of the selected configuration interaction plus perturbation theory (SCI+PT) methods [55, 17, 40, 27, 50, 18, 58, 29, 39, 89, 44, 68, 105], the earliest of which being the configuration interaction by perturbatively selecting iteratively (CIPSI) method [55, 40] of Malrieu and collaborators. SCI+PT methods have two stages. In the first stage a variational wavefunction is constructed iteratively, starting from a determinant that is expected to have a significant amplitude in the final wavefunction, e.g., the Hartree-Fock determinant. Each iteration of the variational stage has three steps: selection of important determinants, construction of the Hamiltonian matrix, and iterative diagonalization of the Hamiltonian matrix. In the second stage, 2^{nd}-order perturbation theory is used to improve upon the variational energy.

The SHCI algorithm has greatly improved the efficiency of both stages. First, as discussed in Section 2.2.1, it greatly speeds up the determinant selection, and,

second, as discussed in Section 2.2.2, it drastically reduces the central processing unit (CPU) cost as well as the memory cost of performing the perturbation step by using a semistochastic algorithm. These two modifications have allowed SHCI to be used for systems as large as hexatriene in an ANO-L-pVDZ basis (32 correlated electrons in 118 orbitals) which has a Hilbert space of 10^{38} determinants [25]. SHCI has also recently been extended to (a) calculate not just the ground state but also the low-lying excited states [54], (b) perform self-consistent field orbital optimization in very large active spaces [97], and (c) include relativistic effects including the spin-orbit coupling using "one-step" calculations with two-component Hamiltonians [77]. Since SHCI has greatly reduced the time required to select determinants, we find, for large systems, that Hamiltonian construction is the most time-consuming step of the variational stage. For around 10^8 variational determinants, it takes two orders of magnitude more time to construct the Hamiltonian matrix than to select the determinants for most molecules. In addition, if a small stochastic error is required, the perturbative stage can be expensive, particularly on computer systems that do not have enough memory. Hence, in this paper, we present an improved SHCI algorithm that greatly speeds up these two steps. For the variational stage, we introduce a fast Hamiltonian construction algorithm that allows us to use two orders of magnitude more determinants in the wavefunction. For the perturbative stage, we introduce the 3-step batch perturbation method that further speeds up the calculation and reduces the memory requirement. We also describe important implementation details of the algorithm, including the key data structures and parallelization.

We organize the paper as follows: In section 2.2, we review the SHCI method. In section 2.3, we introduce our faster Hamiltonian construction al-

gorithm. In section 2.4, we introduce our 3-step batch perturbation algorithm. In section 2.5, we describe the key data structures in our implementation. In section 2.6, we describe the parallelization strategy and demonstrate its scalability. Section 2.7 concludes the paper.

2.2 SHCI Review

In this section, we review the semistochastic heat-bath configuration interaction method (SHCI) [52, 93, 54], emphasizing the two important ways it differs from other SCI+PT methods.

In the following, we use $\mathcal{V}$ for the set of variational determinants, and $\mathcal{P}$ for the set of perturbative determinants, that is, the set of determinants that are connected to the variational determinants by at least one non-zero Hamiltonian matrix element but are not present in $\mathcal{V}$.

2.2.1 Variational Stage

SHCI starts from an initial determinant and generates the variational wave function through an iterative process. At each iteration, the variational wavefunction, Ψ_V, is written as a linear combination of the determinants in the space $\mathcal{V}$

$$\Psi_V = \sum_{D_i \in \mathcal{V}} c_i \, |D_i\rangle \tag{2.1}$$

and new determinants, D_a, from the space $\mathcal{P}$ that satisfy the criterion

$$\exists \; D_i \in \mathcal{V}, \text{ such that } |H_{ai} c_i| \geq \epsilon_1 \tag{2.2}$$

are added to the $\mathcal{V}$ space, where H_{ai} is the Hamiltonian matrix element between determinants D_a and D_i, and ϵ_1 is a user-defined parameter that controls the accuracy of the variational stage [2]. (When $\epsilon_1 = 0$, the method becomes equivalent to FCI.) After adding the new determinants to $\mathcal{V}$, the Hamiltonian matrix is constructed, and diagonalized using the diagonally preconditioned Davidson method [32], to obtain an improved estimate of the lowest eigenvalue, E_V, and eigenvector, Ψ_V. This process is repeated until the change in E_V falls below a certain threshold, e.g., 1 μHa.

Other SCI methods, such as CIPSI [55, 40] use different criteria, usually based on either the first-order perturbative coefficient of the wavefunction,

$$\left|c_a^{(1)}\right| = \left|\frac{\sum_i H_{ai} c_i}{E_0 - E_a}\right| > \epsilon_1 \tag{2.3}$$

or the second-order perturbative correction to the energy.

$$-\Delta E_2 = -\frac{\left(\sum_i H_{ai} c_i\right)^2}{E_0 - E_a} > \epsilon_1. \tag{2.4}$$

The reason we choose instead the selection criterion in Eq. 2.2 is that it can be implemented very efficiently without checking the vast majority of the determinants that do not meet the criterion, by taking advantage of the fact that most of the Hamiltonian matrix elements correspond to double excitations, and their values do not depend on the determinants themselves but only on the four orbitals whose occupancies change during the double excitation. Therefore, before performing an HCI run, for each pair of spin-orbitals, the absolute values of the

[2] Since the absolute values of c_i for the most important determinants tends to go down as more determinants are included in the wavefunction, a somewhat better selection of determinants is obtained by using a larger value of ϵ_1 in the initial iterations.

Hamiltonian matrix elements obtained by doubly exciting from that pair of orbitals is computed and stored in decreasing order by magnitude, along with the corresponding pairs of orbitals the electrons would excite to. Then the double excitations that meet the criterion in Eq. 2.2 can be generated by looping over all pairs of occupied orbitals in the reference determinant, and traversing the array of sorted double-excitation matrix elements for each pair. As soon as the cutoff is reached, the loop for that pair of occupied orbitals is exited. Although the criterion in Eq. 2.2 does not include information from the diagonal elements, the HCI selection criterion is not significantly different from either of the two CIPSI-like criteria because the terms in the numerator of Eq. 2.3 span many orders of magnitude, so the sum is highly correlated with the largest-magnitude term in the sum in Eq. 2.3. It was demonstrated in Ref. [52] that the selected determinants give only slightly inferior convergence to those selected using the criterion in Eq. 2.3. This is greatly outweighed by the improved selection speed. Moreover, one could use the HCI criterion in Eq. 2.2 with a smaller value of ϵ_1 as a preselection criterion, and then select determinants using the criterion in Eq. 2.4, thereby having the benefit of both a fast selection method and a close to optimal choice of determinants.

2.2.2 Perturbative Stage

In common with most other SCI+PT methods, the perturbative correction is computed using Epstein-Nesbet perturbation theory [38, 78]. The variational wavefunction is used to define the zeroth-order Hamiltonian, H_0 and the per-

turbation, V,

$$H_0 = \sum_{i,j \in \mathcal{V}} H_{ij} |D_i\rangle\langle D_j| + \sum_{a \notin \mathcal{V}} H_{aa} |D_a\rangle\langle D_a|.$$

$$V = H - H_0. \tag{2.5}$$

The first-order energy correction is zero, and the second-order energy correction ΔE_2 is

$$\Delta E_2 \;=\; \langle \Psi_0 | V | \Psi_1 \rangle \;=\; \sum_{a \in \mathcal{P}} \frac{\left(\sum_{i \in \mathcal{V}} H_{ai} c_i\right)^2}{E_0 - E_a}, \tag{2.6}$$

where $E_a = H_{aa}$.

It is expensive to evaluate the expression in Eq. 2.6 because the outer summation includes all determinants in the space $\mathcal{P}$ and their number is $O(n^2 v^2 N_{\mathcal{V}})$, where $N_{\mathcal{V}}$ is the number of variational determinants, n is the number of electrons and v is the number of virtual orbitals. For the calculation on Cr_2, described in Section 4.2, $n = 28$, $v = 62$ and $N_{\mathcal{V}} = 2 \times 10^9$, so the number of determinants in $\mathcal{P}$ is huge. The straightforward and time-efficient approach to computing the perturbative correction requires storing the partial sum $\sum_{i \in \mathcal{V}} H_{ai} c_i$ for each a, while looping over all the determinants $i \in \mathcal{V}$. This creates a severe memory bottleneck.

Various schemes for improving the efficiency have been implemented, including only exciting from a rediagonalized array of the largest-weight determinants [40], and its efficient approximation using diagrammatic perturbation theory [27]. However, this is both more complicated than necessary (requiring a double extrapolation with respect to the two variational spaces to reach the Full CI limit) and is more computationally expensive than necessary since even the largest weight determinants have many connections that make only small con-

tributions to the energy. The SHCI algorithm instead uses two other strategies to reduce both the computational time and the storage requirement.

First, SHCI screens the sum [52] using a second threshold, ϵ_2 (where $\epsilon_2 < \epsilon_1$) as the criterion for selecting perturbative determinants $\mathcal{P}$,

$$\Delta E_2(\epsilon_2) = \sum_a \frac{\left(\sum_{D_i \in \mathcal{V}}^{(\epsilon_2)} H_{ai} c_i\right)^2}{E_V - H_{aa}} \tag{2.7}$$

where $\sum^{(\epsilon_2)}$ indicates that only terms in the sum for which $|H_{ai}c_i| \geq \epsilon_2$ are included. Similar to the variational stage, we find the connected determinants efficiently with precomputed arrays of double excitations sorted by the magnitude of their Hamiltonian matrix elements [52]. Note that the vast number of terms that do not meet this criterion are *never evaluated*.

Even with this screening, the simultaneous storage of all terms indexed by a in Eq. 2.7 can exceed computer memory when ϵ_2 is chosen small enough to obtain essentially the exact perturbation energy. The second innovation in the calculation of the SHCI perturbative correction is to overcome this memory bottleneck by evaluating this perturbative correction semistochastically [93]. The most important contributions are evaluated deterministically and the rest are sampled stochastically. The total perturbative correction is

$$\Delta E_2(\epsilon_2) = \left[\Delta E_2^{\mathrm{s}}(\epsilon_2) - \Delta E_2^{\mathrm{s}}\left(\epsilon_2^{\mathrm{d}}\right)\right] + \Delta E_2^{\mathrm{d}}\left(\epsilon_2^{\mathrm{d}}\right) \tag{2.8}$$

where ΔE_2^{d} is the deterministic perturbative correction obtained by using a larger threshold $\epsilon_2^{\mathrm{d}} \geq \epsilon_2$ in Eq. 2.7. ΔE_2^{s} is the stochastic perturbative correction from randomly selected samples of the variational determinants, and is given

by

$$\Delta E_2^{\mathrm{s}}(\epsilon_2) = \frac{1}{N_d\,(N_d-1)} \left\langle \sum_{D_a \in \mathcal{P}} \left[\left(\sum_{D_i \in \mathcal{V}}^{N_d^{\mathrm{uniq}},(\epsilon_2)} \frac{w_i c_i H_{ai}}{p_i} \right)^2 + \sum_{D_i \in \mathcal{V}}^{N_d^{\mathrm{uniq}},(\epsilon_2)} \left(\frac{w_i\,(N_d-1)}{p_i} - \frac{w_i^2}{p_i^2} \right) c_i^2 H_{ai}^2 \right] \frac{1}{E_0 - E_a} \right\rangle \tag{2.9}$$

where N_d is the number of variational determinants per sample and N_d^{uniq} is the number different determinants in a sample. p_i and w_i are the probability of selecting determinant D_i and the number of copies of that determinant in a sample, respectively. The N_d determinants are sampled from the discrete probability distribution

$$p_i \;=\; \frac{|c_i|}{\sum_j^{N_{\mathcal{V}}} |c_j|}, \tag{2.10}$$

using the Alias method [108, 61], which allows samples to be drawn in $O(1)$ time. (The more commonly used heatbath method requires $O(\log(n))$ time to do a binary search of an array of cumulative probabilities.) $\Delta E_2^{\mathrm{s}}[\epsilon_2]$ and $\Delta E_2^{\mathrm{s}}[\epsilon_2^{\mathrm{d}}]$ are calculated using the same set of samples, and thus there is significant cancellation of stochastic error. Furthermore, because these two energies are calculated simultaneously, the additional cost of performing this calculation, compared to a purely stochastic summation, is very small. Clearly, in the limit that $\epsilon_2^{\mathrm{d}} = \epsilon_2$, the entire perturbative calculation becomes deterministic.

The perturbative stage of the SHCI algorithm has the interesting feature that it achieves super-linear speedup with the number of computer nodes used. There are two reasons for this, both having to do with the increase in the total computer memory. First, a larger fraction of the perturbative energy can be computed deterministically, using a smaller value of ϵ_2^{d} in Eq. 2.8. Second, a larger value of N_d in Eq. 2.9 can be used. For a given total number of samples, the

statistical error is smaller for a small number of large samples, than for a large number of small samples, because the number of sampled contributions to the energy correction is a quadratic function of the number of sampled variational determinants. For example, N_s samples, each of size N_d, will have $N_s N_d^2$ contributions to the energy, whereas $N_s/2$ samples, each of size $2N_d$, will have $2N_s N_d^2$ contributions. Consequently, this too contributes to a super-linear speedup.

2.2.3 Other features of SHCI

We note that although SHCI has a stochastic component, it has the advantages compared to quantum Monte Carlo algorithms that there is no sign problem, and that each sample is independent. Another feature of the method is that if the calculation is done for various values of the variational threshold ϵ_1, a plot of the total energy (variational plus perturbative correction) plotted versus the perturbative correction yields a smooth curve that can be used to assess the convergence and extrapolate to the Full CI limit, $\Delta E = 0$ [54]. We typically use a quadratic fit, with the points weighted by $(\Delta E)^{-2}$ [25].

As is typical in many quantum chemistry methods, we note that the convergence of both the variational energy and the total (variational plus perturbative) energy depends on the choice of orbitals. Natural orbitals, calculated within HCI, are typically a better choice than Hartree Fock orbitals, and optimized orbitals [97] are a yet better choice. For systems with more than a few atoms, split-localized optimized orbitals lead to yet better convergence [25].

We describe, in Sections 2.3 and 2.4, improvements we have made to the variational and the perturbative stages of the SHCI algorithm, which speed up

the calculations by an order of magnitude or more for large systems.

2.3 Fast Hamiltonian Construction

The Hamiltonian matrix is stored in upper-triangular sparse matrix form. At each variational iteration, we have a set of old determinants, and a set of new determinants. We have already calculated the Hamiltonian matrix for the old determinants, and need to calculate the old-new and the new-new matrix elements.

The SHCI algorithm greatly speeds up the step of finding the important determinants and one can very quickly generate hundreds of millions or more. With this many determinants, the construction of the Hamiltonian matrix is expensive. Most of the matrix elements are zero, but finding the non-zero Hamiltonian elements quickly is challenging because the determinants in the variational wavefunction do not exhibit any pattern. (Efficient construction of the Hamiltonian matrix of the same size in FCI is much more straightforward than in SCI.) There are two straightforward "brute force" approaches to building the Hamiltonian matrix: a) looping over all pairs of determinants to find those pairs that are related by single or double excitations, and, b) generating all connections of each determinant in $\mathcal{V}$ and searching for the connections in the sorted array of variational determinants. When the number of determinants is not very large, the former is more efficient. Both of these are much too expensive for the very large number of variational determinants that we use.

The original SHCI algorithm introduced auxiliary arrays [93] to speed up the Hamiltonian construction, but it still spends considerable time on elements that

Table 2.1: The notation for the data structures in our current algorithm for efficiently constructing the Hamiltonian matrix. Analogous data structures with the alpha and beta roles reversed are also used. The text gives details of how they are constructed efficiently.

Notation	Description
$\vec{D}$	The array of determinants in $\mathcal{V}$, in the order they were generated.
$\vec{\alpha}$	The array of all alpha strings, without repeats that are present in at least one determinant in $\mathcal{V}$, in the order they were generated.
$i_\alpha(\alpha)$	Hash map that takes an alpha string, α, and returns its index, i_α, in $\vec{\alpha}$.
$\vec{i}_{D\alpha}(i_\alpha)$	The array of determinant indices in $\vec{D}$ such that the alpha strings of those determinants have index i_α in $\vec{\alpha}$. The elements of $\vec{i}_{D\alpha}(i_\alpha)$ are sorted either by their values, or by the indices of their beta strings in $\vec{\beta}$. (See text for details.)
$\vec{i}_{\beta\alpha}(i_\alpha)$	The array of all beta string indices in β that appear with α_{i_α} in a determinant. It is sorted so that the elements of $\vec{i}_{D\alpha}(i_\alpha)$ and $\vec{i}_{\beta\alpha}(i_\alpha)$ are always in correspondence.
$\vec{i}_\alpha(\alpha^{(-1)})$	Hash map that takes an alpha string with one less electron, $\alpha^{(-1)}$, and returns an array of indices of alpha strings in $\vec{\alpha}$ that can give $\alpha^{(-1)}$ upon removing an electron. These are generated only for the α's present in the new determinants.
$\vec{i}_{\alpha\alpha}(j_\alpha)$	The array of indices of alpha strings in the array $\vec{\alpha}$, connected by a single excitation to the j_α^{th} alpha string of array $\vec{\alpha}$, sorted in ascending order. These are generated only for the j_α's present in the new determinants.
$i_D, j_D, \cdots$	Indices of $\vec{D}$.
$i_\alpha, j_\alpha, \cdots$	Indices of $\vec{\alpha}$.
$\alpha(D_i)$	Alpha string of determinant D_i.

ALGORITHM 2.1: Hamiltonian matrix update for determinants connected by single or double alpha excitations. The algorithm for single or double beta excitations is very similar.

for D_i in $\vec{D}$ **do**

  Use hash map $i_\beta(\beta)$ to find i_β, the index of $\beta(D_i)$

  s is the index of the first new determinant with $s > i$.

  for j in $\vec{i}_{D\beta}(i_\beta)$ **do**

    if $j \geq s$ and D_i, D_j are connected **then**

      Compute and add H_{ij} to the Hamiltonian

    end if

  end for

end for

are zero. In our improved SHCI algorithm, we use a larger number of auxiliary arrays to further reduce the time. All the relevant data structures are shown in Table 2.1. Some of these are appended to at each variational iteration because they contain information about all the variational determinants currently included in the wavefunction, whereas others are constructed from scratch since part of their information content pertains to only the new determinants.

The auxiliary arrays are constructed by looping over just the new determinants. First, each new α encountered is appended to array $\vec{\alpha}$ and hash map $i_\alpha(\alpha)$. Also, each new determinant is appended to the arrays $\vec{i}_{D\alpha}(i_\alpha)$ and $\vec{i}_{\beta\alpha}(i_\alpha)$. In order to speed up the generation of the Hamiltonian matrix (described later) these are sorted by i_D when the number of new determinants is much smaller than the number of old determinants, and by i_β otherwise. Then, the hash map

ALGORITHM 2.2: Hamiltonian matrix update for determinants connected by an opposite-spin double excitation.

for D_i in $\vec{D}$ **do**

Use hash maps $i_\alpha(\alpha)$ and $i_\beta(\beta)$ to find i_α, i_β,

the indices of $\alpha(D_i)$ and $\beta(D_i)$.

s is the index of the first new determinant with $s > i$.

for k_α in $\vec{i}_{\alpha\alpha}(i_\alpha)$ **do**

if number of new determinants is small **then**

for $j \geq s$ in $\vec{i}_{D\alpha}(k_\alpha)$ (reverse loop) **do**

if $i_\beta(\beta(D_j)) \in \vec{i}_{\beta\beta}(i_\beta)$ (binary search) **then**

Compute and add H_{ij} to the Hamiltonian

end if

end for

else

Find the intersection $\vec{j}_\beta$ of sorted arrays

$\vec{i}_{\beta\alpha}(k_\alpha)$ and $\vec{i}_{\beta\beta}(i_\beta)$ in $O(n)$ time.

Since $\vec{i}_{\beta\alpha}(k_\alpha)$ and $\vec{i}_{D\alpha}(k_\alpha)$ are in

1-to-1 correspondence, this provides the

corresponding determinants $\vec{j}_D$

for j in $\vec{j}_D$ **do**

if $j \geq s$ **then**

Compute and add H_{ij} to the Hamiltonian

end if

end for

end if

end for

end for

$\vec{i}_\alpha(\alpha^{(-1)})$ is constructed, and finally the array $\vec{i}_{\alpha\alpha}(j_\alpha)$. The purpose of $\vec{i}_\alpha(\alpha^{(-1)})$ is simply to speed up the construction of $\vec{i}_{\alpha\alpha}(j_\alpha)$. Note that if two α strings are a single excitation apart, they will be simultaneously present under one, and only one key of the hash map $\vec{i}_\alpha(\alpha^{(-1)})$.

Then, we update the Hamiltonian matrix using these auxiliary arrays and a loop over all the determinants. Algorithms 2.1 and 2.2 describe the algorithm using pseudocode. The Hamiltonian matrix elements are nonzero only for determinants that are at most two excitations apart, namely diagonal elements, same-spin single excitations, same-spin double excitations and opposite-spin double excitations. For finding the same-spin connections, we use a method closely related to that in Ref. [89]. Finding the opposite-spin connections is more computationally expensive and our algorithm speeds this up significantly.

Same-spin excitations: For determinants connected by single or double alpha excitations to a given determinant D_i, the beta strings must be the same as $\beta(D_i)$. Hence, we simply loop over the determinants in the $\vec{i}_{D\beta}(i_\beta)$ array and check if the alpha strings are related by a single or a double excitation, and if they are, we compute that Hamiltonian matrix element. Similarly, we can find the single and double beta excitations by looping over the determinants in the $\vec{i}_{D\alpha}(i_\alpha)$ array.

Opposite-spin excitations: For the opposite-spin double excitations, we first loop over all k_α in the $\vec{i}_{\alpha\alpha}(i_\alpha)$ array, i.e., the indices of $\vec{\alpha}$ connected by single excitations to $\alpha(D_i)$. The determinants that have alpha string k_α are in $\vec{i}_{D\alpha}(k_\alpha)$, but since only some of these have beta strings that are single excitations of $\beta(D_i)$, we need to filter $\vec{i}_{D\alpha}(k_\alpha)$ to find the connected determinants. This is done in two different ways as described in Algorithm 2.2 depending on the number of new determinants. When the number of new determinants is less than 20% of the to-

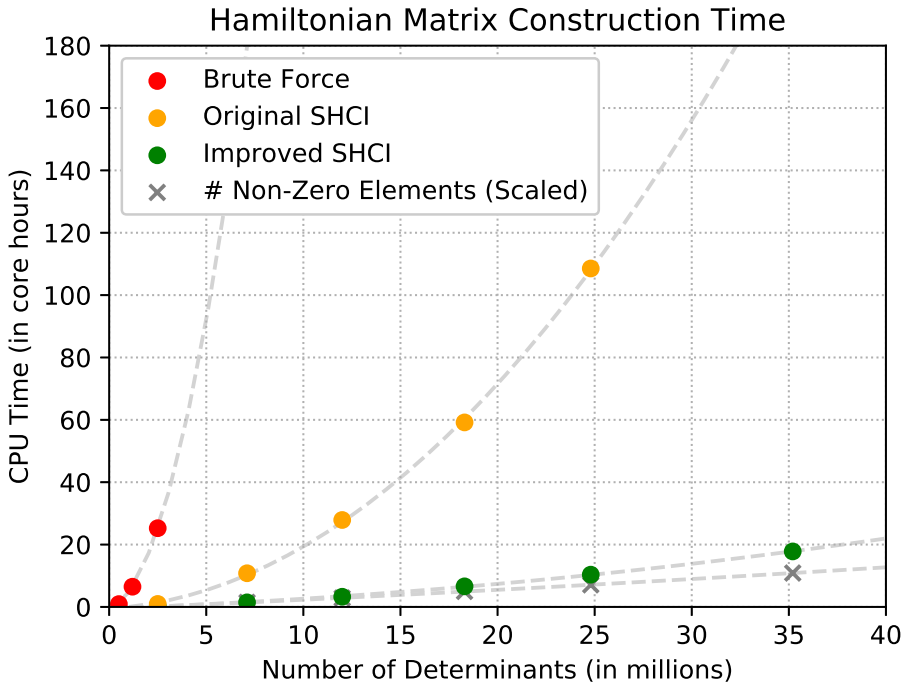

Figure 2.1: Hamiltonian matrix construction time for a copper atom in a cc-pVTZ basis. Hamiltonian construction is the performance bottleneck in the variational stage. Hamiltonian construction in our improved SHCI algorithm is an order of magnitude faster than in our original SHCI algorithm, and several orders of magnitude faster than the faster of the two brute force approaches (loop over each pair of determinants). Also shown is the number of nonzero elements in the Hamiltonian, scaled so that the first point coincides with the first point of the improved SHCI CPU time.

tal number of determinants (e.g. in the later iterations of a given ϵ_1), $\vec{i}_{D\alpha}(i_\alpha)$ and $\vec{i}_{\beta\alpha}(i_\alpha)$ are sorted by i_D, otherwise, they are sorted by i_β. The remaining determinants after filtering are the determinants connected to the given determinant through opposite-spin double excitations. Each connection is visited only once during this process, which was not the case in the original SHCI method.

In Fig 2.1, we use a copper atom with a pseudopotential [103] [3] and the cc-pVTZ basis to compare the improved SHCI algorithm to the original SHCI algorithm, and to the brute force algorithm where we loop over each pair of determinants. The improved algorithm is about an order of magnitude faster than the original SHCI for medium size calculations. For large calculations, e.g. the Cr_2 calculation described in Section 4.2 where we use billions of variational determinants, the speedup is even greater.

2.4 3-step Perturbation Energy

As described in Section 2.2, our original SHCI algorithm [93] solves the memory bottleneck problem of SCI+PT methods by introducing a semistochastic algorithm for computing the perturbative correction to the energy. In Section 2.2 we have emphasized that the statistical error can be dramatically reduced by decreasing the value of ϵ_2^{d} in Eq. 2.8, and by increasing the size of the stochastic samples, N_d in Eq. 2.9. However, decreasing ϵ_2^{d} or increasing N_d can quickly lead to very large memory requirements, making the calculations impractical even on large computers. In this situation, one is left with no choice but to run relatively inefficient calculations with larger ϵ_2^{d} and smaller N_d. For example, in Table 2.4 rows 3 and 4 show a comparison of the total CPU time of the perturbation stage for a copper atom in a cc-pVTZ basis on a machine with large memory versus on a machine with small memory. When we decrease the memory by a factor of four, the total CPU time of the original SHCI algorithm increases by almost a factor of 8.

[3] We employ the Trail-Needs pseudopotential [103], but the conclusions of this paper would be the same for an all-electron calculation.

Method	Memory	CPU Time (core hours)	Error (μHa)
HCI (deterministic)	3TB	145.0	0
Original SHCI	32GB	116.6	10
	128GB	14.5	10
Improved SHCI	32GB	4.2	9
	128GB	3.7	9
	128GB	5.9	1.8

Table 2.2: Computational cost of perturbative correction for a copper atom in a cc-pVTZ basis. The variational space has 19 million determinants for $\epsilon_1 = 5 \times 10^{-5}$ Ha and the perturbative space has 35 billion determinants for $\epsilon_2 = 10^{-7}$ Ha. HCI uses the deterministic perturbation of Ref. [52]. SHCI uses the 2-step semistochastic perturbation algorithm of Ref. [93]. Improved SHCI introduces the 3-step batch perturbation that significantly improves the efficiency of SHCI, especially for memory constrained cases. The timings for the 32GB machine are obtained by running on the same 128GB large memory machine but intentionally tuning the parameters so that the memory usage is kept below 32GB throughout the run. We also provide the timing to reach a 1.8 μHa uncertainty to illustrate that our statistical error goes down much faster than $1/\sqrt{T}$ since we use smaller $\epsilon_2^{\mathrm{dtm}}$ and $\epsilon_2^{\mathrm{psto}}$ values for smaller target errors.

For a given target error, assuming we have infinite computer memory, there is an optimal choice of ϵ_2^{d} and N_d for reaching that target error using the least computer time. Our improved algorithm is designed to have an efficiency that depends only weakly on the available computer memory. It is always more efficient than the original algorithm, especially when running on computers with small memory, in which case the gain in efficiency can be orders of magnitude. To achieve that we replace the original 2-step SHCI algorithm with a 3-step algorithm. In each of the three steps, the perturbative determinants are divided into batches using a hash function [3, 1], and the energy correction is computed either by adding, in succession, the contribution from each batch, or by estimat-

ing their sum by evaluating only a subset of these batches. The hash function maps a determinant to a 64-bit integer h. A batch contains all the determinants that satisfy $h \mod n = i$, where i is the batch index and n is the number of batches. We use a high-quality hash function which ensures a highly-uniform mapping, so each batch has about the same number of determinants, i.e., the fluctuations in the number of determinants in the various batches is the square root of the average number of determinants in each batch. The contributions of the various batches fluctuate both because the contributions of the perturbative determinants within a batch fluctuate and the number of perturbative determinants in a batch fluctuate. For both contributions, the ratio of the fluctuation to the expected value is $\sim \sqrt{N}/N \to 0$ for large N, where N is the average number of determinants in a batch.

In brief, our improved SHCI algorithm has the following 3 steps:

1. A deterministic step with cutoff $\epsilon_2^{\text{dtm}} (< \epsilon_1)$, wherein all the variational determinants are used, and all the perturbative batches are summed over.
2. A "pseudo-stochastic" step, with cutoff $\epsilon_2^{\text{psto}} (< \epsilon_2^{\text{dtm}})$, wherein all the variational determinants are used, and typically only a small fraction of the perturbative batches need be summed over to achieve an error much smaller than the target error.
3. A stochastic step, with cutoff $\epsilon_2 (< \epsilon_2^{\text{psto}})$, wherein a few stochastic samples of variational determinants, each consisting of N_d determinants, are sampled using Eq. 2.10 and only one of the perturbative batches is randomly selected per variational sample.

The total perturbative correction is

$$\begin{aligned}\Delta E_2(\epsilon_2) &= \left[\Delta E_2^{\text{sto}}(\epsilon_2) - \Delta E_2^{\text{sto}}\left(\epsilon_2^{\text{psto}}\right)\right] \\ &+ \left[\Delta E_2^{\text{psto}}\left(\epsilon_2^{\text{psto}}\right) - \Delta E_2^{\text{psto}}\left(\epsilon_2^{\text{dtm}}\right)\right] \\ &+ \Delta E_2^{\text{dtm}}\left(\epsilon_2^{\text{dtm}}\right) \end{aligned} \tag{2.11}$$

The choice of these parameters depends on the system and the desired statistical error, but reaonable choices for a target error around 10^{-5} Ha are $\epsilon_2^{\text{dtm}} = 2 \times 10^{-6}$ Ha, $\epsilon_2^{\text{psto}} = 10^{-7}$ Ha, and, $\epsilon_2 = \epsilon_1/10^6$. Of course, if $\epsilon_1 \leq \epsilon_2^{\text{dtm}}$ the deterministic step is skipped. We next describe each of the 3 steps in detail.

The first step is a deterministic step similar to the original SHCI's deterministic step, except that when there is not enough memory to afford the chosen ϵ_2^{dtm}, we divide the perturbative space into batches according to the hash value of the perturbative determinants and evaluate their contributions batch by batch. The total deterministic correction is simply the sum of the corrections from all the batches

$$\Delta E_2\left(\epsilon_2^{\text{dtm}}\right) = \sum_B \sum_{\substack{D_a \in \mathcal{P} \\ h(D_a) \in B}} \frac{\left(\sum_{D_i \in \mathcal{V}}^{(\epsilon_2^{\text{dtm}})} H_{ai} c_i\right)^2}{E_V - H_{aa}} \tag{2.12}$$

where $h(D)$ is the hash function and B is the hash value space for a batch. This method solves the memory bottleneck in a different way than the original SHCI algorithm. We could do the full calculation in this way, i.e., use a very small value for ϵ_2^{dtm} and a large number of batches, but it is much more efficient to only evaluate the large contributions here and leave the huge number of small contributions to the later stochastic steps.

The second step is a pseudo-stochastic step. It is similar to the deterministic step, except for the following differences: a) we use an ϵ_2^{psto} much smaller than

$\epsilon_2^{\mathrm{dtm}}$ as the selection criterion, b) we divide the perturbative space into as many batches as is needed in order for one batch to fit in memory, with the constraint that there are at least 16 batches, c) we use the corrections from the perturbative determinants in a small subset of the batches (often one is enough) to estimate the total correction from all the perturbative determinants, as well as its standard error. Looping over batches, for each batch, we calculate the correction from each unique perturbative determinant in that batch. We accumulate the number of unique determinants, the sum and the sum of squares of the corrections from these determinants. At the end of each batch iteration, we calculate the mean and standard deviation of the corrections from all the evaluated perturbative determinants and use these to estimate the total correction from all the perturbative determinants. Note that the standard deviation of the total correction is the standard deviation of the sum of only the unevaluated determinants. If we process all the batches, the pseudo-stochastic step becomes deterministic and has zero standard deviation. When the standard deviation of the total correction is smaller than 40% of the target error, we exit the loop over batches. However, a single batch is often sufficient to reach a statistical error below that threshold, for the smallest ϵ_1 values that we typically use.

The third step is a stochastic step that is similar to the stochastic step of the original SHCI algorithm, except that instead of keeping all the perturbative determinants that satisfy the ϵ_2 criterion we keep only one randomly selected batch out of several. The available computer memory constrains the number of perturbative determinants, and one can obtain the same number sampling a certain number of variational determinants and all the perturbative determinants that satisfy the ϵ_2 criterion (the original SHCI algorithm), or, by using a larger number of variational determinants and selecting just one batch of the pertur-

bative determinants. The latter allows us to use much larger variational samples. Using larger variational samples is advantageous because we find that the additional fluctuations due to sampling the perturbative determinants is much smaller than the reduction in the fluctuations due to having larger variational samples. Typically, we use $\epsilon_2 = 10^{-6}\epsilon_1$. Since the smallest ϵ_1 that we use is typically around 10^{-5} Ha, this value is in fact much smaller than is needed to ensure that the perturbative correction is fully converged. For a statistical error of 10^{-5} Ha, 128 batches is usually a good choice to start with. The size of the variational sample is chosen so that a single perturbative batch fits in the available memory. We use a minimum of 10 samples in our stochastic step in order to get a meaningful estimate of the uncertainty. On large memory machines, we often achieve a much smaller statistical error than the target with 10 samples. In that case, we can decrease the size of the variational sample in later runs for similar systems.

In Table 2.4, the last four rows compare the original SHCI to the improved version with 3-step batch perturbation. In the memory-constrained case, the improved SHCI runs more than an order of magnitude faster than the original SHCI. Even when memory is abundant, the improved SHCI is still a few times faster.

The main reasons that the improved SHCI is much faster are: (1) It computes a larger fraction of the perturbative correction in the deterministic step. (2) A small fraction of the batches in the pseudo-stochastic step is usually sufficient to give an accurate estimate of the total correction. (3) It uses much larger samples of variational determinants in the stochastic step.

We now comment on a couple of aspects of our algorithm that may not be

obvious:

1) The value of the perturbative correction depends only on ϵ_2 and not ϵ_2^{dtm} and ϵ_2^{psto}. The latter two quantities affect only the efficiency of the calculation. By using batches in the stochastic step, we can use a much smaller ϵ_2^{sto} and thereby include almost the entire perturbative space. In our calculations, we usually set $\epsilon_2^{\text{sto}} = 10^{-6}\epsilon_1$, which is much smaller than is possible using our previous 2-step perturbation method, and much smaller than necessary to keep the systematic error within the target statistical error.

2) In the pseudo-stochastic step, we estimate the fluctuations of the unevaluated perturbative determinants from the fluctuation of the evaluated perturbative determinants. This relies on having a sufficiently uniform hash function. Note that since we are using all the variational determinants in this step, the fluctuations come just from the perturbative determinants. In contrast, in the stochastic step, the fluctuations come both from the choice of variational determinants and the choice of batches. In that case, one cannot simply use the standard deviation of the corrections from the evaluated perturbative determinants to estimate the standard deviation of the total correction. So, in the stochastic step we use a minimum of 10 samples, calculate the correction from each of these samples, and use the standard deviation of these sample corrections to estimate the standard deviation of the total correction.

3) In the stochastic step, the fluctuation between batches of perturbative determinants is much smaller than the fluctuation between samples of variational determinants. The reason for this is that there are many more perturbative determinants in a batch (each making only a small contribution) than there are variational determinants in a sample. Further, the variational determinants

vary greatly in importance. This is why we use importance sampling as described by Eq. 2.10 when selecting variational determinants, and why we precede the stochastic step with the deterministic and pseudo-stochastic steps, but even with these improvements the fluctuations from the choice of variational samples is much larger than the fluctuation from the choice of batches. Hence, we use only one randomly selected batch of perturbative determinants (typically out of 128 batches) per variational sample.

4) The use of batches carries a small computational overhead of having to regenerate the perturbative determinants for each batch. Using our method, generating determinants is sufficiently fast that the increase in computational cost would be substantial only if this is done many times. If we employed a purely deterministic algorithm, the number of batches would be very large, but with our 3-step semistochastic algorithm the number of batches actually computed is sufficiently small in each of the three steps that there is never a large computational overhead.

Finally, we comment on two other algorithms that have been recently been proposed for calculating the perturbative correction. First, another very efficient semistochastic algorithm has been proposed by Garniron et al. [44]. However, that algorithm has, for each perturbative determinant, a loop over the variational determinants to find those that are connected. For the very large number of variational determinants that we employ here (up to 2×10^9) this is impractical. To avoid confusion, we should mention that the reason that their energy for Cr_2 is very different from ours is that they used a nonrelativistic Hamiltonian. Second, another algorithm that uses batches of perturbative determinants to overcome the memory bottleneck has been proposed very recently [105]. It

is an efficient deterministic algorithm for memory constrained environments, but for a reasonable statistical error tolerance, e.g., 10^{-5} Ha, a semistochastic approach is usually much faster, as we can see from Table 2.4. Also, in our Cr_2 calculation, we stochastically estimate the perturbative correction from at least trillions of perturbative determinants, for $\epsilon_2 = 3 \times 10^{-12}$ Ha, which probably involves quadrillions of contributions ($n^2v^2N_{\mathcal{V}} = 9 \times 10^{15}$), which is infeasible with a deterministic algorithm.

2.5 Key Data Structures

In this section, we discuss three key data structures used to store the determinants, the distributed Hamiltonian matrix, and, the distributed partial sums in the perturbative stage of the calculation.

2.5.1 Determinants

We use two different representations of determinants. For storing and accessing determinants locally in memory, we use arrays of bit-packed 64-bit unsigned integers. Each bit represents a spin-orbital. The n-th orbital is represented by the (n mod 64)-th bit of the `(n / 64)`-th integer, where "`/`" means integer (Euclidean) division and the counting starts from zero. (n mod 64) can be implemented as `(n & 63)`, and `(n / 64)` can be implemented as `(n >> 6)`, where "`&`" is the bitwise "and" and "`>>`" is the bitwise right shift. Both operations cost only one clock cycle on modern CPUs.

For transferring the determinants to other nodes or saving them to disk, we

use base-128 variable-length integers (VarInts) [98] to compress the 64-bit integers. VarInts take only a few bit operations to compute and reduce the memory footprint by up to 87.5% for small integers, which reduces the network traffic and the size of the wavefunction files considerably, especially for large basis sets.

2.5.2 Hamiltonian Matrix

We store only the upper triangle of the Hamiltonian matrix. The rows are distributed to each node in a round-robin fashion: the first row goes to the first node, the second row goes to the second node, and when we reach the end of the node array, we loop back and start from the first node again. Each row is a sparse vector, represented by two arrays, one stores the indices of the nonzero elements and the other stores the values.

During the matrix-vector multiplication, each node will apply its own portion of the Hamiltonian to the vector to get a partial resulting vector. The partial results are then merged together using a binomial tree reduction. The work on each node is distributed to the cores with dynamic load balancing. To save space, we store only one copy of the partial resulting vector on each node and each thread updates that vector with hardware atomic operations. In addition, we cache the diagonal of the matrix on each node to speed up the Davidson diagonalization [32].

2.5.3 Partial Sums

In the perturbative stage, we loop over the variational determinants $\{D_i\}$ to compute the partial sum $\sum_i H_{ai} c_i$ for each perturbative determinant D_a. The map from D_a to $\sum_i H_{ai} c_i$, is stored in a distributed hash table [2]. This choice is dictated by the enormous number of perturbative determinants we employ. The time complexity of inserting one element into the hash table is $O(1)$, while for a sorted array it is $O(\log(n))$. For large calculations, the prefactor from using hash tables is small compared to the $\log(n)$ cost from using a sorted array.

The distributed hash table is based on lock-free [5] open-addressing [6] linear-probing [4] concurrent hash tables [24] specifically designed for intensive commutative insertion and update operations. The linear-probing technique for conflict resolution has better efficiency than separate chaining during parallel insertion, and the lock-free implementation allows all the threads to almost always operate at their full speed.

On each node, we have n of these concurrent hash tables, where n is the number of nodes. One of them stores the entries belonging to that node, and the other $(n - 1)$ tables store the entries belonging to other nodes pending synchronization. Each concurrent hash table is implemented as lots of segments (at least four times the number of hardware threads) and each segment can be modified by only one thread at a time. When a thread wants to insert or update a (key, value) pair, it first checks whether the segment that the key belongs to is being used by other threads. If the segment is being used, the thread will insert or update the entry to a thread-local hash table, which will be merged to the main table later periodically. We can do this because the insertion and the update operations of the partial sums are commutative. Hence, each insertion

and update is guaranteed to finish within $O(1)$ time without getting blocked, even for perturbative determinants with lots of connections to the reference determinants. The inter-node synchronization runs periodically so that most of the perturbative determinants will have only one copy during the entire run on the entire cluster, except for those with lots of connections to the reference determinants.

2.6 Parallelization

All the critical parts of SHCI are parallelized with MPI+OpenMP. This section describes the parallelization and the scalability of each part.

When finding the connected determinants, performing the matrix-vector multiplication during the diagonalization, and constructing the Hamiltonian matrix from the auxiliary arrays, we use the round-robin scheme to distribute the load across the nodes and use dynamic load balancing for all the cores on the same node.

We parallelize the construction of the α-singles, $\vec{i}_{\alpha\alpha}(j_\alpha)$, and β-singles, $\vec{i}_{\beta\beta}(j_\beta)$, arrays on each node, which is the most time-consuming part of constructing the auxiliary arrays. For each entry of $\vec{i}_{\alpha\alpha}$ and $\vec{i}_{\beta\beta}$, we initialize a lock to ensure exclusive modification. We loop over all the $\vec{i}_\alpha(\alpha^{(-1)})$ arrays and for each (i_α, j_α) pair (which are one excitation away) inside a particular $\vec{i}_\alpha(\alpha^{(-1)})$ array, we lock and append j_α to $\vec{i}_{\alpha\alpha}(i_\alpha)$, and we lock and append i_α to $\vec{i}_{\alpha\alpha}(j_\alpha)$. When both i_α and j_α occur only in the new determinants, the smaller of the two does both appends.

In Figs. 2.2 and 2.3, we demonstrate the parallel scalability of our SHCI implementation when applied to a copper atom in a cc-pVTZ basis. We use up to 16 nodes, and each node has 6 cores.

For the variational part, our implementation scales almost linearly up to 4 nodes. At 16 nodes we have 75% parallel efficiency.

For the perturbative stage, two major factors determine the speedup. One is the additional communication associated with shuffling perturbative determinants across the nodes, which increases with the number of nodes. The other is the speedup from having more cores. We can see from Fig. 2.3 that from 1 to 4 nodes, the first factor dominates and there is significant deviation from ideal speedup. Starting from 8 nodes, we have to shuffle almost all the perturbative determinants from the spawning node to the storage node that each determinant belongs to, so there is little change in the first factor and the second factor starts to dominate, pushing the speedup curve upward and producing almost perfect scaling. Note that in the original SHCI algorithm [93], there is a superlinear speedup from using more nodes because many stochastic samples are needed when running in a memory-constrained environment. Here we have solved this problem with the 3-step batch perturbation, for which the number of stochastic samples is almost always 10. (We require a minimum of 10 samples in order to have a reasonable estimate of the stochastic error.) Consequently, on memory constrained environments, we achieve a few orders of magnitude speedup.

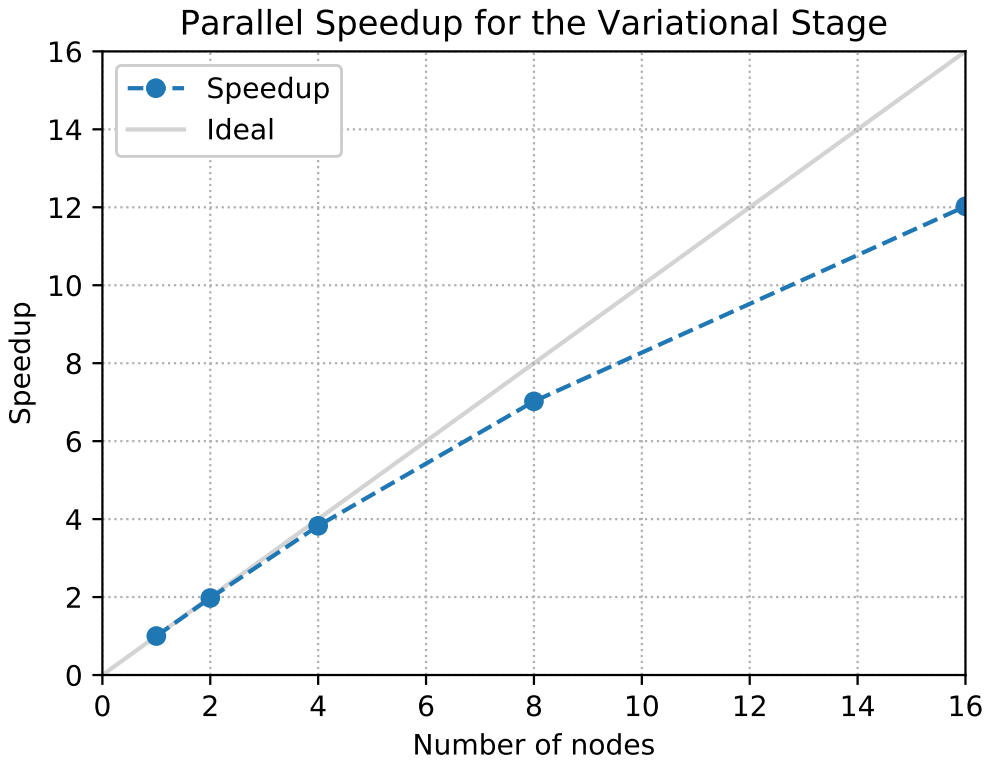

Figure 2.2: Parallel speedup of the variational stage for a copper atom in a cc-pVTZ basis. There is almost perfect scaling for up to 4 nodes and 75% parallel efficiency at 16 nodes.

2.7 Conclusion

In this paper, we introduced our fast semistochastic heat-bath configuration interaction algorithm, an efficient and essentially exact algorithm for estimating the Full-CI energy. We introduced a new Hamiltonian generation algorithm and a 3-step batch perturbation algorithm to overcome the bottlenecks in the original SHCI algorithm. We also presented the key data structures and parallelization strategy, which are also crucial to the performance. These improvements allowed us to use 2×10^9 variational determinants, which is more than one order of magnitude larger than the 9×10^7 determinants used in our earlier

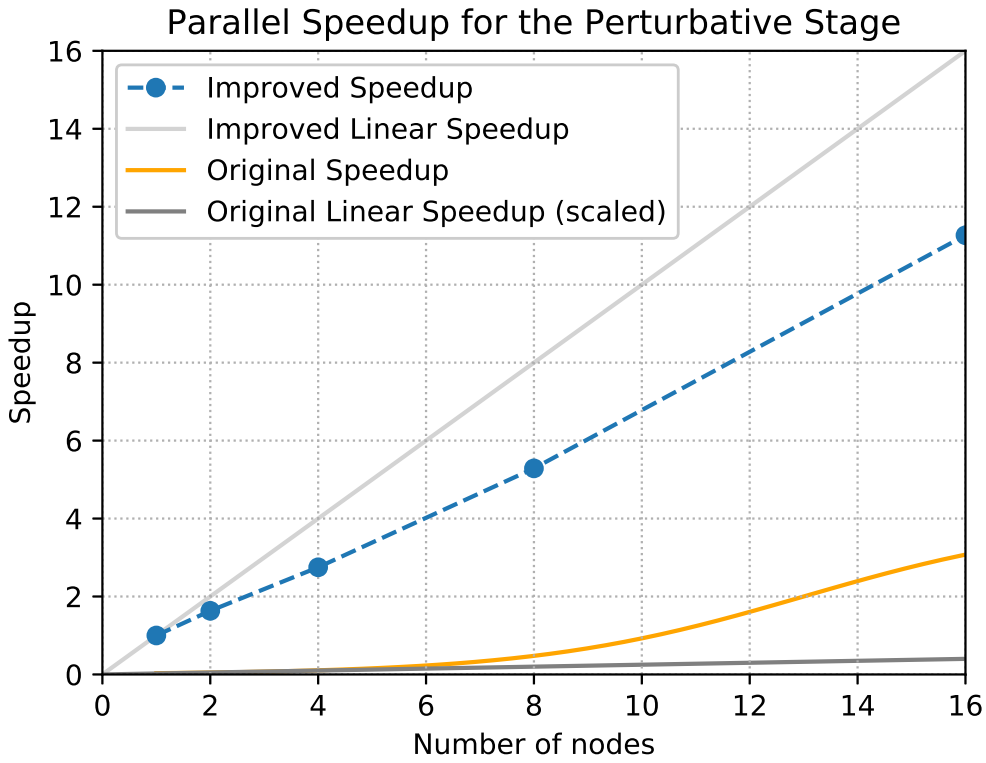

Figure 2.3: Parallel speedup for improved SHCI compared to the original SHCI for the perturbative stage of the calculation for a copper atom in a cc-pVTZ basis. From 1 node to 4 nodes, we see a significant deviation from linear speedup due to the additional communication from shuffling the perturbative determinants across nodes. Starting from 8 nodes, the number of shuffles approaches a constant and we can see an almost linear speedup from using more processors. The estimate of the speedup of the original SHCI is based on the assumption that the total memory of 10 nodes is enough to support the optimal choice of ϵ_2^{dtm} and N_d, in Eqs. 2.8 and 2.9. The "Original" curves are scaled to reflect the relative speed of original SHCI algorithm to that of the improved algorithm.

SHCI calculation [25], and two orders of magnitude larger than the largest variational space of 2×10^7 determinants employed to date in any other selected CI method [44].

Future extensions of the method include going to yet larger variational spaces using a *direct* method [59, 56], wherein the Hamiltonian matrix is recalculated at each Davidson iteration and therefore need not be stored. Although this increases the computational cost, the increase is not overwhelming because of the use of the auxiliary arrays introduced in this paper. Other extensions include increasing the range of applicability of the method to larger systems by combining SHCI with range-separated density functional theory [88], and the use of SHCI as an impurity solver in embedding theories. Recently, a selected coupled cluster method has been developed [115]. Although the current version has only been used with small basis sets, it is possible that with further development this will become a highly competitive method, especially for weakly correlated systems.

Possible applications of the SHCI method include providing benchmark energies for a variety of organic molecules, as well as for transition metal atoms, dimers, and monoxides, and calibration or training data for large scale methods, e.g., to calibrate interatomic potentials for molecular dynamics and exchange-correlation functionals for density functional theory, and to train machine learning based quantum chemistry solvers. Calculations on the homogeneous electron gas are also underway.

CHAPTER 3

HOMOGENEOUS ELECTRON GAS SIMULATION WITH FAST SHCI

We apply the recently developed semistochastic heatbath configuration interaction (SHCI) method to the homogeneous electron gas (HEG) in the mid-to-high density regime. In this regime, the basis-set incompleteness error is the dominant error for basis set methods so we extend the SHCI method to handle large basis sets. To obtain highly accurate results, we use up to 39,886 orbitals, which is more than an order of magnitude larger than the number used in previous state-of-the-art calculations using methods that yield essentially exact results within a basis set. This enables accurate extrapolation to the complete basis set limit. We calculate HEG correlation energies for the 14-electron supercell with Wigner-Seitz radius r_s ranging from 0.5 to 5.0, and for the 54-electron supercell with $r_s = 0.5$ and compare our results with earlier calculations.

3.1 Introduction

The homogeneous electron gas (HEG) is one of the most fundamental models in condensed-matter physics. With a tunable parameter, the Wigner-Seitz radius r_s, controlling the density of the electrons, the HEG provides a simple paradigm for studying interacting fermion problems ranging from weakly correlated to strongly correlated. In addition, the HEG is a cornerstone of density functional theory (DFT). The exchange-correlation energy of the HEG is the entire exchange-correlation energy in the local density approximation (LDA) of DFT, and it is an important component of more accurate exchange-correlation functionals [81, 36].

The LDA is usually obtained from a fit to the diffusion Monte Carlo (DMC) exchange-correlation energy per electron, ϵ_{xc}, of the HEG with various r_s values [20]. The DMC random walk is performed in real space (the coordinates of the electrons) and consequently the computed energies are directly in the infinite-basis limit. However, in order to control the Fermion sign problem, DMC is usually performed with the fixed-node approximation and consequently, the energies have a fixed-node error which depends on the quality of the trial wavefunctions. Backflow wavefunctions have been used to reduce the error [63, 86] but the magnitude of the remaining error is unclear.

In order to assess the magnitude of the DMC fixed-node error, other quantum Monte Carlo (QMC) methods have been applied to the HEG. In contrast to DMC, the random walk in the full configuration interaction quantum Monte Carlo (FCIQMC) method [16, 28, 94, 94] is in the space of occupation numbers of a finite number of orbitals. The finite basis set energies need to be extrapolated to obtain the complete basis set (CBS) energies. The Fermion sign problem in FCIQMC is controlled by having walker cancellations and by making an *initiator approximation*. The initiator error disappears in the infinite walker population limit, and for many systems it is practical to use a sufficiently large population to make the initiator error negligible. In the mid-to-high density regime, the basis set incompleteness error is the dominant error. For example, for the HEG with $r_s = 0.5$ and 14 electrons in each periodic cell, the two largest FCIQMC calculations, which use 778 and 1850 orbitals yield correlation energies $-0.5893(3)$ Ha and $-0.5936(3)$ Ha respectively, a 4.3 mHa difference, which is larger than the estimated initiator and statistical errors.

The coupled cluster Monte Carlo (CCMC) method [79] is closely related to

FCIQMC but yields coupled cluster energies in the infinite walker limit. It has been used [79] to compute the correlation energy of the HEG with a 14-electron periodic cell for r_s in the range $0.5 - 5.0\ a_0$ with excitation orders up to five, i.e., CCSDTQ5, and 358 orbitals. As expected, high excitation orders are necessary to get an agreement with FCIQMC at the larger r_s values.

The method used in this paper is a modified version of the recently proposed heat-bath configuration interaction (SHCI) method [52, 93, 65]. SHCI is a selected configuration interaction plus perturbation method (SCI+PT) method. It differs from SCI+PT methods in that it uses precomputed double excitation lists to avoid ever looking at determinants that do not satisfy the threshold for contributing to the variational wavefunction or the perturbative correction [52], and it evaluates the perturbative correction semistochastically to avoid a memory bottleneck [93]. Recently, SHCI has been sped up by another order of magnitude by introducing a fast Hamiltonian generation algorithm and replacing the 2-step semistochastic perturbative correction by a 3-step algorithm [66].

This paper adapts the SHCI algorithm to the homogeneous electron gas problem (HEG). We introduce a more space-efficient data structure for representing determinants when using very large basis sets. In addition, we optimize the storage and the usage of the double excitation lists, taking into consideration the momentum conservation property of the plane-wave orbitals. Finally, we apply the revised algorithm to the HEG with up to 39,886 orbitals, which is more than an order of magnitude larger than previously used in methods that yield essentially exact energies within a basis, in order to get accurate extrapolated energies in the complete basis limit.

We organize this chapter as follows: In section 3.2, we formulate the HEG

problem and give the Hamiltonian of HEG in a periodic boundary condition with a plane wave basis. In section 3.3, we revise SHCI for the homogeneous electron gas problem (HEG). In section 3.4, we use the revised algorithm to obtain accurate HEG results in the mid-to-high density regime. Section 3.5 concludes the paper.

3.2 Homogeneous Electron Gas Hamiltonian

We consider a system of $N_\uparrow$ spin-up electrons and $N_\downarrow$ spin-down electrons in a d-dimensional hypercube of side length L with periodic boundary conditions in all directions and a uniform positive background such that the whole system is neutral.

The Hamiltonian of this system can be expressed in terms of the Yukawa potential $V(\mathbf{r};\kappa) = \frac{e^{-\kappa r}}{r}$

$$\hat{H} = \sum_{i=1}^{N} -\frac{1}{2}\nabla_i^2 + \frac{1}{2}\sum_{i\neq j}^{N}\frac{1}{L^d}\sum_{\mathbf{q}\neq 0} V(\mathbf{q})e^{i\mathbf{q}\cdot(\mathbf{r}_i-\mathbf{r}_j)}$$

where $N = N_\uparrow + N_\downarrow$, $\mathbf{q} = \frac{2\pi}{L}(n_1, n_2, \cdots, n_d)$, $n_i \in \mathbb{Z}$. And $V(\mathbf{q})$ is the Fourier transform of the Yukawa potential at $\kappa \to 0$. For $d = 3$, $V(\mathbf{q}) = \frac{4\pi}{q^2}$. [45]

To convert the Hamiltonian into its second quantization form

$$\hat{H} = \sum_{PQ} f_{PQ} a_P^\dagger a_Q + \frac{1}{2}\sum_{PQRS} g_{PQRS} a_P^\dagger a_Q^\dagger a_R a_S \tag{3.1}$$

we use the planewave basis set

$$\phi_P(x) = \frac{1}{\sqrt{L^d}} e^{-i\mathbf{k}_P\cdot\mathbf{r}}\sigma_P(m_s) \tag{3.2}$$

where $\sigma_P(m_s)$ is the spin eigenfunction and $\mathbf{k}_P = \frac{2\pi}{L}(n_{P_1}, n_{P_2}, \cdots, n_{P_d})$, $n_{P_i} \in \mathbb{Z}$.

After simplification, we can get the coefficients of the second quantization terms

$$
\begin{aligned}
f_{PQ} =& \frac{\mathbf{k}_Q^2}{2}\delta_{\sigma_P\sigma_Q}\delta_{\mathbf{k}_P\mathbf{k}_Q} \\
g_{PQRS} =& \frac{1}{L^d}\delta_{\sigma_P\sigma_R}\delta_{\sigma_Q\sigma_S}V(\mathbf{k}_{PR}) \\
& (1-\delta_{\mathbf{k}_R\mathbf{k}_P})\delta_{\mathbf{k}_R+\mathbf{k}_S,\mathbf{k}_P+\mathbf{k}_Q}
\end{aligned}
$$

Hence, the Hamiltonian matrix elements between a pair of Slater determinants are

$$
\begin{aligned}
\langle i|\hat{H}|i\rangle =& \sum_P i_P\frac{\mathbf{k}_P^2}{2} - \frac{1}{2L^d}\sum_{P\neq Q} i_P i_Q \delta_{\sigma_P\sigma_Q}V(\mathbf{k}_{QP}) \\
\langle i_1|\hat{H}|i_2\rangle =& \Gamma_I^{i_1}\Gamma_J^{i_1}\Gamma_K^{i_2}\Gamma_L^{i_2}\delta_{\mathbf{k}_K+\mathbf{k}_L,\mathbf{k}_I+\mathbf{k}_J}\frac{1}{L^d} \\
& [\delta_{\sigma_I\sigma_K}\delta_{\sigma_J\sigma_L}V(\mathbf{k}_{IK}) - \delta_{\sigma_I\sigma_L}\delta_{\sigma_J\sigma_K}V(\mathbf{k}_{IL})]
\end{aligned}
$$

where $i_P = 1$ if and only if orbital P is occupied, $\Gamma_P^i = \sum_{l=1}^{P-1}(-1)^{i_l}$. $|i_1\rangle$ has orbitals I, J occupied while $|i_2\rangle$ has orbitals K, L occupied, $I < J$, $K < L$, and all the other orbitals of $|i_1\rangle$ and $|i_2\rangle$ are the same.

3.3 Revising SHCI for Homogeneous Electron Gas

In this section, we describe how we adapt our SHCI algorithm (previously used for chemical systems) to treat the HEG at mid to high densities. The adaptations are necessitated by the use of a plane wave basis and the large number of orbitals required for an accurate extrapolation to the complete basis limit. Appendix 3.2 gives the Hamiltonian matrix elements of HEG.

3.3.1 Large Basis Set

The usual representation of determinants involves using bit-packing, i.e., one bits denote occupied orbitals and zero bits denote unoccupied orbitals. This becomes inefficient when the number of orbitals is much larger than the number of electrons. To reduce memory and time requirements when using a large basis set, we introduce a hybrid representation of the determinants. First, the orbitals are sorted into descending order of importance (occupancy). Then, bit-packing is used to represent the occupancy of the first few orbitals, and a self-balancing binary search tree [111], namely a red-black tree [112], is used to store the indices of the remaining orbitals. Fig. 3.1 illustrates this hybrid structure.

The red-black tree is a common data structure for storing a set of distinct objects, which in our case are the indices of the occupied orbitals. An excitation corresponds to a deletion of one or two occupied orbitals and insertion of one or two unoccupied orbitals. The worst case time complexity for this operation on a red-black tree (including the cost of rebalancing the tree) is $O(\log N)$. Another common operation on determinants is going through the occupied orbitals in order, which corresponds to an inorder traversal [113] of the tree. The time complexity of this traversal is $O(N)$. This hybrid structure gives us high performance and compact storage when there are a small number of important orbitals and a large number of unimportant orbitals.

3.3.2 Orbital Momentum Conservation

We employ a plane-wave basis set and periodic boundary conditions. Since the orbitals are momentum eigenstates, momentum conservation can be used

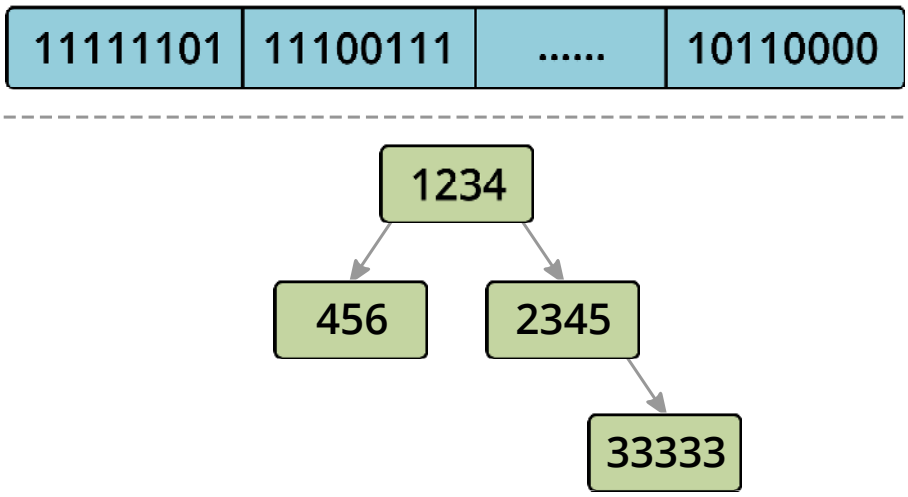

Figure 3.1: Hybrid Representation for Determinants. The occupancy of the most important orbitals is represented with bit-packing. The indices of the other occupied orbitals, which are 456, 1234, 2345, and 33333 in this case, are stored in a self-balancing binary search tree [112].

to reduce the storage of the HCI double excitation helper lists.

First, we find all the possible differences between the momenta of two orbitals. Let M be the number of orbitals ($\mathbf{k}$ points), then the number of distinct differences between them is also of order $O(M)$. For an orbital i, we denote the momentum of that orbital to be $\mathbf{k}_i$. We use p and q to denote the pair of occupied orbitals during excitation, and r and s to denote the pair of unoccupied orbitals to which the electrons are excited. Then we have $\mathbf{k}_p + \mathbf{k}_q = \mathbf{k}_r + \mathbf{k}_s$. For HEG, as shown in Appendix 3.2, $\mathbf{k}_p - \mathbf{k}_q$ and $\mathbf{k}_p - \mathbf{k}_r$ uniquely determines the magnitude of the Hamiltonian matrix element associated with excitation $pq \to rs$. Hence, for each possible momentum difference $\mathbf{k}_{pq}$, we associate with it a list of $\langle \mathbf{k}_{pr}, |H_{pqrs}| \rangle$ pairs in descending order of $|H_{pqrs}|$. Fig. 3.2 illustrates the structure of these helper lists. Finally, when using these helper lists to find important connected

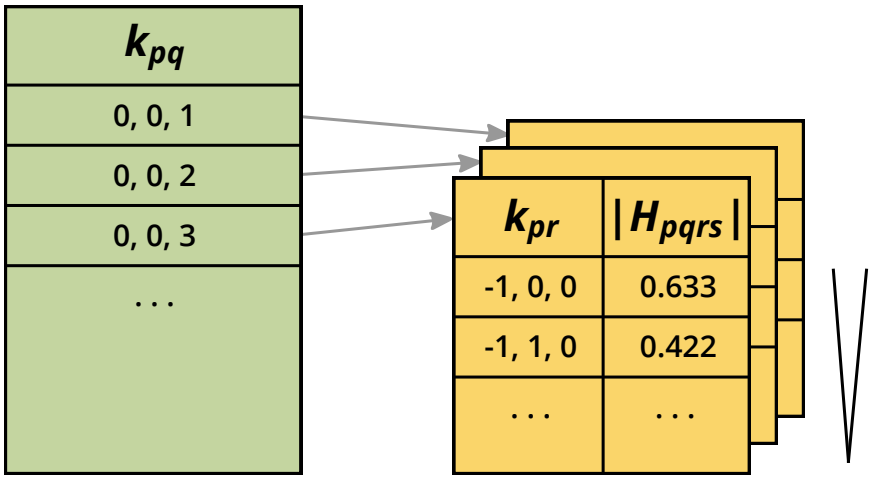

Figure 3.2: SHCI Helper Lists for HEG. For each $\mathbf{k}_{pq}$, we generate a list of $\langle \mathbf{k}_{pr}, |H_{pqrs}| \rangle$ pairs and sort them in descending order of $|H_{pqrs}|$. When trying to find connected determinants from a given spawning determinant, we go through each occupied pair of orbitals p and q, calculate their momentum difference $\mathbf{k}_{pq}$, and go through the corresponding list until $|H_{pqrs}|$ falls below a certain threshold. For each entry that we go through in the list, we can obtain r and s by using $\mathbf{k}_{pr}$ and momentum conservation.

determinants to a given reference determinant, we perform the following for each pair of occupied orbitals p and q: we go through the list associated with $\mathbf{k}_{pq} = \mathbf{k}_p - \mathbf{k}_q$ to get $\mathbf{k}_{pr}$ until the corresponding $|H_{pqrs}|$ falls below a given threshold. Using $\mathbf{k}_{pr}$ and momentum conservation, we can easily get r and s, and thus the connected determinant which satisfies the SHCI criteria.

The storage complexity of these helper lists is $O(M^2)$, as opposed to $O(M^4)$ for chemistry systems. The time complexity of finding determinants connected to a given determinant in descending order of importance is the same as in chemistry, which is $O(n_e^2 + n_D)$, where n_e is the number of electrons and n_D is the number of new determinants found.

3.4 Results

We apply our revised algorithm to HEG of several different r_s values in the mid- to high-density region and both 14-electron and 54-electron supercells. In each case, we calculate the correlation energies in several basis sets with different momentum cutoffs, and perform a complete basis set (CBS) extrapolation.

When the electron density is high, the correlation energies depend significantly on the momentum cutoff. Hence, in order to obtain more accurate results, we use up to 39,886 orbitals in our calculations. In the high-density region, this decreases the CBS extrapolation distance in the previous literature by more than an order of magnitude, and thus gives us much shorter extrapolation distances and more accurate results.

3.4.1 14-Electron Supercell

Fig. 3.3 to 3.6 show our CBS extrapolation curves and Table 3.1 reports our extrapolated correlation energies. Here M is the number of spin orbitals included in a plane-wave basis set.

We use quadratic extrapolations weighted by $1/M$ for r_s from 0.5 to 2.0. For $r_s = 5.0$, since it is already converged at the size of the basis sets that we use, we take the average of the last three points.

We can see from this table that the results from SHCI are significantly more accurate than previous results. This is mainly because SHCI can use large basis sets, enabling us to go much closer to the infinite basis set limit. Fig. 3.7 which

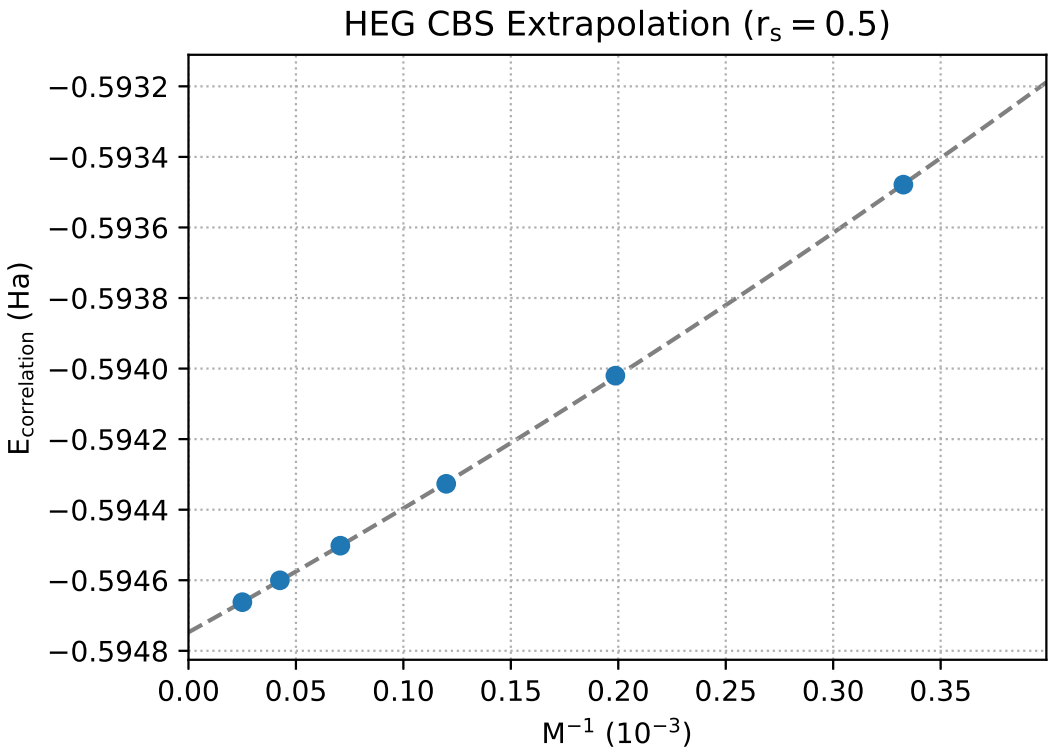

Figure 3.3: Complete basis set extrapolation for HEG 14-electron supercell with $r_s = 0.5$. The extrapolated correlation energy is −0.594748(12) Ha.

plots the raw data points from FCIQMC [95] and SHCI, illustrates this.

3.4.2 54-Electron Supercell

We also apply SHCI to the 54-electron supercell case. Fig. 3.8 shows the CBS extrapolation and Table 3.2 compares the SHCI results with FCIQMC and DMC.

We can see that our SHCI result agrees well with FCIQMC, but slighly lower than FCIQMC-TC, and all of them are much higher then BF-DMC, which probably used a poor trial wavefunction. In these SHCI calculations, we use an order

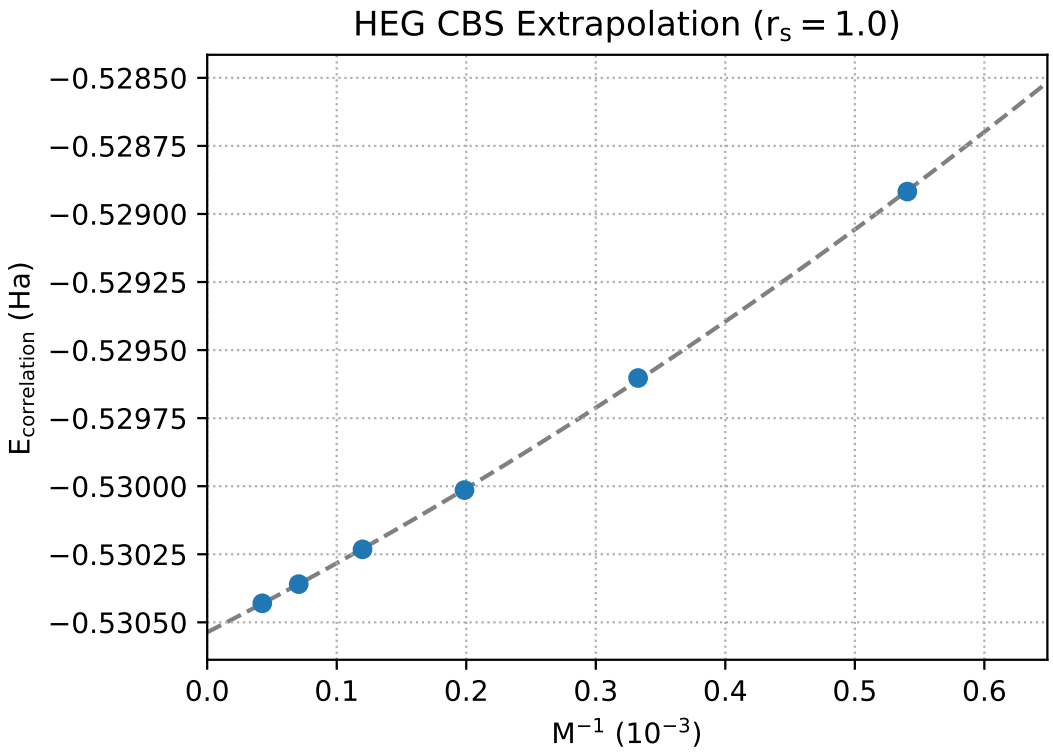

Figure 3.4: Complete basis set extrapolation for HEG 14-electron supercell with $r_s = 1.0$. The extrapolated correlation energy is −0.530536(18) Ha.

of magnitude more orbitals than FCIQMC and FCIQMC-TC, and the extrapolation distance of SHCI is about an order of magnitude smaller than FCIQMC and four times smaller than FCIQMC-TC. Hence, we believe the extrapolated value from SHCI is likely to be more accurate and reliable than from FCIQMC or FCIQMC-TC.

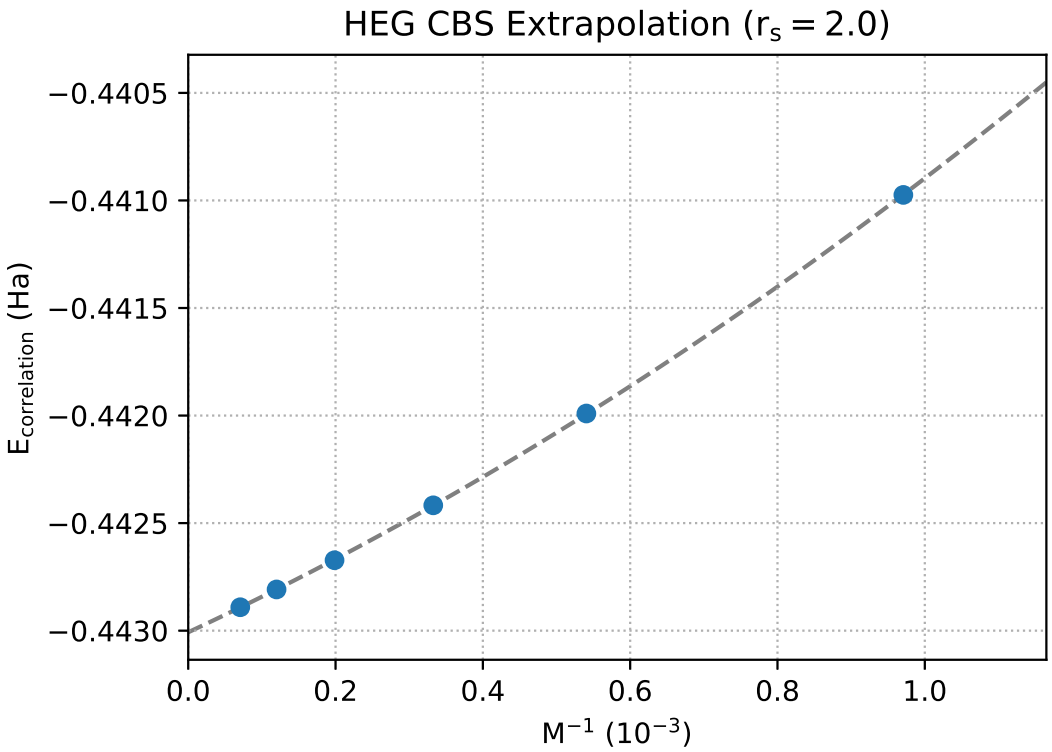

Figure 3.5: Complete basis set extrapolation for HEG 14-electron supercell with $r_s = 2.0$. The extrapolated correlation energy is −0.443007(12) Ha.

3.5 Conclusions

In this paper, we applied our fast semistochastic heat-bath configuration interaction algorithm (SHCI) to the homogeneous electron gas (HEG) problem in the mid to high density regime. In this regime, the basis set extrapolation is the primary source of uncertainty in the energy. By using SHCI with up to 39886 orbitals, we reduced the extrapolation distance significantly and achieved more accurate results than other state-of-the-art methods.

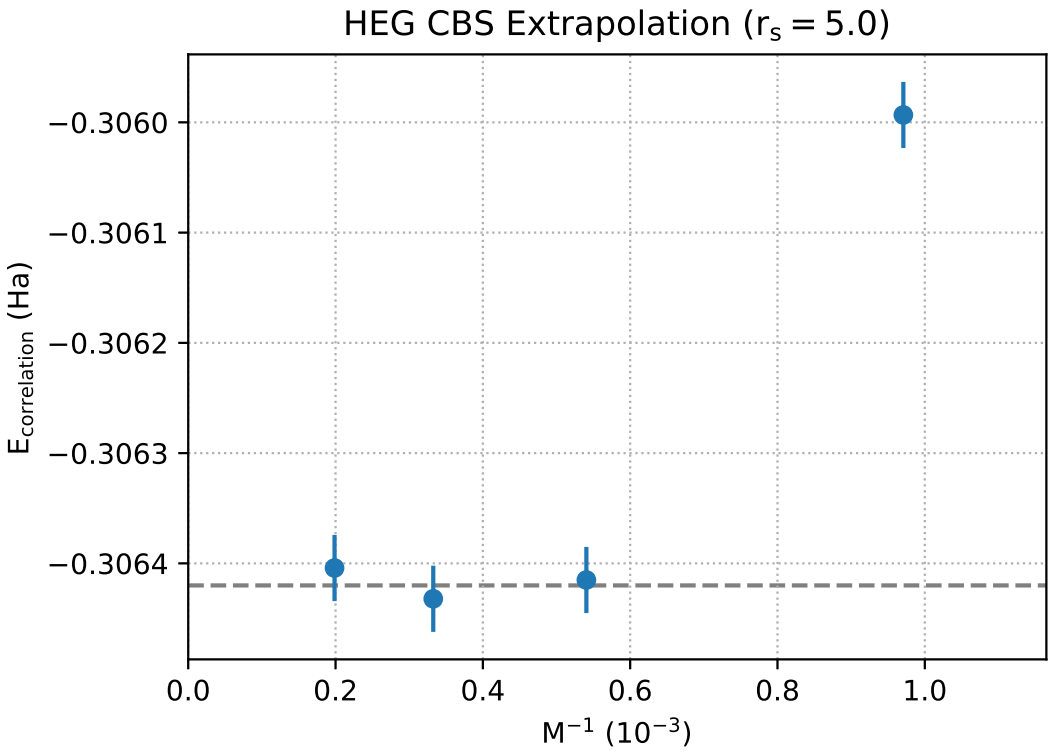

Figure 3.6: Complete basis set extrapolation for HEG 14-electron supercell with r_s = 5.0. Since the correlation energy is already converged after 2000 orbitals, we use the average of the last 3 points in this case. The extrapolated correlation energy is −0.30642(5) Ha.

Table 3.1: Summary of the HEG Total Correlation Energies (Ha). CCMC [79] uses quantum Monte Carlo to evaluate coupled cluster wavefunctions with up to 1030 orbitals and 5^{th} order excitation (CCSDTQ5). FCIQMC [95] and its recent improvement FCIQMC-TC [70] use up to 2368 orbitals. SHCI uses up to 39886 orbitals, which give shorter extrapolation distances and much more accurate results than CCMC and FCIQMC.

r_s	SHCI	CCMC	FCIQMC	FCIQMC-TC
0.5	-0.594748(12)	-0.5947(2)	-0.5959(7)	-0.5948(2)
1.0	-0.530536(18)	-0.5311(2)	-0.5316(4)	-0.5309(2)
2.0	-0.443007(12)	-0.4434(10)	-0.444(1)	-0.4440(3)
5.0	-0.30642(5)	-0.3025(4)	-0.307(1)	-0.3078(3)

Table 3.2: Summary of the HEG Total Correlation Energies (Ha) with 54-Electron Supercells. DMC [87] uses real space basis and backflow wave function. FCIQMC [95] and its recent improvement FCIQMC-TC [70] use up to 1850 orbitals. SHCI uses up to 23506 orbitals, which give much shorter extrapolation distances than FCIQMC and thus more accurate results.

r_s	SHCI	FCIQMC	FCIQMC-TC	BF-DMC
0.5	-2.4313(11)	-2.435(7)	-2.425(1)	-2.387(2)

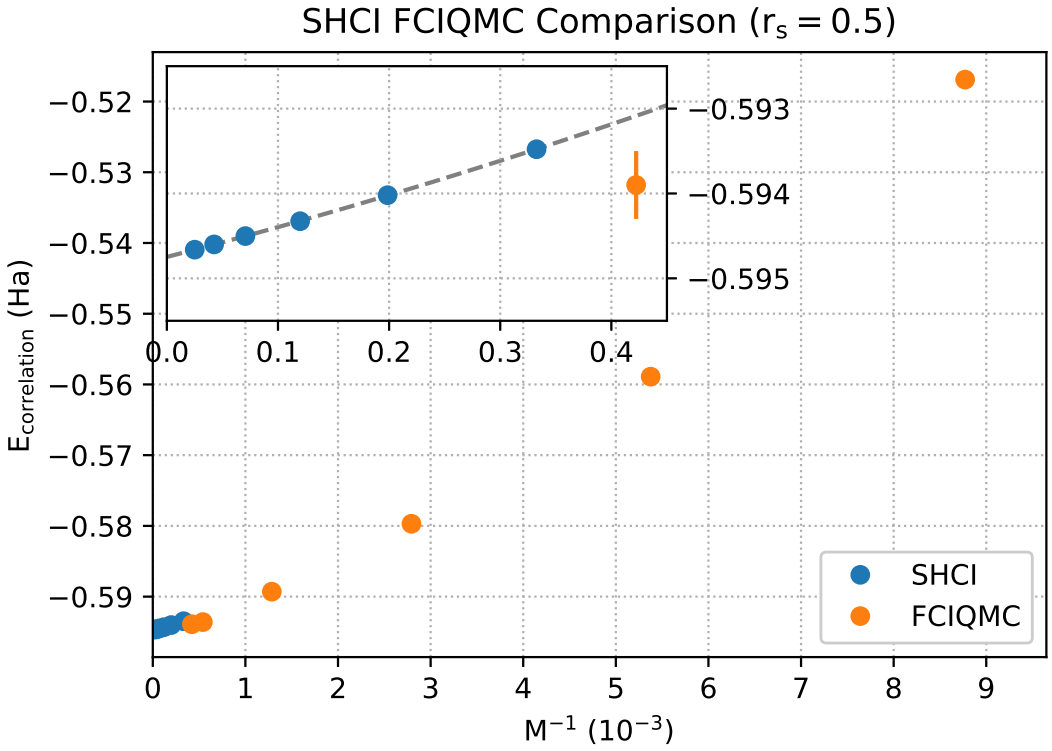

Figure 3.7: Comparison between SHCI results and FCIQMC results for $r_s = 0.5$. Note that all the points have error bars smaller than the size of the points themselves except for the FCIQMC point on the zoomed-in view. SHCI goes much closer to the infinite basis set and thus achieves more accurate and reliable extrapolated results.

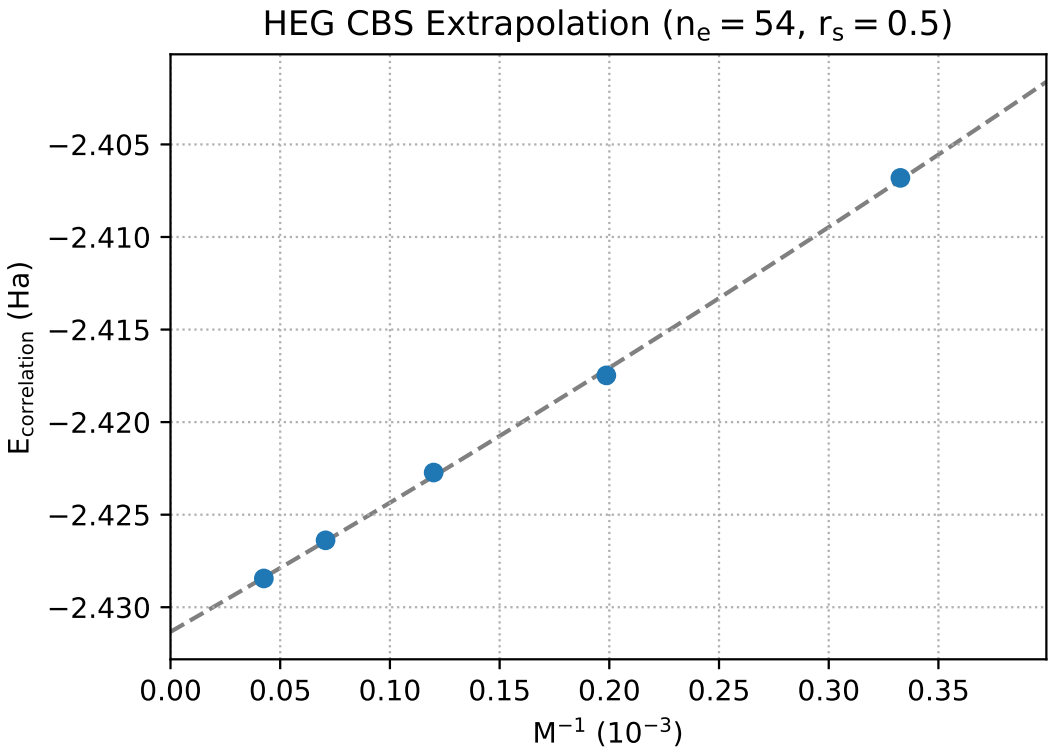

Figure 3.8: Complete basis set extrapolation for HEG 54-electron supercell with $r_s = 0.5$. The extrapolated correlation energy is −2.4313(11) Ha.

CHAPTER 4

CHROMIUM DIMER SIMULATION WITH FAST SHCI

In this chapter, I apply the fast SHCI method to the chromium dimer. I first perform an accurate calculation at the equilibrium geometry, and then perform a slightly less accurate calculation for several squeezed or stretched geometries and obtain the entire potential energy curve.

4.1 Introduction

The chromium dimer is a challenging strongly-correlated system that has been used as a benchmark molecule for a variety of methods [91, 62, 84, 71, 106, 49]. In this chapter, we use the fast heat-bath configuration interaction method to calculate both the energy at the equilibrium geometry and entire potential energy curve.

For the equilibrium geometry, we include in our variational wavefunction two billion most important determinants, which are two orders of magnitudes more than ever done in other selected-CI based methods. This allows us to achieve significantly higher accuracy, which even beats the accuracy of well-developed methods, such as the density matrix renormalization group (DMRG). Section 4.2 presents the results.

For the entire potential energy curve calculation, we correlate up to 28 electrons in a cc-pVDZ basis. The resulting Hilbert space of 5×10^{29} determinants is several orders of magnitude larger than that used in other systematically improvable methods. Our results match well with the experiments at and near the

equilibrium geometry. Section 4.3 presents the results.

4.2 Equilibrium Geometry

In this section[1], we examine the chromium dimer at the equilibrium bond length of 1.68 Å.

We use a relativistic exact two-component (X2C) Hamiltonian, the cc-pVDZ-DK basis, and we correlate the valence and the semi-core electrons. This gives an active space of (28e, 76o) and a Hilbert space of 5×10^{29} determinants, which is far beyond the reach of FCI. We show how we obtain an accurate estimate of the FCI energy in this large active space with our improved SHCI algorithm.

We use PySCF [99] to generate the molecular orbital integrals for orbitals that minimize the HCI variational energy for $\epsilon_1 = 2 \times 10^{-4}$ Ha, using the method of Ref. [97]. We perform SHCI with several ϵ_1 values from $5{\times}10^{-5}$ to $3{\times}10^{-6}$ Ha. The Hamiltonian matrix is stored in memory distributed over computer nodes, and so needs to be constructed only once. We use very small values of $\epsilon_2 = 10^{-6}\epsilon_1$ to ensure that the perturbative correction is exceedingly well converged, and choose the target error for the stochastic perturbation energy to be 10^{-5} Ha.

The improved SHCI is fast enough that we can use over two billion variational determinants, and stochastically include the contributions of at least trillions of perturbative determinants. The largest variational calculation, where we iteratively find and diagonalize 2 billion determinants for $\epsilon_1 = 3.0 \times 10^{-6}$ Ha, takes only one day on 8 nodes, each of which has 4 Intel Xeon E7-8870 v4 CPUs.

[1] This section was published in Ref. [66]

The corresponding perturbative calculation takes only 6 hours using only one of these nodes. During that perturbative calculation, we skip the deterministic step, perform a pseudo-stochastic step with $\epsilon_2^{\text{psto}} = 1 \times 10^{-7}$ Ha, and a stochastic step with $\epsilon_2 = 3 \times 10^{-12}$ Ha. We skip the deterministic step here because $\epsilon_1 = 3 \times 10^{-6}$ is already close to our default ϵ_2^{dtm} of 2×10^{-6} so skipping this won't affect the efficiency of subsequent steps much. The pseudo-stochastic step uses 25 batches, each of which has about 8.9 billion determinants. We evaluate only one of them, from which we obtain an estimate of the total correction for all the 25 batches (223 billion determinants) to be -0.011681(1) Ha. Since the estimated error is already much smaller than our target error, we skip the remaining 24 batches. The pseudo-stochastic step takes 1.6 hours. The stochastic step uses 128 batches and 6 million variational determinants in each sample, which results in about 3.7 billion determinants per batch. We use 10 samples and obtain the additional correction from $\epsilon_2 = 3.0 \times 10^{-12}$ Ha to be -0.001203(6) Ha. The combined uncertainty of the entire semistochastic perturbation stage is 6 μHa. It is hard to estimate how many determinants are stochastically included for $\epsilon_2 = 3 \times 10^{-12}$ Ha, so we estimate a lower bound with $\epsilon_2^{\text{psto}} = 1.4 \times 10^{-8}$ Ha and obtain 1.8 trillion unique perturbative determinants. Hence, with $\epsilon_2 = 3 \times 10^{-12}$ Ha (the value we are actually using) we stochastically estimate contributions from at least trillions of unique perturbative determinants and obtain better than 10^{-5} Ha statistical uncertainty in 6 hours using only one node.

These large calculations enable us to obtain an estimate of the FCI energy with sub-millihartree uncertainty in this large active space. Table 4.1 reports the results.

We extrapolate our results using a weighted quadratic fit and obtain for the

ϵ_1 (Ha)	$N_{\mathcal{V}}$	E_{var} (Ha)	E_{total} (Ha)
5.0×10^{-5}	24M	-2099.863816	-2099.909741(7)
3.0×10^{-5}	53M	-2099.875327	-2099.912356(7)
2.0×10^{-5}	102M	-2099.883027	-2099.914132(8)
1.0×10^{-5}	309M	-2099.893761	-2099.916595(1)
7.0×10^{-6}	539M	-2099.898165	-2099.917540(1)
5.0×10^{-6}	911M	-2099.901781	-2099.918306(3)
3.0×10^{-6}	2.00B	-2099.906322	-2099.919205(6)
0.0 (Extrap.)	-	-2099.9224(6)	

Table 4.1: Results for Cr_2 at r=1.68 in the cc-pVDZ-DK basis. The active space is (28e, 76o). $N_{\mathcal{V}}$ is the number of variational determinants. $\epsilon_2 = 10^{-6}\epsilon_1$. We use weighted quadratic extrapolation, shown in Fig. 4.1, to obtain the FCI limit corresponding to $\Delta E = 0$.

ground state energy, −2099.9224 Ha as $\Delta E \to 0$. The weight of each point is $(E_{\text{var}} - E_{\text{tot}})^{-2}$. Fig. 4.1 shows the computed energies and the extrapolation. We also perform a weighted linear fit and use the difference of the extrapolated values from the quadratic and the linear fits (0.6 mHa) as the uncertainty. In summary, the estimated FCI energy of Cr_2 in the cc-pVDZ-DK basis with 28 correlated electrons and the relativistic X2C Hamiltonian is −2099.9224(6) Ha.

We compare our result with DMRG and p-DMRG, which are the only essentially exact methods that have been applied to this large active space of the chromium dimer. The DMRG calculations use up to bond dimension $M = 16000$ and obtain an extrapolated energy of −2099.9195(27) Ha (default schedule) and −2099.9192(24) (reverse schedule) [48]. These two values are close to the SHCI energy obtained with the smallest ϵ_1, but higher than the extrapolated SHCI energy by 3 mH, which is about the estimated error of the DMRG results. The p-DMRG calculations used up to $M = 4000$ and extrapolated energy obtained from a linear fit is −2099.9201 Ha [48]. If instead, we perform a weighted quadratic fit (shown in Fig. 4.1), the extrapolated energy is −2099.9225 Ha, in perhaps for-

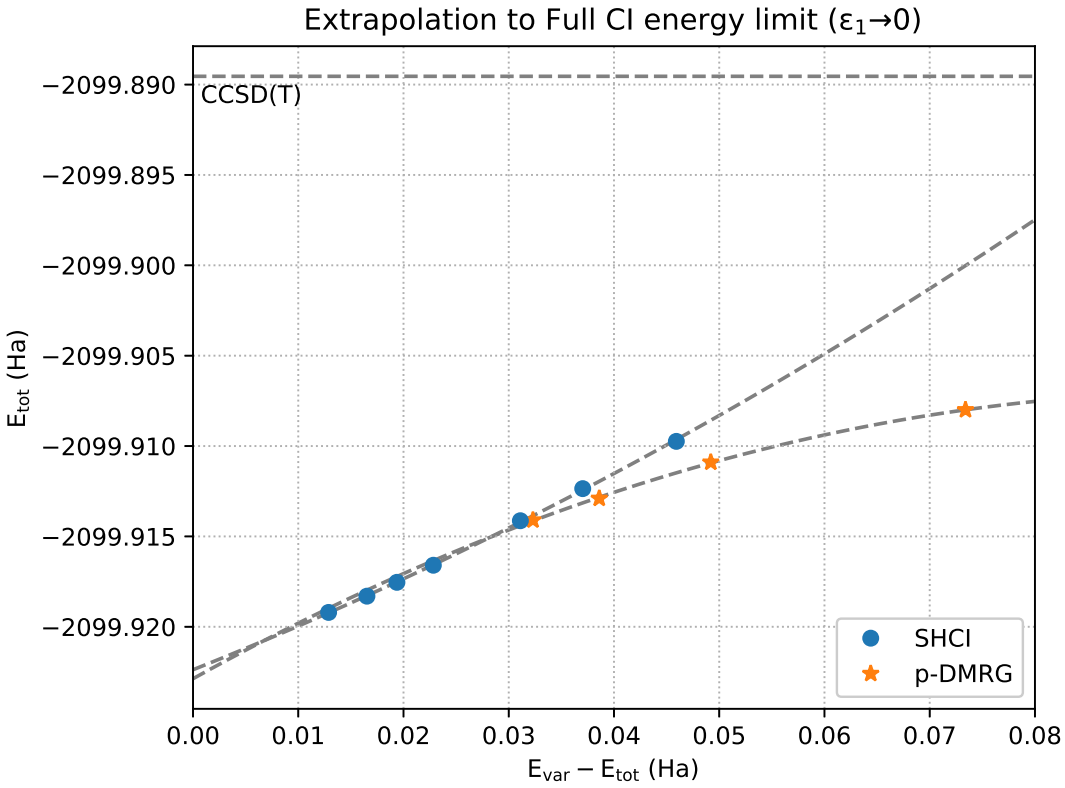

Figure 4.1: Weighted quadratic extrapolation of the Cr_2 ground state energy. The weight of each point is $(E_{\text{var}} - E_{\text{tot}})^{-2}$. The extrapolated energy is −2099.9224(6), where the uncertainty comes from the difference between linear extrapolation and quadratic extrapolation. The p-DMRG extrapolation and the CCSD(T) value are also shown.

tuitously good agreement with our result of −2099.9224(6) Ha. However, the extrapolation uncertainty is larger than the SHCI extrapolation uncertainty. In contrast, the CCSD(T) energy is considerably higher and DMRG and SHCI.

One of the merits of selected-CI methods is the ability to include all excitations, regardless of excitation level. To see the contribution from each excitation level we plot the number of selected determinants and the $\sum_i |c_i|^2$ versus excitation level in Fig. 4.2. Determinants with excitation levels up to 15 excitations are

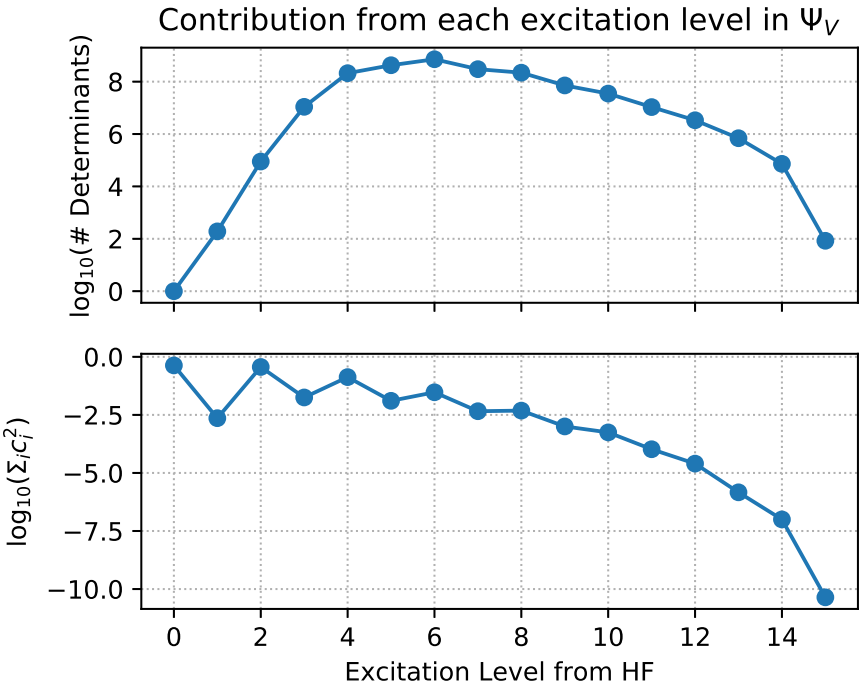

Figure 4.2: Contribution from each excitation level to the variational wavefunction for Cr_2 with 2×10^9 determinants. Determinants with up to 15 excitations are present in the variational wavefunction.

present in the variational wavefunction even though we are using optimized orbitals. (Using Hartree-Fock orbitals, we expect that determinants with even higher excitation levels will be present.) This implies that truncating the CI expansion at the double, triple or quadruple excitation levels (which is the most that is usually done in systematic CI expansions), will give poor energies for such strongly correlated systems.

4.3 Potential Energy Surface

In addition to the energy at equilibrium geometry, I also obtain the entire potential energy curve of the chromium dimer.

We calculate the potential energy curves, first correlating just the 12 valence (3d, 4s) electrons, and then correlating also the semi-core (3s, 3p) electrons, making a total of 28 correlated electrons. For 12 correlated electrons, we study the basis set dependence by calculating the curves for cc-pVDZ, cc-pVTZ, and cc-pVQZ basis sets. The largest active space in our calculations is (28e, 76o), which gives a Hilbert space of 5×10^{29} determinants, far beyond the reach of FCI.

We use both PySCF [99] and our program to generate the molecular orbital integrals for orbitals that minimize the HCI variational energy for $\epsilon_1 = 2 \times 10^{-4}$ Ha, using an improved version of the method of Ref. [97]. For 12 correlated electrons, we use CAS-core orbitals, and for 28 correlated electrons, we use the HF-core orbitals. For the former, the potential energy curves obtained from HF-core and CAS-core orbitals differ greatly, whereas for the latter there is essentially no difference between HF-core and CAS-core curves. We perform SHCI with several ϵ_1 values from 2×10^{-4} to 5×10^{-6} Ha. The sparse Hamiltonian matrix is stored in memory and so needs to be constructed only once for each geometry and each active space. For the perturbative correction calculation, we set $\epsilon_2 = 0$ to include the entire Hilbert space.

The improved SHCI is fast enough that we can use more than one billion variational determinants and stochastically include the corrections from at least trillions of perturbative determinants for each geometry.

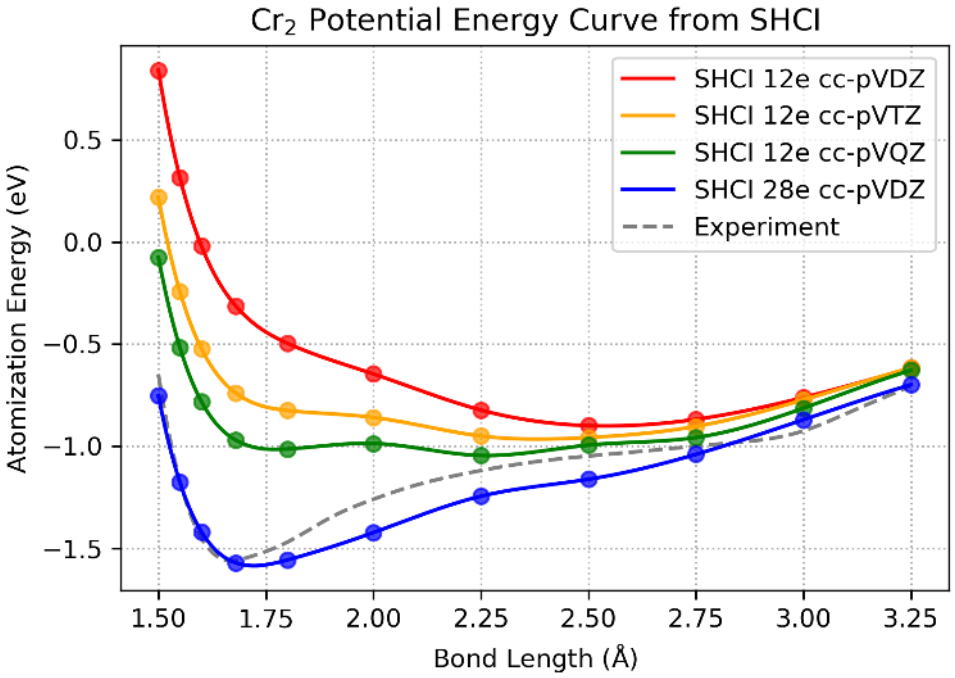

Figure 4.3: Comparison of SHCI potential energy curves of Cr_2, correlating 12 or 28 electrons, with experiment. The shape of the experimental data is deduced from measuring 29 vibrational states using negative-ion photoelectron spectroscopy [19]. The potential energy curves from the 12-correlated-electron calculations agree poorly with experiment, though the agreement improves upon increasing the basis size. The 28-correlated-electron calculation agrees much better with experiment.

We extrapolate the energies for each geometry with a weighted quadratic fit and obtain for the ground state energy as $\Delta E \to 0$. The weight of each point is $(E_{\text{var}} - E_{\text{tot}})^{-2}$. Then we interpolate the points for each active space with cubic spline interpolation. Fig. 4.5 presents the most accurate raw data points and the extrapolated curve for the 28 correlated electrons case.

Fig. 4.3 compares the energies calculated with different active spaces to the experimental data. There is some uncertainty in the experimental curve due to

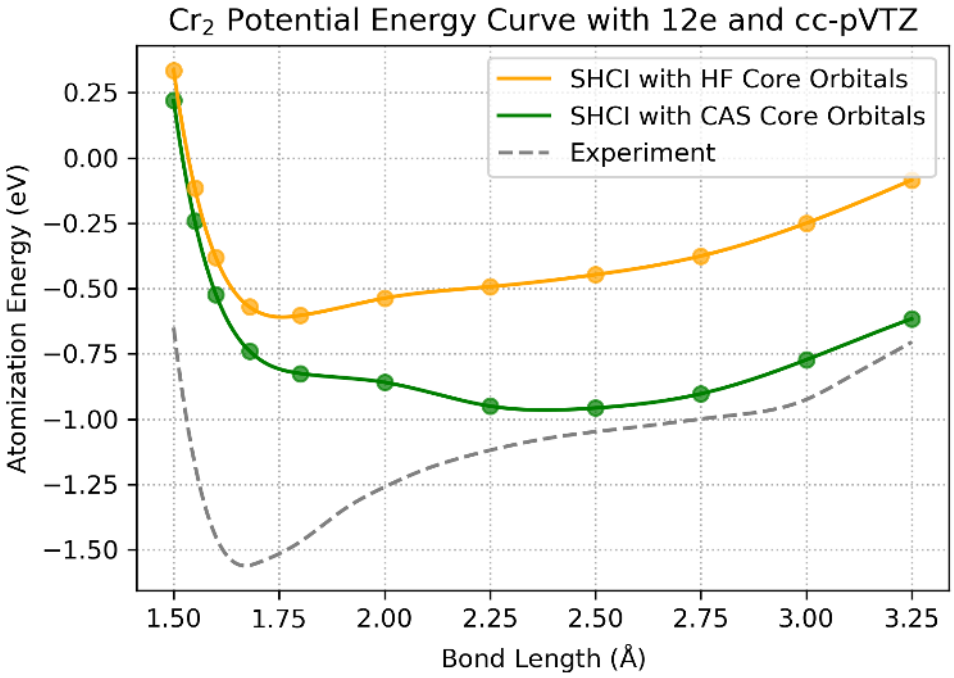

Figure 4.4: Comparison of SHCI potential energy curves of Cr_2 with 12 correlated electrons and a cc-pVTZ basis, using either HF-core orbitals or CAS-core orbitals. The CAS-core curve gives a lower potential energy, closer to experiment, but the HF-core calculation results in a potential energy curve whose shape (in particular the location of the minimum) is closer to experiment.

uncertainty in the assignment of the higher vibrational levels as well as due to uncertainty in the depth of the potential. The shape of the experimental curve is deduced from 29 vibrational states using negative-ion photoelectron spectroscopy [19]. The measured 29 vibrational states were assigned to $\nu = 1 - 9$ and $\nu = 24 - 43$. There is uncertainty in the shape of the experimental curve because of the missing vibrational levels as well as uncertainty about the assignment of the observed peaks to the higher vibrational levels. The depth of the experimental curve is estimated from adding an estimate of the zero-point energy to experimentally determined bond dissociation energies. Unfortunately, there is

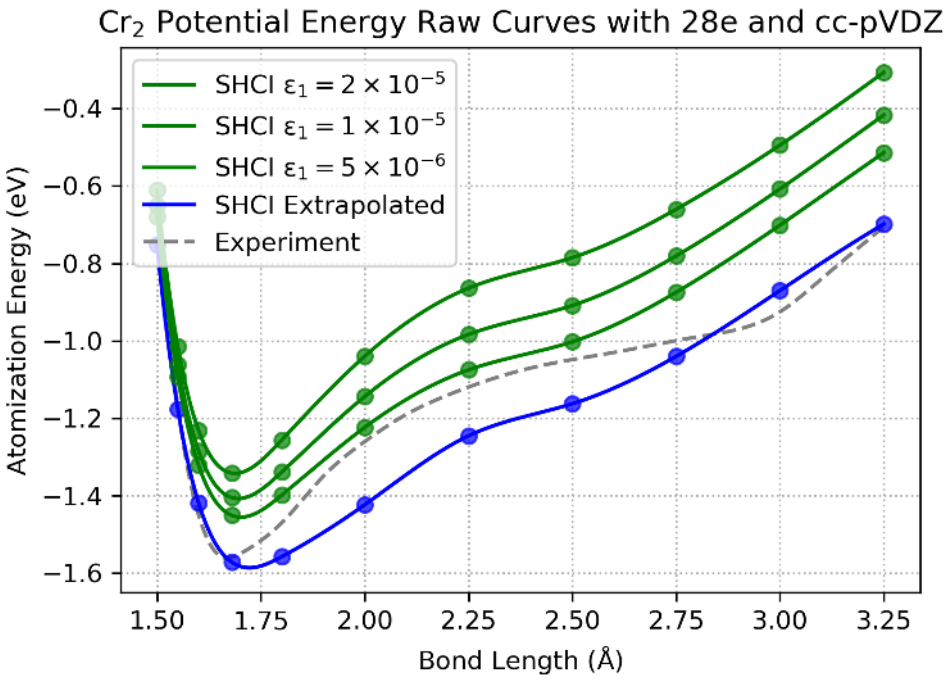

Figure 4.5: Raw data and extrapolation for Cr_2 with 28 correlated electrons and a cc-pVDZ basis.

a considerable spread in the estimates of the latter. Here we use the well depth estimate of 1.56 eV from Ref. [106], which is based on the bond dissociation energy of Ref. [96]. This well depth is deeper than the estimate of 1.47 eV used in Ref. [49] based on the bond energy of 1.44 eV reported in Ref. [19].

Fig. 4.3 shows that the 12-active-electron curves do not agree well with experiment. Using a larger basis sets helps bring the curve closer to the experimental data, but is not enough to produce a good agreement. By including the 10 additional electrons from the semi-core in the active space, SHCI achieves much better agreement, demonstrating that semi-core correlation plays an important role in the molecular bond of Cr_2. At large bond lengths one would expect that semi-core correlation is not important and in fact the 12 active elec-

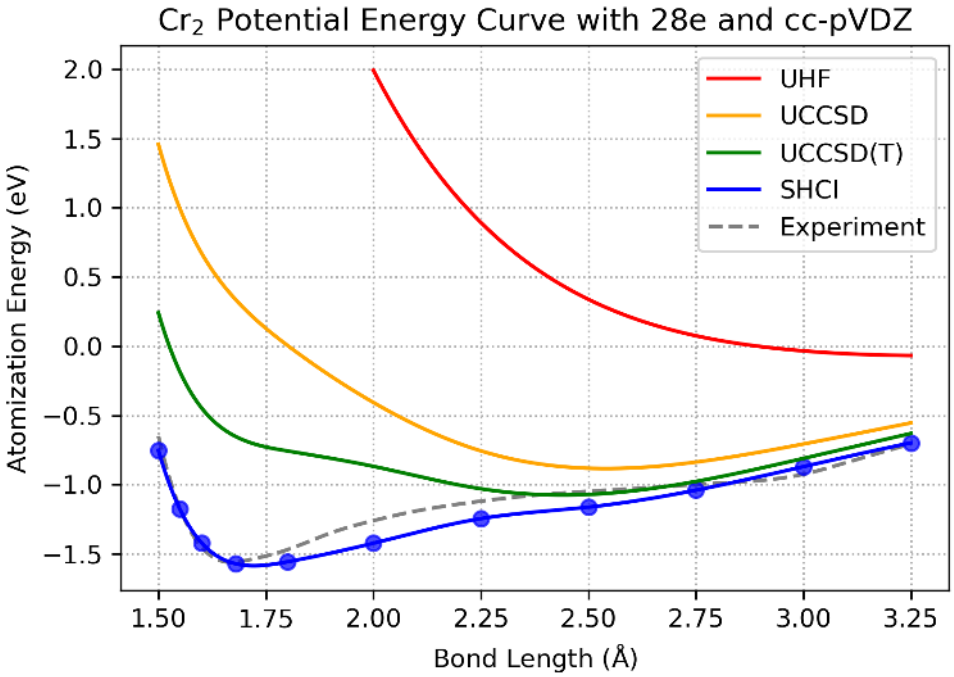

Figure 4.6: Potential energy curve of Cr_2 with 28 electrons in a cc-pVDZ basis calculated from various methods. The experimental data come from negative-ion photoelectron spectroscopy [19]. The UHF curve bears no resemblance to the experimental curve. The UCCSD and UCCSD(T) curves are better, especially at long bond lengths, but even UCCSD(T), which is considered to be the "gold standard" for single-reference systems, agrees poorly with experiment. In contrast, the SHCI curve is in reasonable agreement with experiment.

tron and the 28 active electron energies become nearly coincident there.

Fig. 4.4 shows the difference between using HF-core orbitals and using CAS-core orbital, under the setting of 12 correlated electrons and cc-pVTZ basis sets. We can see that the CAS-core curve gives a lower potential energy, closer to experiment, but has the wrong minimum location. The HF-core calculation results in a potential energy curve whose overall shape and the location of the minimum are much closer to the experiment.

In Fig. 4.6, we compare SHCI results to unrestricted Hartree-Fock (UHF) and unrestricted coupled cluster methods, UCCSD and UCCSD(T). As expected, the UHF curve bears no resemblance to the experimental curve. The UCCSD and UCCSD(T) curves are better, but even UCCSD(T), which is considered to be the "gold standard" for single-reference systems, agrees poorly with experiment. In contrast, the SHCI curve is in reasonable agreement with experiment, perhaps fortuitously so, since the cc-pVDZ basis used is expected to have a significant finite-basis error. With our current computer resources, we cannot obtain a well-converged curve with larger basis sets, but, with further improvements in the method and larger computers, we hope to obtain basis set converged curves that agree yet better with experiment, and perhaps even provide information about experimental inaccuracies.

CHAPTER 5

TRANSITION METAL SYSTEMS BENCHMARK

5.1 Introduction

I participated in a large multi-institutional collaboration that used 21 different electronic structure methods on 21 transition metal atoms, ions and oxides, each of them for basis sets ranging from $d\zeta$ to 5ζ. The goal of the project was to have 3 methods that provide essentially exact energies for these systems (within the chosen basis sets) but at a cost that scales exponentially with system size, and use these energies to assess the accuracy of 18 other methods that are approximate but exhibit better scaling. My role in the project was to provide the SHCI energies (one of the 3 methods capable of providing the exact energies, and in fact the only method that was able to compute all the systems for all the basis sets).

5.2 Methods

We consider transition metal systems, with the core electrons removed using effective core potentials[102, 104]. These potentials are an accurate representation of the core[12] for many-body simulations and allow all the methods considered in this work to be directly comparable. These potentials are available for Sc, Ti, V, Cr, Mn, Fe, and Cu, which defines our test set. We consider these transition metal atoms, their ions, and their monoxide molecules. To simplify the comparison, the molecules were computed at their equilibirium geometry with bond

Table 5.1: A list of abbreviations used in this benchmark. In Column A, the maximum basis set performed by that method for the transition metal atoms is listed, and in Column B the same for the monoxide molecules.

Abbreviation	Method	A	B
AFQMC	Auxilliary field quantum Monte Carlo	5	5
iFCIQMC	Full configuration interaction quantum Monte Carlo	q	d
DMRG	Density matrix renormalization group	t	d
SHCI	Semistochastic heatbath configuration interaction	5	5
CCSD(T)	Coupled cluster with singles, doubles, and perturbative triples	5	5
SEET	Self-energy embedding theory	q	q
CISD	Configuration interaction with singles and doubles	5	5
QSGW	quasiparticle self-consistent GW approximation	t	t
HF+RPA	Hartree-Fock random phase approximation	t	t
SC-GW	Self-consistent GW approximation	d	-
GF2	Second order Green function	q	q
CCSD	Couple cluster with singles and doubles	5	5
MRLCC	multireference localized coupled cluster	5	5
DMC	Diffusion Monte Carlo (single determinant)	c	c
LDA	DFT in the local density approximation	5	5
PBE	DFT in the PBE approximation	5	5
HSE06	DFT with the HSE06 functional	t	t
B3LYP	DFT with the B3LYP functional	5	5
SCAN	DFT with SCAN functional	5	5
HF	Hartree-Fock	5	5

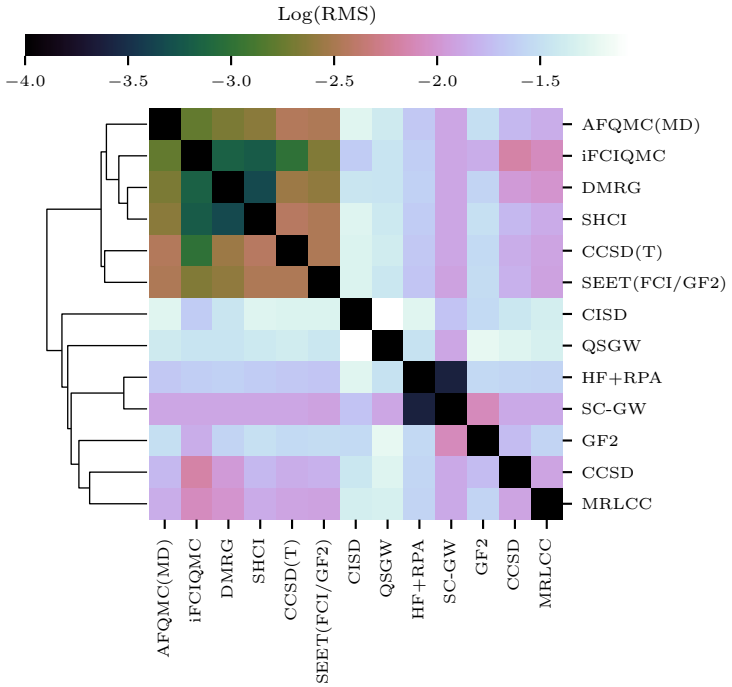

Figure 5.1: Cluster analysis of electronic structure methods in this work. The matrix values are the logarithm of the RMS deviation of the total energy in Hartrees (Eqn 5.1) between the two methods.

lengths in as follows: ScO: 1.668, TiO: 1.623,VO: 1.591, CrO: 1.621, MnO: 1.648, FeO: 1.616, CuO: 1.725.

Almost every electronic structure method works in a finite basis. Here, we follow the chemistry convention of defining an ascending basis set denoted by the ζ value, ranging from 2 to 5; i.e., dζ, tζ, qζ, and 5ζ. For each system, we consider the first principles Hamiltonian projected onto that basis. While these results are only comparable to experiment in the complete basis set limit (cbs), for each basis set there corresponds a projected Hamiltonian, which also has an exact solution. We thus can compare methods *within a basis* since the Hamiltonian is defined precisely.

The deviation in the total energy between two methods m and n is computed as

$$\sigma(m,n) = \sqrt{\frac{\sum_{i\in\text{systems}}(E_i(n) - E_i(m))^2}{N}}. \tag{5.1}$$

This is a measure of how well the output total energies between two methods agree; it is possible for two methods with large σ to agree on energy *differences*.

To compare total energy between methods and systems in a consistent way, we use the concept of percent of correlation energy, commonly used in quantum chemistry:

$$\%\ \text{correlation energy}(m) = 100 \times \frac{E_{HF} - E_m}{E_{HF} - E_{SHCI}}, \tag{5.2}$$

where E_{HF} is the Hartree-Fock energy, m stands for the method under consideration, and E_{SHCI} is the total energy computed in the basis by the SHCI method.

Energy differences are assessed by considering the ionization potential of a transition metal M: IP $= E(M+) - E(M)$ and the dissociation energy of a metal oxide molecule MO: DE $= E(M) + E(O) - E(MO)$. These quantities have been studied in detail for these systems in the past, for example Refs [11, 43, 35, 107, 116, 14, 74, 76, 101, 100, 57], among others. However, none of these previous studies have attained reference results as well-converged as the ones in this paper, and none compare a large number of techniques on the same Hamiltonian.

In Table 5.1, we classify the methods tested in this work. The ensemble of techniques includes many of the most common techniques to address the many-electron problem, as well as some emerging methods. We also include a few methods such as CISD which are no longer commonly used in calculations, but have historical relevance. The methods in this benchmark vary dramatically in their computational cost; the density functional theory methods required only

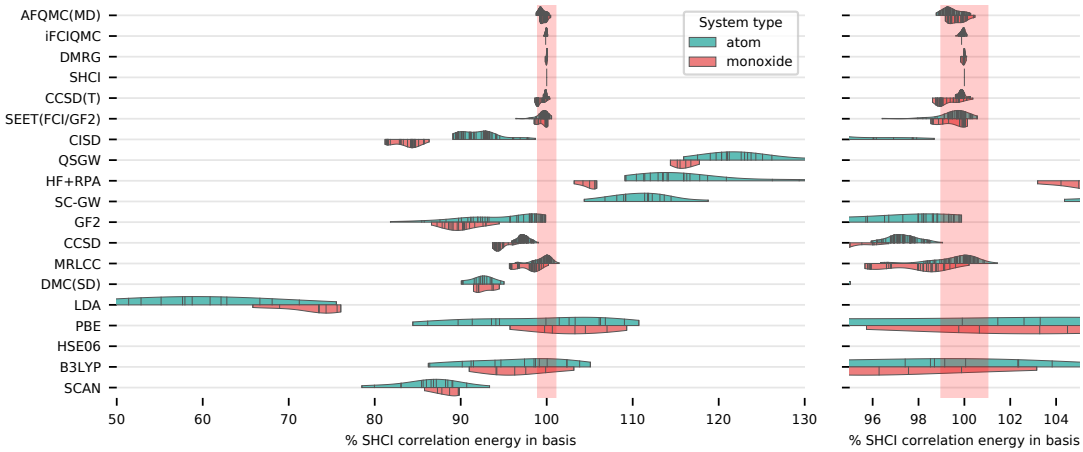

Figure 5.2: Kernel density estimation of the percent of the SHCI-computed correlation energy within each basis obtained by each of the methods in the benchmark set. All basis sets available are plotted; individual data points are indicated by small lines.

a few minutes to complete the test set, while some of the more advanced techniques were not able to treat every basis for every system with the available amount of computer time. The methods also scale very differently, ranging from $O(N_e^3)$ to exponential in the number of electrons.

5.3 Results

In Fig 5.1, we show a cluster analysis of the total energy results using Eqn 5.1 as the distance metric. Three methods agree to better than 1 mHa on all attained basis sets and systems: DMRG, iFCIQMC, and SHCI. The iFCIQMC and DMRG techniques were not optimal for these systems, and so could only be completed to high accuracy for small basis sets. These three methods each have single parameters that, when converged, theoretically should result in exact re-

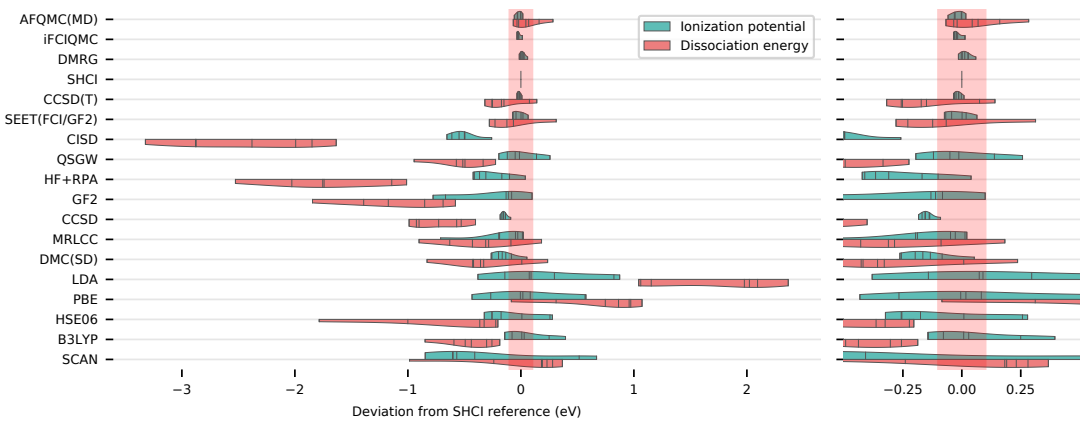

Figure 5.3: Kernel density estimation plot of binding energy and ionization potential of molecules and atoms to SHCI complete basis reference calculations. Each technique is listed with the largest basis set available, so long as the basis set is triple-ζ or larger. Methods are ordered according to the clustering in Fig 5.1.

sults. Because of this 3-fold agreement, we can take any of these results as the exact ground state energy in a given basis set to within about 1 mHa, which is approximately what is termed "chemical accuracy." Among these three, only SHCI was performed for all basis sets and all systems and so we use that as the reference total energy.

Using this reference, one can compute the percent of correlation energy obtained by a given method, shown in Fig 5.2. At 100% of the correlation energy, the exact result is obtained.

We can see that all the systematic methods, including FCIQMC, DMRG, and SHCI, agree exceptionally well with each other, for those basis sets where the FCIQMC and DMRG calculations were feasible. This is as expected since they use the same Hamiltonian and all of these methods should give the full-CI energies without systematic errors.

However, systematic methods like these can only be applied to relatively small systems due to their high computational cost, so it is also important to study the errors in other more efficient methods. These errors are often canceled out in previous studies when comparing to experimental data, and the results from our study make these errors directly accessible.

From these figures, we can see that both AFQMC and CCSD(T) give reasonably good agreement to the reference values and their computational costs increase much slower with the system size than the three essentially exact methods. Other methods are much less accurate and thus may not produce reliable results for the energies. However, they may still provide sufficient accuracy for some systems.

In many cases, using less accurate methods can be the only option due to the high computational cost of more accurate methods. In such cases, benchmark results from some sample systems of a similar kind can provide useful information on which approximate method to choose.

CHAPTER 6

SIMPLIFIED HIGH PERFORMANCE CLUSTER COMPUTING

MapReduce and its variants have significantly simplified and accelerated the process of developing parallel programs. However, most MapReduce implementations focus on data-intensive tasks while many real-world tasks are compute intensive and their data can fit distributedly into the memory. For these tasks, MapReduce programs can be much slower than hand-optimized ones. We have developed Blaze, a C++ library that makes it easy to develop high performance parallel programs for such compute intensive tasks. At the core of Blaze is a highly-optimized in-memory MapReduce function, which has three main improvements over conventional MapReduce implementations: eager reduction, fast serialization, and special treatment for a small fixed key range. We also offer additional conveniences that make developing parallel programs similar to developing serial programs. These improvements make Blaze an easy-to-use cluster computing library that approaches the speed of hand-optimized parallel code. We apply Blaze to some common data mining tasks, including word frequency count, PageRank, k-means, expectation maximization (Gaussian mixture model), and k-nearest neighbors. Blaze outperforms Apache Spark by more than 10 times on average for these tasks, and the speed of Blaze scales almost linearly with the number of nodes. In addition, Blaze uses only the MapReduce function and 3 utility functions in its implementation while Spark uses almost 30 different parallel primitives in its official implementation.

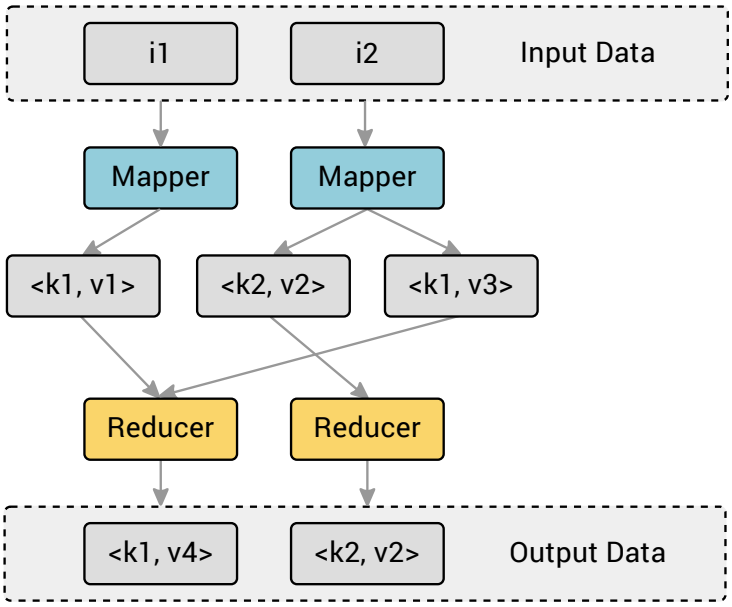

Figure 6.1: MapReduce Programming Model. The map function generates a set of intermediate key/value pairs for each input. The reduce function merges the values associated with the same key. Numerous data mining and machine learning algorithms are expressible with this model.

6.1 Introduction

Cluster computing enables us to perform a huge amount of computations on big data and get insights from them at a scale that a single machine can hardly achieve. However, developing parallel programs to take advantage of a large cluster can be very difficult.

MapReduce [33, 34] greatly simplified this task by providing users a high-level abstraction for defining their computation, and taking care of the intricate low-level execution steps internally. Fig. 6.1 illustrates the MapReduce programming model. Logically, each MapReduce operation consists of two phases: a map phase where each input is mapped to a set of intermediate key/value

pairs, and a reduce phase where the pairs with the same key are put together and reduced to a single key/value pair according to a user specified reduce function.

Many data mining algorithms are expressible with this model, such as PageRank [10, 83, 37], k-means [123, 26, 31, 8, 47], Gaussian mixture model [26], and k-nearest neighbors [9, 72, 69, 118].

Although logically expressible, achieving similar efficiency as a hand-optimized parallel code is hard, especially when the data can be fit distributed into the memory. In such cases, the file system is no longer the bottleneck and the overhead from MapReduce can make the execution much slower than hand-optimized code.

Google's MapReduce [33, 34] and most of its variants [41, 21, 73, 7, 13, 30, 37, 46, 51, 67, 121, 117] save intermediate data and result to the file system even when the data can be fit into the memory. Hence, its MapReduce performance is severely limited by the performance of the file system.

Spark [42, 120, 122, 119] offers an in-memory implementation of MapReduce, which is much faster than Google's MapReduce. However, it uses a similar algorithm as Google's MapReduce, which is designed for disk-based data intensive use cases and does not consider the computational overheads of MapReduce seriously. Hence, the performance of Spark is often far from the performance of hand-optimized code.

To achieve better performance while preserving the high-level MapReduce abstraction, we develop Blaze, a C++ based cluster computing library that focuses on in-memory high performance MapReduce and related operations.

Blaze introduces three main improvements to the MapReduce algorithm: eager reduction, fast serialization, and special treatment for a small fixed key range. Section 6.2.3 provides a detailed description of these improvements.

We apply Blaze to several common data mining tasks, including word frequency count, PageRank, k-means, expectation maximization (Gaussian mixture), and k-nearest neighbors. Our results show that Blaze is on average 10 times faster than Spark on these tasks.

The main contributions of this research are listed as follows:

1. We develop Blaze, a high performance cluster computing library that allows users to write parallel programs with the high-level MapReduce abstraction while achieving similar performance as hand-optimized code for compute intensive tasks.
2. We introduce three main performance improvements to the MapReduce algorithm to make it more efficient: eager reduction, fast serialization, and special treatment for a small fixed key range.
3. We apply Blaze to 5 common data mining tasks and demonstrate that Blaze programs are easy to develop and can outperform Apache Spark programs by more than 10 times on average for these tasks.

The remaining sections are organized as follows: Section 6.2 describes the Blaze framework and the details of the optimization. Section 6.3 present the details of how we implement several key data mining and machine learning algorithms with Blaze and compare the performance with Apache Spark. Section 6.4 concludes the paper.

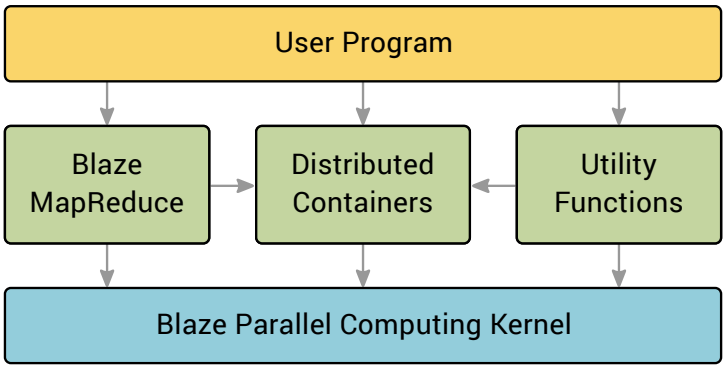

Figure 6.2: Blaze Architecture.

6.2 The Blaze Library

The Blaze library offers three sets of APIs: 1) a high-performance MapReduce function, 2) distributed data containers, and 3) parallel computing utility functions. These APIs are built based on the Blaze parallel computing kernel, which provides common low-level parallel computing primitives.

6.2.1 Distributed Containers

Blaze provides three distributed data containers: *DistRange*, *DistVector*, and *DistHashMap*. DistRange does not store the whole data but only the start, the end, and the step size of the data. DistVector distributedly stores an array of elements. DistHashMap distributedly stores key/value pairs.

All three containers support the `foreach` operation, where a custom function can be applied to each of its element in parallel. This function can either change the value of the element, or use the value of the element to perform

external operations.

Both the DistVector and the DistHashMap can be converted to and from C++ standard library containers with Blaze utility functions `distribute` and `collect`. DistVector can also be created from the `load_file` utility function, which can load text files from the file system in parallel into a distributed vector of lines.

DistVector also has a `topk` method, which can return the top k elements from the distributed vector in $O(n + k \log k)$ time and $O(k)$ space. Users can provide a custom comparison function to determine the priority of the elements.

6.2.2 MapReduce

The MapReduce function uses a functional style interface. It takes four parameters:

1. Input. One of the Blaze distributed containers.
2. Mapper. When the input is a DistRange, the mapper should be a function that accepts two parameters: a value from the DistRange and a handler function for emitting key/value pairs. When the input is a DistVector or a DistHashMap, the mapper should be a function that accepts three parameters: a key from the input, the corresponding value, and an emit handler.
3. Reducer. The function that reduce two values to one value. Blaze provides several built-in reducers, including `sum`, `prod`, `min`, and `max`, which can cover most use cases. These reducers can be used by providing the

reducer name as a string, for example, `"sum"`. Users can also provide custom reduce functions, which should take two parameters, the first one is a reference to the existing value which needs to be updated, and the second one is a constant reference to the new value.

4. Target. One of the Blaze distributed containers or a vector from the standard library. The target container should be mutable and it is not cleared before performing MapReduce. New results from the MapReduce operation are merged/reduced into the target container.

Blaze MapReduce also takes care of the serialization of common data types so that the map function can emit non-string key/value pairs, and the reduce function no longer requires additional logic for parsing the serialized data. Using custom data types as keys or values is also supported. For that, users only need to provide the corresponding serialize/parse methods and a hash function (for keys).

We provide two examples of using Blaze MapReduce in section 6.5.1 and 6.5.2.

6.2.3 Optimization

We introduce several optimizations to make the MapReduce function faster, including eager reduction, fast serialization, and special treatment for cases where the resulting key range is small and fixed.

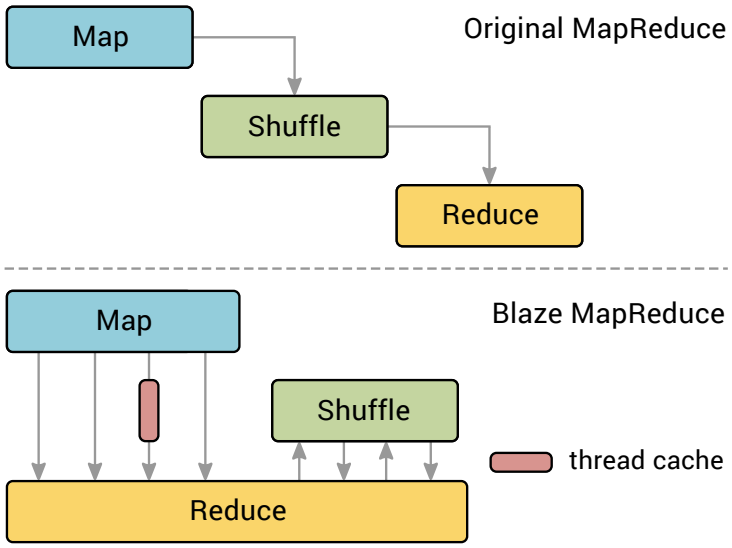

Figure 6.3: Eager Reduction in Blaze MapReduce.

Eager Reduction

Conventional MapReduce performs the map phase first and saves all the emitted pairs from the mapper function. Then, it shuffles all the emitted pairs across the networks directly, which could incur a large amount of network traffics.

In Blaze MapReduce, we perform machine-local reduce right after the mapper function emits a key/value pair. For popular keys, Blaze automatically reduces new values to a thread-local cache instead of the machine-local copy. The cross-machine shuffle operates on the locally reduced data which substantially reduces the network communication burden. During the shuffle operations, reduce functions are also operating asynchronously to maximize the throughput. Fig. 6.3 illustrates the difference between the conventional MapReduce and Blaze MapReduce with eager reduction.

Fast Serialization

During the shuffle/reduce phase, we serialize the messages into a compact binary format before casting them across the network.

Although these two fields allow missing fields and support serializing the fields in arbitrary order, this additional flexibility is not needed in MapReduce. On the other hand, these two fields can have a significant impact on both the performance and the serialized message size, especially when the content size of a field is small, which is common for MapReduce key/value pairs. For example, when both the key and value are small integers, the serialized message size of each pair from Protocol Buffers will be 4 bytes while the message from Blaze fast serialization will be only 2 bytes, which is 50% smaller than the one from Protocol Buffers. Removing the fields tags and wire types does not cause ambiguity as long as we always serialize the fields in the same order, which is easy to achieve in MapReduce. The smaller size in the serialized message means less network traffics, so that Blaze can scale better on large clusters when the cross-rack bandwidth becomes the bottleneck.

Optimization for Small Key Range

For small key range, we create a thread-local cache for each key at the beginning and set that as the reduce target during the local map/reduce phase. After the local map/reduce phase finishes, we perform parallel tree based reduce operations: first locally and then across multiple machines. The resulting execution plan is essentially the same as hand-optimized parallel for loops with thread-local intermediate results.

Table 6.1: Monte Carlo Pi Estimation Performance. We can see that Blaze MapReduce has almost the same speed as hand-optimized MPI+OpenMP parallel for loops while requires much fewer source lines of code (SLOC).

Samples	Blaze MapReduce	MPI+OpenMP
10^7	0.14 ± 0.01 s	0.14 ± 0.01 s
10^8	1.44 ± 0.07 s	1.42 ± 0.09 s
10^9	14.2 ± 1.3 s	14.6 ± 1.7 s
SLOC	8	24

We benchmark the performance of Blaze MapReduce against hand-optimized parallel for-loop on the Monte Carlo Pi estimation task. In this task, the mapper function first generates two random numbers x and y in the range $[0, 1]$, and then emits 1 to key 0 when $x^2 + y^2 < 1$. Cases like this where we reduce big data to a small number of keys are commonly seen in data mining and are not efficient with the original MapReduce algorithm. However, by using a thread-local copy as the default reduce target for each thread, Blaze MapReduce can achieve similar performance as hand-optimized code based on raw MPI and OpenMP. Table 6.1 reports the result and Section 6.5.2 lists our implementation. The tests are performed on a local machine with Ubuntu 16.04, GCC 5.4 -O3, and an Intel i7-8550U processor.

6.3 Applications

In this section, we benchmark Blaze against a popular data mining package Spark, on common data mining tasks, including word frequency count, PageR-

ank, k-means, expectation maximization (with the Gaussian Mixture model), and k-nearest neighbors search.

6.3.1 Task Description and Implementation

In this section, we describe the data mining tasks and how we implement them in Blaze and Spark. All the source code of our implementation is included in our GitHub repository [64].

Word Frequency Count

This task counts the number of occurrences of each unique English word in a text file. We use the Bible and Shakespeare's works as the testing text. Since Spark has significant overhead in starting the MapReduce tasks, we repeat the Bible and the Shakespeare 200 times, so that the input file contains about 0.4 billion words.

We use MapReduce in both Blaze and Spark. The mapper function takes a single line and emits multiple (word, 1) pairs. The reducer function sums the values. Section 6.5.1 contains the full Blaze implementation for this example.

PageRank

This task calculates the PageRank score, which is defined as the stationary value of the following equation:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)} \tag{6.1}$$

where $M(p_i)$ is the set of pages that link to p_i, $L(p_j)$ is the number of outbound links from page p_j, N is the total number of pages, and $d = 0.15$. When a page has no outbound links, it is called a sink and is assumed to connect to all the pages. We use the graph500 generator to generate the input graph which contains 10 million links. We set the convergence criterion to 10^{-5}, which results in 27 iterations for our input. The links are stored distributedly across multiple machines.

For Blaze, we use 3 MapReduce operations per iteration to implement this task. The first one calculates the total score of all the sinks. The second one calculates the new PageRank scores according to Eq. 6.1. The third one calculates the maximum change in the scores of all the pages. For Spark, we use the built-in PageRank module from the Spark GraphX library [114].

K-Means

K-Means is a popular clustering algorithm. The algorithm proceeds by alternating two steps until the convergence. The first step is the assignment step where each point is assigned to the nearest clustering center. The second step is the refinement step where each clustering center is updated based on the new mean of the points assigned to the clustering center.

We generate 100 million random points around 5 clustering centers as the testing data, and use the same initial model and convergence criteria for Spark and Blaze. The points are stored distributedly across multiple machines.

For Blaze, we use a single MapReduce operation to perform the assignment step. The update step is implemented in serial. For Spark, we use the built-in

implementation from the Spark MLlib library [75].

Expectation Maximization

This task uses the expectation maximization method to train the Gaussian Mixture clustering model (GMM). Starting from an initial model, we first calculate the Gaussian probability density of each point for each Gaussian component

$$p_k(\vec{x}|\theta_k) = \frac{1}{(2\pi)^{d/2}|\Sigma_k|^{1/2}} e^{-\frac{1}{2}(\vec{x}-\vec{\mu}_k)^T \Sigma_k^{-1}(\vec{x}-\vec{\mu}_k)} \tag{6.2}$$

where μ_1 to μ_K are the centers of these Gaussian components and Σ_1 to Σ_K are the covariance matrices. Then we calculate the membership of each point for each Gaussian component

$$w_{ik} = \frac{p_k(\vec{x}_i|\theta_k) \cdot \alpha_k}{\sum_{m=1}^{K} p_m(\vec{x}_i|\theta_m) \cdot \alpha_m} \tag{6.3}$$

where α_k is the weights of the Gaussian component. Next, we calculate the sum of membership weights for each Gaussian component $N_k = \Sigma_{i=1}^{K} w_{ik}$. After that, we update the parameters of the Gaussian mixtures

$$\alpha_k = \frac{N_k}{N} \tag{6.4}$$

$$\vec{\mu}_k = \left(\frac{1}{N_k}\right) \sum_{i=1}^{N} w_{ik}\vec{x}_i \tag{6.5}$$

$$\Sigma_k = \left(\frac{1}{N_k}\right) \sum_{i=1}^{K} w_{ik}(\vec{x}-\vec{\mu}_k)^T(\vec{x}-\vec{\mu}_k) \tag{6.6}$$

Finally, we calculate the log-likelihood of the current model for these points to determine whether the process is converged.

$$\sum_{i=1}^{N} \log p(\vec{x}_i|\Theta) = \sum_{i=1}^{N} \left(\log \sum_{k=1}^{K} \alpha_k p_k(\vec{x}_i|\theta_k)\right) \tag{6.7}$$

We generate 1 million random points around 5 clustering centers as the testing data and use the same initial model and convergence criteria for Spark and Blaze. The points are stored distributedly across multiple machines.

For Blaze, we implement this algorithm with 6 MapReduce operations per iteration. The first MapReduce calculates the probability density according to Eq. 6.2. The second MapReduce calculates the membership according to Eq. 6.3. The third MapReduce accumulates the sum of memberships for each Gaussian component N_k. The next two MapReduce perform the summations in Eq. 6.5 and Eq. 6.6. The last MapReduce calculates the log-likelihood according to Eq. 6.7. For Spark, we use the built-in implementation from the Spark MLlib library [75].

Nearest 100 Neighbors

In this task, we find the 100-nearest neighbors of a point from a huge set of other points. This is a common procedure in data analysis and recommendation systems. We use 200 million random points for this test.

For both Spark and Blaze, we implement this task with the *top k* function of the corresponding distributed containers and provide custom comparison functions to determine the relative priority of two points based on the Euclidean-distance.

6.3.2 Performance Analysis

We test the performance of both Spark and Blaze for the above tasks on Amazon Web Services (AWS). The time for loading data from the file system is not included in our measurements. Spark is explicitly set to use the `MEMORY_ONLY` mode and we choose memory-optimized instances r5.xlarge as our testing environments which have large enough memory for Spark to complete our tasks. Each r5.xlarge has 4 logical cores, 32GB memory, and up to 10 Gbps network performance.

For Spark, we use the AWS Elastic MapReduce (EMR) service version 5.20.0, which comes with Spark 2.4.0. Since, in the default setting, Spark changes the number of executors on the fly, which may obscure the results, we set the environment variable for maximizing resource allocation to true to avoid the change. We also manually specify the number of partitions to 100 to force the cross-executor shuffle on the entire cluster. For Blaze, we use GCC 7.3 with -O3 optimization and MPICH 3.2. For both Spark and Blaze, we perform warmup runs before counting the timings. Timings are converted to more meaningful results for each task.

The detailed performance comparison are shown in Fig. 6.4 to 6.8. "Spark", "Spark (MLlib)", "Spark (GraphX)", "Blaze", "Blaze TCM" denote the original Spark implementation, the MLlib library in Spark, the GraphX library in Spark, original Blaze, and Blaze linked with Thread-Caching Malloc (TCMalloc), respectively.

As shown in Fig. 6.4 to 6.8, Blaze outperforms Spark significantly on all five data mining applications. On average, Blaze is more than 10 times faster than

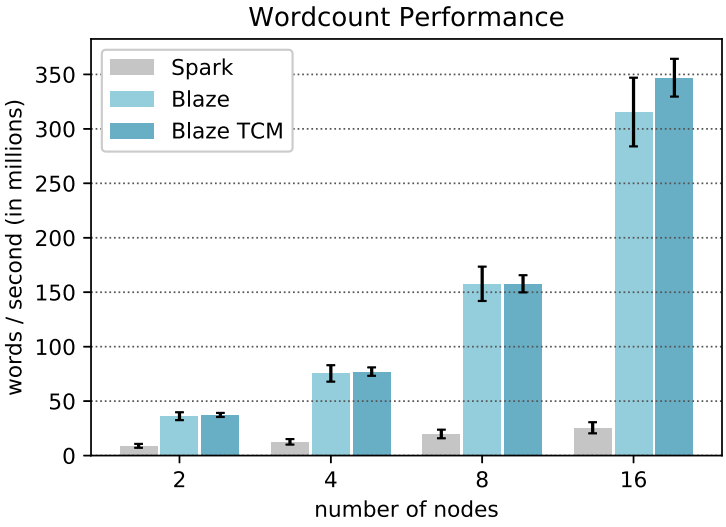

Figure 6.4: Performance of the word frequency count measured in the number of words processed per second.

Spark. The superior performance of Blaze shows that our highly-optimized implementation suits these data mining applications well. The performance difference between Blaze and Blaze TCM is negligible. However, without using TCMalloc, the performance has more fluctuations and can occasionally experience a significant drop of up to 30%.

6.3.3 Memory Consumption

We measure the memory consumption for running these tasks on a single local machine of 12 logical cores, using the same versions for all the software as the tests on AWS. As shown in Fig 6.9, we can see that both Blaze and Blaze TCM consume much less memory than Spark during the runs, especially for PageRank, K-Means, and expectation maximization (GMM), where Spark uses 10 times more memory than Blaze. The only case where the memory consump-

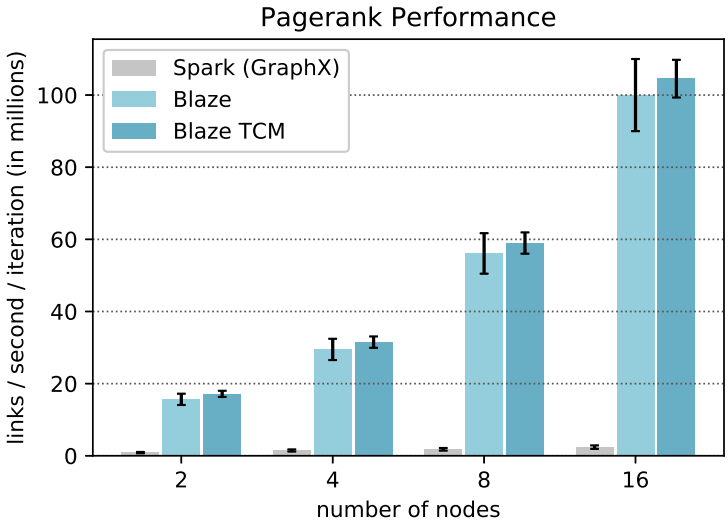

Figure 6.5: Performance of the PageRank algorithm measured in number of links processed per second per iteration.

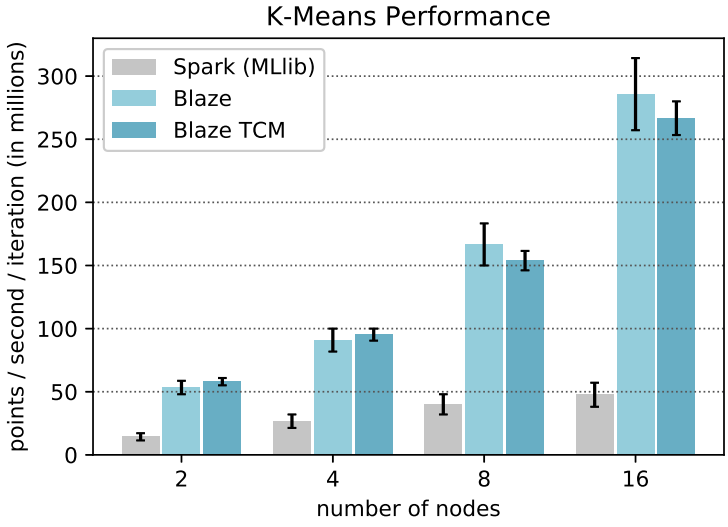

Figure 6.6: Performance of the K-Means algorithm measured in the number of points processed per second per iteration.

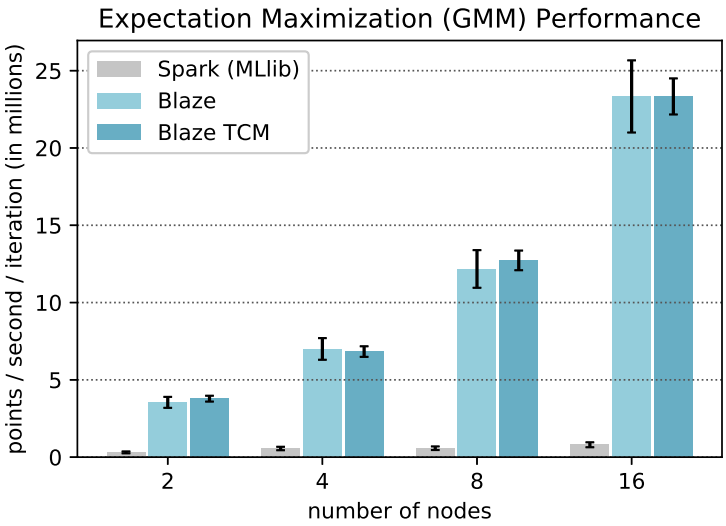

Figure 6.7: Performance of the Expectation Maximization algorithm for the Gaussian Mixture Model measured in the number of points processed per second per iteration.

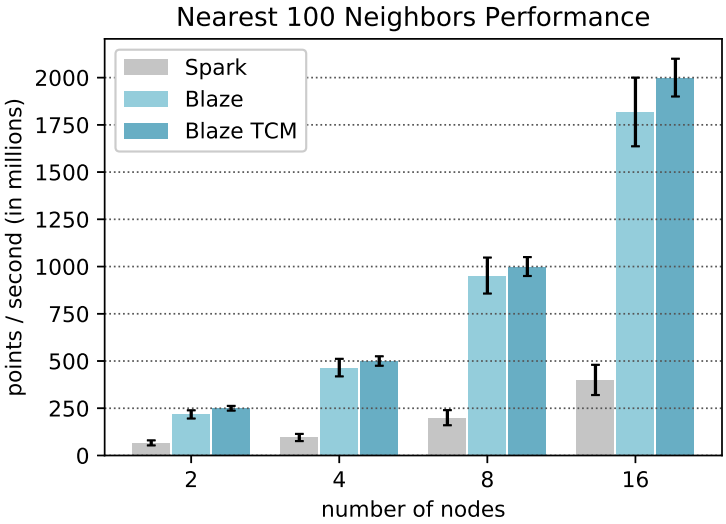

Figure 6.8: Performance of the Nearest 100 Neighbors search measured in the number of total points processed per second.

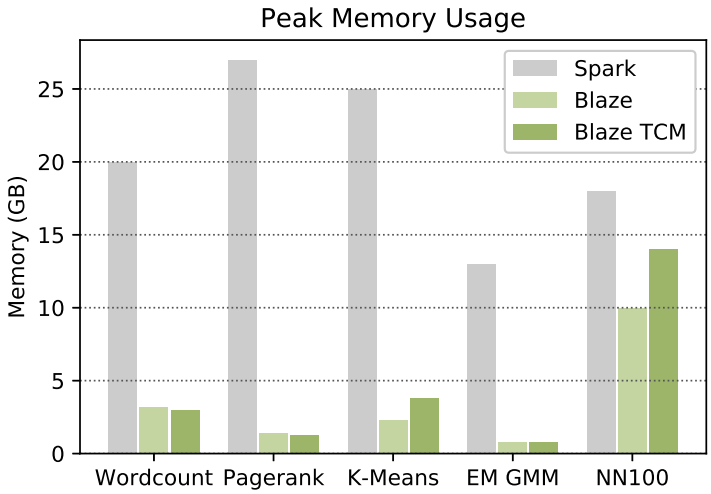

Figure 6.9: Peak memory usage on a single node.

tion between Spark and Blaze is close is the k-nearest neighbors search, which does not involve intermediate key/value pairs.

The memory consumption between Blaze and Blaze TCM are always on the same order of magnitude, although in one case, Blaze consumes 40% more memory when linked against TCMalloc.

6.3.4 Cognitive Load

Cognitive load refers to the effort needed to develop or understand the code. Minimizing the cognitive load is the ultimate goal that MapReduce and its variants try to achieve.

There are lots of different measures for cognitive effort. Source lines of code is not a good measure here as Spark/Scala supports chaining functions and can put several consecutive operations on a single line. Hence, a line of Spark/Scala

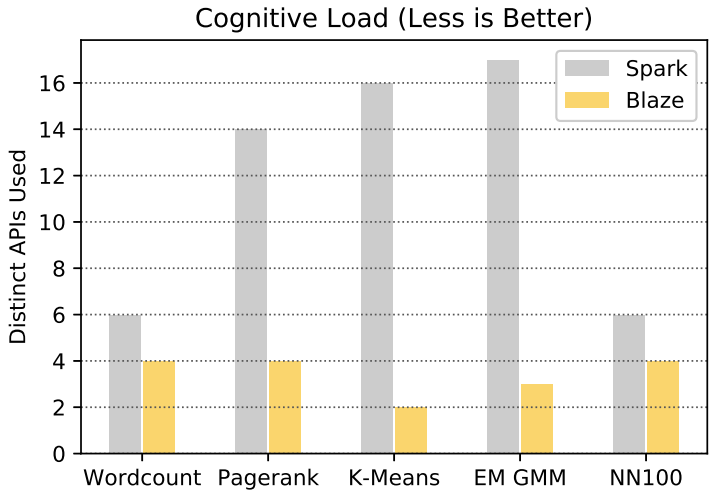

Figure 6.10: Cognitive load comparison between Blaze and Spark.

may be much more difficult to understand than a line of C++. Here we use the number of distinct APIs used as the indicator for cognitive load. It is a legitimate indicator because people will have to do more searches and remember more APIs when a library requires more distinct API calls to accomplish a task.

Spark's built-in implementation uses about 30 different parallel primitives for different tasks, while Blaze only uses the MapReduce function and less than 5 utility functions. We can see from Fig. 6.10 that the cognitive load of using Blaze is much smaller than using Spark.

6.4 Conclusion

Blaze provides a high performance implementation of MapReduce. Users can write parallel programs with Blaze's high-level MapReduce abstraction and achieve similar performance as the hand-optimized parallel code.

We use Blaze to implement 5 common data mining algorithms. By writing only a few lines of serial code and applying the Blaze MapReduce function, we achieve over 10 times higher performance than Spark on these compute intensive tasks, even though we only use the MapReduce function and 3 utility functions in our Blaze implementation while Spark uses almost 30 different parallel primitives for different tasks in its official implementation.

The high-level abstraction and the high performance makes Blaze an appealing choice for compute intensive tasks in data mining and related fields.

6.5 Examples

In this section, we provide two examples to illustrate the usage of Blaze. All the source code of our implementation is included in our GitHub repository [64].

6.5.1 Word frequency count

In this example, we count the number of occurrences of each unique word in an input file with Blaze MapReduce. We save the results in a distributed hash map, which can be used for further processing.

To compile this example, you can clone our repository [64], go to the `example` folder and type `make wordcount`.

```
#include <blaze/blaze.h>
#include <iostream>
```

```
int main(int argc, char** argv) {
  blaze::util::init(argc, argv);

  // Load file into distributed container.
  auto lines =
      blaze::util::load_file("filepath...");

  // Define mapper function.
  const auto& mapper = [&](
      const size_t,  // Line id.
      const std::string& line,
      const auto& emit) {
    // Split line into words.
    std::stringstream ss(line);
    std::string word;
    while (getline(ss, word, ' ')) {
      emit(word, 1);
    }
  };

  // Define target hash map.
  blaze::DistHashMap<std::string, size_t> words;

  // Perform mapreduce.
  blaze::mapreduce<
      std::string, std::string, size_t>(
          lines, mapper, "sum", words);

  // Output number of unique words.
```

```
  std::cout << words.size() << std::endl;

  return 0;
}
```

6.5.2 Monte Carlo Pi Estimation

In this example, we present a MapReduce implementation of the Monte Carlo π estimation.

To compile this example, you can clone our repository [64], go to the `example` folder and type `make pi`.

```
#include <blaze/blaze.h>
#include <iostream>

int main(int argc, char** argv) {
  blaze::util::init(argc, argv);

  const size_t N_SAMPLES = 1000000;

  // Define source.
  blaze::DistRange<size_t> samples(0, N_SAMPLES);

  // Define mapper.
  const auto& mapper =
      [&](const size_t, const auto& emit) {
    // Random function in std is not thread safe.
    double x = blaze::random::uniform();
```

```
    double y = blaze::random::uniform();
    // Map points within circle to key 0.
    if (x * x + y * y < 1) emit(0, 1);
  };

  // Define target.
  std::vector<size_t> count(1);  // {0}

  // Perform MapReduce.
  blaze::mapreduce<size_t, size_t>(
      samples, mapper, "sum", count);

  std::cout << 4.0 * count[0] / N_SAMPLES
      << std::endl;

  return 0;
}
```

In conventional MapReduce implementations, mapping big data onto a single key is usually slow and consumes a large amount of memory during the map phase. Hence, in practice, people usually hand-code parallel for loops in such situations. However, by using Blaze, the above code has similar memory consumption and performance as the hand-optimized parallel for loops. In short, Blaze frees users from dealing with low-level data communications while ensuring high performance.

CHAPTER 7

SCIENTIFIC SOFTWARE ENGINEERING IN SMALL RESEARCH GROUPS

With the development of computer technology, more and more scientific research involves developing and using scientific software. From the natural sciences, such as physics and chemistry, to the social sciences, such as economics and finance, software opens new fields for researchers and provides new results and insights that are impossible or too costly with conventional theoretical and experimental approaches. Therefore, many researchers are devoting increasingly more time and resources to software development, and this scientific software is becoming more and more complicated.

Software engineering is a computer science subject that studies how to manage complicated software development in a systematic and structured way so that software is easier to develop, maintain, and extend, and the software developers can cooperate more efficiently. Software engineering is especially crucial to scientific software development, because if the software has poor quality, the results may be wrong, which can lead to wrong conclusions and mislead future research.

In this chapter, I look back on my experience of developing quantum chemistry software for my PhD research and my experiences of working at Google, a software company famous for its excellent engineering practices, to provide a brief discussion of the differences between industrial software engineering and scientific software engineering, common industrial software engineering tools and practices, and their applicability in scientific software engineering.

7.1 Differences Between Industrial and Scientific Software

There are many differences between industrial and scientific software development. This section tries to discuss the ones that matter the most to the software engineering practices.

7.1.1 Feature Requirements

The clarity of the feature requirements is one of the biggest difference between industrial and scientific software development. In industrial software engineering, before writing the first line of code, project managers or user experience teams first come up with detailed designs through market research, and the software requirements for implementing the design are clearly documented in a design document written by professional managers and technical leaders. This is completely different from scientific research, where the input and output of the software often evolve during the research, developers have to make feature decisions when writing the code, and the algorithms involved are constantly changing to improve the accuracy and speed of the calculations.

In some rare cases, features in the industrial software also change during the development process. However, additional features can usually be incorporated without changing the overall architecture. In contrast, during the development of scientific software, as researchers dive deeper into the problem they are solving, it is common that they may use their existing code as a component to explore more general problems or related problems, which can require an overall restructuring of the program.

7.1.2 Users

We focus on software development in small research groups. For these groups, the software being developed is usually used internally. The users know each other, and the communication cost between developers and users is extremely low. Developers often focus on implementing new algorithms and using them to answer new scientific problems that have not been solved before and pay less attention to the user experience. However, in industrial software engineering, the communication cost between users and developers is extremely high and sometimes even impossible. Therefore, many industrial software engineering practices focus a lot on creating a smooth user experience, both for common use cases and edge cases. Many companies may even sacrifice new features or technologies until they become mature in order to avoid unexpected behaviors.

7.1.3 Lifecycle

In industry, the lifecycle of software is usually from several months to several years.

However, in scientific software engineering, depending on the nature of the research, some software developed in small research groups is used only during the period of a specific research project, while other software based on mature algorithms serve as basics tools for application-based research and may be used for much longer. It is worthwhile for researchers to determine the likely lifecycle of the software under development and apply software engineering techniques accordingly.

7.2 Applicability of Industrial Software Engineering Practices

7.2.1 Object Oriented Design

Object-oriented design groups related data and methods together into objects so that the software can have a modular structure, which makes the code easier to read, maintain, and extend. Many successful industrial products use object-oriented design, such as the Windows operating system and the Google internet services.

In scientific software engineering, there are also many related data and functions, and by grouping them into higher-level logical units with object-oriented programming, we can get a similar benefit as in the industrial cases.

Most object-oriented languages allow granular access control of each member variable and each member function of the objects. From my experience of developing SHCI, it is usually more convenient and efficient to relax the access control of data related objects in scientific programming. There are two reasons for this: 1) the nature of scientific research imposes lots of uncertainty as to whether some member variables in an object may be related to another object or not and making the access control more relaxed can make it much easier to experiment with new algorithms and speed up the research. 2) most software in scientific programming is only for internal users, and thus, there are many fewer risks associated with accessing private variables.

7.2.2 Unit Tests

Unit tests are common practices in industrial software engineering. Upon submitting new functions or classes, many software companies require each edge case of each function in each class to be addressed by a separate test case. Unit tests seem to slow down the development process in the short run, but can eventually speed up the entire product development process in the long run and increase the robustness of the products.

However, many codes in scientific software are often for experimental proposes and are likely to be used only once. Unit tests are mainly for preventing errors in the future when using that unit as a component, so there is no need to write unit tests for units that are not likely to be used in the future or used in other components.

For utility classes that are used as components in other classes, whether unit tests are beneficial depends mainly on the background of the group members. In many engineering research groups, members are proficient in programming, and students in these groups may frequently write unit tests after they graduate and enter the industrial world. In this case, enforcing unit tests on these utility components will not take much time, and can be a good opportunity for students to practice the skills of writing unit tests. For natural science or social science research groups, due to the lack of comprehensive computer science training, enforcing unit tests even just on the utility classes may be too costly, and it may be more efficient not to use them but instead put in the option to generate verbose outputs in each component.

7.2.3 Code Review

Code review is the practice of letting other team members review changes to a software system before merging these changes. It can help teams to identify defects and hard to read parts in the code earlier, and improve the quality of software significantly. It also makes sure that each line of code is known to several team members so that in the absence of some team members, the team can still make changes to the code quickly.

The main drawback of code review is that it takes some time to understand other's code, but for industrial software engineering, it almost always saves time in the long run due to the increased quality and high knowledge coverage of the code.

For scientific software engineering, code review is also helpful for the same reasons as for industrial software engineering. However, for some codes which involve fancy algorithms or specialized knowledge of a certain team member, other team members may need to spend a significant amount of time to acquire the background knowledge. Code review, in this case, can slow down the overall progress of the research significantly, but sometimes it is may still be beneficial to keep doing code review in such case for educational purposes.

7.2.4 Refactoring

Software development is usually an iterative process, and during incremental development, software changes drastically. These changes often cause the quality of the code to decrease. In order to improve the quality, many large software

companies, including Google, perform periodic refactoring of their codebase. During the refactoring periods, engineers focus on implement existing functions and features in a more readable, more maintainable way, and more robust way. Although they seem to stop making progress during these periods, long term productivity is improved.

Refactoring is especially important to scientific software engineering because when doing scientific programming, we are often not sure whether a new change is useful or not, so we tend to care less about the quality of the code and focus more on implementing and testing new ideas as soon as possible. However, this will result in a poor code quality, which makes software less maintainable. Hence, by keeping and refactoring what has been tested to be useful, the entire code base can become much more readable and reusable in the future. Even in the case when we are developing production components rather than experimental functions, it may still be more efficient to first focus on implementation to produce the correct results, and then refactor the implementation to make it easier to read and extend in the future.

7.2.5 Continuous Integration

Continuous integration automatically builds and tests the codebase before a change from a developer can be integrated into the codebase. This allows developers to detect errors in the code quickly before it causes trouble to end-users or other developers working on the same repository.

In scientific software engineering, many researchers focus on moving fast and getting results out as soon as possible. While efficiency is important to

small research groups, chasing efficiency too much may have an adverse effect and slow down the speed in the long run. Continuous integration is an easy way to make sure that after applying the new changes, the entire software can still build successfully, and the basic tests produce expected results. Depending on how stable and mature an algorithm is, developers can decide how many tests to include in the continuous integration so that they can achieve a balance between not breaking mature functions and testing new ideas quickly. In addition, the continuous integration configuration file can also be used as a guideline for setting up the software on a new computing environment, and a successful integration status ensures users that the software can be built successfully on a clean environment.

CHAPTER 8

CONCLUSION

Fast semistochastic heatbath configuration interaction (Fast SHCI) is an efficient algorithm for electronic structure calculations. It is more than an order of magnitude faster than other selected configuration interaction plus perturbation theory algorithms, and also much faster than other essentially exact quantum chemistry methods, such as DMRG or FCIQMC, in many cases.

The key reasons that fast SHCI is fast are:

- The heatbath criterion is easy to evaluate, and important determinants that meet this criterion can be found efficiently by using precomputed double excitation lists.
- The fast Hamiltonian construction algorithm significantly speeds up the Hamiltonian construction process by using helper arrays.
- The 3-step perturbation algorithm speeds up the perturbation correction calculation regardless of the memory of the system.

SHCI achieves accurate results on homogeneous electron gas in the mid to high-density region due to its high efficiency and ability to work with large basis sets. SHCI also successfully produces accurate results on huge Hilbert spaces of Chromium dimers.

Modular design makes the SHCI code easy to maintain and extend, and also produces several generic distributed computing building blocks, which makes writing high-performance parallel software much easier than using raw MPI routines directly.

Finally, regarding scientific software engineering, since scientific software is becoming more and more complex, it is crucial to keep the quality of the code high so that the scientific results from this software are reliable and the process of extending the software in order to experiment on new scientific ideas is efficient. We can borrow the experience from industrial software engineering, but due to the differences between industrial and scientific research, some common practices which are generally believed to be beneficial to industrial software engineering will have to be adjusted to really benefit scientific software development.

BIBLIOGRAPHY

[1] Boost C++ Libraries, https://www.boost.org/doc/libs/, (2012).

[2] Distributed hash table. https://en.wikipedia.org/wiki/Distributed_hash_table. Accessed: 2018-10-12.

[3] Hash functions. http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/1997/9709/9709n/9709n.htm. Accessed: 2019-05-20.

[4] Linear probing. https://en.wikipedia.org/wiki/Linear_probing. Accessed: 2018-10-12.

[5] Non-blocking algorithm. https://en.wikipedia.org/wiki/Non-blocking_algorithm. Accessed: 2018-10-12.

[6] Open addressing. https://en.wikipedia.org/wiki/Open_addressing. Accessed: 2018-10-12.

[7] Foto N Afrati and Jeffrey D Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 99–110. ACM, 2010.

[8] Prajesh P Anchalia, Anjan K Koundinya, and NK Srinath. Mapreduce design of k-means clustering algorithm. In *Information Science and Applications (ICISA), 2013 International Conference on*, pages 1–5. IEEE, 2013.

[9] Prajesh P Anchalia and Kaushik Roy. The k-nearest neighbor algorithm using mapreduce paradigm. In *Intelligent Systems, Modelling and Simulation (ISMS), 2014 5th International Conference on*, pages 513–518. IEEE, 2014.

[10] Bahman Bahmani, Kaushik Chakrabarti, and Dong Xin. Fast personalized pagerank on mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 973–984. ACM, 2011.

[11] Charles W. Bauschlicher and Phillippe Maitre. Theoretical study of the first transition row oxides and sulfides. *Theoretica chimica acta*, 90(2):189–203, January 1995.

[12] M. Chandler Bennett, Cody A. Melton, Abdulgani Annaberdiyev, Guangming Wang, Luke Shulenburger, and Lubos Mitas. A new generation of

effective core potentials for correlated calculations. *The Journal of Chemical Physics*, 147(22):224106, December 2017.

[13] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A Acar, and Rafael Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 7. ACM, 2011.

[14] Thomas Bligaard, R. Morris Bullock, Charles T. Campbell, Jingguang G. Chen, Bruce C. Gates, Raymond J. Gorte, Christopher W. Jones, William D. Jones, John R. Kitchin, and Susannah L. Scott. Toward Benchmarking in Catalysis Science: Best Practices, Challenges, and Opportunities. *ACS Catalysis*, 6(4):2590–2602, April 2016.

[15] G. H. Booth, A. Grüneis, G. Kresse, and A. Alavi. Towards an exact description of electronic wavefunctions in real solids. *Nature*, 493:365, 2013.

[16] George H Booth, Alex JW Thom, and Ali Alavi. Fermion monte carlo without fixed nodes: A game of life, death, and annihilation in slater determinant space. *J. Chem. Phys.*, 131(5):054106, 2009.

[17] Robert J Buenker and Sigrid D Peyerimhoff. Individualized configuration selection in ci calculations with subsequent energy extrapolation. *Theor. Chim. Acta*, 35(1):33–58, 1974.

[18] L. Bytautas and K. Ruedenberg. A priori identification of configurational deadwood. *Chem. Phys.*, 356:64–75, 2009.

[19] Sean M Casey and Doreen G Leopold. Negative ion photoelectron spectroscopy of chromium dimer. *The Journal of Physical Chemistry*, 97(4):816–830, 1993.

[20] D. M. Ceperley and B. J. Alder. *Phys. Rev. Lett.*, 45:566, 1980.

[21] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[22] Garnet Kin-Lic Chan and Martin Head-Gordon. Highly correlated calculations with a polynomial cost algorithm: A study of the density matrix renormalization group. *J. Chem. Phys.*, 116(11):4462–4476, 2002.

[23] Garnet Kin-Lic Chan and Sandeep Sharma. The density matrix renormalization group in quantum chemistry. *Annu. Rev. Phys. Chem.*, 62:465–481, 2011.

[24] Zhiwen Chen, Xin He, Jianhua Sun, Hao Chen, and Ligang He. Concurrent hash tables on multicore machines: Comparison, evaluation and implications. *Future Generation Computer Systems*, 82:127–141, 2018.

[25] Alan D. Chien, Adam A. Holmes, Matthew Otten, C. J. Umrigar, Sandeep Sharma, and Paul M. Zimmerman. Excited states of methylene, polyenes, and ozone from heat-bath configuration interaction. *J. Phys. Chem. A*, 122:2714, 2018.

[26] Cheng-Tao Chu, Sang K Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Kunle Olukotun, and Andrew Y Ng. Map-reduce for machine learning on multicore. In *Advances in neural information processing systems*, pages 281–288, 2007.

[27] Renzo Cimiraglia and Maurizio Persico. Recent advances in multireference second order perturbation ci: The cipsi method revisited. *J. Comp. Chem.*, 8(1):39–47, 1987.

[28] Deidre Cleland, George H Booth, and Ali Alavi. Communications: Survival of the fittest: Accelerating convergence in full configuration-interaction quantum monte carlo. *J. Chem. Phys.*, 132(4):041103, 2010.

[29] J.P. Coe, P. Murphy, and M.J. Paterson. Applying Monte Carlo configuration interaction to transition metal dimers: Exploring the balance between static and dynamic correlation. *Chem. Phys. Lett.*, 604:46, 2014.

[30] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Nsdi*, volume 10, page 20, 2010.

[31] Xiaoli Cui, Pingfei Zhu, Xin Yang, Keqiu Li, and Changqing Ji. Optimized big data k-means clustering using mapreduce. *The Journal of Supercomputing*, 70(3):1249–1259, 2014.

[32] Ernest R Davidson. Super-matrix methods. *Computer Physics Communications*, 53(1-3):49–60, 1989.

[33] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[34] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[35] Katharina Doblhoff-Dier, Jörg Meyer, Philip E. Hoggan, Geert-Jan Kroes, and Lucas K. Wagner. Diffusion Monte Carlo for Accurate Dissociation Energies of 3d Transition Metal Containing Molecules. *Journal of Chemical Theory and Computation*, 12(6):2583–2597, June 2016.

[36] R. M. Dreizler and E. K. U. Gross. *Density Functional Theory*. Springer-Verlag, Berlin, 1990.

[37] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM international symposium on high performance distributed computing*, pages 810–818. ACM, 2010.

[38] P. S. Epstein. *Phys. Rev.*, 28:695, 1926.

[39] F. A. Evangelista. Adaptive multiconfigurational wave functions. *J. Chem. Phys.*, 140:124114, 2014.

[40] Stefano Evangelisti, Jean-Pierre Daudey, and Jean-Paul Malrieu. Convergence of an improved cipsi algorithm. *Chem. Phys.*, 75(1):91–102, 1983.

[41] Apache Software Foundation. Apache hadoop. https://hadoop.apache.org/, 2019. Accessed: 2019-02-01.

[42] Apache Software Foundation. Apache spark - unified analytics engine for big data. https://spark.apache.org/, 2019. Accessed: 2019-02-01.

[43] Filipp Furche and John P. Perdew. The performance of semilocal and hybrid density functionals in 3d transition-metal chemistry. *The Journal of Chemical Physics*, 124(4):044103, January 2006.

[44] Yann Garniron, Anthony Scemama, Pierre-François Loos, and Michel Caffarel. Hybrid stochastic-deterministic calculation of the second-order perturbative contribution of multireference perturbation theory. *J. Chem. Phys.*, 147:034101, 2017.

[45] Gabriele Giuliani and Giovanni Vignale. *Quantum theory of the electron liquid*. Cambridge university press, 2005.

[46] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 383–397. ACM, 2015.

[47] Satish Gopalani and Rohan Arora. Comparing apache spark and map reduce with performance analysis using k-means. *International journal of computer applications*, 113(1), 2015.

[48] Sheng Guo, Zhendong Li, and Garnet Kin-Lic Chan. A perturbative density matrix renormalization group algorithm for large active spaces. *J. Chem. Theory Comput.*, 2018.

[49] Sheng Guo, Mark A. Watson, Weifeng Hu, Qiming Sun, and Garnet Kin-Lic Chan. N-electron valence state perturbation theory based on a density matrix renormalization group reference function, with applications to the chromium dimer and a trimer model of poly(p-phenylenevinylene). *J. Chem. Theory Comput.*, 12(4):1583–1591, 2016.

[50] R. J. Harrison. Approximating full configuration interaction with selected configuration interaction and perturbation theory. *J. Chem. Phys.*, 94:5021–5031, 1991.

[51] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Parallel Architectures and Compilation Techniques (PACT), 2008 International Conference on*, pages 260–269. IEEE, 2008.

[52] A. A. Holmes, Norm M. Tubman, and C. J. Umrigar. Heat-bath configuration interaction: An efficient selected ci algorithm inspired by heat-bath sampling. *J. Chem. Theory Comput.*, 12:3674, 2016.

[53] Adam A. Holmes, Hitesh J. Changlani, and C. J. Umrigar. Efficient heat-bath sampling in fock space. *J. Chem. Theory Comput.*, 2016.

[54] Adam A. Holmes, C. J. Umrigar, and Sandeep Sharma. Excited states using semistochastic heat-bath configuration interaction. *J. Chem. Phys.*, 147(16):164111, 2017.

[55] B Huron, JP Malrieu, and P Rancurel. Iterative perturbation calculations of ground and excited state energies from multiconfigurational zeroth-order wavefunctions. *J. Chem. Phys.*, 58(12):5745–5759, 1973.

[56] Joseph Ivanic and Klaus Ruedenberg. Identification of deadwood in configuration spaces through general direct configuration interaction. *Theor Chem Acc*, 106:339, 2001.

[57] Erin R. Johnson and Axel D. Becke. Communication: DFT treatment of strong correlation in 3d transition-metal diatomics. *The Journal of Chemical Physics*, 146(21):211105, June 2017.

[58] Thomas P. Kelly, Ajith Perera, Rodney J. Bartlett, and James C. Greer. Monte Carlo configuration interaction with perturbation corrections for dissociation energies of first row diatomic molecules: C_2, N_2, O_2, CO. *J. Chem. Phys.*, 140:084114, 2014.

[59] P.J. Knowles and N.C. Handy. A new determinant-based full configuration interaction method. *Chem. Phys. Lett.*, 111(4-5):315–321, 1984.

[60] Walter Kohn. Nobel lecture: Electronic structure of matterwave functions and density functionals. *Rev. Mod. Phys.*, 71(5):1253, 1999.

[61] Richard A Kronmal and Arthur V Peterson Jr. On the alias method for generating random variables from a discrete distribution. *Amer. Statist.*, 33(4):214–218, 1979.

[62] Y Kurashige and T Yanai. Second-order perturbation theory with a density matrix renormalization group self-consistent field reference function: Theory and application to the study of chromium dimer. *J. Chem. Phys.*, 135(9):094104, 2011.

[63] Yongkyung Kwon, DM Ceperley, and Richard M Martin. Effects of backflow correlation in the three-dimensional electron gas: Quantum monte carlo study. *Physical Review B*, 58(11):6800, 1998.

[64] Junhao Li. Blaze. https://github.com/junhao12131/blaze, 2019. Accessed: 2019-02-01.

[65] Junhao Li, Matthew Otten, Adam A. Holmes, Sandeep Sharma, and C. J. Umrigar. Fast semistochastic heat-bath configuration interaction. *J. Chem. Phys.*, 148:214110, 2018.

[66] Junhao Li, Matthew Otten, Adam A Holmes, Sandeep Sharma, and Cyrus J Umrigar. Fast semistochastic heat-bath configuration interaction. *The Journal of chemical physics*, 149(21):214110, 2018.

[67] Songze Li, Mohammad Ali Maddah-Ali, and A Salman Avestimehr. Coded mapreduce. In *Communication, Control, and Computing (Allerton), 2015 53rd Annual Allerton Conference on*, pages 964–971. IEEE, 2015.

[68] Pierre-François Loos, Anthony Scemama, Aymeric Blondel, Yann Garniron, Michel Caffarel, and Denis Jacquemin. A mountaineering strategy to excited states: Highly accurate reference energies and benchmarks. *J. Chem. Theory Comput.*, 14:43604379, 2018.

[69] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *Proceedings of the VLDB Endowment*, 5(10):1016–1027, 2012.

[70] Hongjun Luo and Ali Alavi. Combining the transcorrelated method with full configuration interaction quantum monte carlo: application to the homogeneous electron gas. *Journal of chemical theory and computation*, 14(3):1403–1411, 2018.

[71] Dongxia Ma, Giovanni Li Manni, Jeppe Olsen, and Laura Gagliardi. Second-Order Perturbation Theory for Generalized Active Space Self-Consistent-Field Wave Functions. *J. Chem. Theory Comput.*, 12:3208, 2016.

[72] Jesús Maillo, Isaac Triguero, and Francisco Herrera. A mapreduce-based k-nearest neighbor approach for big data classification. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 2, pages 167–172. IEEE, 2015.

[73] Cascading maintainers. Cascading. https://www.cascading.org/, 2019. Accessed: 2019-02-01.

[74] Narbe Mardirossian and Martin Head-Gordon. Thirty years of density functional theory in computational chemistry: an overview and extensive assessment of 200 density functionals. *Molecular Physics*, 115(19):2315–2372, October 2017.

[75] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[76] Yury Minenkov, Edrisse Chermak, and Luigi Cavallo. Troubles in the Systematic Prediction of Transition Metal Thermochemistry with Contemporary Out-of-the-Box Methods. *Journal of Chemical Theory and Computation*, 12(4):1542–1560, April 2016.

[77] Bastien Mussard and Sandeep Sharma. One-step treatment of spin–orbit coupling and electron correlation in large active spaces. *J. Chem. Theory Comput.*, 14(1):154–165, 2017.

[78] R. K. Nesbet. *Proc. R. Soc. London, Ser. A.*, 230:312, 1955.

[79] Verena A Neufeld and Alex JW Thom. A study of the dense uniform electron gas with high orders of coupled cluster. *arXiv preprint arXiv:1706.09923*, 2017.

[80] Roberto Olivares-Amaya, Weifeng Hu, Naoki Nakatani, Sandeep Sharma, Jun Yang, and Garnet Kin-Lic Chan. The ab-initio density matrix renormalization group in practice. *J. Chem. Phys.*, 142(3):034102, 2015.

[81] R. G. Parr and W. Yang. *Density-Functional Theory of Atoms and Molecules*. Oxford University Press, New York, 1989.

[82] F. R. Petruzielo, A. A. Holmes, Hitesh J. Changlani, M. P. Nightingale, and C. J. Umrigar. Semistochastic projector monte carlo method. *Phys. Rev. Lett.*, 109(23):230201, 2012.

[83] Steven J Plimpton and Karen D Devine. Mapreduce in mpi for large-scale graph algorithms. *Parallel Computing*, 37(9):610–632, 2011.

[84] Wirawan Purwanto, Shiwei Zhang, and Henry Krakauer. An auxiliary-field quantum monte carlo study of the chromium dimer. *J. Chem. Phys.*, 142:064302, 2015.

[85] Krishnan Raghavachari, Gary W Trucks, John A Pople, and Martin Head-Gordon. A fifth-order perturbation comparison of electron correlation theories. *Chemical Physics Letters*, 157(6):479–483, 1989.

[86] P. Lopez Rios, A. Ma, N. D. Drummond, M. D. Towler, and R. J. Needs. Inhomogeneous backflow transformations in quantum Monte Carlo calculations. *Phys. Rev. E*, 74, Dec 2006.

[87] P López Ríos, A Ma, ND Drummond, MD Towler, and RJ Needs. Inhomogeneous backflow transformations in quantum monte carlo calculations. *Physical Review E*, 74(6):066701, 2006.

[88] A. Savin. On degeneracy, near degeneracy and density functional theory. In J. M. Seminario, editor, *Recent Developments of Modern Density Functional Theory*, pages 327–357. Elsevier, Amsterdam, 1996.

[89] Anthony Scemama, Thomas Applencourt, Emmanuel Giner, and Michel Caffarel. Quantum monte carlo with very large multideterminant wavefunctions. *J. Comp. Chem.*, 37(20):1866–1875, 2016.

[90] Ulrich Schollwöck. The density-matrix renormalization group. *Rev. Mod. Phys.*, 77(1):259, 2005.

[91] Gustavo E. Scuseria. Analytic evaluation of energy gradients for the singles and doubles coupled cluster method including perturbative triple excitations: Theory and applications to FOOF and Cr_2. *J. Chem. Phys.*, 94:442, 1991.

[92] Sandeep Sharma and Garnet Kin-Lic Chan. Spin-adapted density matrix renormalization group algorithms for quantum chemistry. *J. Chem. Phys.*, 136(12), Mar 28 2012.

[93] Sandeep Sharma, Adam A. Holmes, Guillaume Jeanmairet, Ali Alavi, and C. J. Umrigar. Semistochastic heat-bath configuration interaction method: Selected configuration interaction with semistochastic perturbation theory. *J. Chem. Theory Comput.*, 13(4):1595–1604, 2017.

[94] James J. Shepherd, George Booth, Andreas Grueneis, and Ali Alavi. Full configuration interaction perspective on the homogeneous electron gas. *Phys. Rev. B*, 85(8), 2012.

[95] James J Shepherd, George Booth, Andreas Grüneis, and Ali Alavi. Full configuration interaction perspective on the homogeneous electron gas. *Physical Review B*, 85(8):081103, 2012.

[96] Benoit Simard, Marie-Ange Lebeault-Dorget, Adrian Marijnissen, and JJ Ter Meulen. Photoionization spectroscopy of dichromium and dimolybdenum: Ionization potentials and bond energies. *J. Chem. Phys.*, 108(23):9668–9674, 1998.

[97] James ET Smith, Bastien Mussard, Adam A Holmes, and Sandeep Sharma. Cheap and near exact casscf with large active spaces. *J. Chem. Theory Comput.*, 13(11):5468–5478, 2017.

[98] Stephen Stuart and Rex Fernando. Encoding rules and mime type for protocol buffers. *https://tools.ietf.org/html/draft-rfernando-protocol-buffers-00*, 2012.

[99] Qiming Sun, Timothy C Berkelbach, Nick S Blunt, George H Booth, Sheng Guo, Zhendong Li, Junzi Liu, James McClain, Sandeep Sharma, Sebastian Wouters, and Garnet Kin-Lic Chan. Pyscf: The python-based simulations of chemistry framework. *WIREs Comput. Mol. Sci.*, 8:e1340, 2018.

[100] David P. Tew. Explicitly correlated coupled-cluster theory with Brueckner orbitals. *The Journal of Chemical Physics*, 145(7):074103, August 2016.

[101] Robert E. Thomas, George H. Booth, and Ali Alavi. Accurate Ab Initio Calculation of Ionization Potentials of the First-Row Transition Metals with the Configuration-Interaction Quantum Monte Carlo Technique. *Physical Review Letters*, 114(3):033001, January 2015.

[102] J. R. Trail and R. J. Needs. Pseudopotentials for correlated electron systems. *The Journal of Chemical Physics*, 139(1):014101, July 2013.

[103] J. R. Trail and R. J. Needs. Correlated electron pseudopotentials for 3d-transition metals. *J. Chem. Phys.*, 142:064110, 2015.

[104] J. R. Trail and R. J. Needs. Correlated electron pseudopotentials for 3d-transition metals. *The Journal of Chemical Physics*, 142(6):064110, February 2015.

[105] Norm M. Tubman, Daniel S. Levine, Diptarka Hait, Martin Head-Gordon, and K. Birgitta Whaley. An efficient deterministic perturbation theory for selected configuration interaction methods. *https://arxiv.org/pdf/1808.02049.pdf*.

[106] Steven Vancoillie, Per Ake Malmqvist, and Valera Veryazov. Potential energy surface of the chromium dimer re-re-revisited with multiconfigurational perturbation theory. *J. Chem. Theory Comput.*, 12:1647, 2016.

[107] Pragya Verma, Zoltan Varga, Johannes E. M. N. Klein, Christopher J. Cramer, Lawrence Que Jr, and Donald G. Truhlar. Assessment of Elec-

tronic Structure Methods for the Determination of the Ground Spin States of Fe(II), Fe(III) and Fe(IV) Complexes. *Physical Chemistry Chemical Physics*, May 2017.

[108] Alastair J Walker. An efficient method for generating discrete random variables with general distributions. *ACM Trans. on Math. Software (TOMS)*, 3(3):253–256, 1977.

[109] Steven R White. Density-matrix algorithms for quantum renormalization groups. *Physical Review B*, 48(14):10345, 1993.

[110] Steven R White and Richard L Martin. Ab initio quantum chemistry using the density matrix renormalization group. *J. Chem. Phys.*, 110(9):4127–4130, 1999.

[111] Wikipedia contributors. Binary search tree — wikipedia, the free encyclopedia, 2019. [Online; accessed 23-April-2019].

[112] Wikipedia contributors. Redblack tree — wikipedia, the free encyclopedia, 2019. [Online; accessed 23-April-2019].

[113] Wikipedia contributors. Tree traversal — wikipedia, the free encyclopedia, 2019. [Online; accessed 23-April-2019].

[114] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[115] Enhua Xu, Motoyuki Uejima, and Seiichiro Lenka Ten-no. Full Coupled-Cluster Reduction for Accurate Description of Strong Electron Correlation. *Phys. Rev. Lett.*, 121:113001, 2018.

[116] Xuefei Xu, Wenjing Zhang, Mingsheng Tang, and Donald G. Truhlar. Do Practical Standard Coupled Cluster Calculations Agree Better than Kohn–Sham Calculations with Currently Available Functionals When Compared to the Best Available Experimental Data for Dissociation Energies of Bonds to 3d Transition Metals? *Journal of Chemical Theory and Computation*, 11(5):2036–2052, May 2015.

[117] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D Stott Parker. Mapreduce-merge: simplified relational data processing on large clusters. In

Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pages 1029–1040. ACM, 2007.

[118] Takuya Yokoyama, Yoshiharu Ishikawa, and Yu Suzuki. Processing all k-nearest neighbor queries in hadoop. In *International Conference on Web-Age Information Management*, pages 346–351. Springer, 2012.

[119] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[120] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[121] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Osdi*, volume 8, page 7, 2008.

[122] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[123] Weizhong Zhao, Huifang Ma, and Qing He. Parallel k-means clustering based on mapreduce. In *IEEE International Conference on Cloud Computing*, pages 674–679. Springer, 2009.