INAUGURAL - DISSERTATION

zur Erlangung der Doktorwürde der Gesamtfakultät

für Mathematik, Ingenieur- und Naturwissenschaften der

Ruprecht-Karls-Universität Heidelberg

Superlinear Parallel Scaling of Quadrature Evaluated Second-Order Møller-Plesset Perturbation Theory

vorgelegt von

Benjamin Thomitzni, M. Sc. Chemie

aus Öhringen

im Januar 2023

Gutachter:

Prof. Dr. Andreas Dreuw

Prof. Dr. Oriol Vendrell

Datum der mündlichen Prüfung: 28.02.2023

The source code presented in this thesis is freely available at

https://github.com/BenTho-Uni/libqqc

Abstract

Most resources of modern computer clusters are locked behind the need for a high degree of parallel efficiency, having millions of processing units on heterogeneous hardware, e.g. having both CPUs and GPUs. Post-HF wave function-based methods offer a high accuracy while having high computational scaling and memory requirements. Their need for storing, transforming, manipulating, and communicating their underlying high-dimensional quantities makes them a bad fit for modern computer architecture.
The second-order Møller-Plesset perturbation theory (MP2) is a commonly used method to recover the electron correlation energy that the Hartree-Fock method is missing. Its computational scaling is $\mathcal{O}(N^5)$ (N representing the number of atomic orbitals), as its two-electron integrals have to be transformed from their atomic orbital representation to their molecular orbital representation, and its memory requirements scale with $\mathcal{O}(N^4)$.
This thesis applies the ideas of a quadrature scheme[1–4] to the MP2 method to arrive at a lower scaling form that is embarrassingly parallel. The full Q-MP2 energy integral scales with $\mathcal{O}(P^2OV)$ (spatial grid points P, occupied molecular orbitals O, virtual molecular orbitals V) with its largest entity needing $\mathcal{O}(PN^2)$ memory. In this thesis, an efficient efficient implementation in the form of the open-source `libqqc` library is presented.
The setup of the integration grids is the deciding factor of the accuracy of the Q-MP2 method. To investigate this a benchmark of small molecules is shown, consisting of seven molecules, three basis sets, and 28 different grid combinations. All but the smallest grid combinations are shown to be within the magnitude of the target chemical accuracy, with only four points on the one-dimensional integration grid being necessary. The error of the spatial grid falls asymptotically with the number of grid points, with the third smallest grid of 20 radial and 38 angular points being chosen for further tests as it is the smallest well-behaved one.
The total single-node performance, measured as the number of floating point operations performed per second compared to the theoretical maximum, of the different variants of the algorithm is found to be below 15%. The algorithm is memory bound. The parallelisation strategy shows near-perfect load balancing over the computational nodes. The single-node parallel efficiency is shown to be superlinear for large systems, as a higher percentage of memory can be stored in low-level and fast memory caches with an increasing number of cores. This trend is followed

at the multi-node level, which was investigated for up to 960 cores/20 nodes on the `JUSTUS 2` computer cluster.
Future optimisation strategies will be focused on optimising the integration grids to lower the number of necessary integration grid points, integral screening, better utilisation of temporal locality, and exploitation of matrix sparsity. Finally, the quadrature scheme was extended to coupled-cluster theory (Q-CC2) and the algebraic-diagrammatic construction scheme for the polarisation propagator (Q-ADC(2)). For the latter method, the computational scaling associated with the solution of the particle-hole state was lowered from originally $\mathcal{O}(N^5)$ to $\mathcal{O}(P^2OV^2)$ and the memory requirement can be additionally lowered from $\mathcal{O}(N^4)$ to $\mathcal{O}(PN^2)$ by folding of the doubles space into the singles space. Compared to the performance of Q-MP2, a future implementation is expected to have a better single-node performance as more computational work needs to be done per memory transaction, and to have similar parallel efficiency as little additional node-to-node communication is necessary.

Zusammenfassung

Moderne Rechencluster besitzen üblicherweise Millionen von Recheneinheiten auf heterogener Hardware, wie etwa CPUs oder GPUs. Damit ein Programm diese Leistung vollständig nutzen kann, ist ein hohes Maß an paralleler Effizienz gefordert. Wellenfunktionsbasierte Methoden auf Grundlage der Hartree-Fock-Nährung ermöglichen eine hohe Genauigkeit, sind jedoch üblicherweise gekennzeichnet durch eine hohe Skalierung ihres Rechenaufwands und hohe Speicheranforderungen. Die ihnen zugrunde liegenden hoch-dimensionalen Datenstrukturen, wie z.Bsp. Zweielektronenintegrale, müssen gespeichert, transformiert, manipuliert und kommuniziert werden, was sie ungünstig für moderne Computerarchitektur macht.
Die Møller-Plesset-Störungstheorie der zweiten Ordnung (MP2) ist eine oft verwendete Methode zur Berechnung der Elektronenkorrelationsenergie, welche der Hartree-Fock Methode fehlt. Aufgrund der notwendigen Transformation der Zweielektronenintegrale von der Atomorbital- in die Molekülorbitalbasis skaliert ihr Rechenaufwand mit $\mathcal{O}(N^5)$ (mit der Anzahl an Atomorbitalen N). Ihr Speicherbedarf skaliert mit $\mathcal{O}(N^4)$.
Die vorliegende Arbeit wendet die Ideen eines Quadraturschemas[1–4] auf die MP2-Methode an und erreicht damit eine niedrigere Skalierung und eine natürlich parallelisierbare Form. Das komplette Q-MP2 Energieintegral hat einen Rechenaufwand welcher mit $\mathcal{O}(P^2OV)$ (räumliche Gitterpunkte P, besetzte Molekülorbitale O, virtuelle Molekülorbitale V) skaliert und einen Speicherbedarf von $\mathcal{O}(PN^2)$. Der Autor präsentiert mit dieser Arbeit eine effiziente Implementierung in Form der Open-Source `libqqc` Bibliothek.
Die Struktur der Integrationsgitter ist der für die Genauigkeit entscheidende Faktor der Q-MP2 Methode. Um diese zu untersuchen, wird ein Benchmark von kleinen Molekülen vorgestellt, welcher aus sieben Systemen, drei Basissätzen und 28 verschiedenen Integrationsgittern besteht. Alle bis auf das kleinste Gitter konnten die angepeilte chemische Genauigkeit erreichen, wobei nur vier Punkte für das eindimensionale Integrationsgitter ausreichend sind. Der Fehler des räumlichen Gitters nimmt mit steigender Anzahlan Quadraturpunkten asymptotisch ab, wobei das dritt-kleinste Gitter mit 20 Radial- und 38 Winkelpunkten für die weiteren Tests verwendet wurde, da diese Kombination das kleinste Gitter mit regulärem Verhalten ist.
Die absolute Leistung der verschiedenen Varianten des Algorithmus auf einem

einzelnen Rechenknoten, gemessen an der Anzahl ausgeführter Fließkommarechenoperationen pro Sekunde im Vergleich zum theoretischen Maximum, liegt bei unter 15%. Der Algorithmus ist durch seine Speicheranforderung limitiert. Die Parallelisierungsstrategie zeigt eine beinahe perfekte Verteilung des Arbeitsaufwandes zwischen den Rechenknoten. Die Ein-Knoten-Paralleleffizienz zeigt superlineares Verhalten für große Systeme, da ein größerer Anteil der benötigten Daten mit steigender Anzahl an Kernen in schnellerem Low-Level-Speicher verbleibt. Dieser Trend zeigt sich ebenfalls bei der gleichzeitigen Verwendung mehrerer Rechenknoten. Dabei wurde die Verwendung auf bis zu 960 Kernen/20 Rechenknoten des `JUSTUS 2` Rechenclusters getestet.
Zukünftige Optimierungsstrategien werden darauf fokussiert sein die Integrationsgitter zu optimieren, um die Anzahl an Gitterpunkten zu reduzieren, unnötige Integrale auszusieben, die zeitliche Lokalität der Daten und die Struktur dünnbesetzter Matritzen besser auszunutzen. Schlussendlich wurde das Quadraturschema auch auf die Coupled-Cluster Methode (Q-CC2) und das algebraisch-diagrammatische Konstruktionsschema des Polarisationspropagators (Q-ADC(2)) angewandt. Der Rechenaufwand zur Berechnung einfach angeregter Zustände von Letzterem wurde von $\mathcal{O}(N^5)$ auf $\mathcal{O}(P^2OV^2)$ reduziert und der Speicherbedarf kann durch das zusätzliche Falten des Doppelanregungsblocks in den Einfachanregungsblock von $\mathcal{O}(N^4)$ auf $\mathcal{O}(PN^2)$ reduziert werden. Verglichen mit Q-MP2 kann von einer zukündigen Q-ADC(2)-Impelentierung erwartet werden, dass sie eine höhere absolute Ein-Knoten-Leistung zeigt, da mehr Rechenarbeit pro Datentransaktion ausgeführt wird. Ebenso wird erwartet, dass sie eine ähnliche parallele Effizienz zeigt, da nur geringe zusätzliche Knoten-zu-Knoten-Kommunikation anfällt.

Danksagung

Ich bedanke mich bei Prof. Andreas Dreuw, ohne dessen Vertrauen, Geduld und Rat ich diese Arbeit nicht zum Abschluss gebracht hätte. Ein Dank geht ebenso an Adrian Dempwolff, der mir stehts Rat und Hilfe anbot. Und an Sebastian Thielen und meine restlichen Kollegen des Arbeitskreises für die andauernde Hilfe und tolle Zeit, die leider viel zu schnell vorbei war.
Weiterer Dank geht an Liam Keegan und das restliche Team des Scientific Software Centers des IWR für ihre Hilfe in der ersten parallelisierten Version der Implementierung.
Schließlich geht mein Dank an Peter Bastian für hilfreiche Diskussionen sowie an Nathanial Bloomfield, Peter Gill und Giuseppe Barca, auf deren Arbeit meine aufbaut.

Contents

Abstract **v**

List of Figures **xvii**

List of Tables **xxi**

Abbreviations **xxv**

Symbols **xxvii**

1 General Introduction **1**
1.1 Thesis Overview 4

2 General Theory **7**
2.1 Mathematical Foundations 8
2.1.1 Vectors, Matrices and Operators 8
2.1.1.1 Dirac Notation for Vector Spaces 10
2.1.1.2 Eigenvalues, Eigenvectors and Eigenfunctions of Operators 11
2.1.2 Hilbert Space 13
2.1.3 The Variational Principle 13
2.1.4 Numerical Integration 15
2.1.4.1 Monte Carlo Integration 16
2.2 Failures of Classical Mechanics 17
2.3 Quantum Chemistry Foundations 20
2.3.1 Schrödinger Equation 20
2.3.2 The Adiabatic Approximation 22
2.3.3 Description of the Wave Function 24
2.3.4 The Mathematics of Orbitals 26
2.3.5 The Hartree-Fock Approximation 28
2.3.5.1 Restricted Closed-Shell Roothaan-Hall 31
2.3.5.2 The SCF Algorithm 34
2.3.6 Recovering Electronic Correlation 34
2.3.6.1 Configuration Interaction 35

2.3.6.2 Perturbation Theory 36

3 Derivation of Quadrature MP2 43

3.1 State of the Art 43

3.1.1 Resolution-of-the-Identity Approximation 44

3.1.1.1 Cholesky Decomposition 45

3.1.2 Laplace-transformed MP 46

3.1.3 Monte Carlo Stochastic MP2 (MC-MP2) 48

3.1.4 MPI/OpenMP hybrid parallel quadrature MP2-F12 49

3.1.5 Quadrature of opposite-spin MP2 (Q-MP2-OS) 50

3.2 Derivation of Q-MP2 52

3.3 Energy Benchmark 58

3.3.1 Computational Details 58

3.3.2 Results 58

4 Implementation 61

4.1 Design Goals 61

4.1.1 Portability 62

4.1.2 Modularity 62

4.1.3 Sustainability 62

4.2 Hardware Architecture and
Supporting Software 65

4.3 The `libqqc` library 66

4.3.1 General Structure 66

4.3.1.1 Repository Structure 66

4.3.1.2 `libqqc/` Structure 68

4.3.1.3 Library Usage Example 69

4.3.2 The `utils` Module 70

4.3.2.1 `printers` 71

4.3.2.2 `load_from_file` 71

4.3.2.3 `ttimer` 75

4.3.3 The `grid` Module 76

4.3.3.1 One and Multi-Dimensional Grid Generation . . . 76

4.3.3.2 Voronoi Cell Generation and Supporting Functions[5] 77

4.3.4 The `loader` Module 77

4.3.5 The `vault` Module 81

4.3.6 The `methods` Module 83

4.3.6.1 `Do_qmp2` in Detail 84

4.3.6.2 `Qmp2_energy` in Detail 86

4.3.7 The Test Suite 91

5 Grid Benchmark 93

5.1 Introduction to Integration Grids 93

5.1.1 Gauss Quadrature 93

5.1.2 Molecular Becke Grid 95

5.1.2.1 Delaunay Triangulation 96
5.1.2.2 Voronoi Cells 96
5.1.2.3 Construction of Becke Weights 97
5.2 Grid Benchmark 99
5.2.1 Computational Details 99
5.2.2 Energy Benchmark 99

6 High-Performance Optimisations 103
6.1 High-Performance Concepts 103
6.1.1 Processor Performance 104
6.1.2 Memory Performance 106
6.1.3 Locality 108
6.1.4 Amdahl's Law 109
6.1.5 Exploiting Parallelism 110
6.1.5.1 Shared Memory Parallelism 112
6.1.5.2 Distributed Memory Parallelism 115
6.1.5.3 Accelerator-based Parallelism 117
6.2 Implementation of Optimisations and Parallelism in `libqqc` 121
6.2.1 General Optimisations 122
6.2.2 Adding Shared-Memory Parallelism: `OpenMP` 123
6.2.3 Adding Distributed-Memory Parallelism: `MPI` 123
6.2.4 Accelerator Parallelism: `OpenMP target` 126
6.3 Performance Benchmarking 127
6.3.1 Computational Details 127
6.3.2 Node-Level Performance 127
6.3.3 Multi-Node-Level Performance 130

7 Further Work and Outlook 133
7.1 Increasing Performance 133
7.1.1 Investigation into Optimised Grids 133
7.1.2 Exploiting Spatial Symmetry 135
7.1.3 Exploiting Temporal Locality 136
7.1.4 Exploiting Matrix Sparsity 137
7.1.5 Integral Screening 138
7.2 Application to New Methods 139
7.2.1 Coupled Cluster Method Family 139
7.2.1.1 Approximated second-order Coupled Cluster: CC2 142
7.2.1.2 Applying the Quadrature: Q-CC2 143
7.2.2 Algebraic Diagrammatic Construction Method Family 145

8 Conclusion 153

A Supporting Information: Q-MP2 Benchmarks 165

A.1 Energy Benchmark 166
A.2 Grid Benchmark 167
A.3 Performance Benchmark 170

B Additional Code Listings 175
B.1 Implementation 175

Bibliography 181

List of Figures

2.1 Example of simple numerical integration by uniformly sampling the function space[6]. . . . 16

2.2 Example use of Monte Carlo method. By calculating the ratio of randomly sampled points in and out of the area π can be calculated. 17

2.3 Simplified summary of the Stern-Gerlach experiment. A beam source (furnace) expells a beam of silver atoms, which is sent through an inhomogeneous field and detected afterwards. . . . 19

2.4 Classical view of the atom as a solar-system-like arrangement of electrons orbiting the nucleus. . . . 19

2.5 Slater-type and Gaussian-type 1s orbital example[6]. . . . 27

2.6 Minimal basis set example approximating the Slater-type orbital[6]. . 28

3.1 Cholesky algorithm of the Q-Chem implementation by Epifanovsky et al.[7,8]. . . . 47

3.2 Absolute error for the correlation energy of Q-MP2 compared to the parent MP2 method on the 20/38 Becke weighted grid with $N_K = 4$. 59

3.3 Relative absolute error for the correlation energy of Q-MP2 compared to the parent MP2 method on the 20/38 Becke weighted grid with $N_K = 4$. . . . 60

4.1 General modular structure of the `libqqc` library as of date. . . . 68

4.2 Full class relationship diagram of all relevant classes of the `libqqc` library for the calculation. . . . 71

4.3 Class diagram for the `Printer_qmp2` class of the `utils` module. . . 72

4.4 Flowchart representation of the `print_final` method of the `Printer_qmp2` class. . . . 73

4.5 Flowchart representation of the `load_from_file` utility function. . 74

4.6 Class diagram of the `Tclock` single timer class of the `utils` module. 75

4.7 Class diagram of the `Ttimer` class of the `utils` module. . . . 76

4.8 Class diagram of the base `Grid` class of the `grids` module which handles the storage and traversion of the integration grid. . . . 77

4.9 Class diagram of the base `Loader` class of the `loader` module which handles the interfacing with the host program and the data input . 78

4.10 Flowchart sketch of `Loader_qmp2` method loading in the atomic orbital representation of the coulomb integral and its transformation. 79

4.11 Class diagram of the base `Vault_qmp2` class of the `vault` module which handles the storage of the data. . . . 82

4.12 Class diagram of the base `Do_qmp2` class of the `methods` module which handles the setup and the necessary calculation before calling the requested property calculation methods. 84
4.13 Flowchart of the `run` method used in the `Do_qmp2` method instance. 85
4.14 Class diagram of the base `Qmp2_energy` class of the `methods` module which run the Q-MP2 energy calculation. 87

5.1 Simplified example of grid point generation in standard Becke weighted molecular grids on the H_2O molecule. 96
5.2 Example of the Voronoi cells (green) resulting from the Delaunay triangulation (blue) for a simplified H_2O molecule. 97
5.3 Average absolute correlation energy correction error of the benchmark set for the STO-3G basis. 100
5.4 Average absolute correlation energy correction error of the benchmark set for the 3-21g basis. 100
5.5 Average absolute correlation energy correction error of the benchmark set for the 6-31g basis. 101
5.6 Average absolute correlation energy correction error of the benchmark set comparing basis sets for $N_K = 4$. 102

6.1 Historical overview of processor clock speed over the decades. 104
6.2 Overview of the memory hierarchy of modern CPUs with typical sizes and speeds of commercial desktop hardware and cluster server. 107
6.3 Example how a row-major matrix is saved in memory. 109
6.4 Difference between the overall concept of concurrency, the execution of multiple programs at the same time and true parallelism, the execution of commands at the same time. Parallelism is a subfield of concurrency. 110
6.5 An overview of the address space of a process and the thread substructure. 112
6.6 Fork-join model of `OpenMP`, where the master thread of the process creates the thread team, which split off and joins back after the code execution. 113
6.7 Example architecture of cluster system utilising multiple single servers (nodes) connected over a high throughput network interface. 115
6.8 General architecture of modern graphics processing units. 118
6.9 Overview of the communication and workload distribution of the `MPI` implementation of the `libqqc` library. 125
6.10 Speedup of the hybrid `MPI` + `OpenMP` variant of the algorithm. . . . 129
6.11 Speedup of the `MPI` only variant of the algorithm. 130
6.12 Workload balancing example for the anthracene system on three nodes. 130
6.13 Speedup of the hybrid `MPI` + `OpenMP` variant of the algorithm. . . . 131
6.14 Speedup of the `MPI` only variant of the algorithm. 131

7.1 Different Strategies implemented and tested during the project. . . 134

7.2 Exploiting the spatial symmetry of systems to lower the PQ dimensionality. 135
7.3 Example strategy to increase temporal locality in the PQ point pair supermatrix. 136
7.4 Linked list representation example and memory calculation of a sparse matrix. 137
7.5 Schematic view of the ADC matrices for different orders of approximation[9]. The orders of terms contributing to each block are shown. 146
7.6 Matrix-Vector-Product form of the ADC algorithm which is used by the Davidson solver. 147

List of Tables

4.1 Hardware of early stages of development, mainly during single-node performance optimization. 65
4.2 Hardware of later stages of development, used during multi-node (`MPI`) performance optimization. 65
4.3 Summary of supporting software on the different architectures. . . . 65

6.1 Example GFLOPs performance of a Q-MP2 porphyrin calculation compared to the theoretical performance of the equivalent number of cores of the Intel® Xeon® Gold processor 6252 with advertised 2.1 GHz frequency. 128

7.1 List of necessary quantities for a PQ pair calculation and their memory contributions. 136

A.1 Correlation energy errors of the Q-MP2 method for smaller and medium size molecules using different basis sets and $20, 38$ molecular Becke weighted grid points. Energy difference against the parent MP2 method in Hartree. 166
A.2 Average error (in Hartree) and average relative error (in %) of the Q-MP2 correlation energy compared to the parent MP2 method for the STO-3G basis set. 167
A.3 Average error (in Hartree) and average relative error (in %) of the Q-MP2 correlation energy compared to the parent MP2 method for the 3-21g basis set. 168
A.4 Average error (in Hartree) and average relative error (in %) of the Q-MP2 correlation energy compared to the parent MP2 method for the 6-31g basis set. 169
A.5 Main workload timings of Q-MP2 for the porphyrine system in ms on the JUSTUS2 cluster system. 170
A.6 Main workload timings of Q-MP2 for the anthracene system in ms on the JUSTUS2 cluster system. 171
A.7 Main workload timings of Q-MP2 for the H_3COH system in ms on the JUSTUS2 cluster system. 172
A.8 Main workload timings of Q-MP2 for the H_2O system in ms on the JUSTUS2 cluster system. 173

Listings

4.1 This code is not very readable. 63
4.2 This code is much more readable. 63
4.3 Including the necessary header files for the calculation. 69
4.4 Generating the method specific `loader` interface class. 70
4.5 Generating the method specific `vault` storage class. 70
4.6 Generating the desired `method` class and executing the calculation. . 70
4.7 Example call to load in the AO integral data from file 81
4.8 Constructor of the `Vault_qmp2` class using the standard `Loader_qmp2` interface. 82
4.9 Code excerpt of the precalculation for the exponential factors used for scaling in the algorithm. 84
4.10 Example matrix constructs used in the `Eigen` variant of the algorithm. 89
4.11 Example usage of the utilised `Eigen` methods. 90
4.12 Example calculation of the C term for the only-array and for the `Eigen` implementation. 90
6.1 Example of bad spatial locality while multiplying two matrices. . . 108
6.2 Example of good spatial locality while multiplying two matrices. . . 108
6.3 Example use of `OpenMP` paradigm to increment 512 double values. . 113
6.4 Example use of the `MPI` standard. 116
6.5 Example use of the `CUDA` language. 119
6.6 Utilising the `OpenMP` target directive to offload the work to the GPU. 121
6.7 Utilising the `OpenMP` compiler directive to introduce shared-memory parallelism to the library main work loop of the `qmp2_energy` module. 123
6.8 Correct setup for simultanious use of `MPI` and `OpenMP` to guarantee thread safety. 124
6.9 Utilisinge the `target` directive of `OpenMP` to offload work to the GPU. 126
B.1 Example `GitHub` action executing the CI pipeline 175
B.2 Example run output for the anthracene $(6-31G)$ test system, using the array variant and the hybrid `OpenMP+MPI` parallelization scheme. The run was executed on 19 nodes. 176
B.3 Example call to load in the AO integral data `Q-Chem` 177
B.4 Example fail state of the testing suite automatically executed by each push to the main branch of the repository. 178

Abbreviations

ADC, ADC(2)-pp	Algebraic-Diagrammatic Construction scheme, –"– of second order for the polarization propagator
ALU	Arithmetic Logic Unit
AO	Atomic Orbital
API	Application Programming Interface
AVX	Advanced Vector Extensions
BLAS	Basic Linear Algebra Subprograms
BO	Born-Oppenheimer approximation
CC, CC2, CCSD	Coupled-Cluster, –"– Second Order Approximation, –"– Singles Doubles
CD	Cholesky-Decomposition
CGTO	Contracted Gaussian-Type Orbital
CI, CIS, CISD	Configuration Interaction, –"– Singles Doubles
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DF	Density Fitting
DFT	Density Functional Theory
DNA	Deoxyribonucleic Acid
ERI	Electron-Repulsion-Integral
FLOPS	Floating Point Operations Per Second
FMA	Fused Multiplay and Add
GB	Gigabyte
GCC	GNU Compiler Collection

GPL	GNU General Public License
GPU	Graphics Processing Unit
GTO	Gaussian-Type Orbital
HF	Hartree-Fock method
HOMO	Highest Occupied Molecular Orbital
HPC	High-Performance Computing
IP	Ionisation Potential
ISR	Intermediate State Representation
KAM	Kolmogorov-Arnold-Moser theorem
L1, L2, L3, LX	first, . . . level cache
LUMO	Lowest Unoccupied Molecular Orbital
MO	Molecular Orbital
MP2	Møller-Plesset perturbation theory of second order
MPI	Message Passing Interface
MP, MPPT	Møller-Plesset perturbation theory
NDA	Non-Disclosure Agreement
PCM	Point Charge Model
Q-ADC(2)	quadrature evaluated ADC(2)
Q-CC2	quadrature evaluated CC2
Q-MP2	quadrature evaluated MP2
RAM	Random Access Memory
RI	Resolution-of-the-Identity
RS	Rayleigh-Schrödinger perturbation theory
SCF	Self-Consistent-Fiel
SE	time-independent (electronic) Schrödinger Equation
SIMD	Single Instruction, Multiple Data
SMP	Symmetric Multi-Processing
SSD	Solid State Drive
STO	Slater-type Orbital
TB	Terabyte

Symbols

α	Gaussian orbital exponent
χ	one-electron spin wave function
δ	Kronecker delta
λ	Lagrangian multiplyer, AO index
ω	grid weight
ψ	basis set function
Φ	wave function
ϕ	one-electron wave function
μ, ν	AO indices
ε	MO energy (in a.u.)
ζ, η	Slater orbital exponent
a.u.	atomic units/Hartree energy
$a, b, \ldots$	index of virtual MO
$\hat{f}(i)$	Fock operator
$\hat{h}(i)$	one-electron Hamiltonian
$i, j, \ldots$	index of occupied MO
$p, q, r, s, \ldots$	index of general MO
r	distance, coordinates
x_p	spacial and spin coordinates
A, B, ...	nuclei index
E	energy
$\hat{\mathcal{E}}$	energy operator

$\hat{H}$	Hamiltonian operator
I	Coulomb potential integral
$\hat{J}$	Coulomb operator of HF
$\hat{K}$	exchange operator of HF
K, k	number/index of points on the one-dimensional grid
L	number of Lebedev/angular points on the spatial grid
M_{Aa}	ratio of mass between nuclei and electron
N	number of basis functions
$\mathcal{O}$	'Big-O' complexity/scaling
O	number of occupied orbitals
P, Q	number/index of spatial grid point
$\hat{\mathcal{P}}_{12}$	permutation operator
R	number of radial points on the spatial grid
$\hat{\mathcal{T}}_e, \hat{\mathcal{T}}_N$	kinetic energy operator of electrons/nuclei
V	number of virtual orbitals
$\hat{\mathcal{V}}_{Ne}, \hat{\mathcal{V}}_{ee}, \hat{\mathcal{V}}_{NN}$	potential energy operator
Z_A	charge of nuclei

For my family,

who encouraged me to do it.

For my friends
who helped me to persevere.

Chapter 1

General Introduction

The field of theoretical chemistry has become a trusted ally of many areas of research since its inception at the end of the 19th century. The modern computational chemist is involved in protein research[10–13], drug design[10,14,15], development of improved batteries for electric cars[16–19], improvement of photovoltaic cells[20–23], and much more.

The traditional methods of the field can be roughly split into two categories. Methods relying on classical mechanics use classical and semi-classical force fields to calculate molecular properties. They are often used in areas where very large molecular systems, such as proteins[24–26] or bulk materials[27–29], are included explicitly in the calculation. This is possible because of their comparatively low cost in terms of computational resources and time. The downside of these force field methods is that they often have to be parameterised on training data, leading to overfitting and less general applicability.

In contrast to this, methods exist which try to model the quantum mechanical nature of real systems intrinsically. These methods aim to solve the equation postulated by Erwin Schrödinger[30] in the early 20th century. This equation is of the same fundamental importance to the field of quantum mechanics as Newton's equation on motion are to classical mechanics. Theoretically, solving Schrödinger's equation for the exact wave function of a system would give access to accurate properties.

Due to its nature as a system of coupled differential equations, a direct solution is not accessible for systems of more than two particles. To remedy this, approximations are employed to the exact theory, which then can be solved feasibly. This

approach results in a multitude of different method families, each with their advantages and disadvantages depending on the underlying approximations.
Generally speaking, the higher the level of the employed theory, the more accurate is the description of the system. The trade-off for this accuracy comes in form of higher computational complexity and resource requirements. For example, the Hartree-Fock approximation is the most basic of the wave function-based methods regularly used by quantum chemists. The number of mathematical operations that need to be performed scales formally with the system size, represented by the number of atomic orbitals N, to the fourth power ($\mathcal{O}(N^4)$) and it recovers approx. 97% of the total (ground state) energy of the system. The missing three percent, which stem from the only approximately described electron-electron correlation, has to be recovered by using even higher-scaling levels of theory. The most common first step in this direction is done through the Møller-Plesset perturbation theory[31].
Two factors therefore ultimately decide if a calculation is feasible: The scaling of the method and the amount of available and usable resources. Much work is invested in introducing additional approximations to lower the scaling of a calculation. This can be done, for example, by partitioning the system into an active core and its surrounding environment. Each part is dealt with different levels of theory. An example would be the description of a solvated molecule, where the solvent is approximated with a lower level of theory (e.g. PCM[32–34]). Many approximations, however, aim to lower the scaling of the method directly.
There are two main paths to lowering the scaling. The first one consists of lowering the system size N. This can be achieved, for example, through numerical optimization. This means that parts of the initial integrals are not calculated at all, as they are expected to contribute little to the answer (e.g. through integral screening or exploitation of interelectronic distances in R12/F12 methods[35–39]). Another ansatz partitions the orbital space into a relevant and a not-relevant part. An example would be the separation of the full occupied orbital space into lower-lying core orbitals and outer valence orbitals[40–42]. Disconnecting the two orbital spaces gives easier access to the core orbital properties (CVS-ADC[43–45], CVS-EOM-CC[46]).
The second main path to lowering the scaling aims at decreasing the scaling exponent. An example (for the scaling of the required memory space) would be the reexpression of the underlying electron-repulsion integrals with the help of an

auxiliary basis, through the resolution-of-the-identity[47,48]/density fitting approximation (e.g. RI-MP[49], RI-CC[50]). Most wave function-based methods consist of expressions of the form

$$\sum_{pqrs} \frac{\langle pq|rs\rangle}{\varepsilon_p + \varepsilon_q + \varepsilon_r + \varepsilon_s} \tag{1.1}$$

with p, q, r, s being arbitrary molecular orbital indices and ε_p denoting the corresponding energy. These indices are coupled through the energy denominator, meaning that the computational scaling of eq. 1.1, $\mathcal{O}(N^4)$, cannot be lowered.
Almlöw et al.[1,51,52] have shown that the energy denominator can be expressed through a numerical quadrature, which allows for the decoupling of the indices. This idea is the starting point for the quadrature scheme presented in this thesis. Ishimura et al.[2] then showed that this can be used to reexpress certain quantities into lower scaling forms. Hirata et al.[53,54] presented an embarrassingly parallel Monte-Carlo evaluated Møller-Plesset perturbation theory formulation using this ansatz. Barca, Bloomfield and Gill[3] used the general idea to treat the opposite-spin part of the second-order Møller-Plesset perturbation (MP2-OS) energy integral with a numerical quadrature directly. They arrived through the heavy use of grid pruning and integral screening at a quadratic scaling, embarrassingly parallel version of the MP2 method (Q-MP2-OS). This ansatz is used in the following thesis to arrive at an expression for the full energy integral and is later extended to other, higher-scaling methods.
Lowering the scaling through any of the above-mentioned methods means losing accuracy. The only way around this conundrum is to increase of the available resources instead. Modern computational chemistry, especially of large systems and/or higher levels of theory, is done on extensive computer clusters. These are systems of dozens to tens of thousands of computational nodes, interconnected with high-bandwidth, low-latency networks. Until the end of the last century, one could simply rely on the steady advancement in processing frequency for faster calculations on one of these nodes. Systems that could not be treated were made quickly possible by the steady increase in computational power, with minimal changes to the algorithms necessary. This is often expressed in one of the different forms of Moore's Law, which can be stated as the doubling of transistor density on chips every two years.
However, many argue that Moore's Law has come to an end[55,56]. For the last

two decades, the frequencies of central processing units (CPUs) have roughly stabilised. There are a lot of reasons for that, but one of the major ones is that the ever-increasing demand for power leads to more and more heat that has to be dispersed. Another problem is that at the scale that modern chip dies are manufactured at ($< 15\ nm$), quantum effects start to become prevalent[57].
The manufacturers' answer to these problems was the addition of multiple processing units per CPU die. Modern computer nodes can have more than 64 physical cores. The fastest supercomputer cluster system, in terms of FLOPS, is as of November 2022 the `FRONTIER - HPE CRAY EX235A`[58] at the Oak Ridge National Laboratory. It features 8.730.112 processing units with a total theoretical peak performance of 1.6×10^{18} FLOPS. The term processing units here refers to the available cores of the central processing units (CPUs), as well as the cores available through graphical processing units (GPUs). Most modern supercomputer architectures have some degree of this hardware heterogeneity. This staggering amount of calculations per second shows that today most computational performance is locked behind the ability of a program to execute efficiently in parallel.
The need for both, a lower scaling and higher parallel efficiency, is the main motivation for this thesis. The following chapters will give an introduction to their respective concepts and the state of the art before showcasing the derivation, implementation and testing for a low scaling, highly parallel efficient implementation of quadrature evaluated second-order Møller-Plesset perturbation theory. Additional relevant projects as well as an outlook of extending this quadrature scheme to new methods are showcased at the end.

1.1 Thesis Overview

The thesis is organised as follows: Chapter 2 introduces the basic theory to the reader. It starts by introducing general mathematical concepts used in theoretical chemistry, such as matrix/vector algebra, function operators and the variational principle, before introducing general numerical integration strategies. These concepts are then used in the derivation of the most basic building blocks of quantum chemistry up to electron-correlated wave function-based methods. From there the quadrature scheme presented in this thesis is introduced in chapter 3. First, the state of the art is summarised, and then previous relevant works are closely inspected. Finally, the derivation of the quadrature evaluated second-order Møller-Plesset perturbation theory is presented and some example results are shown. The grid dependency of the method is discussed in chapter 5. To do so, the utilised molecular grids are introduced and a benchmark for the different grid setups is presented. The main work of the thesis consists of the efficient and massively parallel implementation of this method. The basic concepts used to achieve this are discussed at the beginning of chapter 4, which is then followed by a detailed look into the structure of the open-source `libqqc` library. In chapter 6 the basic theory necessary for high-performance computation is discussed, the utilised strategies are presented and finally, the performance of the implementation is shown. The remaining smaller projects the author contributed to along with an outlook into future possibilities for the quadrature scheme are presented in chapter 7. It finishes with providing some working equation for a new quadrature evaluated ADC(2) method. Finally, the results of this thesis are summarised in chapter 8.

Chapter 2

General Theory

Interdisciplinary research is ingrained in the DNA of theoretical chemistry. Much, if not all, of the underlying theory, is based more on physics than chemistry and much work lives in the abstract world of mathematics. And while this may once have been the whole sum of fields connected to theoretical chemistry, it is so no more. Modern research blurs the line not only between chemistry and physics, but between biology, toxicology, and medicine, to only name a few. The field has become so interconnected, that basically all modern work in the lab requires some backup from a theoretician. The advancements talked about in the introductory chapter has made it possible to utilise theoretical chemistry in many applications. And so, the modern chemist needs a broad understanding of numerous different topics.

The work presented in this thesis lives in the intersection between method development and high-performance computing. One would be very optimistic to expect in-depth knowledge of both subjects from the general chemist or computer scientist. As such, the following chapter tries to bring both sides of the equation to a common understanding of the presented work.

In the opening section 2.1 some general introductions to the key mathematical concepts of the following theory are presented, as well as some introduction to the numerical integration aspect. General physics concepts are introduced in section 2.2. Afterwards, the foundation of theoretical chemistry is summarised in section 2.3.

2.1 Mathematical Foundations

Not all fundamental concepts will be discussed here. However, the concepts of linear algebra and operator-based mathematics are heavily used both in the electronic structure theory and the optimisation work discussed later. Therefore, what follows is a short reminder. This recap follows largely the introductory chapter of Szabo and Ostlund's 'Modern Quantum Chemistry'[59].

2.1.1 Vectors, Matrices and Operators

The math behind electronic structure theory is often expressed in terms of integrals and operators. The underlying calculations however are generally done by vector and matrix representation of these concepts.

We define a vector $\vec{a}$ as a collection of scalars representing a single entity, which can be expressed in terms of the sum of scaled unit vectors $\{\vec{e_p}\}$.

$$\vec{a} = \sum_p a_p \vec{e_p} \qquad\qquad \vec{e_p} \cdot \vec{e_j} = \delta_{ij} \tag{2.1}$$

These are orthonormal to each other and represent the vectors of the basis in which the vector is expressed. The symbol δ_{ij} is called the Kronecker delta.

$$\delta_{ij} = \begin{cases} 1 & \text{for } i = j \\ 0 & \text{for } i \neq j \end{cases} \tag{2.2}$$

Neither the set of scaling factors a_p nor the unit vectors $\vec{e_p}$ are generally unique. Therefore the same vector can have a different representation in another set of basis vectors.

An operator $\hat{\mathcal{O}}$ is an entity capable of transforming vector $\vec{a}$ into a new vector $\vec{b}$ by

$$\hat{\mathcal{O}}\vec{a} = \vec{b} \tag{2.3}$$

When a linear operator acts on a unit vector $\vec{e_p}$ it produces another vector which can be expressed in the same basis.

$$\hat{\mathcal{O}}\vec{e_p} = \sum_j \vec{e_j} O_{ij} \tag{2.4}$$

The rectangular array of elements is called a matrix. A matrix of all components O_{ij} is called the matrix representation of the operator in this basis.

$$\mathbf{O} = \begin{pmatrix} O_{11} & O_{12} & \dots \\ O_{21} & O_{22} & \dots \\ \dots & \dots & \dots \end{pmatrix} \tag{2.5}$$

This matrix representation completely defines the behaviour of the operator acting on a vector in the same basis as the matrix, as any vector can be represented in a linear combination of unit vectors of this basis. From the general matrix formalism and multiplication, rules follow that the order of operators acting on a vector is generally not commutative.

$$\mathcal{A}\mathcal{B}\vec{a} \neq \mathcal{B}\mathcal{A}\vec{b} \tag{2.6}$$

The adjoint of a matrix $\mathbf{A_{ij}}$ is the matrix $\mathbf{A}^\dagger$ where the rows and columns of the matrix have been transposed and each new element is the complex conjugate of the old element.

$$\mathbf{A_{ij}}^\dagger = \mathbf{A}^\star_{\mathbf{ji}} \tag{2.7}$$

A square matrix has the same number of rows and columns. A square matrix is called unitary if its inverse is equal to its adjoint matrix $\mathbf{A^{-1}} = \mathbf{A}^\dagger$.

A square matrix is called hermitian when it is self-adjoint. This means that the adjoint of the matrix returns the same matrix $\mathbf{A}^\dagger = \mathbf{A}$. If all elements of the matrix are real it is called symmetric.

2.1.1.1 Dirac Notation for Vector Spaces

A vector space is a set of all possible vectors $\{\vec{v}\}$ over a field F of scalars. The space has two binary operations (addition and scalar multiplication) and fulfils eight axioms for these operations for all members of the space.

Theoretical chemistry concepts such as quantum states are often expressed in the Dirac notation[59–61]. The interpretation of this notation is interchangeably between vector spaces and functions in a Hilbert space, which will be explained shortly.

An n-dimensional complex vector in the Dirac notation is noted in terms of unit vectors $|i\rangle$, so the vector $\vec{a}$ becomes

$$|a\rangle = \sum_{p}^{N} |i\rangle \, a_p = \begin{pmatrix} a_1 \\ a_2 \\ \dots \end{pmatrix} \tag{2.8}$$

This is called a 'ket' and represents a column vector. The adjoint row vector of $|a\rangle$ can be represented by the equivalent 'bra'

$$\langle a| = \sum_{p}^{N} a_p^{\star} \langle i| \tag{2.9}$$

The scalar product $\langle a|b\rangle$ represents the known scalar product between the two vectors $a\vec{*}$ and $\vec{b}$ or the matrix product of their representations $\mathbf{a}{*}\mathbf{b}$.

An operator in this notation becomes thus

$$\hat{\mathcal{O}}\,|a\rangle = |b\rangle \qquad \hat{\mathcal{O}}\,|i\rangle = \sum_{j} |j\rangle \, \mathbf{O}_{ji} \tag{2.10}$$

Multiplying the operator with a bra $\langle k|$ 'from the left', meaning building the scalar product with this bra, gives us the element of the operator representation in the basis with the indices ki.

$$\langle k|\hat{\mathcal{O}}|i\rangle = \sum_{j} \langle k|j\rangle \, (\mathbf{O})_{ji} = \sum_{j} \delta_{kj} = (\mathbf{O})_{ki} \tag{2.11}$$

as $\langle a|b\rangle = \sum_{ij} a_p^\star \langle i|j\rangle b_j$ can only be satisfied if $\langle i|j\rangle = \delta_{ij}$.

The adjoint of the operator $\hat{\mathcal{O}}$ acts in turn on a bra instead of a ket $\langle a| \mathcal{O}^\dagger = \langle b|$.

A self-adjoint operator $\hat{\mathcal{O}} = \mathcal{O}^\dagger$ is called Hermitian and the following relationship holds for the elements of its matrix representation: $\langle a|\hat{\mathcal{O}}|b\rangle = \langle a|\hat{\mathcal{O}}|b\rangle = \langle b|\hat{\mathcal{O}}|a\rangle^\star$.

Many operators in theoretical chemistry are Hermitian and much of what the methods in the following chapters try to achieve is to find the eigenvalue or the eigenvector of a given operator.

2.1.1.2 Eigenvalues, Eigenvectors and Eigenfunctions of Operators

The eigenvector of an operator is a vector $|a\rangle$ such that letting the operator act on the vector returns the same vector times a scalar, $\hat{\mathcal{O}}\,|a\rangle = \omega_a\,|a\rangle$. The scalar is called the eigenvalue of the operator. Eigenvectors of Hermitian operators are orthogonal to each other if their eigenvalues are not the same (non-degenerate) and those values are real.

Not every matrix representation of a Hermitian operator is diagonal. This is however the case if the representation is in the basis of the eigenvectors of the operator. Finding the eigenvectors of an operator to use in its diagonalization is one of the core computational problems in theoretical chemistry. The eigenvectors can then be used to form a unitary transformation matrix $\mathbf{U}$ with which the matrix representation of the operator can be transformed into its diagonal form $\mathbf{\Omega}$ through $\mathbf{\Omega} = \mathbf{U}^\dagger\mathbf{O}\mathbf{U}$. A Hermitian operator has n eigenvalues.

As mentioned before, many formulas in theoretical chemistry are generally discussed in terms of Dirac formalism. It represents either a vector space or a function space. Dirac formalism can be used interchangeably between these two domains. The Fourier series shows that a well-behaved function $a(x)$ on an interval $[x_1, x_2]$ can be described by a linear combination of sinuses and cosines with its own set of expansion parameters for each function. This is equivalent to the idea of expanding a vector in terms of the unit vectors of a basis. With the functions ψ_p, which fulfil the orthonormality condition

$$\int_{x_1}^{x_2} dx \psi_p \psi_j = \delta_{ij} \tag{2.12}$$

on the interval we can construct any function $a(x)$ in terms of these functions

$$a(x) = \sum_p \psi_p a_p \tag{2.13}$$

which, as mentioned, is analogous to the expansion of a vector in the set of basis functions in equation 2.8.

$$\psi_p(x) = |i\rangle \qquad \psi_p^*(x) = \langle i| \tag{2.14}$$

With this any function $a(x)$ can be written in terms of Dirac notation as $a(x) = |a\rangle$. The scalar product becomes

$$\int d\tau a(x)b(x) = \langle a|b\rangle \tag{2.15}$$

Let an operator now act upon the function $a(x)$ to produce $b(x)$ analogously to equation 2.3. This becomes in Dirac notation equivalent to equation 2.10. A function $|a\rangle$ which, after an operator acts upon it returns itself multiplied by a scalar is called an eigenfunction.

$$\hat{\mathcal{O}} \, |a\rangle = \omega_a \, |a\rangle \tag{2.16}$$

One can obtain the prefactor for normalised eigenfunctions ϕ_a through

$$\omega_a = \int d\tau \phi_a^* \mathcal{O} \phi_a = \langle a|\hat{\mathcal{O}}|a\rangle \tag{2.17}$$

Integrals of the form

$$\langle a|\hat{\mathcal{O}}|b\rangle = \int d\tau \phi_a^* \hat{\mathcal{O}} \phi_b \tag{2.18}$$

will become important in later sections discussing the Schrödinger equation and its variants. For a Hermitian operator, the following relationship follows from this

$$\langle a|\hat{\mathcal{O}}|b\rangle = \langle b|\hat{\mathcal{O}}|a\rangle^* \tag{2.19}$$

2.1.2 Hilbert Space

The underlying quantum mechanics of theoretical chemistry, which will be confined to the view of the common chemist in this work, exists in a complex, separable so-called Hilbert space. A Hilbert space is a vector space with a defined inner product between its vectors. In this methodology, these vectors represent the quantum mechanical state of a system and can in turn be expressed in terms of unit state vectors. Each desired observable resulting from the measurement of a system can be represented as linear Hermitian operators acting on the unit vectors. An eigenstate of the measurements is therefore an eigenvector of the operator and its eigenvalue the observable.

The inner product described in the Hilbert space is defined as

$$\langle \psi_p | \psi_j \rangle = \int d\tau \psi_p^* \psi_j \tag{2.20}$$

and is generally interpreted as the probability amplitude of the initial systems ψ_p collapsing into the specific eigenstate of the final system ψ_j at the point of measurement, with the result of the measurement representing the eigenvalue of the eigenstate.

2.1.3 The Variational Principle

The time-independent Schrödinger equation, which will be fully introduced later, is in essence an eigenvalue problem of the form

$$\hat{\mathcal{H}} |\Phi\rangle = \varepsilon |\Phi\rangle \tag{2.21}$$

where $\hat{\mathcal{H}}$ is a Hermitian operator acting on the wave function of the system $|\Phi\rangle$, which is an orthonormal eigenfunction of the operator, and ε represents the energy of the system. The Schrödinger equation is generally not exactly solvable, as it represents a many-body problem, which calls for approximate solutions to find the eigenvalues. One of the most used methods to solve the above equation is called the variational principle.

The variational principle relies on trial functions $|\tilde{\Phi}\rangle$ which can be expressed as a linear combination of eigenfunctions $|\phi_a\rangle$ with their respective eigenvalues ε_a

$$|\tilde{\Phi}\rangle = \sum_a |\phi_a\rangle\, c_a = \sum_a |\phi_a\rangle \langle\phi_a|\tilde{\Phi}\rangle \tag{2.22}$$

and respectively

$$\langle\tilde{\Phi}| = \sum_a c_a^* \langle\phi_a| = \sum_a \langle\tilde{\Phi}|\phi_a\rangle \langle\phi_a| \tag{2.23}$$

If this trial function is normalised $\langle\tilde{\Phi}|\tilde{\Phi}\rangle = 1$ then any eigenvalue formed from letting it interact with the operator $\hat{\mathcal{H}}$ has to be equal or higher then the lowest lying eigenvalue.

$$\langle\tilde{\Phi}|\hat{\mathcal{H}}|\tilde{\Phi}\rangle \geq \varepsilon_0 \tag{2.24}$$

This would only be true if the trial function is identical to the lowest lying eigenfunction $|\phi_0\rangle$. This ansatz is typically transformed into a diagonalisation problem.

Assuming that the trial function consists of a linear combination of n fixed, real, orthonormal basis functions

$$|\tilde{\Phi}\rangle = \sum_p^n c_p\, |\phi_p\rangle \tag{2.25}$$

we can transform the operator in its matrix representation in the basis of these functions

$$(\mathbf{H})_{ij} = H_{ij} = \langle\phi_p|\hat{\mathcal{H}}|\phi_j\rangle \tag{2.26}$$

Because of the Hermeticity of the operator this matrix is symmetric, $H_{ji} = H_{ij}$. With the assumption of the trial function being normalised the expectation value becomes

$$\langle\tilde{\Phi}|\hat{\mathcal{H}}|\tilde{\Phi}\rangle = \sum_{ij} c_p \langle\phi_p|\hat{\mathcal{H}}|\phi_j\rangle\, c_j = \sum_{ij} c_p c_j H_{ij} \tag{2.27}$$

This function can now be minimised, which boils down to solving the set of equations which follow

$$\mathbf{H}\vec{c} = E\vec{c} \tag{2.28}$$

with $\vec{c}$ being the column vector of the coefficients c_p. This can be solved in the usual manner to gain the eigenvalues and eigenfunctions, which results in n solutions for trial wavefunctions $|\tilde{\Phi}_a\rangle$, with the eigenvalues E_a representing the expectation value for $\langle\tilde{\Phi}_a|\hat{\mathcal{H}}|\tilde{\Phi}_a\rangle$. The lowest eigenvalue corresponds to the best approximation of the so-called ground state energy when speaking specifically about the Schrödinger equation. The approximation is here that it is the ground state energy in the finite basis $|\phi_p\rangle$ and it is an upper bound to the 'real' ground state energy in an infinite basis.

$$E_0 = \langle\tilde{\Phi}_0|\hat{\mathcal{H}}|\tilde{\Phi}_0\rangle \geq \varepsilon_0 \tag{2.29}$$

2.1.4 Numerical Integration

Analytical evaluation of integrals is not always a feasible path, as will become one of the main aspects of this thesis. To remedy this we need numerical integration techniques. The most simple form of approximating the analytical solution of an integral is the sum over sample points in the integral space.

$$F(x) = \int dx f(x) \approx \sum_{P}^{n_P} \omega^P f(x^P) \tag{2.30}$$

From here on out if not noted otherwise the index of a grid point on which a function has been evaluated will be written with an upper case exponent, typically P. The number of sampling points x^P is n^P. ω^P is a weighting factor which decides the amount of contribution of the point to the overall integral. This is the fundamental idea behind quadrature[62] techniques, of which the simplest form is the trapezoid rule learned in school

$$F(x) = \int dx f(x) \approx \sum_{P}^{n^P} \frac{f(x^{P-1}) + f(x^P)}{2}(x^P - x^{P-1}) \tag{2.31}$$

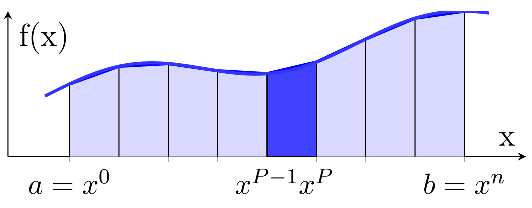


FIGURE 2.1: Example of simple numerical integration by uniformly sampling the function space[6].

A simple example is demonstrated in fig. 2.1. A further look into the numerical integration techniques important for this work is given in chapter 5.

2.1.4.1 Monte Carlo Integration

One final numerical technique, which is used in one of the publications leading to the derivation of this thesis' main subject, shall be quickly sketched here. Quadrature techniques go through all the points on their integration grid, which all have to be evaluated and weighted. If the integration space is erratic the grid has to be finer, which increases the number of points which have to be evaluated on it. This ansatz is deterministic, which means for the same input parameters and the same problem it will always return the same answer.

Another technique for the approximation of integrals stems from the Monte Carlo method family. It is not dissimilar to the trapezoid rule discussed above. The main difference lies in the choice of evaluation points. In the Monte Carlo integration, the points are drawn from a random distribution, evaluated on the integral then multiplied by a randomly sampled width $(b-a)$ from the integral space.

$$F(x) = \int_a^b f(x)dx \approx \frac{1}{N}(b-a)\sum_{k=1}^{N} f(x_p) \tag{2.32}$$

This idea relies on the law of large numbers that for a sufficiently large N the integral will be sufficiently well approximated

$$\lim_{N\to\infty} \frac{1}{N}(b-a)\sum_{k=1}^{N} f(x_p) = F(x) \tag{2.33}$$

The error for this integral, for a not infinite number of points, scales $O(\frac{1}{N})$ with the number of samples.

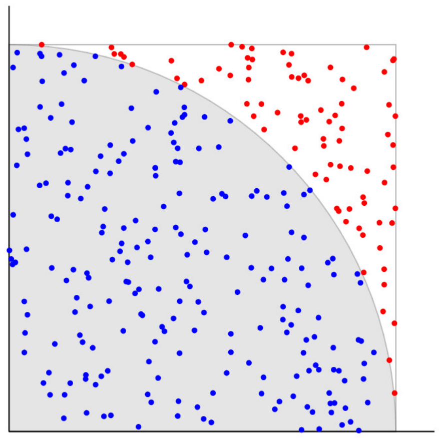


FIGURE 2.2: Example use of Monte Carlo method. By calculating the ratio of randomly sampled points in and out of the area π can be calculated.

An often-used example to demonstrate a Monte Carlo method is shown in figure 2.2. To calculate π random samples are drawn from the area. Both the area of the quarter circle A_C and the area of the square A_S is known

$$\frac{A_C}{A_S} = \frac{\frac{\pi r^2}{4*4}}{r^2} = \frac{\pi}{16} \tag{2.34}$$

π can then be estimated as

$$\frac{n_{\text{in Circle}}}{n_{\text{total}}} = \frac{\pi}{16} \tag{2.35}$$

Simple Monte Carlo integration uses a uniform distribution for the sampling points, which is not optimal for all applications. Importance sampling Monte Carlo generalises this ansatz to any distribution, which, if cleverly chosen, can be used to decrease the variance and with that the error of the method. This leads to the use of fewer points for the same accuracy. One of the most used algorithms is the Metroplis-Hasting[63,64] algorithm.

2.2 Failures of Classical Mechanics

Theoretical chemistry is a field resulting from the developments of quantum mechanics. This happened relatively recently, beginning at the start of the last century.

Until about 150 years ago the area of physics was completely defined. The laws of Newton and Maxwell, among others, fully explained the movements, light and energy of an object. If one has the starting conditions of a moving body, its kinetic energy or impulse, and sufficient information about the environment, one could make precise predictions of the position, energy, momentum and other properties of this body.

Until the end of the 19th century, when experimental data pointed to problems which could not be explained by classical physics. The reigning theory of the time, investigated by Rayleigh and Jeans among others, was that the electromagnetic field behaved as a collection of harmonic oscillators with an infinite amount of variable frequencies. This however meant that the light spectrum of a heated body should show that oscillators with a short wavelength are excited at any temperature. Which would mean that a body always glows and that no such thing as darkness exists.

In the 1890s Max Planck proposed a solution to this problem. If energy was not a continuous spectrum, but a discrete one, one could explain why certain wavelengths would only be excited at higher temperatures. This idea sounded counter-intuitive to our understanding of nature, which we observe to be continuous. However, this packaging of energy, or *quantising*, proved to be correct.

This quantisation was shown to be not only a property of energy. In the 1920s Stern and Gerlach reported the observation of the quantisation of the angular momentum of atoms. They send a beam of silver atoms through a strongly inhomogeneous electric field, see figure 2.3, and recorded the interaction with a glass screen. One could see two distinct dots on the screen. This could be explained by the structure of the silver atom: silver's most outer electron sits in an incomplete $5s$ shell with a total angular momentum of $l = 0$. However, the electron itself has an underlying property which produces this effect, called spin. The term 'spin' is reminiscent of the rotation of a charged classical body, which would result in similar deflection behaviour, although the electron does not spin in the classical sense.

This intrinsic spin of the electron, a charged particle, interacting with the magnetic field results in a force acting on the particle, diverting its path. Classically,

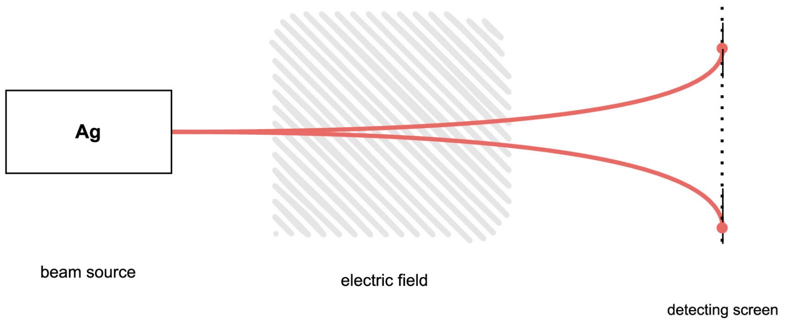


FIGURE 2.3: Simplified summary of the Stern-Gerlach experiment. A beam source (furnace) expells a beam of silver atoms, which is sent through an inhomogeneous field and detected afterwards.

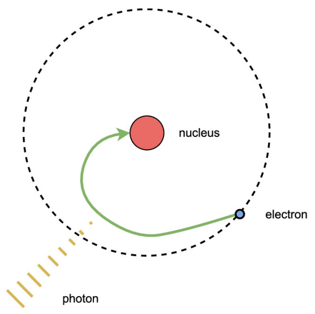


FIGURE 2.4: Classical view of the atom as a solar-system-like arrangement of electrons orbiting the nucleus.

one would have assumed this spin to be continuous. What the Stern-Gerlach experiment showed however was that the electron spin only had two discrete values and the resulting angular momentum was therefore quantised.

One more problem of classical physics shall be discussed, as it has more relevance to quantum chemistry. At the time one traditional view of the atom was the *solar system* model (fig. 2.4). The atomic nucleus sits positively charged at the centre, and the negatively charged electron orbits it with a speed large enough to keep it in orbit.

Already at the end of the 18th century, the first experiments suggested that light behaved like a wave. Maxwell connected this behaviour to his theory of electromagnetic waves and correctly identified light as such. Together with the experiments by Hertz, this posed a question: Maxwell's equation proposed that a moving charge in an electric field emits an electromagnetic wave, which we know as light. Light however carries energy, and that energy can only come from the original charged particle. Electrons move in circular motions and should therefore emit light. This should cause the electron to lose energy, which means losing speed and collapsing into its nucleus. A process which should take about 10^{-11} s. Seeing as the universe still exists, this cannot be the nature of the atom, nor that of the electron.

Other experiments both for the electron, such as the electron diffraction experiment, and for light itself, such as the Compton-effect and the photoelectric effect, would soon point to a new theory: particles of small enough size behave not only as a particle, but also as a wave. This new idea of the *particle-wave duality* of matter, later generalised by de Broglie, was another foundational change to the classical concepts of physics. This, together with the idea of quantisation and the discovery of the Uncertainty principle by Heisenberg, lead to a new area of physics called *quantum mechanics.*

2.3 Quantum Chemistry Foundations

Quantum chemistry, uses the ideas and equations of quantum mechanics to model and explain chemical processes.

2.3.1 Schrödinger Equation

Electronic structure theory is concerned with solutions to the Schrödinger equation. Published by Erwin Schrödinger in 1926[30] it is the foundational equation of quantum mechanics, similar to what Newton's equation is to classical mechanics.

$$\hat{\mathcal{H}}(r)\left|\Phi(r,t)\right\rangle = \hat{\mathcal{E}}(t)\left|\Phi(r,t)\right\rangle = -i\frac{\delta}{\delta t}\left|\Phi(r,t)\right\rangle \tag{2.36}$$

with the operator $\hat{\mathcal{H}}$ being the Hermitian Hamiltonian, the wave function of the system $|\Phi\rangle$ and the energy operator $\mathcal{E}$. The parameter r depicts the dependency of the spatial configuration of the wave function, and the coordinates of its underlying particles. Going forward it will be dropped unless relevant.

This thesis is mostly about non-relativistic ground state methodology. Going forward only the time-independent form of the Schrödinger equation will be discussed, if not noted explicitly otherwise.

$$\hat{\mathcal{H}} |\Phi\rangle = \varepsilon |\Phi\rangle \tag{2.37}$$

with the Hamiltonian consisting of all operators acting on the system.

$$\hat{\mathcal{H}} = \mathcal{T}_e + \mathcal{T}_N + \mathcal{V}_{Ne} + \mathcal{V}_{ee} + \mathcal{V}_{NN} \tag{2.38}$$

Going forward parameters written in upper case, M, R, will denote properties of the nuclei and parameters written in lower case, n, r, those of the electrons. Equation 2.38 consists of the kinetic energy of the electrons $\mathcal{T}_e$ and the nuclei $\mathcal{T}_N$, the repulsive potential between the electrons V_{ee} and the nuclei V_{NN} themselves and the attractive forces between nuclei and electrons $\mathcal{V}_{Ne}$. The operator equations are respectively

$$\mathcal{T}_e = -\sum_{a=1}^{n} \frac{1}{2}\nabla_a^2 \quad , \quad \mathcal{T}_N = -\sum_{A=1}^{M} \frac{1}{2M_A}\nabla_A^2 \tag{2.39}$$

$$\mathcal{V}_{ee} = +\sum_{a=1}^{n}\sum_{b>i} \frac{1}{r_{ab}} \quad , \quad \mathcal{V}_{NN} = +\sum_{A=1}^{M}\sum_{B>A}^{M} \frac{Z_A Z_B}{R_{AB}} \tag{2.40}$$

$$\mathcal{V}_{Ne} = -\sum_{a=1}^{n}\sum_{A=1}^{M} \frac{Z_A}{r_{aA}} \tag{2.41}$$

All equations in this thesis are written in terms of atomic units.

The wave function $|\Phi\rangle$ consists of the wave functions of the particles of the system, which are in turn expanded in a set of basis function $|\phi\rangle$, which will be discussed shortly. There is no true physical interpretation of the wave function $|\Phi\rangle$. However, its square, $\langle\Phi|\Phi\rangle$, is generally interpreted as the probability of finding the described system at a specific point in space. This interpretation, which is analogous to the square of a light wave function describing the intensity of light, is called *Born's Interpretation*. For this reason, the wave function is generally normalised over all space

$$\int_{-\infty}^{\infty} d\tau \Phi(\tau) = 1 \tag{2.42}$$

as the probability of finding the system anywhere should be exactly one.

The quantisation property of quantum mechanics stems from this and other boundary conditions to the wave function. To fulfil the normalisation criteria the wave function must not be infinite at any point. The Schrödinger equation is also a differential equation of the second order. This means that the second, and following from that the first, derivation of the wave function must exist and the wave function has to be continuous. Finally, only one probability of finding the system at a point in space should exist. This means that the wave function has to be unambiguous.

The major problem with the Schrödinger equation stems from its differential equation nature. Solving it for any system with more than two particles, e.g. a helium atom (two protons, two electrons), is not only not feasible, but generally assumed to be impossible. No analytical solution exists for these many body problems, including the two-body problem for specific setups. This is covered for example by the *KAM theory*[65–67] and it is not a special property of the Schrödinger equation but rather a general problem with, among other things, multi-body Hamiltonian partial differential equations in Hilbert spaces.

To remedy this the field of quantum chemistry has introduced a series of approximations to the Schrödinger equation. There are two major branches of Schrödinger equation based quantum chemistry, density functional theory and electronic structure theory. Most of the theory in this work is covered by the latter.

2.3.2 The Adiabatic Approximation

The first major approximation to the Schrödinger equation is the adiabatic Approximation, also called Born-Oppenheimer[68] Approximation. It aims to decouple the electronic and nuclear problems. To do so it argues that, because the mass of a proton is around 1800 times that of an electron, the speed at which the electrons move has to be much higher than that of the heavy nuclei. Therefore, the movement of the latter can be neglected and the system is only parametrically dependent on the nuclei's position. This means that one can separate the full Hamiltonian into the kinetic energy of the nuclei and an electronic contribution.

$$\hat{\mathcal{H}} = T_N + \hat{\mathcal{H}}_e \tag{2.43}$$

with

$$\hat{\mathcal{H}}_e = T_e + V_N + V_{ee} + V_{Ne} \tag{2.44}$$

Where the potential energy of the nuclei is a constant and the potential energy between the electrons and nuclei only depends parametrically on the nuclei's position.

This electronic Hamiltonian has its own set of eigenfunctions, which are called the electronic wave function of the system $|\Phi_e\rangle$. The V_N term of the electronic Hamiltonian is constant and any addition of a constant to the eigenvalues of an operator does not effect its eigenfunctions. This term is therefore often dropped for derivations. The wave function and the energy of the electronic Schrödinger equation

$$\hat{\mathcal{H}}_e \Phi_e(r : R) = \varepsilon_e(R)\Phi_e(r : R) \tag{2.45}$$

now only has a parametric dependency on the positions of the nuclei R. Additionally, the wave function explicitly depends on the electronic positions r. The electronic wave function is once again assumed to be normalised. The full wave function can be recovered as a sum of superpositions of the nuclei and electronic wave functions

$$\Phi_{exact}(r, R) = \sum_n \Phi_N(R)\Phi_n(r : R) \tag{2.46}$$

The actual approximation is that the electronic wave function is not, or only very weakly, dependent on the nuclei coordinates. Or, that the movement of the nuclei does not influence the energy of the electrons.

$$T_N \Phi_e \approx \Phi_e T_N \tag{2.47}$$

This approximation holds for many systems in theoretical chemistry. It breaks down if external fields are applied to a system, which couple the electronic and the nuclei movement, or when there is a degeneracy in the potential surfaces of the electronic states, such as with conical intersections[69–71].

The total energy of the system is the sum of the electronic energy plus the potential of the nuclei. Analogously to the electronic problem, one can formulate a Schrödinger equation for the nuclei.

If not explicitly mentioned, all following derivations will be in terms of the electronic Schrödinger equation.

2.3.3 Description of the Wave Function

Up until now the structure of the wave function, apart from its wave characteristic, was left unmentioned. They are however of importance for the following discussions.

What is the wave function of a single electron? It has a position in space, described by its coordinates, e.g. $r = \{x, y, z\}$, and a quantity called spin. The wave function of the electron, therefore, has to be made up of both of these quantities. Let $\phi(r)$ denote the set of spatial functions for an electron p, which we call spatial orbitals, which is dependent on the spatial coordinates and $\{\alpha(s),\ \beta(s)\}$ a set of spin functions dependent on the spin of the electron. The full function can then be described as a spin orbital $\chi(x)$

$$\chi(x) = \begin{cases} \phi(r)\alpha(s) & \text{for spin up} \\ \phi(r)\beta(s) & \text{for spin down} \end{cases} \tag{2.48}$$

where the parameter $x = \{r, s\}$ denotes both the spin and spatial coordinates of the orbital. The set of spatial orbitals is assumed to be orthonormal, as are the spin orbitals.

The question now becomes: What is the wave function of a many-electron system? Assuming one could transform the electron-electron repulsion $\mathcal{V}_{ee}$ term of the Hamiltonian into a form where one can separate the contribution for each electron, either by neglecting or averaging it, the electronic Hamiltonian could be written as the sum of its one-electron parts

$$\hat{\mathcal{H}} = \sum_{p}^{n_e} \hat{h}(p) \tag{2.49}$$

the one-electron Hamiltonian $\hat{h}(p)$ (p being the index for a general electron) would then have a set of eigenfunctions that one could connect to the spin orbitals from before

$$\hat{h}(p)\chi(p) = \varepsilon\chi(p) \tag{2.50}$$

The eigenfunction Φ of the full Hamiltonian of eq. 2.49 would then be simply a product of the one-electron spin orbitals

$$\Phi(x_1, x_2, \ldots, x_n) = \chi_1(x_1)\chi_2(x_2)\ldots\chi_n(x_n) \tag{2.51}$$

This ansatz for the wave function is called the Hartree product.

However, one problem arises. In Born's interpretation of the wave function the probability of these eigenfunctions would be equal to the product of the probability of the spin orbitals, so

$$\langle\Phi(x_1, x_2, \ldots, x_n)|\Phi(x_1, x_2, \ldots, x_n)\rangle = \langle\chi_1(x_1)|\chi_1(x_1)\rangle \langle\chi_2(x_2)|\chi_2(x_2)\rangle \ldots \langle\chi_n(x_n)|\chi_n(x_n)\rangle \tag{2.52}$$

This in turn would mean that the probability of finding one electron is independent of all the other electrons, which is not the physical behaviour. Electrons repel each other, so their movement and position are correlated. Another problem is that the Pauli anti-symmetric principle[72] is not represented in the wave function. The wave function of a fermionic system, a system of identical particles with spin $\frac{1}{2}$, must be anti-symmetric with respect to the interchange of the position of two of its particles. This is not the case for the Hartree product.

To remedy this the wave function is typically formed in terms of a Slater determinant.

$$\Phi(x_1, x_2, \ldots, x_n) = \frac{1}{\sqrt{N!}} \begin{vmatrix} \chi_1(x_1) & \chi_2(x_1) & \ldots \chi_n(x_1) \\ \chi_1(x_2) & \chi_2(x_2) & \ldots \chi_n(x_2) \\ \ldots & \ldots & \ldots \\ \chi_1(x_n) & \chi_2(x_n) & \ldots \chi_n(xn) \end{vmatrix} = |\chi_1\chi_2, \ldots, \chi_n\rangle \tag{2.53}$$

This determinant, together with the normalisation factor, now introduces the anti-symmetric principles to the wave function.

$$\Phi(x_1, x_2, \ldots, x_n) = -\Phi(x_2, x_1, \ldots, x_n) \tag{2.54}$$

This also introduces an exchange-correlation between the electrons, as the final probability of the system should not depend on the exchange of two electrons. This ultimately correlates the movement of electrons of parallel spin.

2.3.4 The Mathematics of Orbitals

Before the next foundational approximation step, a quick mention of the mathematical structure of the above-mentioned orbitals shall be given. The set of orthonormal orbitals is the basis in which the quantum chemical system is expressed. But what do they look like mathematically? For a single electron, the choice of basis follows phenomenological arguments: the electronic density of a system should have a sharp peak at the position of the nuclei and fall off asymptotically for an increasing range. This function can be expressed in terms of a Slater-type orbital function. Such a function for a $1s$ orbital is given in the following equation

$$\phi^{\mathrm{STO}_{abc}}(\zeta, d) = \sqrt{\frac{\zeta^3}{\pi}} e^{-\eta d} \tag{2.55}$$

with the parameter ζ describing the size or diffuseness of the orbital and d denoting the distance of the electron to the nuclei. Figure 2.5 shows that this expression observes the correct behaviour, with a cusp at the central coordinate and an asymptote towards zero at increasing range.

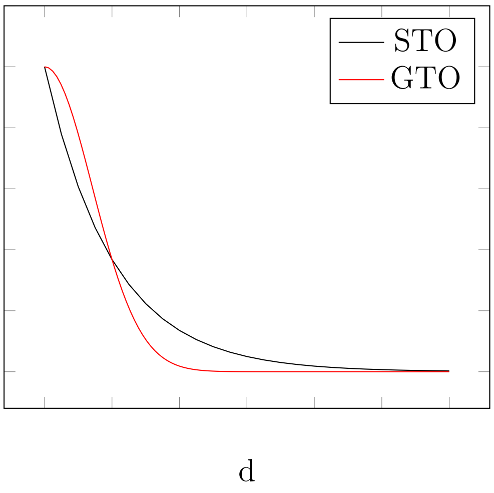


FIGURE 2.5: Slater-type and Gaussian-type $1s$ orbital example[6].

While these Slater-type orbitals exhibit correct behaviour they are generally not used. The reason behind it is that the product of two of these orbitals is comparatively hard to evaluate.

A workaround is the use of Gaussian functions. The product of two Gaussian functions produces another Gaussian function, which is very easy to evaluate. An example for the $1s$ orbital is given in the following equation

$$\phi_{abc}^{GTO}(\alpha, d) = \frac{2\alpha^{3/4}}{\pi} e^{-\alpha d^2} \tag{2.56}$$

However, they do not exhibit the correct behaviour, as can be seen in figure 2.5, having a wrong maximum and long-range behaviour. To remedy this a contraction of multiple types of Gaussian function is used to approximate the Slate-type orbital and recover the cusp and long-range behaviour, as can be seen in figure 2.6.

There are many categories of basis sets which are commonly used. The smallest ones are called minimal basis sets, as they consist of the minimum amount of orbitals needed to represent the occupied orbitals in a system. They follow the naming convention $STO-NG$, with N being the number of primitive Gaussians that approximate the Slater function. Another set of basis functions uses multiple Gaussians with different parameters. They are summarised under the category of multiple-ζ basis function (double-, triple-). Basis sets which use different numbers of Gaussians for different regions of the atom are called split-valence basis sets. An example of these kinds of sets is the ones developed by Pople[73–75], which follow the naming scheme $X-YZ\ldots G$. The first number describes the amount of Gaussians

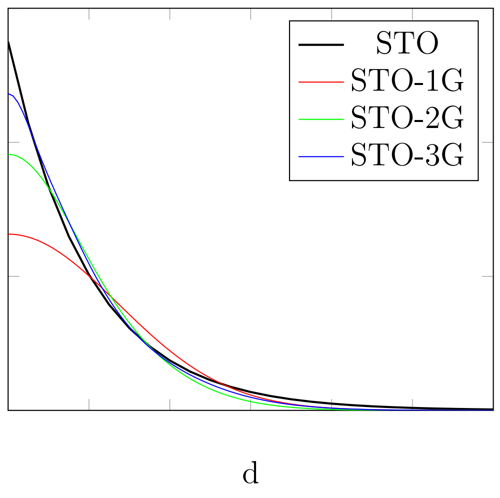


FIGURE 2.6: Minimal basis set example approximating the Slater-type orbital[6].

used in the core region of the atom and the parameters behind the dash describe the multiple-ζ functions of the valence region. Over the years many additions to these categories have been developed. Diffuse functions in the outer regions of the basis sets have been added to help with large excited state descriptions (such as charge-transfer states). Other examples of commonly used basis sets are the ones by Dunning et al.[76] and Ahlrich et al.[77].

2.3.5 The Hartree-Fock Approximation

One of the earliest, still used, methods of computational quantum chemistry, stems from the approximation introduced by Hartree[78] and Fock[79]. It is an ab-initio method, meaning that the calculation is done solely on principles of quantum mechanics without any pre-existing information or parametrisation. The Hartree-Fock method is the starting point of most more sophisticated methods that follow, which includes the method derived in the next chapter.

This section aims to derive the Hartree-Fock approximation in its commonly used matrix form to explain the underlying algorithm and have a foundation of where to go on from. The derivation will not be step by step, but will only cover the most important parts.

Starting from a single determinant wave function $|\Phi\rangle = |\chi_1\chi_2\ldots\chi_n\rangle$ where the spin orbitals are restrained to remain orthonormal $\langle a|b\rangle = \delta_{ab}$ the energy of the electronic system is

$$\begin{aligned}E_0(\{\chi_a\}) = \langle\Phi_0|\hat{\mathcal{H}}|\Phi_0\rangle = &\sum_a \langle\chi_a|\hat{h}|\chi_a\rangle \\ &+ \frac{1}{2}\sum_{ab} \big[\, \langle a(1)b(2)|\frac{1}{r_{12}}|a(1)b(2)\rangle \\ &- \langle a(1)b(2)|\frac{1}{r_{12}} b(1)a(1)\rangle \,\big]\end{aligned} \tag{2.57}$$

where the variational principle is used to minimize the energy. The spin-orbital was shortened to $\chi_a(x_1) = a(1)$. The only flexibility stems from the choice of spin orbitals χ_a. The aim is to minimise the energy equation 2.57. This can be done using Lagrangian multipliers because of the constraint of orthogonality. The Lagrangian function is

$$\mathcal{L}(\{\chi_a\}) = E_0(\{\chi_a\}) - \sum_{ab} \lambda_{ba}(\langle a|b\rangle - \delta_{ab}) \tag{2.58}$$

To minimise the Lagrangian function its derivative $\delta\mathcal{L} = 0$ is calculated. With the energy expression for the Slater determinant 2.57 this leads to the expression

$$\delta\mathcal{L}(\{\chi_a\}) = 0 = \hat{h}(1)a(1) + \sum_b \int \big[b^*(2)\frac{1}{r_{12}}a(1)b(2) - b(2)\frac{1}{r_{12}}b(1)a(2)\big]d2 - \sum_a \lambda_{ab}b(1) \tag{2.59}$$

which can be rewritten as

$$\left[\hat{h}(1) + \sum_b \langle b(2)|\frac{1-\hat{\mathcal{P}}_{12}}{r_{12}}|b(2)\rangle\right] a(1) = \sum_b \lambda_{ab}b(1) \tag{2.60}$$

using the permutation operator

$$\hat{\mathcal{P}}_{12}\chi_a(x_1)\chi_b(x_2) = \chi_a(x_2)\chi_b(x_1) \tag{2.61}$$

This leads to an expression of the form

$$\hat{F}(1)\,|a\rangle = \underline{\lambda}\,|b\rangle \tag{2.62}$$

with $\underline{\lambda}$ denoting the matrix of Lagrangian multipliers. The expression in eq. 2.61 can be transformed to an eigenvalue equation by the transformation of this matrix through the use of a unitary transformation matrix $\mathbf{U} \dagger \mathbf{U} = 1$.

$$\underline{\varepsilon} = \mathbf{U}^{\dagger}\underline{\lambda}\mathbf{U} \tag{2.63}$$

This step also transforms the one-electron Hamiltonian and the spin orbitals. Inserting unity and multiplying with $\mathbf{U}^{\dagger}$ from the left

$$\mathbf{U}^{\dagger}\hat{h}(1)\mathbf{U}\mathbf{U}^{\dagger}\left|\chi(1)\right\rangle + \left[\langle\chi(2)\mathbf{U}|\frac{1-\hat{\mathcal{P}}_{12}}{r_{12}}|\mathbf{U}^{\dagger}\chi(2)\rangle\right]\mathbf{U}^{\dagger}\left|\chi(1)\right\rangle = \mathbf{U}^{\dagger}\underline{\lambda}\mathbf{U}\mathbf{U}^{\dagger}\left|\chi(1)\right\rangle \tag{2.64}$$

$$\hat{h}'(1)\left|\chi'(1)\right\rangle + \left[\langle\chi'(2)|\frac{1-\hat{\mathcal{P}}_{12}}{r_{12}}|\chi'(2)\rangle\right]\left|\chi'(1)\right\rangle = \underline{\varepsilon}\left|\chi'(1)\right\rangle \tag{2.65}$$

The component form of this expression is commonly called the 'canonical' Hartree-Fock expression. The prime mark will be dropped from now on.

$$\left[\hat{h}(1) + \sum_{b}\langle b(2)|\frac{1-\hat{\mathcal{P}}_{12}}{r_{12}}|b(2)\rangle\right]\chi_a(1) = \varepsilon_k\chi_a(1) \tag{2.66}$$

It is common to define the following two quantities

$$\hat{\mathcal{J}}(1) = \sum_{b}\langle b(2)|\frac{1}{r_{12}}|b(2)\rangle \qquad \hat{\mathcal{K}}(1) = \sum_{b}\langle b(2)|\frac{\hat{\mathcal{P}}_{12}}{r_{12}}|b(2)\rangle \tag{2.67}$$

Here the operator $\hat{\mathcal{J}}$ is called the Coulomb operator. It can be interpreted as the average electrostatic repulsion an electron at position 1 feels against an electron at position 2. This is because the integration turns the two-electron operator $\frac{1}{r_{12}}$ into a one-electron operator.

$$\hat{\mathcal{J}}(1)\chi_a(1) = \left(\int d2\frac{\chi_b^*(2)\chi_b(2)}{r_{12}}\right)\chi_a(1) = \left(\int d2\frac{|\chi_b(2)|^2}{r_{12}}\right)\chi_a(1) \tag{2.68}$$

The other operator $\hat{\mathcal{K}}$ is called the exchange operator. It does not have a simple physical interpretation and stems from the antisymmetrical nature of the original one-Slater-determinant wave function. Both operators are commonly bundled with the one-electron operator $\hat{h}$ into the full Fock operator $\hat{\mathcal{F}}$. The main approximation of the Hartree-Fock equation is that the wave function is only a single Slater determinant.

The canonical form of the Hartree-Fock equation is not the one that is used in actual computation. This is done by transforming equation 2.66 into a matrix representation in the basis of a finite basis set (which has basis functions of the form discussed in the previous chapter). Different variants are branching out from this point, depending on the nature of the system described. For closed-shell systems, with no unpaired electrons, one can simplify the equations further. If the additional restriction, that the spin functions for the spin orbitals are the same for alpha and beta spin, is chosen then the most commonly used Hartree-Fock flavour, the restricted closed-shell Roothaan-Hall[59,80,81] matrix equation, is generated.

2.3.5.1 Restricted Closed-Shell Roothaan-Hall

First, the restricted closed-shell form of the Hartree-Fock equation is derived. Assuming that the spin function part of the spin-orbital is the same function for both α and β electrons and that $\langle\alpha(s)|\alpha(s)\rangle = \langle\beta(s)|\beta(s)\rangle = 1$ and $\langle\alpha(s)|\beta(s)\rangle = \langle\beta(s)|\alpha(s)\rangle = 0$ one has

$$f(x_1)\phi_p(r_1)\alpha(s_1) = \epsilon_p\phi_p(r_1)\alpha(s_1) \tag{2.69}$$

where $f(x_1)$ is the one-particle Fock operator

$$f(x_1) = h(r_1) + \sum_p^N \int dx_2 \chi_p^* \frac{1 - \hat{\mathcal{P}}_{12}}{r_{12}} \chi_p(x_2) \tag{2.70}$$

The derivation is identical for the β spin function. Multiplying equation 2.69 with $\alpha^*(s)$ from the left and integrating over the spin results in

$$h(r_1)\phi_q(r_1) + \sum_p \int ds_1 dx_2 \alpha^*(s_1)\chi_p^*(x_2)\alpha \frac{1}{r_{12}}\chi_p(c_2)\alpha(s_1)\phi_p(r_1) \tag{2.71}$$
$$-\sum_p \int ds_1 dx_2 \alpha^*(s_1)\chi_p^*(x_2)\frac{1}{r_{12}}\chi_p(x_1)\alpha(s_2)\phi_p(x_2) = \varepsilon_q\phi_q(r_1)$$

The space of spin orbitals n_e is exactly half for the space of those with α spin and those with β spin, leading to

$$h(r_1)\phi_q(r_1) + \sum_p^{n_e/2} \int ds_1 ds_2 dr_2 \alpha^*(s_1)\phi_p^*(r_2)\alpha^*(s_2)\frac{1}{r_{12}}\phi_p(r_2)\alpha^*(s_2)\alpha(s_1)\phi_q(r_1) \tag{2.72}$$
$$+\sum_p^{n_e/2} \int ds_1 ds_2 dr_2 \alpha^*(s_1)\phi_p^*(r_2)\beta^*(s_2)\frac{1}{r_{12}}\phi_p(r_2)\beta^*(s_2)\alpha(s_1)\phi_q(r_1)$$
$$-\sum_p^{n_e/2} \int ds_1 ds_2 dr_2 \alpha^*(s_1)\phi_p^*(r_2)\alpha^*(s_2)\frac{1}{r_{12}}\phi_p(r_1)\alpha^*(s_1)\alpha(s_2)\phi_1(x_2)$$
$$-\sum_p^{n_e/2} \int ds_1 ds_2 dr_2 \alpha^*(s_1)\phi_p^*(r_2)\beta^*(s_2)\frac{1}{r_{12}}\phi_p(r_1)\beta^*(s_1)\alpha(s_2)\phi_1(x_2) = \varepsilon_q\phi_q(r_1)$$

Integrating over the spins s_1 and s_2 eliminates the last line in the equation and simplifies the first two lines as they become identical. What is left is

$$h(r_1)\phi_p(r_1) + 2\left[\sum_p^{n_e/2} \int dr_2 \phi_p(r_2)\frac{1}{r_{12}}\phi_p(r_2)\right]\phi_q(r_1) \tag{2.73}$$
$$-\left[\sum_p^{n_e/2} \int dr_2 \phi_p^*(r_2)\frac{1}{r_{12}}\phi_q(r_2)\right]\phi_p(r_1) = \varepsilon_p\phi_p(r_1) \tag{2.74}$$

which is the closed-shell spatial orbital form of the Hartree-Fock equation with the Fock operator

$$f(r_1) = h(r_1) + \sum_{q}^{n_e/2} 2J(r_1) - K(r_1) \tag{2.75}$$

These equations are not solved analytically or numerically with a computer, but rather as a set of matrix equations. For this, the spatial orbitals are represented in a finite, given basis (as discussed in the previous chapter).

$$\phi_r = \sum_r C_{\mu r} \psi_\mu \tag{2.76}$$

The orbital ϕ_r represents a molecular orbital, whereas the basis functions typically used are evaluated around atomic centres and are therefore atomic orbitals. This expansion is a further approximation, as a finite basis set is incomplete. The molecular orbitals in the closed-shell Hartree-Fock equation can be expanded in this basis

$$f(r_1) \sum_r C_{\mu,r} \psi_\mu(r_1) = \varepsilon_r \sum_r C_{\mu,r} \psi_\mu(r_1) \tag{2.77}$$

Multiplying with the basis function $\psi_n u(r_1)$ from the left and integrating over it we arrive at the matrix equation

$$\mathbf{FC} = \underline{\varepsilon}\mathbf{SC} \tag{2.78}$$

with $\mathbf{S}$ being the matrix of the overlap integral between the two functions

$$S_{\mu\nu} = \int dr_1 \psi_m u^*(r_1) \psi_n u(r_1) \tag{2.79}$$

C the matrix of expansian coefficients, $\mathbf{F}$ the Fock matrix

$$F_{\mu\nu} = \int dr_1 \psi_\mu^*(r_1) f(r_1) \psi_n u(r_1) \tag{2.80}$$

and $\underline{\varepsilon}$ the diagonal matrix of orbital energies ε_p. One additional step is taken before calculation: the basis functions are typically not orthonormal, so the matrix equation is transformed with a unitary transformation matrix $\mathbf{X}$ so that the overlap matrix becomes unity. The final equation is then

$$\mathbf{F}'\mathbf{C}' = \mathbf{C}'\underline{\varepsilon} \tag{2.81}$$

2.3.5.2 The SCF Algorithm

The algorithm to solve the Roothan-Hall equation is called the self-consistent field method. Because the Fock operator $\hat{\mathcal{F}}$ is dependent on the orbitals itself in the coulomb and exchange terms, solving the Roothan-Hall method has to be done iteratively. The SCF algorithm is sketched in algo. 1.

Algorithm 1: General self-consistent-field algorithm used to calculate the Hartree-Fock method family.

```
1  define nuclei coordinates $R_A$, core charges $Z_A$, define total number of
     electrons $n_e$, basis set $\phi_\lambda$
2  guess set of coefficients $\{c_{p\lambda}\}$ for molecular orbitals
3  calculate $S_{pq}$, $F_{pq}$, $\langle\lambda\mu|\nu\sigma\rangle$
4  diagonalise $S_{pq}$ to get transformation matrix $\mathbf{X}$
5  while residual > threshold do
6      form Fock matrix $\mathbf{F}$
7      calculate the transformed Fock matrix $\mathbf{F}' = \mathbf{X}^\dagger\mathbf{F}\mathbf{X}$
8      diagonalise $\mathbf{F}'$ to obtain $\mathbf{C}'$, $\varepsilon$
9      calculate $\mathbf{C}_{\text{new}} = \mathbf{X}\mathbf{C}'$
10     calculate residual $\mathbf{C}_{\text{new}} - \mathbf{C}_{\text{old}}$
11 calculate expected quantities
```

2.3.6 Recovering Electronic Correlation

The Hartree-Fock method results in around 97 % of the total energy of common systems. The last percents are however important in many chemical systems, which often feature flat energy surfaces. It is important to recover the difference between the exact energy and the Hartree-Fock energy to accurately describe these systems.

$$E_{\text{exact}} - E_{\text{HF}} = E_{\text{correlation}} \tag{2.82}$$

This difference is commonly called the electron-correlation energy. The single Slater determinant ansatz of the Hartree-Fock equation results in the average treatment of the electron-electron interaction term. This is a major approximation to physical systems.

The Hartree-Fock method is used often by computational chemists as a first approximation for large systems or implicitly in the use of higher levels of theory. These post-Hartree-Fock methods aim to recover this electron-correlation energy. As there are many, only two will be introduced in this general theory introduction. One will be computationally exact, however realistically unfeasible, and the other will be less exact, but more commonly used.

2.3.6.1 Configuration Interaction

The simplest method to recover the electron-electron correlation is called configuration interaction. Where on the Hartree-Fock level only one Slater determinant of spin-orbitals is used to approximate the wave function of the system, in configuration interaction it is instead expanded in a sum of all possible combinations of such Slater determinants. These additional determinants differ in the exchange of one, two, three ... orbitals. Such as, if orbital l and m are exchanged, the resulting Slater determinant is

$$|\Phi\rangle = |\chi_1\chi_2 \dots \chi_l\chi_m \dots \chi_m\rangle \rightarrow |\Phi_l^m\rangle = |\chi_1\chi_2 \dots \chi_m\chi_l \dots \chi_m\rangle \tag{2.83}$$

This wave function is called singly excited. If this is done for all possible combinations of exchanging orbitals the resulting method is called full configuration interaction (full CI)

$$|\Phi\rangle = c_0\,|\Phi_0\rangle + \left(\frac{1}{1!}\right)^2 \sum_{ia} c_p^a\,|\Phi_p^a\rangle + \left(\frac{1}{2!}\right)^2 \sum_{ijab} c_{ij}^{ab}\,|\Phi_{ij}^{ab}\rangle + \dots \tag{2.84}$$

Where the prefactor has been added to avoid double counting of the determinants. The Hartree-Fock spin-orbitals resulting from the SCF procedure are used as the underlying spin-orbitals in this method. This wave function is then inserted into the Schrödinger Equation and projected on each configuration state function $\langle \Phi_0 | , |\Phi_p^a\rangle , \ldots$. The resulting matrices of the form $\langle \Phi_0 | \hat{\mathcal{H}} - E_0 | \Phi \rangle$, $\langle \Phi_p^a | \hat{\mathcal{H}} - E_0 | \Phi \rangle$, $\ldots$ are then expanded in a set of basis function and an eigenvalue equation is recovered. These matrices are then diagonalised to gain the correlation energy correction of the configuration interaction method. They can be further simplified by employing Brillouin's theorem (resulting in no coupling between the HF ground state and the single excited determinant) and the Slator-Condon rules for one- and two-particle operator matrix elements (all matrix elements between determinants differing by more than two indices are zero). However, they are still generally unusable.

Full configuration interaction is typically only done for systems with few particles. The reason behind it is the numerical explosion of excited Slater determinants in a full-CI system. While this can be somewhat remedied with the exploitation of integral symmetries or clever choice of different diagonalisation schemes, the number of systems which can be treated with full CI is minuscule.

The configuration interaction picture gives instead rise to further approximative methods, by neglecting higher orders of excited determinants. This leads for example to the configuration interaction method for single and double excitations, CISD.

2.3.6.2 Perturbation Theory

The last method that will be discussed in this chapter is the perturbative treatment of the Schrödinger equation which follows the Rayleigh-Schrödinger formalism. It will result in the starting equation for the method derived in this thesis in the next chapter.

The Rayleigh-Schrödinger equation assumes that we know the solution to a part of the full Hamiltonian, $\hat{\mathcal{H}}_0$, with its solution $E^{(0)}$. The full Hamiltonian shall be the sum of this plus a small perturbation for which one does not know the solution for, $\hat{\mathcal{H}} = \hat{\mathcal{H}}_0 + \lambda \hat{\mathcal{V}}$. The prefactor λ is just an addition which will be useful for the math later. The perturbation is chosen to be small, which means that the eigenfunctions of the full Hamiltonian should not differ much from the eigenfunction of the known Hamiltonian. Therefore both the energy and the wave function of the full system can be approximated in a Taylor series around the known eigenfunctions.

$$E_p = E_p^{(0)} + \lambda E_p^{(1)} + \lambda^2 E_p^{(2)} + \dots \tag{2.85}$$

$$|\Phi_p\rangle = |\Phi_p^{(0)}\rangle + \lambda\,|\Phi_p^{(1)}\rangle + \lambda^2\,|\Phi_p^{(2)}\rangle + \dots \tag{2.86}$$

The known eigenfunctions of $\hat{\mathcal{H}}_0$, $|\Phi_p^{(0)}\rangle$ are orthogonal to the n-th order part of this expansion $\langle\Phi_p^{(0)}|\Phi_p^{(n)})\rangle = 0$ for $n \neq 0$. These equations are inserted into the eigenvalue equation of the full system.

$$\hat{\mathcal{H}}\,|\Phi_p\rangle = E_p\,|\Phi_p\rangle \tag{2.87}$$
$$\left(\hat{\mathcal{H}}_0 + \hat{\mathcal{V}}\right)\left(|\Phi_p^{(0)}\rangle + \lambda\,|\Phi_p^{(1)}\rangle + \lambda^2\,|\Phi_p^{(2)}\rangle + \dots\right) \tag{2.88}$$
$$= \left(E_p^{(0)} + \lambda E_p^{(1)} + \lambda^2 E_p^{(2)} + \dots\right)\left(|\Phi_p^{(0)}\rangle + \lambda\,|\Phi_p^{(1)}\rangle + \lambda^2\,|\Phi_p^{(2)}\rangle + \dots\right) \tag{2.89}$$

Expanding all of these brackets gives rise to parts which can be grouped by their power of λ

$$\begin{aligned}
\hat{\mathcal{H}}^{(0)}\,|\Phi_p^{(0)}\rangle &= E_p^{(0)}\,|\Phi_p^{(0)}\rangle && n=0\\
\hat{\mathcal{H}}^{(0)}\,|\Phi_p^{(1)}\rangle + \hat{\mathcal{V}}\,|\Phi_p^{(0)}\rangle &= E_p^{(0)}\,|\Phi_p^{(1)}\rangle + E_p^{(1)}\,|\Phi_p^{(0)}\rangle && n=1\\
\hat{\mathcal{H}}^{(0)}\,|\Phi_p^{(2)}\rangle + \hat{\mathcal{V}}\,|\Phi_p^{(1)}\rangle &= E_p^{(0)}\,|\Phi_p^{(2)}\rangle + E_p^{(1)}\,|\Phi_p^{(1)}\rangle + E_p^{(2)}\,|\Phi_p^{(0)}\rangle && n=2\\
&\dots
\end{aligned} \tag{2.90}$$

Projecting these functions on the unperturbed ground state by multiplying with $\langle\Phi_p^{(0)}|$ from the left results in

$$\begin{aligned}
E_p^{(0)} &= \langle\Phi_p^{(0)}|\hat{\mathcal{H}}|\Phi_p^{(0)}\rangle\\
E_p^{(1)} &= \langle\Phi_p^{(0)}|\hat{\mathcal{V}}|\Phi_p^{(0)}\rangle\\
E_p^{(2)} &= \langle\Phi_p^{(0)}|\hat{\mathcal{V}}|\Phi_p^{(1)}\rangle
\end{aligned} \tag{2.91}$$

Where $E_p^{(n)}$ is the n-th order energy correction to the unperturbed ground state. To arrive at a form for the second-order energy one can expand the first-order wave function in terms of a complete set, which is chosen to be the set of eigenfunction Φ_p of the known unperturbed Hamiltonian $\sum_n |\Phi_n\rangle \langle\Phi_n| = 1$.

$$E_p^{(2)}) = \langle\Phi_p^{(0)}|\hat{\mathcal{V}}|\Phi_p^{(1)}\rangle = \sum_{n\neq p} \langle\Phi_p^{(0)}|\hat{\mathcal{V}}|\Phi_n\rangle \langle\Phi_n|\hat{\mathcal{V}}|\Phi_p^{(1)}\rangle \tag{2.92}$$

With the expansion for $\langle\Phi_n|\hat{\mathcal{V}}|\Phi_p^{(1)}\rangle$

$$\langle\Phi_n|\hat{\mathcal{V}}|\Phi_p^{(1)}\rangle = \frac{\langle\Phi_n|\hat{\mathcal{V}}|\Phi_p^{(0)}\rangle}{E_p^{(0)} - E_n^{(0)}} \tag{2.93}$$

which is gained by reordering the first order part of equation 2.90 and projecting it on $\langle\Phi_n|$. The final second-order energy correction is then

$$E_p^{(2)} = \sum_{n\neq p} \frac{\langle\Phi_p^{(0)}|\hat{\mathcal{V}}|\Phi_n\rangle \langle\Phi_n|\hat{\mathcal{V}}|\Phi_p^{(0)}\rangle}{E_p^{(0)} - E_n^{(0}} = \sum_{n\neq p} \frac{|\langle\Phi_n|\hat{\mathcal{V}}|\Phi_p^{(0)}\rangle|^2}{E_p^{(0)} - E_n^{(0}} \tag{2.94}$$

Up until now, no choice has been made as to what the unperturbed and known Hamiltonian is. If one chooses the Hamiltonian to be the Hartree-Fock Hamiltonian one arrives at the Møller-Plesset perturbation method. Let the Hartree-Fock Hamiltonian be the sum of the one-particle part and the average treatment potential of the system. The perturbation shall be the difference between the exact potential and the average one.

$$\hat{\mathcal{H}}_0 = \sum_p (h(p) + v(p)) \tag{2.95}$$

$$\hat{\mathcal{V}} = \sum_{p<q} \frac{1}{r_{pq}} - \sum_p v(p) \tag{2.96}$$

With the following integrals which will be used from here on out

$$\langle pq|rs\rangle = \int dx_1 dx_2 \phi_p^*(x_1)\phi_q^*(x_2)\frac{1}{r_{12}}\phi_r(x_1)\phi_s(x_2) \tag{2.97}$$

$$\langle pq||rs\rangle = \langle pq|rs\rangle - \langle pq|sr\rangle \tag{2.98}$$

The second-order correction to the energy was singled out because the zeroth and first-order energy only recovers the general Hartree-Fock energy. The Hartree-Fock ground state is in this formulation $|\Phi^{(0)}\rangle$. The states $|\Phi_{pq}^{rs}\rangle$ are double excited Hartree-Fock states

$$\hat{\mathcal{H}}_0 |\Phi_{pq}^{rs}\rangle = \Big(E_0^{(0)} - (\varepsilon_p + \varepsilon_q + \varepsilon_r - \varepsilon_s)\Big) |\Phi_{pq}^{rs}\rangle \tag{2.99}$$

Where the electrons of spin orbitals p, q have been promoted to the orbitals r, s. Together with the expression of for the coulomb operator

$$\langle \Phi_0|\frac{1}{r_{ij}}|\Phi_{pq}^{rs}\rangle = \langle pq||rs\rangle \tag{2.100}$$

and removing the sum restrictions in favour of a full summation with prefactors to avoid double counting one arrives at the second-order correction to the Hartree-Fock energy

$$E^{(2)} = \frac{1}{4}\sum_{pqrs} \frac{|\,\langle pq||rs\rangle\,|^2}{\varepsilon_p + \varepsilon_q - \varepsilon_r - \varepsilon_s} \tag{2.101}$$

Going forward this thesis will adhere to the general naming scheme for orbital indices. The indices $i, j, \ldots$ will denote occupied orbitals and $a, b, \ldots$ unoccupied orbitals. The indices $p, q, \ldots$ will be left for general orbitals.

Equation 2.101 is formulated in the space of spin-orbitals. The spin functions complicate the mathematical treatment during computation, however. It is therefore common to integrate them out for systems where spin effects are not of interest. This is generally the case with closed-shell molecules, which have only paired electrons of both spins in each shell. Starting from the above equation, which in the new naming scheme is

$$E^{(2)} = \frac{1}{4}\sum_{ijab} \frac{|\langle ij||ab\rangle|^2}{\varepsilon_a + \varepsilon_b - \varepsilon_i - \varepsilon_j} \tag{2.102}$$

One can fully expand the numerator in the squared absolute as

$$\begin{aligned}\sum_{ijab} |\langle ij||ab\rangle|^2 &= \sum_{ijab} |\langle ij|ab\rangle - \langle ij|ba\rangle|^2 \\ &= \sum_{ijab} \langle ij|ab\rangle \langle ab|ij\rangle + \sum_{ijab} \langle ij|ba\rangle \langle ba|ij\rangle \\ &- \sum_{ijab} \langle ij|ab\rangle \langle ba|ij\rangle - \sum_{ijab} \langle ij|ba\rangle \langle ab|ij\rangle \\ &= t_1 + t_2 + t_3 + t_4\end{aligned} \tag{2.103}$$

All indices within closed summations are free to be relabelled. The spin-orbitals a, b in integral t_2 can therefore be switched which lets $t_2 = t_1$. Adding both together leads to a prefactor of 2. The only integrals surviving the orthonormality factor of the spin function $\langle a|b\rangle = 0$ after integration are the integrals with the spin combinations $\langle \alpha\alpha|\alpha\alpha\rangle$, $\langle \alpha\beta|\alpha\beta\rangle$, $\langle \beta\alpha|\beta\alpha\rangle$, $\langle \beta\beta|\beta\beta\rangle$. The symmetry of t_1 lets all four variants survive, resulting in an additional factor of $2^2 = 4$ after spin integration.

For the same reason, one can prove $t_3 = t_4$. Because of the cross of indices the only spin combinations surviving are $\langle \alpha\alpha|\alpha\alpha\rangle$ and $\langle \beta\beta|\beta\beta\rangle$. The additional prefactor is therefore 2. The sum over all spin orbitals $\sum^n$ is halved in closed-shell systems as there are no spin functions to produce two spin orbitals per spatial orbital.

$$\begin{aligned}t_1 + t_2 + t_3 + t_4 &= 2t_1 - 2t_3 \\ &= 8\sum_{ijab}^{n/2} \langle ij|ab\rangle \langle ab|ij\rangle - 4\sum_{ijab}^{n/2} \langle ij|ba\rangle \langle ab|ij\rangle\end{aligned} \tag{2.104}$$

Where the summation limits now indicate the spatial character of the wave functions. Inserting this equation in 2.102 results in the closed-shell form of the Møller-Plesset perturbation energy correction of second order.

$$E^{(2)} = \sum_{ijab}^{n/2} \frac{2\langle ij|ab\rangle \langle ab|ij\rangle - \langle ij|ba\rangle \langle ab|ij\rangle}{\varepsilon_a + \varepsilon_b - \varepsilon_p - \varepsilon_j} \qquad (2.105)$$

This form will be the playing ground for the quadrature scheme derived in this thesis in the next chapter.

Chapter 3

Derivation of Quadrature MP2

This chapter contains the derivation of the quadrature method presented in this thesis. Section 3.1 introduces the specific theoretical background to understand the derivation. Afterwards, the derivation will be explained in detail in section 3.2. Finally, the first energy benchmark is given in section 3.3 and shows the performance of the method in terms of recovering the full MP2 energy.

3.1 State of the Art

This thesis is concerned with reformulating integrals and terms used in computational chemistry into lower-scaling massively parallel forms. The derivation in this chapter will do so with the closed-shell form of the Møller-Plesset perturbation theory of second order.

$$E^{(2)} = \sum_{ijab}^{n/2} \frac{2 \langle ij|ab \rangle \langle ab|ij \rangle - \langle ij|ba \rangle \langle ab|ij \rangle}{\varepsilon_a + \varepsilon_b - \varepsilon_i - \varepsilon_j} \tag{3.1}$$

This integral scales naively with $\mathcal{O}(O^2V^2)$ for the calculation, with O and V being the number of occupied and virtual orbitals respectively. This scaling represents the computational complexity. The memory complexity is dominated by the fourth-ranked tensors of the form $\langle pq|rs \rangle$ which have to be evaluated and saved in memory for the derivation.

$$\langle pq|rs\rangle = \int dx_1 dx_2 \psi_p^*(x_1)\psi_q^*(x_2)\frac{1}{r_{12}}\psi_r(x_1)\psi_s(x_2) \tag{3.2}$$

Initial evaluation of these integrals is done in the atomic orbital representation $\langle\mu\nu|\lambda\sigma\rangle$. This integral has to be transformed into the molecular orbital representation, which scales with $\mathcal{O}(NO^2V^2)$ computationally, N being the number of atomic orbitals.

$$\langle pq|rs\rangle = C^T_{\mu p}C^T_{\nu q}\left\langle\mu\nu|\lambda\sigma\right\rangle C_{\lambda r}C_{\lambda s} \tag{3.3}$$

It is also the largest object held in memory and therefore gives the method its total memory complexity of $\mathcal{O}(N^4)$.

There are numerous optimisations focused on improving the computational and memory scaling of the MP2 method. They generally focus on reformulating the above-mentioned tensors in lower scaling forms and/or exploiting local orbital character through additional integral screening to reduce the matrix elements which have to be computed. The following section will summarise some of these techniques leading up to the quadrature reformulation.

3.1.1 Resolution-of-the-Identity Approximation

The Resolution-of-the-Identity (RI) approximation[82–84], which is sometimes also called Density Fitting (DF), aims to express the two-electron-repulsion integrals (ERI) in a lower dimensional form. These matrices are generally sparse, meaning that a large part of their entries are close to zero. One can reformulate the matrix into a form of smaller dimensions and save computational time and storage requirements. They also often have linear dependencies, which can be removed through standard schemes.

The RI approximation does so by casting the fourth-ranked tensors into three-ranked forms with the help of an orthonormal auxiliary basis, X or Y, as per

$$\langle\mu\nu|\lambda\sigma\rangle = \sum_{XY}\left\langle\mu\nu|X\right\rangle\left\langle X|Y\right\rangle^{-1}\left\langle Y|\lambda\sigma\right\rangle \tag{3.4}$$

where the following integral is a Coulomb matrix of the form

$$\langle X|Y\rangle = \int dr_1 dr_2 \frac{X(r_1)Y(r_2)}{r_{12}} \tag{3.5}$$

This reexpression is often written by bundling half of the integral 3.4 into RI vectors

$$Z^X_{\mu\nu} = \sum_Y \langle X|Y\rangle^{-\frac{1}{2}} \langle Y|\mu\nu\rangle \tag{3.6}$$

which approximates the original integral through

$$\langle \mu\nu|\lambda\sigma\rangle = \sum_X B^X_{\mu\nu} B_{\lambda\sigma} \tag{3.7}$$

This approximation is done on the atomic orbital level. It lowers the memory complexity of the integrals to $\mathcal{O}(N^2X)$, X being the size of the auxiliary basis and the computational complexity of the transformation bottleneck step to $\mathcal{O}(NOVX)$. The auxiliary bases are prefitted to atom-centric systems, which may not be the best approximation depending on the system.

This approximation is one of the most widely used ones to lower the complexity of computational chemistry methods. In some software suits it is the standard for the calculation of the Møller-Plesset or the Coupled Cluster method families.

However, the choice of the best auxiliary basis set for a system disturbs the general black-box nature of most ab inito calculations. This is remedied by the on-the-fly generation of the auxiliary basis set through the Cholesky Decomposition technique.

3.1.1.1 Cholesky Decomposition

The Cholesky Decomposition[85,86] is a technique often used in linear algebra to cast Hermitian matrices into a product of their lower triangular matrix and its conjugate transpose. It was first applied to the area of quantum chemistry in 1977 by Bebe and Linderberg[87].

The Cholesky Decomposition algorithm, which is summarised for the Q-Chem[88] implementation by Epifanovksy[7,8], is given in figure 3.1. Diagonal elements of the ERI are computed and the largest one is chosen. The matrix elements with this diagonal and two of the four indices is calculated and a Cholesky vector is calculated, which has the form of the auxiliary basis function used in the RI approximation. This is then subtracted from the diagonal elements. The largest element is chosen once again and checked against a threshold. If it is above the threshold, a new Cholesky vector is calculated and subtracted. Once the threshold is reached the basis set generation is finished and the program starts a RI approximated version of the method with the Cholesky vectors as the auxiliary basis functions.

The Cholesky Decomposition allows for on-the-fly generation of the RI auxiliary basis. It can be arbitrarily accurate, as its algorithm involves a threshold parameter which the generation of the basis function vectors is checked against. This ansatz provides much more flexibility compared to the rigid precalculated basis sets used in standard RI, at the cost of the calculation of the auxiliary vectors for each new system.

3.1.2 Laplace-transformed MP

As mentioned before, the complexity of many post-HF methods stems from the need for transformation and storage of the fourth-ranked ERIs. Additionally, the form of the integral in eq. 3.1 introduces certain constraints to the choice of orbitals, which have to be orthonormal and canonical (diagonalising the Fock matrix). These constraints stem from the interconnection of the orbitals in the integrals with the indices of the energy denominator. They are coupled. This also affects the parallel efficiency of the integral evaluation, as the large tensor object of the fourth-ranked ERI has to be saved and traversed. Almlöf[1,51,52] has shown in a series of papers that one can decouple the indices of the (spin-orbital) MP2 correlation energy by transforming the energy denominator with the Laplace transform, see eq. 3.9.

$$E^{(2)} = -\frac{1}{4}\sum_{ijab}\frac{|\,\langle ij||ab\rangle^2\,|}{(\varepsilon_a+\varepsilon_b-\varepsilon_i-\varepsilon_j)} \tag{3.8}$$

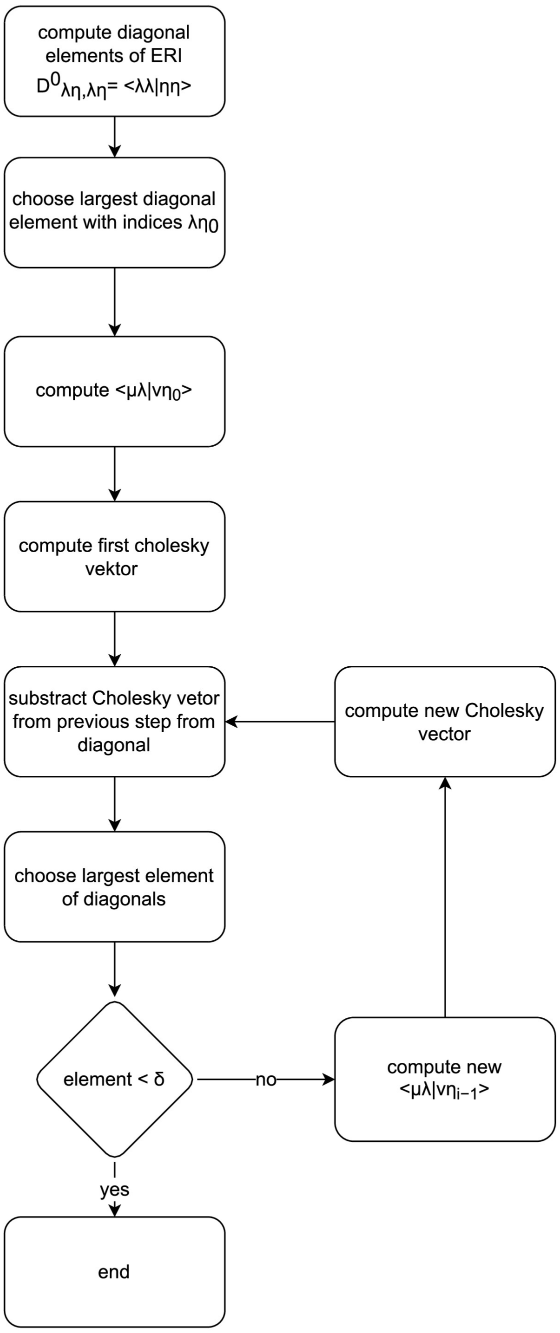

compute diagonal elements of ERI $D^0_{\lambda\eta,\lambda\eta} = \langle\lambda\lambda|\eta\eta\rangle$
choose largest diagonal element with indices $\lambda\eta_0$
compute $\langle\mu\lambda|\nu\eta_0\rangle$
compute first cholesky vektor
substract Cholesky vetor from previous step from diagonal
choose largest element of diagonals
element < δ
no
yes
end
compute new $\langle\mu\lambda|\nu\eta_{i-1}\rangle$
compute new Cholesky vector

FIGURE 3.1: Cholesky algorithm of the Q-Chem implementation by Epifanovsky et al.[7,8].

$$\frac{1}{\varepsilon_a + \varepsilon_b - \varepsilon_i - \varepsilon_j} = -\int_0^\infty d\tau e^{(\varepsilon_a+\varepsilon_b-\varepsilon_i-\varepsilon_j)\tau} \tag{3.9}$$

Where the dependence on the integration grid can be transferred to the orbitals through

$$\psi_p(t) = \psi_p(0)e^{\pm\varepsilon_p t} \tag{3.10}$$

This has the effect that these orbitals do not have to be orthogonal or canonical, making the choice much more flexible. Especially in large, extended systems, transformed orbitals with high localised properties can be chosen and exploited. Localisation increases the sparsity of the resulting matrices, which in turn reduces the number of matrix elements that need to be saved for an accurate representation, ultimately reducing the dimensions of the integrals.

3.1.3 Monte Carlo Stochastic MP2 (MC-MP2)

Willow and Hirata[53] used the above-described decoupling of the energy denominator to reformulate the MP2 integral into the sum of two 13-dimensional integrals. With this, they eliminated the need for the transformation of the full ERI from its atomic orbital representation into its molecular orbital one. The integral is of the form

$$\begin{aligned} E^{(2)} = \int dr_1 dr_2 dr_3 dr_4 \Big(& \frac{o(r_1,r_3,\tau)o(r_2,r_4,\tau)v(r_1,r_4,\tau)o(r_2,r_3,\tau)}{|r_{12}r_{34}|} \\ & - \frac{2o(r_1,r_3,\tau)o(r_2,r_4,\tau)v(r_1,r_3,\tau)v(r_2,r_4,\tau)}{|r_{12}r_{34}|}\Big) \end{aligned} \tag{3.11}$$

with the occupied orbital contributions of

$$o(r_1,r_3,\tau) = \sum_i^{\text{occ.}} \phi_i^*(r_1)\phi_i(r_3)e^{-\varepsilon_i\tau} \tag{3.12}$$

and the virtual orbital contributions of

$$v(r_1, r_3, \tau) = \sum_{a}^{\text{occ.}} \phi_a^*(r_1)\phi_a(r_3)e^{\varepsilon_a \tau} \tag{3.13}$$

where equations 3.12 and 3.13 are the only quantities which have to be manipulated and saved, lowering the memory complexity to $\mathcal{O}(P^2KN)$, with P being the size of the spatial and K the size of the one-dimensional integration grid introduced through the Laplace transformation. However, the curse of dimensionality prohibits the straightforward evaluation of the 13-dimensional integral 3.11. The authors favoured a Monte Carlo integration approach to evaluate the target integral, where samples from a random distribution are drawn, weighted and summed up to approximate the integral through the law of large numbers.

$$I = \int dx f(x) \approx \frac{1}{N}\sum_{n=1}^{N} \frac{f(x_n)}{w(x_n} \tag{3.14}$$

This approach has the advantage over the previous formulation of stochastically evaluated MP2[89–91] of not needing the full four indices ERI tensor, circumventing the largest spatial complexity and the need for the expensive integral transformation. The Monte Carlo integration also exposed massive parallelism for the integral evaluation, as all random walkers act independently in exploring the multi-dimensional space. This was shown in a GPU optimised implementation[54] which shows quadratic speed up for an increase in computational complexity of $\mathcal{O}(N^3)$ or better for smaller systems, and a linear speed up for most larger systems.

The disadvantages of this ansatz lie in its stochastic nature and the singularities of its spatial integration grid where $r_{12} \approx 0$.

3.1.4 MPI/OpenMP hybrid parallel quadrature MP2-F12

Ishimura and Ten-no[2] published a hybrid OpenMP/MPI parallelised algorithm for orbital invariant quadrature-based Møller-Plesset theory (MP2-F12) in 2011, where the two-electron integrals were decomposed in terms of two-dimensional Coulomb integrals and quadrature evaluated molecular orbitals.

$$\langle ij|ab\rangle = \sum_P \bar{\phi}_a(r_P)\phi_i(r_P)\,\langle j|r_{1P}^{-1}|b\rangle \tag{3.15}$$

$$\bar{\phi}_a = \omega(r_P)\phi_a(r_P) \tag{3.16}$$

$$\langle j|r_{1P}^{-1}|b\rangle = \int dr_1 \frac{\phi_b(r_1)\phi_j(r_1)}{|r_1 - r_P|} \tag{3.17}$$

$$\tag{3.18}$$

They achieved near-linear parallel efficiency for their test systems of anthracene, tetracene and coronene on Becke-style integration grids with up to 8192 CPU cores. Their implementation suffered however from the required communication time, which made up $22 - 43\%$ of the total execution time on 8192 cores. As the indices were not decoupled the computational scaling for their implementation is $\mathcal{O}(O^2V^2P)$.

This partitioning of the two-electron integral and their parallel algorithm resembles the pseudospectral formulation of MP2 by Martinez[92].

3.1.5 Quadrature of opposite-spin MP2 (Q-MP2-OS)

The decoupling of the energy denominator was also the starting point for Barca and Gill[3], which used a quadrature treatment to reformulate the opposite-spin part of the MP2 integral into a connected sum of low dimensional entities

$$E_{\mathrm{OS}}^{(2)} \approx +\sum_k^K \sum_{pq}^G T_{pq}^k \bar{T}_{qp}^k \tag{3.19}$$

with

$$
\begin{aligned}
T^k_{pq} &= \sum_{\nu a}^{N} S^{kp}_{a\nu} V^{kq}_{\nu a} \\
V_{\nu a} &= \sum_{\beta}^{N} Y_{\nu\beta} U^q_{a\beta} \\
S^{kp}_{a\nu} &= \sum_{\mu}^{N} X^k_{\mu a} D^p_{\mu\nu}
\end{aligned}
\tag{3.20}
$$

where the quantities U and D are defined as

$$
\begin{aligned}
D^p_{\mu\nu} &= \sqrt{w_p} D_{\mu\nu}(r_p) = \sqrt{w_p}\phi_\mu(r)\phi_\nu(r) \\
U^p_{\lambda\sigma} &= \sqrt{w_p} U_{\lambda\sigma}(r_p) = \sqrt{w_p}\int dr' \frac{\phi_\lambda(r')\phi_\sigma(r')}{|r_p - r'|} dr' \\
\int dr D_{\mu\nu}(r) U_{\lambda\sigma}(r) &\approx \sum_{p}^{G} D^p_{\mu\nu} U^p_{\lambda\sigma}
\end{aligned}
\tag{3.21}
$$

and the only entities that need to be saved are

$$
X_{\mu a}(t) = \sum_{i}^{N_o} C_{\mu i} C_{ai} e^{\varepsilon_i} \tag{3.22}
$$

$$
Y_{\nu\beta}(t) = \sum_{i}^{N_v} C_{\nu a} C_{\beta a} e^{-\varepsilon_a} \tag{3.23}
$$

These sums are carried out over a one-dimensional grid determined by nonlinear min-max optimizations between the lowest and highest energy of its active orbitals. The one-dimensional grid consisted of a low grid point count of 6 and the molecular grid used was similar to the standard one employed in density functional theory, which uses Euler-Maclaurin and Lebedev quadrature rules with Becke weights. It was additionally heavily pruned. They employed multiple screening schemes, such as screening of the relevant shell pairs or cutoff strategies for the molecular grid points, to arrive at an algorithm which scales with a complexity of $\mathcal{O}(N^2)$ with a large prefactor resulting from the molecular grid.

This quadrature ansatz has multiple advantages. It is formulated in the atomic orbital representation, eliminating the need for the costly transformation of the ERI integrals, with the evaluation of the coefficient matrix quantities through the X and Y entities. It is also embarrassingly parallel.

Generally speaking, many methods used in computational chemistry exhibit low parallel efficiency, which is essential for modern high-performance computation clusters. Their testing of the Q-MP2-OS method shows a near-ideal speed up of around $\sim$ 99 % efficiency. They mentioned however that transformation of the same-spin component of the MP2 correlation energy, which is missing for a full MP2 formulation, is more involved and not easily done.

The lower scaling, both in computational and memory complexity, and the ideal parallelisation possibilities are the reason for the next section, which employs a similar scheme for the full MP2 correlation energy integral.

3.2 Derivation of Q-MP2

The starting point of the derivation[6], which follows the Ideas by Almlöf[51], Häser[1,52], Ishimura[2], Barca[3] and Bloomfield[4] is the closed-shell form of the complete Møller-Plesset perturbation theory of second order (eq. 3.25) in its molecular orbital representation. It will be split into two parts for better readability. The summation over the indices i, j and a, b are carried out over the full space of occupied and virtual orbitals.

$$\begin{aligned} E_{MP2} &= \sum_{ij}\sum_{ab} \frac{\langle ij|ab\rangle \langle ab|ji\rangle - 2\langle ij|ab\rangle \langle ab|ij\rangle}{\varepsilon_a + \varepsilon_b - \varepsilon_i - \varepsilon_j} && (3.24) \\ &= B - A && (3.25) \end{aligned}$$

with

$$A = \sum_{ij}\sum_{ab} \frac{2\langle ij|ab\rangle \langle ab|ij\rangle}{\varepsilon_a + \varepsilon_b - \varepsilon_i - \varepsilon_j} \tag{3.26}$$

and

$$B = \sum_{ij}\sum_{ab} \frac{\langle ij|ab\rangle \langle ab|ji\rangle}{\varepsilon_a + \varepsilon_b - \varepsilon_i - \varepsilon_j} \tag{3.27}$$

Following Almlöf[1,52], the energy denominator is transformed and then approximated by a one-dimensional quadrature (eq. 3.28 and 3.29).

$$\frac{1}{A} \approx \int_0^1 t^{A-1} dt \approx \sum_K^{n_K} \omega_K t_K^{A-1} \tag{3.28}$$

Where the summation over K is carried out over all points in the integration grid.

$$\frac{1}{\varepsilon_a + \varepsilon_b - \varepsilon_i - \varepsilon_j} \approx \int_0^1 t^{\varepsilon_a+\varepsilon_b-\varepsilon_i-\varepsilon_j-1} dt \approx \sum_K^{n^K} \omega^K t^{\varepsilon_a+\varepsilon_b-\varepsilon_i-\varepsilon_j-1} \tag{3.29}$$

Secondly, the integrals are expanded and restructured, see equation 3.33. An analytical integration over a coulomb-type integral leads to an integral being dependent only on its second set of coordinates. This integral is readily available after the SCF is carried out in a program and has to be evaluated on the grid. If the above transformation of the energy denominator is carried out the indices can be reorganized into a lower scaling form.

Finally, to deal with the set of three spatial coordinates a quadrature over a spatial grid is employed. Once again, the summations are carried out over the full set of grid points.

$$\sum_{ijab} \langle ij|ab\rangle = \int d\mathbf{r} \int d\mathbf{r}' \frac{\phi_i(\mathbf{r})\phi_j(\mathbf{r}')\phi_a(\mathbf{r})\phi_b(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|} \tag{3.30}$$

$$= \sum_{ijab} \int d\mathbf{r}\phi_i(\mathbf{r})\phi_a(\mathbf{r}) \left(\int d\mathbf{r}' \frac{\phi_j(\mathbf{r})\phi_b(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|} \right) \tag{3.31}$$

$$= \int d\mathbf{r} \left(\sum_i \phi_i(\mathbf{r}) \right) \left(\sum_a \phi_a(\mathbf{r}) \right) \left(\sum_{jb} I_{jb}(\mathbf{r}) \right) \tag{3.32}$$

$$= \sum_P^{n_P} \omega_P \left(\sum_i \phi_i^P \right) \left(\sum_a \phi_a^P \right) \left(\sum_{jb} I_{jb}^P \right) \tag{3.33}$$

These steps will be now carried out for the two parts of the energy expression. Starting with the fragment A the integrals are expanded

$$A = 2\sum_{ij}\sum_{ab}\left(\int dr_1 dr_2 \phi_i(r_1)\phi_j(r_2)\phi_a(r_1)\phi_b(r_2)r_{12}^{-1}\right)\times \tag{3.34}$$

$$\frac{\left(\int dr_1 dr_2 \phi_a(r_1)\phi_b(r_2)\phi_i(r_1)\phi_j(r_2)r_{12}^{-1}\right)}{\varepsilon_a+\varepsilon_b-\varepsilon_i-\varepsilon_j} \tag{3.35}$$

Then the analytical integration is carried out

$$A = 2\sum_{ij}\sum_{ab}\int dr_1 \phi_i(r_1)\phi_a(r_1)\int dr_2\left(\phi_j(r_2)\phi_b(r_2)r_{12}^{-1}\right)\times \tag{3.36}$$

$$\frac{\left(\int dr_1 \phi_a(r_1)\phi_i(r_1)\int dr_2\left(\phi_b(r_2)\phi_j(r_2)r_{12}^{-1}\right)\right)}{\varepsilon_a+\varepsilon_b-\varepsilon_i-\varepsilon_j} \tag{3.37}$$

and the transformation of the energy denominator is employed

$$A = 2\sum_{ij}\sum_{ab}\sum_{K}\omega^K\left(\int dr_1 \phi_i(r_1)\phi_a(r_1)I_{jb}(r_1)\right)\times \tag{3.38}$$

$$\left(\int dr_1 \phi_a(r_1)\phi_i(r_1)I_{jb}(r_1)\right)t_K^{\varepsilon_a+\varepsilon_b-\varepsilon_i-\varepsilon_j-1} \tag{3.39}$$

with the integrals

$$I_{jb}(r_1) = I_{bj}(r_1) = \int dr_2 \frac{\phi_j(r_2)\phi_b(r_2)}{|r_1-r_2|} \tag{3.40}$$

These integrals, in their atomic orbital form, are the largest objects of the method that have to be stored. Now two quadratures are employed to deal with the two sets of three-dimensional coordinates

$$A = 2\sum_{ij}\sum_{ab}\sum_{K}\omega^K \int dr \int dr' \phi_i(r)\phi_a(r)I_{jb}(r)\phi_a(r')\phi_i(r')I_{jb}(r')t_K^{\varepsilon_a+\varepsilon_b-\varepsilon_i-\varepsilon_j-1} \tag{3.41}$$

$$A = 2\sum_{ij}\sum_{ab}\sum_{K}\omega^K \sum_{P}\omega^P \sum_{P'}\omega^{P'} \phi_i(r)\phi_a(r)I_{jb}(r)\phi_a(r')\phi_i(r')I_{jb}(r')t_K^{\varepsilon_a+\varepsilon_b-\varepsilon_i-\varepsilon_j-1} \tag{3.42}$$

the quadrature of the second set of coordinates is renamed for better readability. Additionally, the parameter r will be dropped from now on and the exponents K, P, Q will denote the indices of the quadrature grids, to distinguish them from the molecular orbital indices (except for the one-dimensional integrand t to avoid confusion with the energy exponents).

$$A = 2\sum_{ij}\sum_{ab}\sum_{K}\omega^K \sum_{P}\omega^P \sum_{Q}\omega^Q \phi_i^P \phi_a^P I_{jb}^P \phi_a^Q \phi_i^Q I_{jb}^Q t_K^{\varepsilon_a+\varepsilon_b-\varepsilon_i-\varepsilon_j-1} \tag{3.43}$$

Now the summation can be reorganized and the molecular orbital indices regrouped to arrive at a lower dimensionality in terms of the system size.

$$A = 2\sum_{K}\omega^K \sum_{P}\omega^P \sum_{Q}\omega^Q \left(\sum_{i} \phi_i^P \phi_i^Q t_K{-\varepsilon_i}\right) \times \tag{3.44}$$

$$\left(\sum_{a} \phi_a^P \phi_a^Q t_K^{\varepsilon_a}\right) \times \tag{3.45}$$

$$\left(\sum_{jb} I_{jb}^P I_{jb}^Q t_K^{-\varepsilon_j+\varepsilon_b-1}\right) \tag{3.46}$$

which leads to the final expression of

$$A = 2\sum_K \omega^K \sum_P \omega^P \sum_Q \omega^Q O^{KPQ} V^{KPQ} C^{KPQ} \tag{3.47}$$

These steps are repeated for the second part B. Once again the integrals are expanded

$$B = \sum_{ij}\sum_{ab}\left(\int dr_1 \phi_i(r_1)\phi_a(r_1)\int dr_2 \left(\phi_j(r_j)\phi_b(r_2)r_{12}^{-1}\right)\right)\times \tag{3.48}$$

$$\frac{\left(\int dr_1 \phi_a(r_1)\phi_j(r_1)\int dr_2 \left(\phi_b(r_2)\phi_i(r_2)r_{12}^{-1}\right)\right)}{\varepsilon_a+\varepsilon_b-\varepsilon_i-\varepsilon_j} \tag{3.49}$$

And the analytical integration over the second set of coordinates in each integral carried out

$$B = \sum_{ij}\sum_{ab}\left(\int dr_1 \phi_i(r_1)\phi_a(r_1)I_{jb}(r_1)\right)\times \tag{3.50}$$

$$\frac{\left(\int dr_1 \phi_a(r_1)\phi_j(r_1)I_{ib}(r_1)\right)}{\varepsilon_a+\varepsilon_b-\varepsilon_i-\varepsilon_j} \tag{3.51}$$

The transformation of the energy denominator is then employed

$$B = \sum_{ij}\sum_{ab}\sum_K \omega^K \left(\int dr_1 \phi_i(r_1)\phi_a(r_1)I_{jb}(r_1)\right)\times \tag{3.52}$$

$$\left(\int dr_1 \phi_a(r_1)\phi_j(r_1)I_{ib}(r_1)\right) t^{\varepsilon_a+\varepsilon_b-\varepsilon_i-\varepsilon_j} \tag{3.53}$$

and the two remaining integrations are approximated by the quadratures

$$B = \sum_{ij}\sum_{ab}\sum_K \omega_K \sum_P \omega^P \sum_Q \omega^Q \phi_i^P \phi_a^P I_{jb}^P \phi_a^Q \phi_j^Q I_{ib}^Q t^{\varepsilon_a+\varepsilon_b-\varepsilon_i-\varepsilon_j-1} \tag{3.54}$$

Finally, the summations are reorganized to minimize the scaling

$$B = \sum_K \omega_K \sum_P \omega^P \sum_Q \omega^Q \phi_i^P \left(\sum_a \phi_a^P \phi_a^Q t^{\varepsilon_a}\right) \left(\sum_b \left(\sum_i I_{ib}^P \phi_i^Q t^{-\varepsilon_i+\varepsilon_b/2}\right) \left(\sum_i \phi_i^P I_{ib}^Q t^{-\varepsilon_i+\varepsilon_b/2}\right)\right) \tag{3.55}$$

Which leads to the following expression

$$B = \sum_K \omega_K \sum_P \omega^P \sum_Q \omega^Q V^{KPQ} D^{KPQ} \tag{3.56}$$

The final energy expression for the quadrature-based Møller-Plesset perturbation theory of second order is therefore

$$E^{Q-MP2} = \sum_K \omega_K \sum_P \omega^P \sum_Q \omega^Q (D^{KPQ} - 2O^{KPQ} C^{KPQ}) V^{KPQ} \tag{3.57}$$

$$O^{KPQ} = \sum_i \phi_i^P \phi_i^Q t^{-\varepsilon_i} \qquad V^{KPQ} = \sum_a \phi_a^P \phi_a^Q t^{\varepsilon_a} \tag{3.58}$$

$$C^{KPQ} = \sum_{ib} I_{ib}^P I_{ib}^Q t^{-\varepsilon_i+\varepsilon_b-1} \tag{3.59}$$

$$D^{KPQ} = \sum_b \left(\sum_i I_{ib}^P \phi_i^Q t^{-\varepsilon_i+\varepsilon_b/2}\right) \left(\sum_i \phi_i^P I_{ib}^Q t^{-\varepsilon_i+\varepsilon_b/2}\right) \tag{3.60}$$

The final scaling of the integral employing the quadratures is $\mathcal{O}(KPQOV)$ in terms of computational work and $\mathcal{O}(POV)$ in terms of necessary storage. The one-dimensional integration, as will be shown in chapter 5, can sufficiently be carried out with $K < 10$. The bottleneck of the calculation is therefore the number of three-dimensional grid points P and Q. This leads to a larger realistical scaling compared to the transformation of the largest integral $I^P_{\mu\nu}$, which has the theoretically higher scaling in terms of system size of $\mathcal{O}(PNOV)$, as $P \geq\geq N$ without grid pruning. The choice of the spatial grid is therefore paramount to the performance of the calculation.

The only objects that are needed for the calculation are the Fock matrix $F_{\mu\nu}$ (or the molecular orbital energies), the P grid evaluated orbitals ϕ^P_ν, the evaluated integral $I^P_{\mu\nu}$, the coefficient matrix for the AO-to-MO transformation, the grid weights of P and the grid points and weights of K.

3.3 Energy Benchmark

The method was implemented (see chapter 4) and tested on a set of small and medium-sized molecules.

3.3.1 Computational Details

The accuracy of the method in reproducing the parent MP2 energy was tested with a set of small and medium molecules at different basis set levels. The molecules are: H_2, H_2O, CO, $CH4$, H_2CO, H_2CO, H_3COH, NH_3, anthracene and poryphrine. All calculations were done with a development version of `libqqc`, interfaces with `Q-Chem` 5.2.2 as its host program.

Geometry optimisations were done on the MP2 level of theory with the same basis set (STO-3G, 3-21g, 6-31g) as the individual calculations. The Q-MP2 calculation werewas done on a 20/38 Becke-weighted grid with $N_K = 4$ for the one-dimensional quadrature. All calculations were run on the computational cluster of the Dreuw group (see chapter 4 for details).

3.3.2 Results

As a first step, the accuracy of the Q-MP2 method in reproducing the energy of the MP2 method was tested. For this, consistent integration grids were chosen,

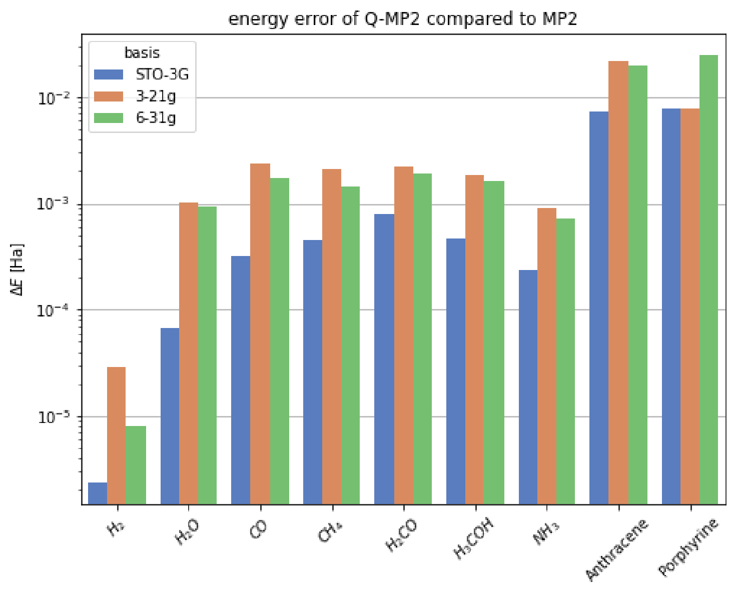


FIGURE 3.2: Absolute error for the correlation energy of Q-MP2 compared to the parent MP2 method on the 20/38 Becke weighted grid with $N_K = 4$.

as well as a set of molecules. The molecules were chosen in such a way to have comparisons between molecules of the same N_P (H_2, CO and NH_3, H_2CO) as well as molecules with similar orbital numbers (CO, CH_4, NH_3 and H_2CO, H_3COH). Two additional larger systems were also included.

The absolute errors and the relative absolute errors for these systems are given in table A.1 and illustrated in figures 3.2 and 3.3.

It is important to compare both the absolute error as well as the relative error of the methods. The error will be larger for systems with higher values for their correlation energy correction, as the absolute quadrature error also scales with the integral value.

Comparing all systems with each other some trends can be concluded. Both the absolute and the relative error seem to be dependent on the choice of basis set for the molecules, with STO-3G giving the lowest error. Going up to the Pople basis set the smaller 3-21g performs worse than the larger set.

The number of orbitals, which decide the matrix dimensions and therefore the number of operations to be carried out, appear to have some correlation with the errors. Comparing CO, NH_3 and CH_4, which have a similar number of orbitals, shows that while CO and CH_4 have similar absolute errors, CH_4 has a much larger absolute error. NH_3 shows both a lower absolute and a lower relative error.

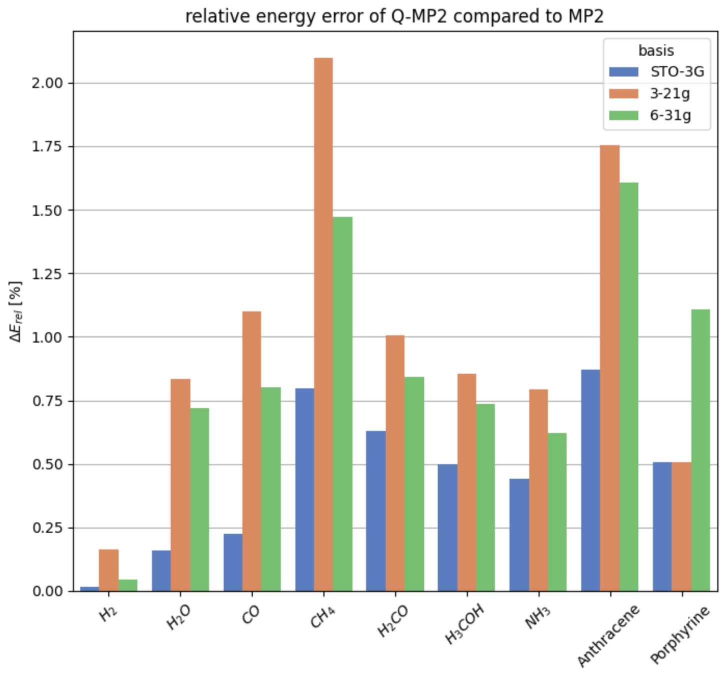


FIGURE 3.3: Relative absolute error for the correlation energy of Q-MP2 compared to the parent MP2 method on the 20/38 Becke weighted grid with $N_K = 4$.

It also has the lowest amount of virtual orbitals of the set, however, it's grid size is between that of the two others.

The size of the system seems to have a big impact on both the absolute and relative error. The minimal H_2 system has the best accuracy, whereas the medium-sized systems have comparable absolute errors. The two larger systems perform the worst in terms of absolute error. They also have the highest absolute value for their energy correction by about a magnitude, which explains the bad performance. The fluctuations of the absolute errors are much larger in the smallest STO-3G basis. However, the relative error does not seem to show much of a correlation. The worst performing system, CH_4, is of medium size and higher grid density because of the small $C - H$ bond lengths.

The deciding factor for the accuracy of the method is the size and make-up of the two quadrature grids. The quadrature grids and their effect on the accuracy will be discussed in a later chapter.

Chapter 4

Implementation

One of the biggest pitfalls many researchers fall into is neglecting the technical side of the research. "Is it science or engineering?" may be a jokingly exclamation, but the reality is that good, reliable, and reproducible science has to be an exchange between good engineering practices and 'pure' scientific endeavor. Once a Ph.D. leaves their position, their code gets buried in not easily accessible folders and soon forgotten. And even if found, the code quality often means that retracing the thoughts and ideas within is very tedious indeed.

Seeing as most of this work is focused on performance optimization modern coding and working principles were put at the center stage. That means spending time on code documentation, collaborative version control systems, a functioning test suit for unit and integration tests, and a continuous integration workflow.

The following chapter tries to sketch the approach and the inner workings of the most important facets of the code in as much clear detail as possible, so that anyone with this thesis may retrace its inception if desired. It starts with the design goals that were put forth for this project in section 4.1 and with a list of the used hardware and underlying supporting software in section 4.2. Finally, we dive into the code of the `libqqc`[93] library in section 4.3.

4.1 Design Goals

As with any coding project, much of the final result evolved over time. During this, the following criteria become important:

4.1.1 Portability

Especially in the early stages of development portable, small, and contained setups that could easily be moved and compiled on new cluster systems without much setup were found necessary. This was realized by designing the first iterations to be as independent of other sources as possible. The first iteration of the algorithm was written with `C`-style arrays and not with high-level algebraic libraries for this reason. In the same line of thinking it was avoided to marry the project to any specific host program and it was decided to export some smaller and larger test systems in human-readable 'column separated value' format. Together with a simple 'read-in' routine, this lead to independence from any larger body of code. This allowed a workflow of fast code iterations and very flexible testing conditions to research the intricacies of high-performance code execution.

4.1.2 Modularity

After the first code iterations were finalized and showing promise the underlying code structure was developed. Early on it was decided that committing the code to one specific quantum chemistry program was to be avoided and to keep it as extensible and flexible as possible. So a modular library structure was developed with an easy-to-use interface and easy-to-extend support for different host programs. The first iteration of the major parts of the code base was drafted with a storage holding facility and a modular setup where each method would be its self-contained, callable part. A utility module was added, housing the 'read-in' routine for the test systems and the setup for the necessary timings.

4.1.3 Sustainability

As mentioned before one large problem (academic) science is facing is the loss of research results and workflows after the researcher has left their position. The author has personally witnessed large and important works vanish into unattainability either through unmaintainable code or simply lost data. This problem is sometimes called coding 'sustainability'.

There are multiple ways to help with this. The first, and arguably most important one, is using a **version control systems** that is accessible to others. Such a version control system allows the user to keep track of any changes they have made to the program throughout a project and to be able to roll back to any previous versions (ideally marked with notes and comments of what changed). This means no version of the code gets lost, even in the case of fully overwriting or losing the data on-site. It was decided to use the `git` tool family and the project was hosted on a local `gitea` repository at the beginning. It was later moved to a public `github` page, where the code was released as an open-source project for anyone to use and modify.

The second most important step is clear code**documentation**. `Doxygen` was used as a documentation back-end, which is a well-known `C/C++` documentation software. Correct documentation is an art in itself, and much discussion is still being held in the community over how much, how little or simple how documentation should be done. Generally, the rules are to keep the variable, function, and class names as close to self-explanatory as possible and only provide written comments where necessary (always assuming the reader has basic knowledge of the language). So instead of writing code like shown in listing 4.1, the code in listing 4.2 is much easier to read.

LISTING 4.1: This code is not very readable.

```
float magic_number = 3.14;
b = function_a (magic_number, 3.0);
graphic(b, 2.4);
```

LISTING 4.2: This code is much more readable.

```
float PI = 3.14;
float radius = 3.0; //in m
float height = 2.4; //in m
circle_area = calculate_circle_area(PI, radius);
//using the circle area constructs a cylinder and outputs an ASCII ↩
    graphic to std::cout
plot_3D_cylinder(circle_area);
```

Documentation is always an ongoing task. However, if you are using a documentation back-end such as `Doxygen`, which can output a handy `html` or `pdf` summary of the documentation, the need for manually recompiling the documentation gets tedious.

The third pillar of code sustainability is an often overlooked one: continues **testing**. Writing good tests is very important and very difficult. Especially larger code basis tend to be so interconnected that no one can know if even benign changes to it will break some functionality. There are many different types of tests, but the three major categories are:

- **unit tests:** test a 'unit' of code, by which is meant the smallest independently callable unit possible, such as a single function call.
- **integration tests:** testing a small number of code 'units' with each other to see if the interplay of them leads to any errors
- **system tests:** testing the behavior of the whole system to see if any problems arise from the interplay of all 'units'

There are more categories (such as different kinds of performance tests), but these three are needed to assure the correct behavior of the program throughout its development. The goal is to have as much code being tested as possible, which often means that writing good tests takes the majority of development time for sustainable software. This is often the reason, especially in such a result-driven environment as research, that this important step gets neglected.

Writing the code, documenting it, recompiling the documentation, writing the tests, and executing them after every change to the code basis takes time, even without any major problems arising. To help with this it is generally favorable to setup what is known as a 'continuous integration/development pipeline. Such a pipeline is used to do any housekeeping that can be automated, e.g. recompiling the documentation and the tests and executing them, after an update to the code base is made. Many `git` based repository providers have easy-to-use tools to set them up. For `github` these are called 'actions'. Such a pipeline was implemented to automate the testing and documentation throughout the project, which will be discussed later in this chapter.

	XPS 13 9360	**IWR CompChem Cluster**
CPU	Intel i5-7200U 4 (physical) at 2.5 GHz min./ 3.10 GHz max.	2× Intel Xeon Gold 5120 14 (physical) at 2.20 GHz min./ 3.20 GHz max.
RAM	8 GB at 1867 MHz	512 GB at 2400 MHz

Table 4.1: Hardware of early stages of development, mainly during single-node performance optimization.

	bwHPC JUSTUS 2[94]
CPU	2× Intel Xeon Gold Prozessor 6252 24 (physical) at 2.10 GHz min./ 3.70 GHz max.
RAM	384 GB
Network	Omni-Path 100 GBit/s

Table 4.2: Hardware of later stages of development, used during multi-node (`MPI`) performance optimization.

	XPS 13	**IWR CompChem**	**JUSTUS 2**
Compiler	GCC 9.4.0	GCC 7.4.0	GCC 10.1.0
OpenMP	4.5 (5.0 partially)	4.5	4.5 (5.0 partially)
CMAKE	3.16.3	3.10.2	3.10.2
MPI	OpenMPI 4.0.3	-	OpenMPI 4.0.5
Eigen	3.4.0	3.4.0	3.4.9

Table 4.3: Summary of supporting software on the different architectures.

4.2 Hardware Architecture and Supporting Software

What follows is a list of the hardware architecture and underlying supporting software used during the different stages of development.

The early stages of development, mainly the single node optimization were done on the hardware listed in table 4.1. Later development, especially the multi-node `MPI`-enabled iterations, were done on the hardware listed in table 4.2.

All results reported in this thesis were run on the hardware listed in 4.2 if not noted otherwise.

The library was compiled using the `GNU` family of compilers[95], using a `CMAKE`[96,97] setup and `OpenMPI`[98–101] to provide the `MPI` functionality through the `mpicc` compiler wrapper. All software versions used can be found in table 4.3, which also lists the supported `OpenMP`[102–104] version and the version of the `Eigen`[105] library used for its implementation variant.

The compilation procedure follows the `configure` script of the repository, meaning that the following flags were set if compiled as a `release` version: `-O3 -ffast-↩ math -march=native`.

Lastly, for the results published in this thesis an interface with the commercial `Q-Chem`[88] software suite was written, which uses some common utilities of the software, especially of the `libdftn`, `libgrid` and `libqints` modules.

4.3 The `libqqc` library

The code used in this thesis was published as an open-source `C/C++` library at `https://github.com/BenTho-Uni/libqqc` under the *GPL-3.0 license*. The main contributor as of the writing of this thesis is the author, Benjamin Thomitzni, with grid implementations by Isabel Vinterbladh. The documentation of the `main` branch can be found at `bentho-uni.github.io/libqqc/`. Parts of the code use the open-source `Eigen`, `OpenMP`, and `OpenMPI` libraries in the above-mentioned versions. Additionally, parts of the `loader` module use code by the commercial `Q-Chem`[88] software suite.

4.3.1 General Structure

The repository of the library consists of the following parts:

4.3.1.1 Repository Structure

`.github/workflows/`

Continuous integration pipeline for automated testing and deployment of documentation after every push to the `main` branch through the use of `Doxygen`. The documentation is deployed to a separate branch, which is then hosted as the above-mentioned `GitHub` page. The continuous integration action executes the `cmake` setup, compiles the code, and then executes `ctest` (see listing B.1 for an example).

`data`/

Four differently sized test systems have been provided to allow stand-alone execution of the library for testing purposes: Water, methanol, anthracene, and porphyrin data exported from `Q-Chem`[88] using the $6-31G$ basis set with the $20,38$ Becke grid of the `libdftn` module. Each folder provides the grid-evaluated atomic orbital integrals and Fock matrix, the evaluated Coulomb integral, as well as the grid data on which the integrals were evaluated. The water case is directly included in the repository. This is not possible for the other systems because of size constraints, but they can be downloaded through the `download_data.sh` script.

`docs`/

Folder for the `Doxygen` documentation.

`host_example`/

Minimal host program examples to showcase the library usage and allow stand-alone testing. One variant for the single-node case, one with the necessary `MPI` setup. The necessary steps for this will be discussed in section 4.3.1.3

`libqqc`/

Folder for the `libqqc` source code. Discussed in detail below.

`runs`/

Outputs of example runs of the provided test systems on the JUSTUS 2 cluster. Each test system runs with the array and with the `Eigen` variant provided and with either the `OpenMP+MPI` and the `MPI only` schemes. Additionally, the submission generator (`make_run_XxY.sh`), and submitter (`multi_submit.sh`), timings-to-csv converter (`process`) and scatter plot scripts (`make_scatter.py`) are provided. One such example run is shown in listing B.2.

`templates`/

This folder consists of some template body and header files which are easily copied and modified to be included in the library.

`tests`/

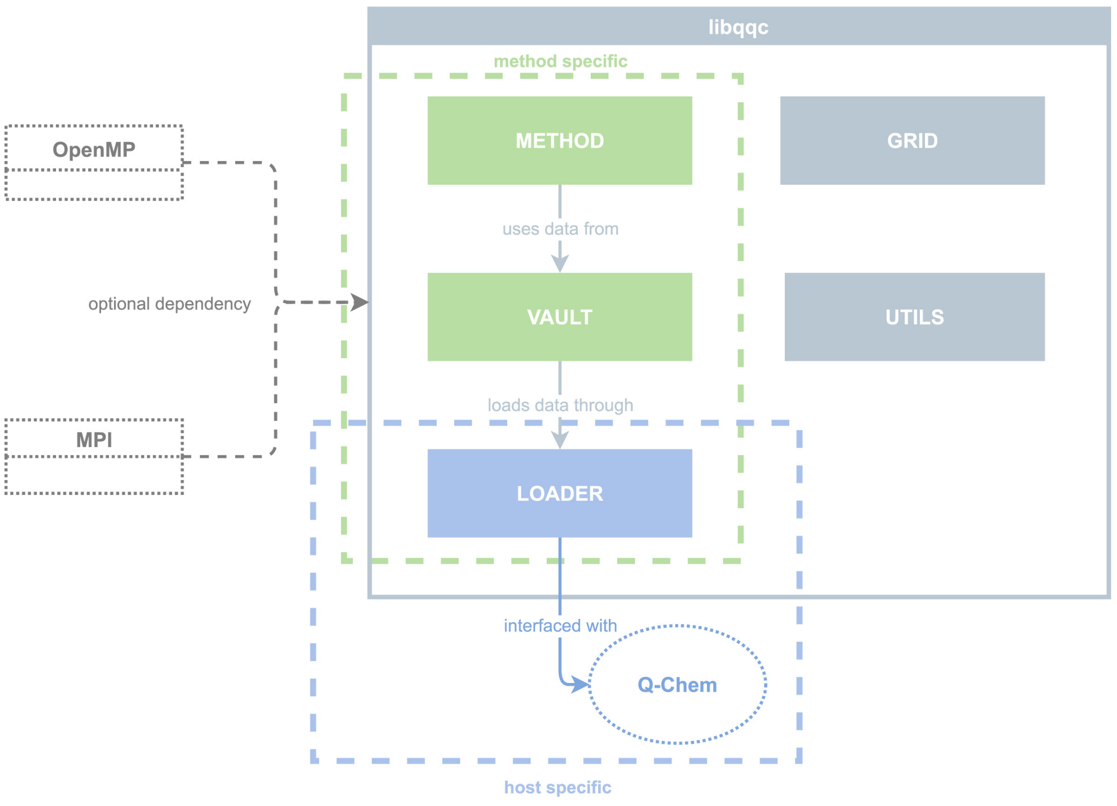


FIGURE 4.1: General modular structure of the `libqqc` library as of date.

Contains the testing suite with the implemented unit and integration tests. Discussed in more detail below.

`configure`

Main cmake configure script to set the correct compiler, flags, and compilation variant options.

4.3.1.2 `libqqc/` Structure

The internal structure of the `libqqc` library is presented in figure 4.1.

The library consists of five internal modules. Two modules are as of now generalized and method independent, namely the `grid` and `utils` modules. The other three modules, `methods`, `vault`, and `loader` have to be provided for their specific use case. However, much can be abstracted from the base classes. Currently, the only implementation published in the library is the Q-MP2 scheme.

The `method` module consists of the routines necessary for the property calculations. It consists of any transformations and pre-calculation before the final function call, as well as the `MPI` setup if necessary. For a deeper explanation see section 4.3.6.

In the `vault` module the classes of the same name function as the data holding facility for a calculation. See section 4.3.5 for further details.

Interfacing and input capabilities are handled by the `loader` module. This is the only part of the code where host-program-specific changes have to be made to facilitate the necessary data for the calculation. Additionally, some integral transformations and operations are carried out in this module. This is done to save as much space as possible and to reduce network communication. For more explanations see section 4.3.4.

The `grid` module consists of the library's grid-generating methods, as well as the base classes for interaction with the properties of the employed grid. This is generalized in the current iteration of the code. For an outlook on future changes to the library see section 7.1.1.

Finally, the last module, `utils`, consists of miscellaneous utility functions, such as the timing and printing facilities and the file interfacing procedures. See 4.3.2 for more detail.

The library has three optional dependencies (`OpenMP, OpenMPI, Eigen`), depending on the requested compilation setup.

4.3.1.3 Library Usage Example

An example of how to use this library is provided in the repository under the `host_example` folder. These examples can be adapted to any host program, e.g. `Q-Chem`.

The only necessary code includes in the host program consists of the method that is used and the timer facility, if desired (listing 4.3).

Listing 4.3: Including the necessary header files for the calculation.

```
#include "../libqqc/methods/do_qmp2.h"
#include "../libqqc/utils/ttimer.h"
```

The usage of the library consists of three parts. First, one has to specify which interface `loader` class should be used (listing 4.4). This may take additional arguments depending on the interface. Then, the data `vault` is generated and the `loader` as an argument is passed to its constructor (listing 4.5), which handles the input of the data. Finally, the `method` class for the desired calculation is generated, which executes the final calculation (listing 4.6.

Listing 4.4: Generating the method specific `loader` interface class.

```
        libqqc::Loader_qmp2_from_file loader("../data/h2o/");
```

Listing 4.5: Generating the method specific `vault` storage class.

```
        libqqc::Vault_qmp2 vault(loader);
```

Listing 4.6: Generating the desired `method` class and executing the calculation.

```
libqqc::Do_qmp2 qmp2(vault);
qmp2.run(out);
```

The `run` method of the `Do_qmp2` class prints the result directly to the output.

A full overview of the classes and their relationships is shown in figure 4.2. Central to the calculation is the method class `Do_qmp2`, which is dependent on the data holding class `Vault_qmp2`. This is in turn dependent both on the interface `Loader_qmp2` and the manager class for the grid data, `Grid`. A closer look into the structure of these classes follows.

The next sections will dive deeper into the code and explain the internal workflow of this setup.

4.3.2 The `utils` Module

The `utils` module houses any utility functions and classes used in the library. As of now, it covers three areas:

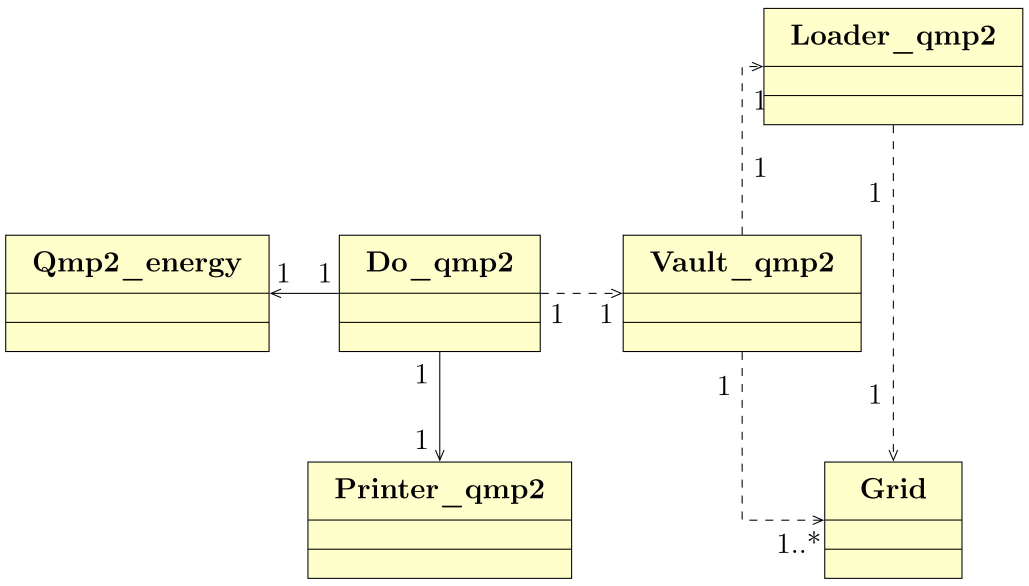


FIGURE 4.2: Full class relationship diagram of all relevant classes of the `libqqc` library for the calculation.

4.3.2.1 `printers`

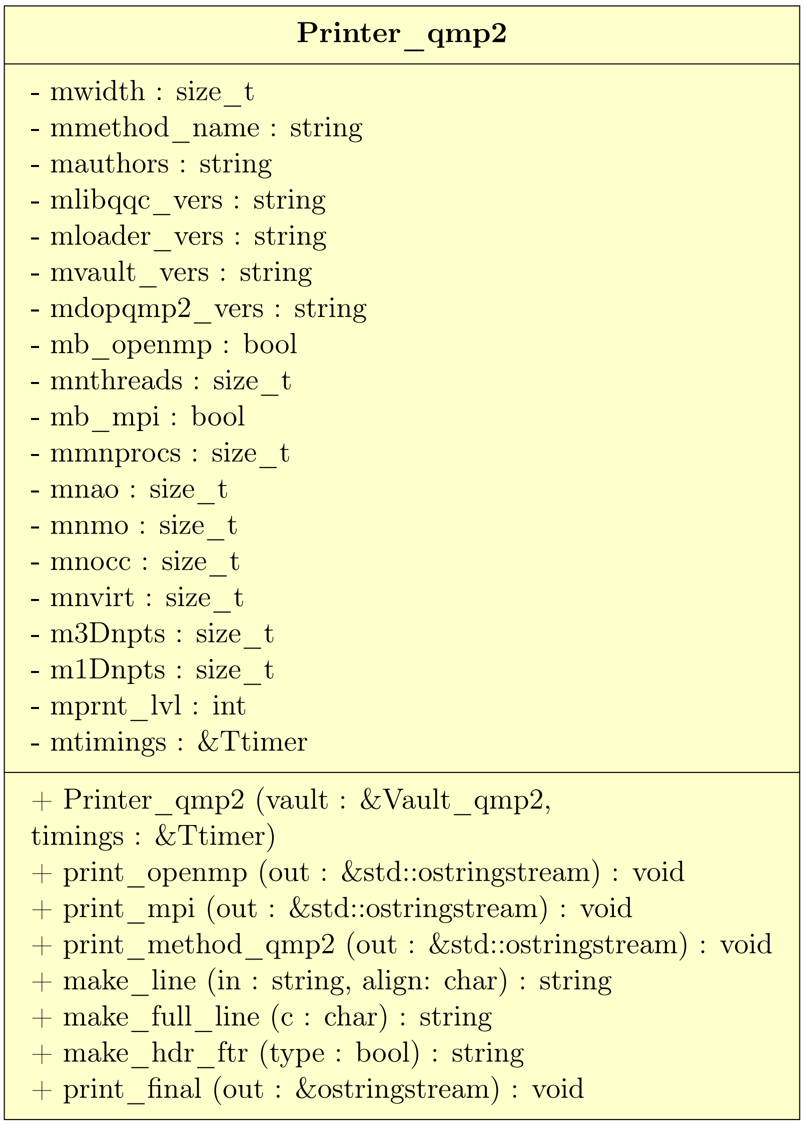
This subsection covers the printing facilities for the different methods. The class diagram is given in figure 4.3.

This class takes the `vault` object as an argument to its constructor which it uses to extract the desired information, such as the number of atomic orbitals for example. The main function `print_final` takes an output stringsstream and clears it. A flowchart diagram of its functionality can be seen in graph 4.4. After it adds general information, such as the authors or the library versions, it calls the method-specific printer which adds the method information. Lastly, it checks the desired printing level and calls the internal printer methods of the `Ttimer` class objects.

Most of the printer functionality, except for the method-specific printer function, will be abstracted in a future release from a base class, as it doesn't change between methods.

4.3.2.2 `load_from_file`

The `load_from_file` facilitates the interaction with a columns-separated-data format file, such as the ones provided in the example systems in this repository. It



Printer_qmp2

- mwidth : size_t
- mmethod_name : string
- mauthors : string
- mlibqqc_vers : string
- mloader_vers : string
- mvault_vers : string
- mdopqmp2_vers : string
- mb_openmp : bool
- mnthreads : size_t
- mb_mpi : bool
- mmnprocs : size_t
- mnao : size_t
- mnmo : size_t
- mnocc : size_t
- mnvirt : size_t
- m3Dnpts : size_t
- m1Dnpts : size_t
- mprnt_lvl : int
- mtimings : &Ttimer

+ Printer_qmp2 (vault : &Vault_qmp2, timings : &Ttimer)
+ print_openmp (out : &std::ostringstream) : void
+ print_mpi (out : &std::ostringstream) : void
+ print_method_qmp2 (out : &std::ostringstream) : void
+ make_line (in : string, align: char) : string
+ make_full_line (c : char) : string
+ make_hdr_ftr (type : bool) : string
+ print_final (out : &ostringstream) : void

FIGURE 4.3: Class diagram for the `Printer_qmp2` class of the `utils` module.

has two function calls, one which loads the data dimensions from a header line in the file (`load_dim_from_file` and one which then reads the file line by line and writes each cell to an array address (`load_array_from_file`).

A flowchart summary of the second function can be found in figure 4.5. The read-in is done by utilizing the file and line stream functionality of `iostream`. The file is opened as a file stream and an exception is thrown if the path does not exist. Any header rows specified are skipped and then the next line is loaded as a line stream object. The function then iterates through the cells by use of the `getline` method and the specified delimiter. If a cell is not empty or consists of white space, a cast to double is attempted. Any exceptions of this cast are caught and the value of the cell is then assumed to be zero. In future iterations, more care

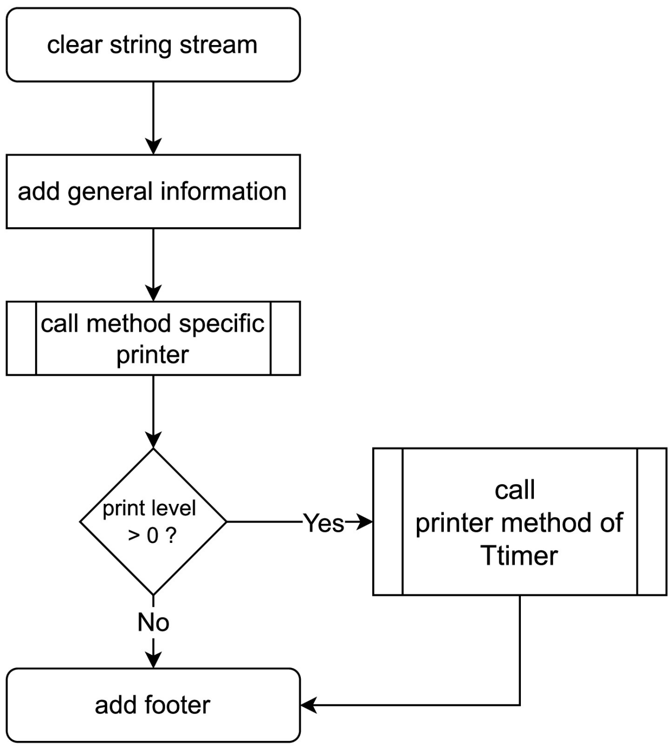


FIGURE 4.4: Flowchart representation of the `print_final` method of the `Printer_qmp2` class.

should be taken with data sanitation at this step. Finally, successful execution is marked by returning a true value.

This utility function is sufficient for the provided data, but, as noted, may have to be expended in terms of data integration in the future. However, most use cases outside of initial testing will involve direct interfacing with a host program and will therefore not involve manual file read-in.

Additionally, this manual file read-in is also a huge bottleneck in the parallel capabilities in testing scenarios. However, as it is not used in real scenarios, the read-in timings can simply be ignored for performance testing.

Parallelising file input is a major problem of the discipline. It has not been attempted in this thesis as of date. Any parallel scheme would involve distributed cluster systems, as not much computational work is done and the major bottleneck is the input/output latency of the local storage. A distributed system could cut the input time by utilizing multiple disk bandwidths. It would however involve additional synchronization and data movement, as the file would have to be distributed over the cluster resources, and the read-in data would then have to be

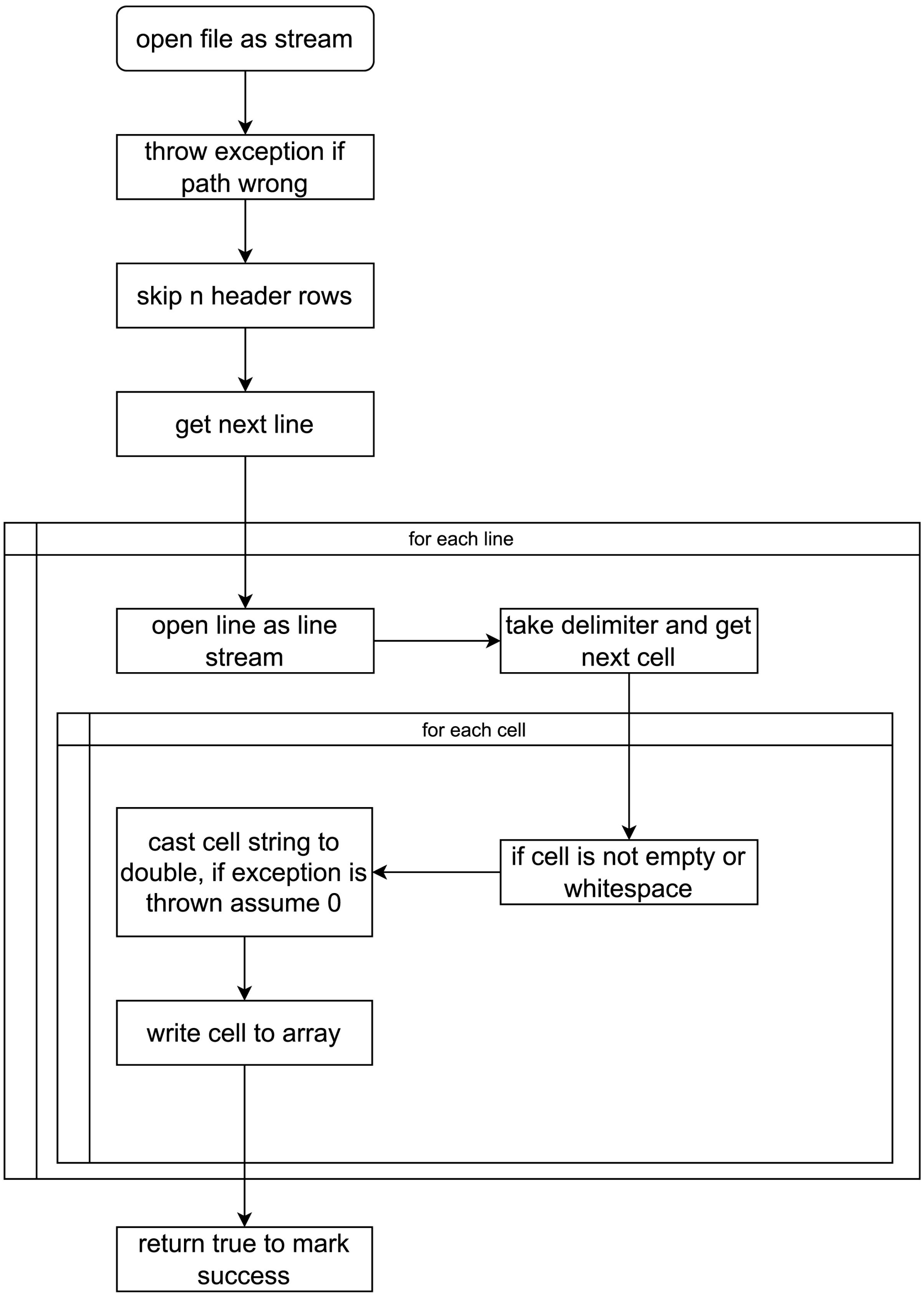

open file as stream
throw exception if path wrong
skip n header rows
get next line
for each line
open line as line stream
take delimiter and get next cell
for each cell
cast cell string to double, if exception is thrown assume 0
if cell is not empty or whitespace
write cell to array
return true to mark success

FIGURE 4.5: Flowchart representation of the `load_from_file` utility function.

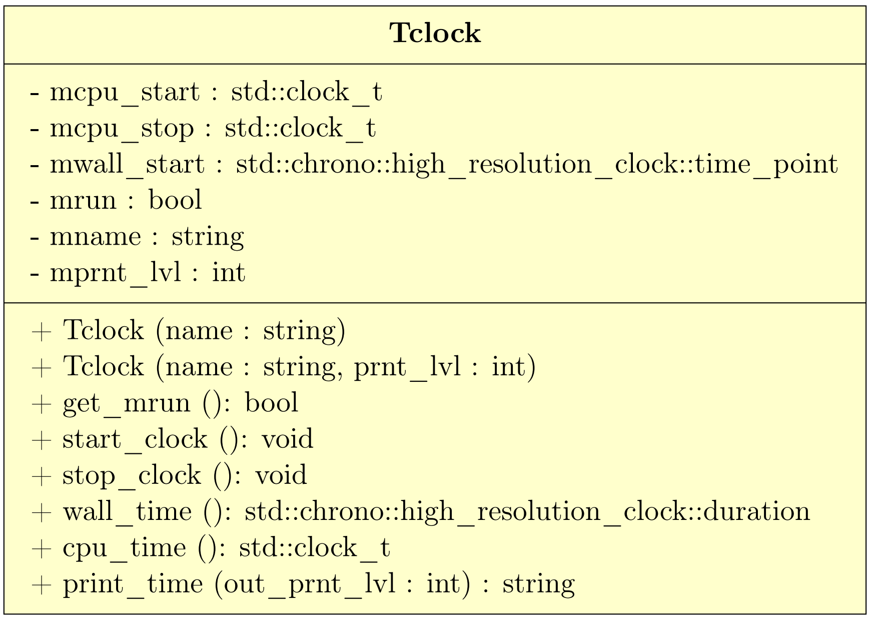

Tclock
- mcpu_start : std::clock_t
- mcpu_stop : std::clock_t
- mwall_start : std::chrono::high_resolution_clock::time_point
- mrun : bool
- mname : string
- mprnt_lvl : int
+ Tclock (name : string)
+ Tclock (name : string, prnt_lvl : int)
+ get_mrun (): bool
+ start_clock (): void
+ stop_clock (): void
+ wall_time (): std::chrono::high_resolution_clock::duration
+ cpu_time (): std::clock_t
+ print_time (out_prnt_lvl : int) : string

FIGURE 4.6: Class diagram of the `Tclock` single timer class of the `utils` module.

exchanged. Finally, the total data object would then have to be constructed from the different parts of the distributed algorithm.

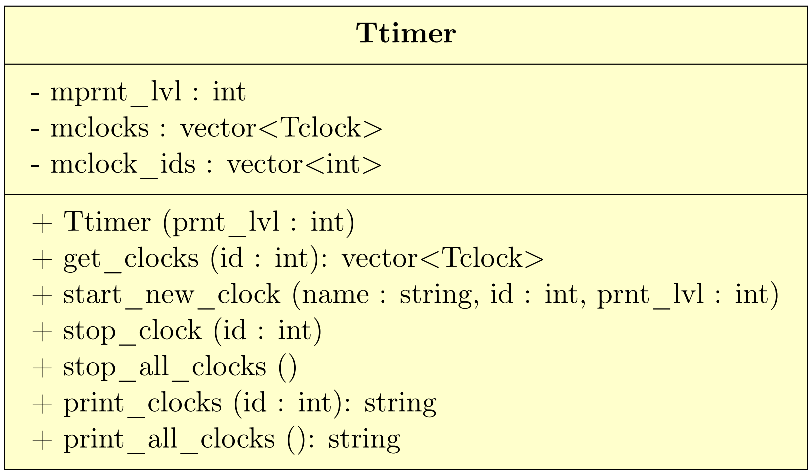
In the current method iteration, the Q-MP2 algorithm iterates over all available data on each cluster node anyway, which is why a complex distributed file input scheme was not deemed necessary for the niche case of manual file input.

4.3.2.3 `ttimer`

The timing functionality of the library is provided by two classes, `Tclock` and `Ttimer`.

The class diagram of `Tclock` can be found in figure 4.6. Both CPU and walltime are being tracked through external classes from standard `C++` libraries. As the wall time carries the information used for the performance testing it uses a high-resolution clock with a resolution of the smallest tick period provided by the implementation of the library.

Timing objects can be single clocks or collection classes called `Ttimer`, with the class diagram given in figure 4.7. The collection class provides a way to start, stop and queue multiple clock objects and print their content in a single call. Every timing in this thesis was done through these timer classes and one collection object was generally used per function call, using multiple clocks.



Ttimer

- mprnt_lvl : int
- mclocks : vector<Tclock>
- mclock_ids : vector<int>

+ Ttimer (prnt_lvl : int)
+ get_clocks (id : int): vector<Tclock>
+ start_new_clock (name : string, id : int, prnt_lvl : int)
+ stop_clock (id : int)
+ stop_all_clocks ()
+ print_clocks (id : int): string
+ print_all_clocks (): string

FIGURE 4.7: Class diagram of the `Ttimer` class of the `utils` module.

4.3.3 The `grid` Module

The `grid` module consists of classes facilitating the storage, generation, and traversion of the integration grids.

The base class is summarised in the class diagram found in figure 4.8. It consists mostly of storage methods and members handling the underlying arrays which store the (multi-) dimensional grid coordinates and their weights.

Further integration work was done by Isabel Vinterbladh during a master thesis project[5] in collaboration with the author. The focus of the thesis was the implementation of common Becke integration grids, their underlying foundations, and some further grid abstractions to test different spatial grid representations.

A summary of her implementations follows but can be seen in full detail in the cited thesis. The theoretical foundations of the Voronoi cell generation and the Becke grid are discussed in section 5.1.

Please also be aware that these implementations have not yet been implemented in the `main` branch of the code repository.

4.3.3.1 One and Multi-Dimensional Grid Generation

A one-dimensional grid generator class, `Grid1D`, is used using the abovementioned `Grid` class as its basis. It differs from the base class in three public methods

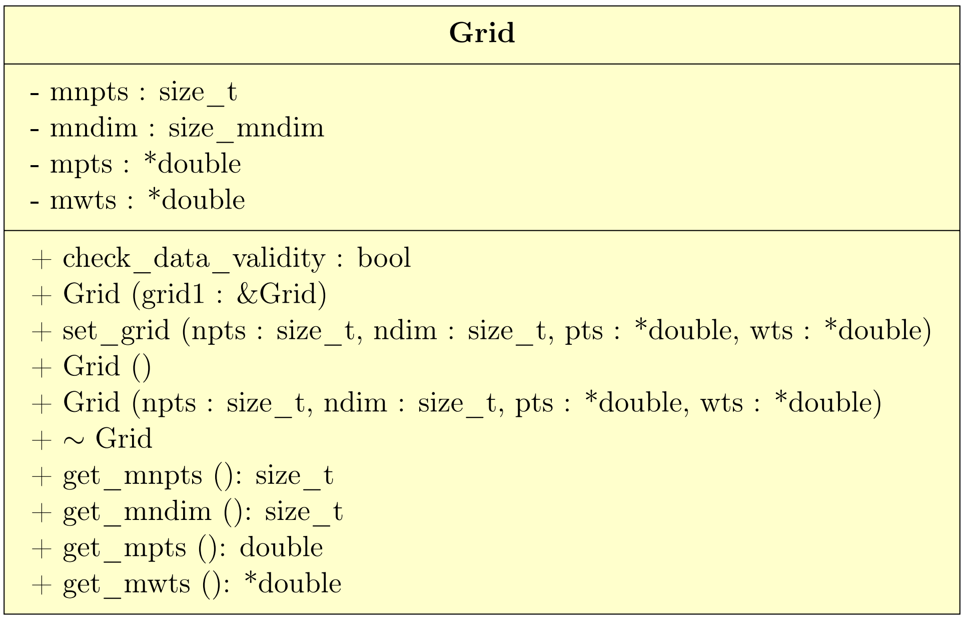


FIGURE 4.8: Class diagram of the base `Grid` class of the `grids` module which handles the storage and traversion of the integration grid.

which are used to generate and to set a uniform and a Gauss-Chebyshev style one-dimensional grid.

Using the one-dimensional case as a template a two and three-dimensional grid class is provided. The latter handles the generation of different spatial grids used in the thesis and is discussed in chapter 5. This includes the Becke grid[106] and abstractions using different spherical arrangements (such as ellipsoidal around the chemical bond) for the grid point coordinates.

4.3.3.2 Voronoi Cell Generation and Supporting Functions[5]

The Delaunay triangulation of the Bowyer-Watson[107–110] algorithm was implemented in the `Voronoi` class, which handles the generation of the Voronoi cells in the procedure published by Becke[106]. Common vector operations and supporting functions, such as cross products, distance, and determinant calculations, were bundled in the `VectorFuncs` class of the same module.

4.3.4 The `loader` Module

The `loader` module covers the interface to the host program and with that the input routines to the `vault` module. For each host program and each method, a

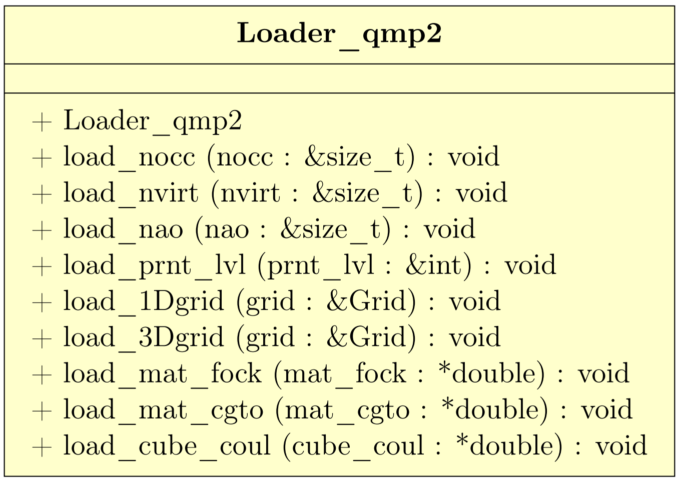

Loader_qmp2
+ Loader_qmp2
+ load_nocc (nocc : &size_t) : void
+ load_nvirt (nvirt : &size_t) : void
+ load_nao (nao : &size_t) : void
+ load_prnt_lvl (prnt_lvl : &int) : void
+ load_1Dgrid (grid : &Grid) : void
+ load_3Dgrid (grid : &Grid) : void
+ load_mat_fock (mat_fock : *double) : void
+ load_mat_cgto (mat_cgto : *double) : void
+ load_cube_coul (cube_coul : *double) : void

FIGURE 4.9: Class diagram of the base `Loader` class of the `loader` module which handles the interfacing with the host program and the data input

loader instance has to be implemented. The repository provides a dummy class example, an example for the manual file-from-disk read-in, and a `Q-Chem` specific implementation.

An example class diagram is given in figure 4.9 for the base implementation of the Q-MP2 algorithm. The only implementation differing from this structure is the loader for the file read-in, which adds private file name attributes to the class and its constructor. As can be seen, the Loader class is simply used to host the different methods which pass references and points from the host program to the caller. However, this class is not purely for data input but also handles the basic integral transformations so that the `vault` module only has to store the user data in its molecular orbital representation.

An example flow chart of one such load function is given in figure 4.10. First, the meta data is gathered, such as the number of atomic, occupied, and virtual orbitals as well as the number of grid points. Then the coefficient matrix C and the integral (in this case the coulomb integral I) are loaded. Finally, the transformation is carried out.

A closer look at the transformation is shown in algorithm 2. The transformation that can be seen here is in its most straightforward form, scaling with $\mathcal{O}(PN^4)$ in regards to system size with a prefactor of P, representing the number of grid points of the integration grid.

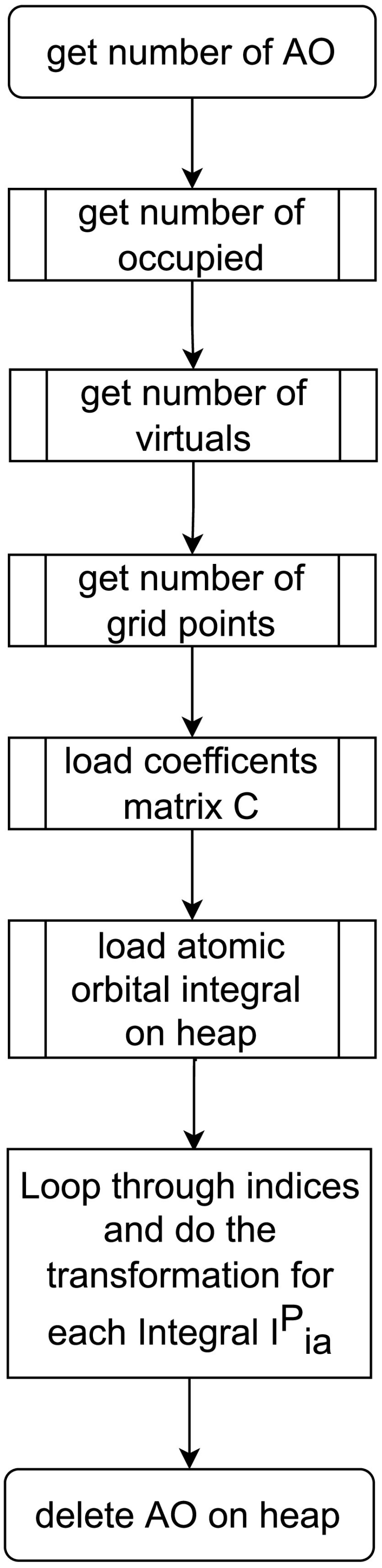

get number of AO
get number of occupied
get number of virtuals
get number of grid points
load coefficents matrix C
load atomic orbital integral on heap
Loop through indices and do the transformation for each Integral I^P_{ia}
delete AO on heap

FIGURE 4.10: Flowchart sketch of `Loader_qmp2` method loading in the atomic orbital representation of the coulomb integral and its transformation.

Algorithm 2: Straight forward transformation of the atomic orbital integral to its molecular orbital representation used in the `Loader_qmp2` class.

```
1 for each point $P$ do
2     for each occupied orbital $i$ do
3         for each virtual orbital $a$ do
4             $I^P_{ia} = 0$ for each atomic orbital $\mu$ do
5                 $t = 0$                          // temporary intermediate
6                 for each atomic orbital $\nu$ do
7                     $t \mathrel{+}= I^P_{\mu\nu} * C_{\nu a}$
8                 $I^P_{ia} \mathrel{+}= C_{i\mu} * t$
```

There is room for improvement of this algorithm and lowering the scaling further. The scaling can be lowered to $\mathcal{O}(PN^3)$ by the introduction of a fully sized intermediate and restructuring of the loops (see algorithm 3. This is achieved by trading memory complexity (from $\mathcal{O}(PN^2+POV)$ to $\mathcal{O}(PN^2+POV+NV)$) for computational complexity.

Algorithm 3: Optimised transformation algorithm with computational scaling of $\mathcal{O}(N^3)$.

```
1 for each point $P$ do
2     for each occupied orbital $i$ do
3         for each virtual orbital $a$ do
4             $I^P_{ia} = 0$
5             $I_{\mu a} = 0$                      // temporary intermediate
6             for each atomic orbital $\nu$ do
7                 $I_{\mu a} \mathrel{+}= I^P_{\mu\nu} * C_{\nu a}$
8             for each atomic orbital $\mu$ do
9                 $I^P_{ia} \mathrel{+}= C_{i\mu} * I_{\mu a}$
```

The highest computationally scaling step of the algorithm is therefore the transformation with $\mathcal{O}(PN^3)$ with this implementation. The transformation step is however not the bottleneck of the calculation and so not much work has been put into lowering the scaling. Work has been invested in increasing its parallel efficiency, as the grid points are independent of each other. All transformation schemes have been fully parallelized with shared and distributed memory techniques and minimal network communications. For a deep dive into the parallel capabilities, see chapter 6.

The only difference in interfacing to other host programs lies within the loading calls shown in figure 4.10. For the from-disk read this consists of simply calling the utility function to load from a local file, mentioned above, as can be seen in listing 4.7 for the coulomb integral.

LISTING 4.7: Example call to load in the AO integral data from file

```
//initializing memory on heap due to size
double* coul_ao = new double[npts * nao * nao];
load_array_from_file(msrc_folder+mfname_coul,
    dim_ao, coul_ao, ' ', 1);
```

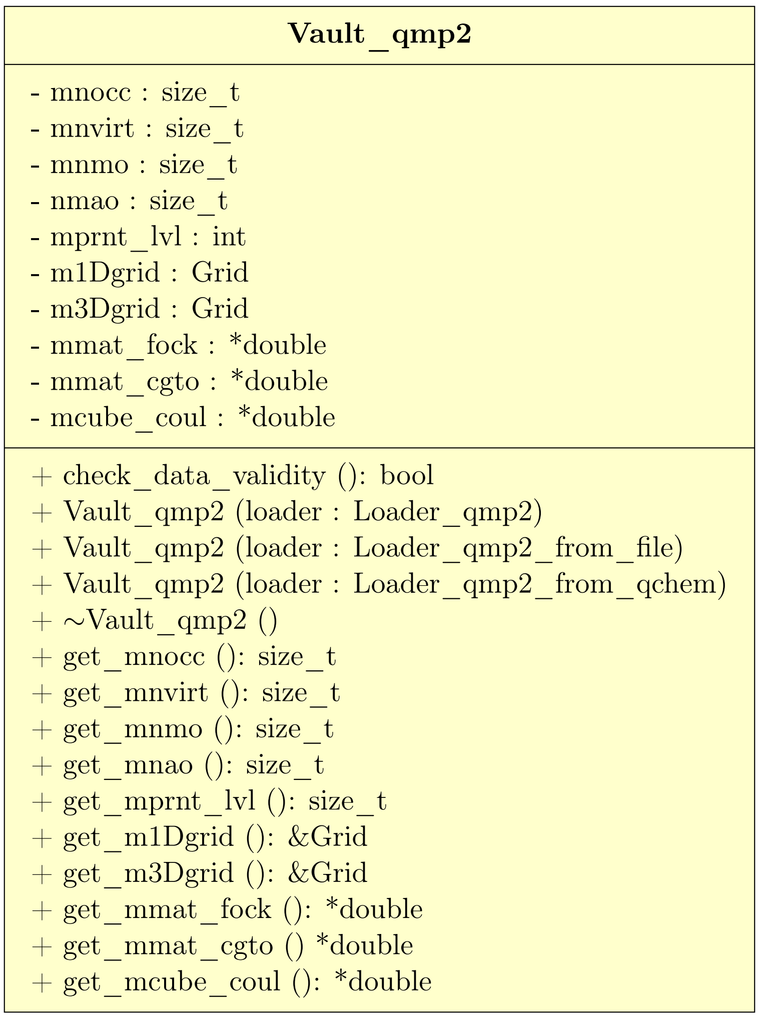
These are the parts that have to be changed to interface host programs. An example is given in listing B.3 for the `Q-Chem` interface of the same integral.

At the start, the memory is allocated on the heap to avoid memory constraints. This memory is later freed again. Then the one-electron-one-center contracted Gaussian-type atomic orbital basis is generated. Then a vector is created which consists of multipole coordinates, which are the coordinates of the integration grid points and a moment of one. this is passed to generate the one-electron-one-center multipole matrix. Array view objects for the multipole moments and the integral data memory is then passed into an integral 'digester' which writes the atomic orbital integral data to the memory locations. The digester consists of `Q-Chem` internal code adapted from the `libfock` and `libqints` modules. This code is not published in the public repository due to a signed NDA.

4.3.5 The `vault` Module

The classes from the `vault` module take the `Loader_` class instance and call it in their constructor to load in the data they store. A class diagram is shown in figure 4.11.

As each method has its own set of data not much abstraction can be done to simplify the class generation. The `Vault_` class consists of the necessary data fields, which are references and pointers for larger sets of data, and the getter methods necessary for access. All data input is done through the constructor. As only molecular orbital data is initially saved and no changes to the data are necessary, no setter functions are needed.



Vault_qmp2

- mnocc : size_t
- mnvirt : size_t
- mnmo : size_t
- nmao : size_t
- mprnt_lvl : int
- m1Dgrid : Grid
- m3Dgrid : Grid
- mmat_fock : *double
- mmat_cgto : *double
- mcube_coul : *double

+ check_data_validity (): bool
+ Vault_qmp2 (loader : Loader_qmp2)
+ Vault_qmp2 (loader : Loader_qmp2_from_file)
+ Vault_qmp2 (loader : Loader_qmp2_from_qchem)
+ ~Vault_qmp2 ()
+ get_mnocc (): size_t
+ get_mnvirt (): size_t
+ get_mnmo (): size_t
+ get_mnao (): size_t
+ get_mprnt_lvl (): size_t
+ get_m1Dgrid (): &Grid
+ get_m3Dgrid (): &Grid
+ get_mmat_fock (): *double
+ get_mmat_cgto () *double
+ get_mcube_coul (): *double

FIGURE 4.11: Class diagram of the base `Vault_qmp2` class of the `vault` module which handles the storage of the data.

The constructor takes a class instance of the different `Loader_` classes as an argument, which it uses to call the routines responsible for extracting and initially transforming data. All of them follow the scheme shown in listing 4.8, with the only difference being which loader class instance is called upon. This will be moved to a template structure in future releases. All heap-declared memory is freed when the destructor is called. After the data is loaded some checks are performed on the data. This covers mainly the metadata (e.g. the number of orbitals shouldn't be zero), the grid data validity, and the memory locations of the pointers (which shouldn't be `NULL`).

LISTING 4.8: Constructor of the `Vault_qmp2` class using the standard `Loader_qmp2` interface.

```
loader.load_nocc (mnocc);
loader.load_nvirt (mnvirt);
mnmo = mnocc + mnvirt;
loader.load_nao (mnao);
// loading input information
loader.load_prnt_lvl (mprnt_lvl);
// load grid object
loader.load_1Dgrid (m1Dgrid);
loader.load_3Dgrid (m3Dgrid);
// loading in matrices
size_t nao2 = mnao * mnao;
size_t npts = m3Dgrid.get_mnpts();
mmat_fock = new double[mnmo * mnmo];
loader.load_mat_fock (mmat_fock);
mmat_cgto = new double[npts * mnmo];
loader.load_mat_cgto (mmat_cgto);
mcube_coul = new double[npts * mnocc * mnvirt];
loader.load_cube_coul (mcube_coul);
check_data_validity();
```

4.3.6 The `methods` Module

The `methods` module consists of the different quadrature methods and their variants depending on the requested parallelization scheme. Right now only the Q-MP2 algorithm is publically available.

There are two distinct parts for each method: a setup class and a class for each property to be calculated. These follow the naming scheme `Do_method` and `Method_property` respectively. For the Q-MP2 scheme, these are the `Do_qmp2` and the `Qmp2_energy` classes.

The optimization and parallelization work done on these classes will be discussed in chapters 6. Their general implementation is discussed in detail below. An initial version was done in collaboration with Liam Keegan of the Scientific Software Centre of the Interdisciplinary Centre for Scientific Computing at Heidelberg University[111] and then developed further by the author.

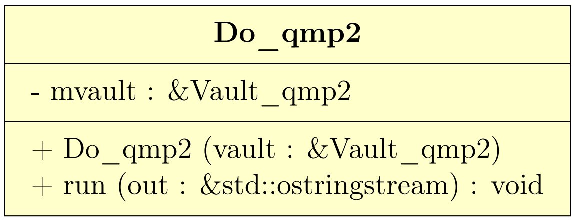

Do_qmp2
- mvault : &Vault_qmp2
+ Do_qmp2 (vault : &Vault_qmp2)
+ run (out : &std::ostringstream) : void

FIGURE 4.12: Class diagram of the base `Do_qmp2` class of the `methods` module which handles the setup and the necessary calculation before calling the requested property calculation methods.

4.3.6.1 `Do_qmp2` in Detail

The class diagram of the setup class for the Q-MP2 method is shown in figure 4.12. It consists of a private reference to the data storing object of the `Vault_qmp2` class. The constructor sets the reference to the vault. All of the setups for the calculation are done in the `run` method, which only takes a string stream to print the results.

Figure 4.13 shows the setup steps for the calculation. First, all references and pointers from the vault are acquired and the orbital data is split into an occupied and a virtual block (solely for ease of handling).

Next, the exponential factors stemming from the one-dimensional integration (e.g. t^{ε_a}) are precalculated (see listing 4.9). It will be discussed in further detail in chapter 6, but for many instruction sets an exponential function is very costly. By precalculating and storing these factors one simplifies the following algorithm to a set of floating point operations which has a higher chance of vectorization through the compiler. This comes at a cost of storing integrals with a maximum size scaling of $\mathcal{O}(KOV)$. However, as mentioned before, K is generally below ten, and therefore this data is much smaller than the initial $\mathcal{O}(PN^2)$ integral data object. It, therefore, does not contribute significantly to the overall memory scaling of the method.

Then the property class instance is initialized with the necessary data passed to it and the property is calculated through the `compute` method. See the next section for further details on the algorithm.

Finally, the Printer instances are initialized and their main method is called to finish up the calculation. At the end of the method, any leftover heap memory is freed.

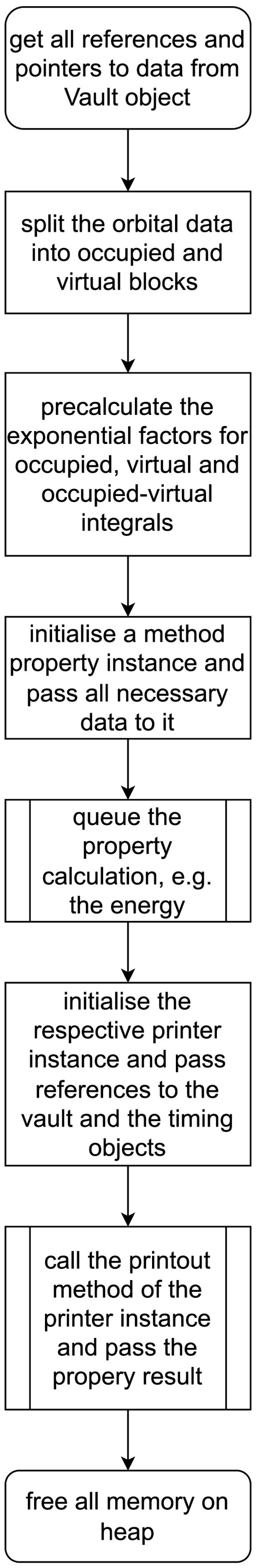

get all references and pointers to data from Vault object
split the orbital data into occupied and virtual blocks
precalculate the exponential factors for occupied, virtual and occupied-virtual integrals
initialise a method property instance and pass all necessary data to it
queue the property calculation, e.g. the energy
initialise the respective printer instance and pass references to the vault and the timing objects
call the printout method of the printer instance and pass the propery result
free all memory on heap

FIGURE 4.13: Flowchart of the `run` method used in the `Do_qmp2` method instance.

Listing 4.9: Code excerpt of the precalculation for the exponential factors used for scaling in the algorithm.

```
double* m1Deps_o = new double[p1Dnpts * nocc]();
double* m1Deps_v = new double[p1Dnpts * nvirt]();
double* c1Deps_ov = new double[p1Dnpts * nocc * nvirt]();
for (size_t k = 0; k < p1Dnpts; k++){
    for (size_t i = 0; i < nocc; i++){
        m1Deps_o[k * nocc+ i] =
            pow((double) v1Dpts[k], (double) (- vf[i]));
    }//for i
    for (size_t a = 0; a < nvirt; a++){
        size_t pos_f = a + nocc;
        m1Deps_v[k * nvirt + a] =
            pow((double) v1Dpts[k], (double) ( vf[pos_f]));
    }//for a
    double inv_t = 1.0 / (double) v1Dpts[k];
    for (size_t i = 0; i < nocc; i++){
        for (size_t a = 0; a < nvirt; a++){
            c1Deps_ov[k * nocc * nvirt + i * nvirt + a] =
                m1Deps_o[k * nocc + i] * m1Deps_v[k * nvirt + a] * ↩
    inv_t;
        }//for a
    }//for i
}//for k
```

4.3.6.2 `Qmp2_energy` in Detail

The diagram for the property instance of the Q-MP2 energy is given in figure 4.14. It consists of private reference and pointer attributed to the necessary data which was passed to it through its constructor.

The constructor does one final setup step before the calculation can be run. This is an optimization specific to the Q-MP2 method, but the coulomb integrals I_{ia}^{P} can be bundled together with the weights of the three-dimensional grid, ω^{P}. Therefore they are scaled once with the appropriate weight for their point P to avoid unnecessary computational work (see algorithm 4).

The implemented algorithm for the Q-MP2 method is given in alg. 5. The algorithm loops through every point K on the one-dimensional integration grid. For each point, it then loops through the larger three-dimensional grid P. Then one

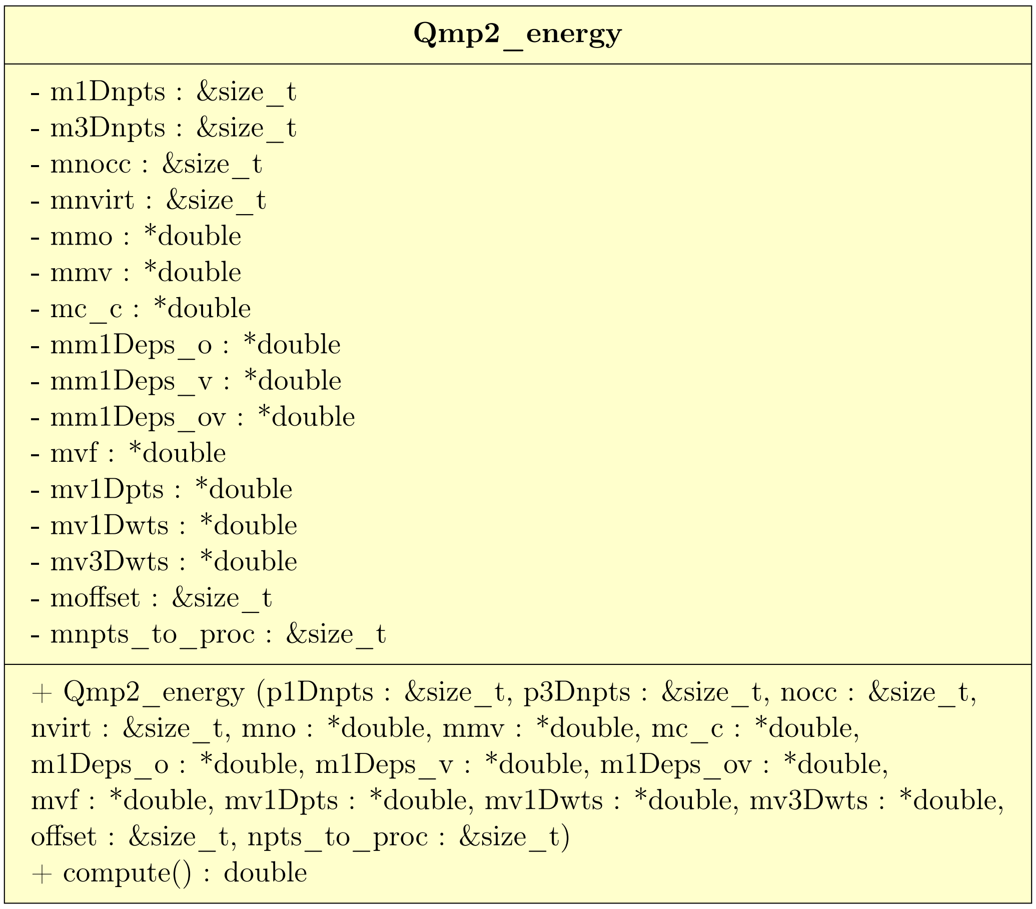

Qmp2_energy
- m1Dnpts : &size_t
- m3Dnpts : &size_t
- mnocc : &size_t
- mnvirt : &size_t
- mmo : *double
- mmv : *double
- mc_c : *double
- mm1Deps_o : *double
- mm1Deps_v : *double
- mm1Deps_ov : *double
- mvf : *double
- mv1Dpts : *double
- mv1Dwts : *double
- mv3Dwts : *double
- moffset : &size_t
- mnpts_to_proc : &size_t
+ Qmp2_energy (p1Dnpts : &size_t, p3Dnpts : &size_t, nocc : &size_t, nvirt : &size_t, mno : *double, mmv : *double, mc_c : *double, m1Deps_o : *double, m1Deps_v : *double, m1Deps_ov : *double, mvf : *double, mv1Dpts : *double, mv1Dwts : *double, mv3Dwts : *double, offset : &size_t, npts_to_proc : &size_t)
+ compute() : double

FIGURE 4.14: Class diagram of the base `Qmp2_energy` class of the `methods` module which run the Q-MP2 energy calculation.

Algorithm 4: Prescaling of the Integral I_{ia}^{P} slices done in the constructor of `Qmp2_energy` to avoid double computation later.

```
1 for each point $P$ do
2     for each occupied orbital $i$ do
3         for each virtual orbital $a$ do
4             $I_{ia}^{P} \mathrel{*}= \omega^{P}$
```

final optimization is done. To avoid unnecessary work the occupied and virtual orbital blocks as well as the coulomb integrals are scaled with their respective precalculated scaling factor shown in listing 4.9. Once again this does not contribute significantly to the memory complexity of the algorithm, as the largest object stored is only of $\mathcal{O}(OV)$, as it is only that single slice P of the integral for this iteration. Then the algorithm loops over the second set of indices of the same spatial grid, Q. However, all of the following calculations consist of fully commutative multiplications. Therefore, only the 'upper diagonal triangle' of the

P, Q pair has to be calculated. This results in only $\frac{P^2}{2}$ points that have to be evaluated. After all parts are brought together, the remaining energy is weighted with the one-dimensional weights ω^K and multiplied by two to double count the nondiagonal elements.

Algorithm 5: Algorithm used in the `Qmp2_energy` method property class in the `methods` module to calculate the energy correction.

```
$E^{\text{Q-MP2}} = 0$
for each point $K$ of one dimensional integration do
    for each point $P$ in spatial integration do
        for each occupied orbital $i$ do
            $o_i^P = \Phi_i^P * S_i^K$        // $S$ being the scaling factor mm1Deps_k
        for each virtual orbital $a$ do
            $v_a^P = \Phi_a^P * S_a^K$        // $S$ being the scaling factor mm1Deps_k
        for each occupied orbital $i$ do
            for each virtual orbital $a$ do
                $c_{ia}^P = I_{ia}^P * S_{ia}^K$   // $S$ being the scaling factor mm1Deps_k
        for each point $Q < P$ in spatial integration do
            $D = 0$
            for each virtual orbital $a$ do
                $t_1 = 0$
                $t_2 = 0$
                for each occupied orbital $i$ do
                    $t_1 \mathrel{+}= o_i^P * I_{ia}^Q$
                    $t_2 \mathrel{+}= \Phi_i^Q * c_{ia}^P$
                $D \mathrel{+}= t_1 * t_2$
            $C = 0$
            for each occupied orbital $i$ do
                for each virtual orbital $a$ do
                    $C = c_{ia}^P * I_{ia}^Q$
            $O = 0$
            for each occupied orbital $i$ do
                $O \mathrel{+}= o_i^P * \Phi_i^Q$
            $V = 0$
            for each virtual orbital $a$ do
                $V \mathrel{+}= v_a^P * \Phi_a^Q$
            $R = (D - 2 * C * O) * V$
            if $P \neq Q$ then
                $R \mathrel{*}= 2.0$
            $E^{\text{Q-MP2}} \mathrel{+}= R * \omega^K$
```

The largest spatial scaling that has to be consistently held in memory at any point of the algorithm is the coulomb integral in its molecular orbital form, which scales with $\mathcal{O}(POV)$. The largest overall memory necessary for the full calculation is its atomic form of $\mathcal{O}(PN^2)$, which is provided by the host program.

This algorithm has a final computational scaling of $\mathcal{O}(KPQOV)$. With $Q = \frac{P}{2}$ and K being in the order of one to ten this reduces to $\mathcal{O}(P^2OV)$. Comparing this to the atomic to molecular orbital transformation, which scales with $\mathcal{O}(POVN)$ one could assume the latter to be the bottleneck. However, as P is linearly dependent on the number of atoms and the grid size, P has a nontrivial connection to the system size as well. The standard Becke weighted grid[106] is generally large enough that the P^2 factor outweighs the PN^2. This makes the grid size to be the sole deciding factor in the computational scaling of the method. This is why the investigation into optimized grids for this algorithm was started. A first look is given in chapter 5.

Using the `Eigen` library

Additionally, a second variant of the Q-MP2 algorithm was implemented. This variant uses the `Eigen 3`[105] library.

`Eigen` provides linear algebra techniques for matrix and vector operations, numerical solvers, and related algorithms. It is a template-based header-only library. This makes it easy to implement, as it does not require complicated compilation setups. It is released under the MPL2 license and is open-source.

This implementation was used as a comparison point for the array variant the author wrote. It is included in the repository and can be optionally activated through a compilation variable and a path to the local `Eigen` location.

The library provides high-level matrix and vector containers, which can also be mapped to continuous memory arrays, as shown in 4.10. The library provides optimized methods for these containers.

LISTING 4.10: Example matrix constructs used in the `Eigen` variant of the algorithm.

```
//on heap declared memory for the occupied orbital block
double* mmo = new double[p3Dnpts * nocc]();
// Using Eigens matrix maps
using MatMap = Map<Matrix <double, Dynamic, Dynamic, RowMajor>>;
using VecMap = Map<VectorXd>;
MatMap map_mmo(mmo, m3Dnpts, mnocc); // occupied orbitals
```

Some of the ones used in this implementation are shown in listing 4.11. The method setup makes the code much more readable, as nested contractions are handled by a single matrix-vector operation command.

LISTING 4.11: Example usage of the utilised `Eigen` methods.

```
// scale each cell of the map with one scalar
o_p = map_mmo.row(p)
    .cwiseProduct(map_mm1Deps_o.row(k));
// transpose a matrix/vector
VectorXd o_q = map_mmo.row(q).transpose();
// form dot product between two vectors
double o = (o_p).dot(o_q);
// form sum of all elements in the matrix/vector
double j = (c2_p.cwiseProduct(map_mc_c_q)).sum();
```

LISTING 4.12: Example calculation of the C term for the only-array and for the `Eigen` implementation.

```
//array only calculation of the X term
double X = 0;
for (size_t i = 0; i < mnocc; i++){
    for (size_t a = 0; a < mnvirt; a++){
        C += m_c_p[i][a]
            * mc_c[q * mnocc * mnvirt + i * mnvirt+a];
    }
}//for i
// Eigen style calculation of the same term
double C = (c2_p.cwiseProduct(map_mc_c_q)).sum();
```

An example of the difference between the two implementations can be seen in listing 4.12. Whereas the only-array implementation has multiple nested for loops to perform the matrix multiplication for each matrix cell manually, the `Eigen` provides the same functionality in a much more concise form. The trade-off is the lost insight and control over the data movement. In the above example, it is clear that the data is iterated in a way that is optimal to avoid cache misses and facilitate vectorization. The `Eigen` implementation does not provide any insight into how the data is traversed. This is the main reason why the low-level implementation was favored for this project.

The performance comparison between the two implementation variants is discussed in chapter 6.

4.3.7 The Test Suite

The test suite is based on `ctest` which is facilitated by the compilation setup software `cmake`. The basic structure follows the `libqqc` library. Each `libqqc` class and each method in this class has its `test_` equivalent in the same setup, plus some testing data to test against. Each module has a collector file that calls each of these tests and checks the returned Booleans for any errors. Executing `ctest`, as is done by the continuous integration setup on GitHub, returns how many and which ones failed. An example can be seen in listing B.4.

Chapter 5

Grid Benchmark

The following chapter will provide an overview of the grid dependency of the Q-MP2 method using standard Becke-weighted molecular grids implemented in `Q-Chem` and `libqqc`. Section 5.1 will provide a basic primer on the integration grids, whereas sections 5.2 and 7.1.1 will summarise the benchmark of the grid performance of Q-MP2.

5.1 Introduction to Integration Grids

This section provides the basic info about the two integration grids that the Q-MP2 was tested on. A one-dimensional integration grid K is introduced by the Laplace transformation of the energy denominator as well as a molecular grid P (or Q) in three-dimensional space on which the integral is evaluated.

5.1.1 Gauss Quadrature

The one-dimensional integration is done through Gauss quadrature. The original integral is approximated by an orthogonal polynomial function $p(x)$ times a weight function $w(x)$.

$$\int_a^b f(x)dx \approx \int_a^b p_n(x)w(x)dx = \sum_{k=1}^{K} p(x_k)\omega_k \tag{5.1}$$

with

$$\int_a^b p_k(x)p_l(x)w(x)dx = \delta_{kl} \tag{5.2}$$

The points at which these functions are evaluated, x_k, are equal to the roots of the polynomial. The choice of the polynomial function and the weight function depend on the specific Gauss quadrature variant. Two of these variants were used in the thesis. The integration to approximate the energy denominator is a Gauss-Legendre type integration. The weight function is

$$w(x) = 1 \tag{5.3}$$

and the polynomial function is a Legendre polynomial of the first kind, which generation function follows

$$p_n(x) = \sum_{k=0}^{n/2}(-1)^k \frac{(2n-2k)!}{(n-k)!(n-2k)!k!2^n} x^{n-2k} \tag{5.4}$$

The integration points follow the form of

$$x_k \approx \cos\left(\pi \frac{4k-1}{4n+2}\right) \qquad k = 1, \ldots, n \tag{5.5}$$

The molecular grid will be discussed later, but its implementation uses the Gauss-Chebyshev integration. The weight function is

$$w(x) = \frac{1}{\sqrt{1-x^2}} \tag{5.6}$$

and the polynomial is a Chebyshev polynomial of the second kind, which is a recursively generated function. Its roots follow

$$x_{k,n} = \cos\left(\frac{2k-1}{2n}\right) \tag{5.7}$$

with the integration weights

$$\omega_{k,n} = \frac{\pi}{n} \tag{5.8}$$

5.1.2 Molecular Becke Grid

The molecular grid used in the following benchmark employs the Becke partitioning scheme[106] for polyatomic molecules. It proposes that any polyatomic molecular function $F(x)$ can be approximated as the sum of its single-center components $F_k(r)$ with a nuclear, relative weight function $w_k(r)$. The sum of all weight functions over the molecule is defined as unity.

$$\sum_n w_n(r) = 1 \tag{5.9}$$

and the full molecular function and its integral can be approximated as

$$F(r) = \sum_k w_k(r)F(r) \tag{5.10}$$

$$I = \sum_k I_k \tag{5.11}$$

Only single-center integrations have to be carried out. The integral for these parts is approximated through quadrature in a spherical coordinate system

$$I_k = \int drd\phi d\psi F_k(r,\phi,\psi)r^2 \tag{5.12}$$

where the two angular dimensions are dealt with as a quadrature over a unit sphere by Gauss-Lebedev quadrature (or Gauss-MacLaurin in the `Q-Chem` implementation) and the radial part is approximated by Gauss-Chebyshev quadrature. This treatment leads to the generation of the integration points on spheres over the atomic center, which has been visualized for a simplified two-dimensional case in figure 5.1.

The main work of the Becke partitioning scheme is focused on deriving fitting and well-behaved nuclear weighting functions for the final integration. To do so, the

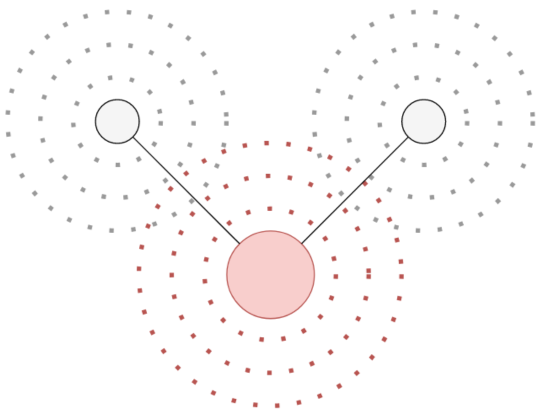


FIGURE 5.1: Simplified example of grid point generation in standard Becke weighted molecular grids on the H_2O molecule.

molecular system is first partitioned into Voronoi cells in which the nuclear weight functions for each center are then defined.

There are numerous algorithms available for the generation of these cells. This thesis will quickly summarise the Delaunay triangulation, which was implemented into `libqqc`[5].

5.1.2.1 Delaunay Triangulation

The Delaunay triangulation generates edge connections between given points in such a way that the resulting triangles have a maximized minimal angle, which avoids triangles with very sharp angles and low area/volume. One algorithm to generate this triangulation is named after papers of Boyer[107] and Watson[108], which is implemented in the library[5] following a master thesis by Isabel Vinterbladh. In the three-dimensional case, this leads to a set of tetrahedrons.

5.1.2.2 Voronoi Cells

From the Delaunay triangles/tetrahedrons, the polyhedrons of the Voronoi cells are generated. Becke uses these cells to determine the appropriate nuclear center for the weighting function, as all points in a cell are closer to the nuclear center that generated the cell than to any other centers.

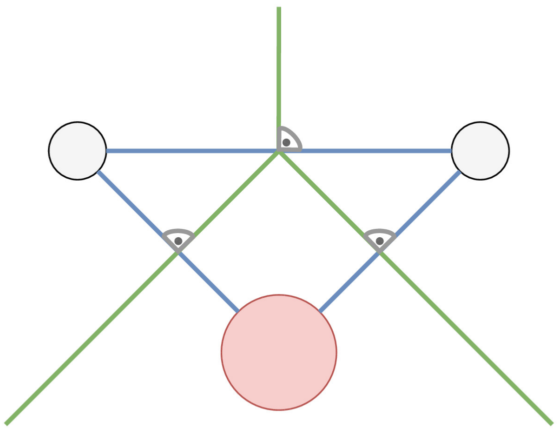


FIGURE 5.2: Example of the Voronoi cells (green) resulting from the Delaunay triangulation (blue) for a simplified H_2O molecule.

Using the Delaunay triangles/tetrahedrons perpendicular bisecting planes are generated on the edge vectors, which form the Voronoi polyhedrons. A simplified view of this scheme is given in figure 5.2.

5.1.2.3 Construction of Becke Weights

The Voronoi polyhedrons are a binary bounding condition for the nuclear weight function, which is not a well-behaved function for the quadrature integration mentioned above. Becke, therefore, smoothed the boundaries of these cells out into overlapping, analytically continuous functions. The final weight for a grid point is a sum of all nuclear weight functions for each atomic center. For a point close to the nuclear center i all other weight contributions from centers j should vanish.

A hyperboloid parameter μ is defined for each center pair i, j

$$\mu_{ij} = \frac{r_i - r_j}{R_{ij}} \tag{5.13}$$

with a range of $-1 < \mu_{ij} < 1$. For each nuclear weight, a step function $s(\mu_{ij})$ is introduced which vanishes if the integration point i is too far away from the respective nucleus j.

$$s(\mu_{ij}) = \begin{cases} 1, & -1 \leq \mu_{ij} \leq 0 \\ 0, & -10 < \mu_{ij} \leq 1 \end{cases} \tag{5.14}$$

To smooth out the step function a cutoff function is employed

$$s(\mu) = \frac{1}{2}(1 - f(\mu)) \tag{5.15}$$

with the smoothing function being a simple two-term polynomial such as

$$f(\mu) = \frac{3}{2}\mu - \frac{1}{2}\mu^3 \tag{5.16}$$

To further smooth out the step function this polynomial is used recursively with three iterations being seen as sufficient by Becke.

$$s(\mu) = \frac{1}{2}(1 - f(f(f(\mu))) \tag{5.17}$$

Using these smooth step functions, a cell function for the Voronoi polyhedron is defined as

$$P_i(r) = \Pi_{j \neq i} s(\mu_{ij}) \tag{5.18}$$

and the resulting nuclear weighting function is constructed using this cell function weighted by the sum of all cell functions

$$w_k(r) = \frac{P_k}{\sum_l(r)} \tag{5.19}$$

The resulting grid of atom-centered spherical grid points is the main integration grid used for many methods, such as the numerical evaluation of density functional theory integrals. It was used as the integration grid for the molecular three-dimensional part of Q-MP2.

In the previous chapter only a proof-of-concept energy benchmark was given for the Q-MP2 method. What follows is a scan over the different integration grid size combinations to investigate the grid dependency of the method.

5.2 Grid Benchmark

5.2.1 Computational Details

To test the grid dependency of the Q-MP2 energy a benchmark set of small and medium-sized molecules was used. The molecules are H_2, H_2O, CO, CH_4, $H2_CO$, H_3COH, NH_3. The geometries of the structures were optimized with MP2 and the same basis set level as the Q-MP2 calculations. The basis sets STO-3G, 3-21g, and 6-31g were chosen to see the effect on the energy correction in going from single-valence to double-valence basis sets and within the Pople basis set[73–75]. The small size of these basis sets allowed fast iteration and consistency with the results in the next chapter, where larger systems are discussed. All calculations were carried out with `Q-Chem` 5.2.2 and a development version of the `libqqc` library on the computer cluster of the Dreuw group.

5.2.2 Energy Benchmark

The average error for the Q-MP2 correlation energy compared to the parent MP2 energy is given in tables A.2, A.3 and A.4 for the set of chosen molecules. Some of these results have been published in a previous thesis of the author[6], but have been recalculated with the current version of the `libqqc` library to ensure consistency. It should be noted that the energies have been compared to the parent method at the same basis set level, so Q-MP2/STO-3G has been compared to MP2/STO-3G.

The results for the minimal STO-3G basis are visualized in figure 5.3. The Becke-weighted P grid is chosen as a tuple of radial and angular grid points. The seven most commonly used combinations in `Q-Chem` have been investigated with varying numbers for the one-dimensional K grid from 4 to 10. For anything but the smallest 20/6 grid the standard chemical accuracy of $< 0.0015\ h$ is reached. Generally, $N_K = 4$ is sufficient for the one-dimensional K grid, as this choice produces the lowest correlation energy error. Only the 20/18 grid falls out of line with this behaviour. Grids with $N_K > 4$ result in similar errors.

Going up to the split-valence basis 3-21g in figure 5.4 lowers the accuracy of the method. As the orbital function density increases the penalty of the incomplete quadrature description also increases. $N_K = 4$ produces the best estimate of the

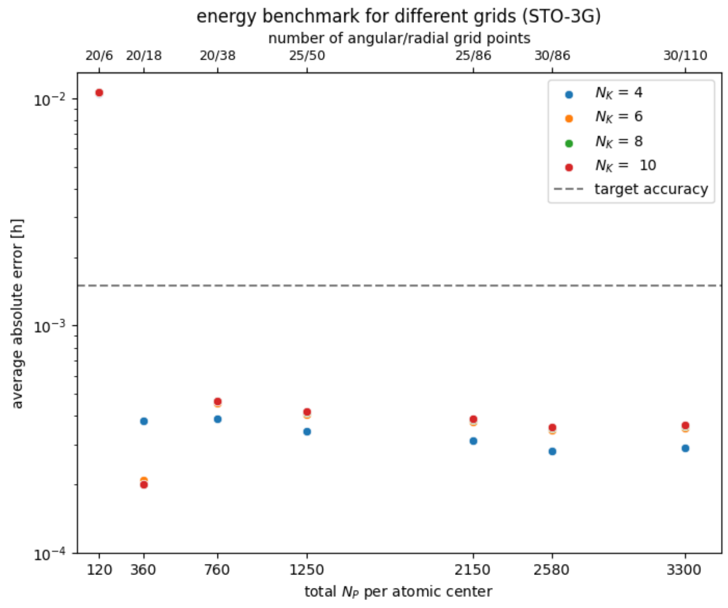


FIGURE 5.3: Average absolute correlation energy correction error of the benchmark set for the STO-3G basis.

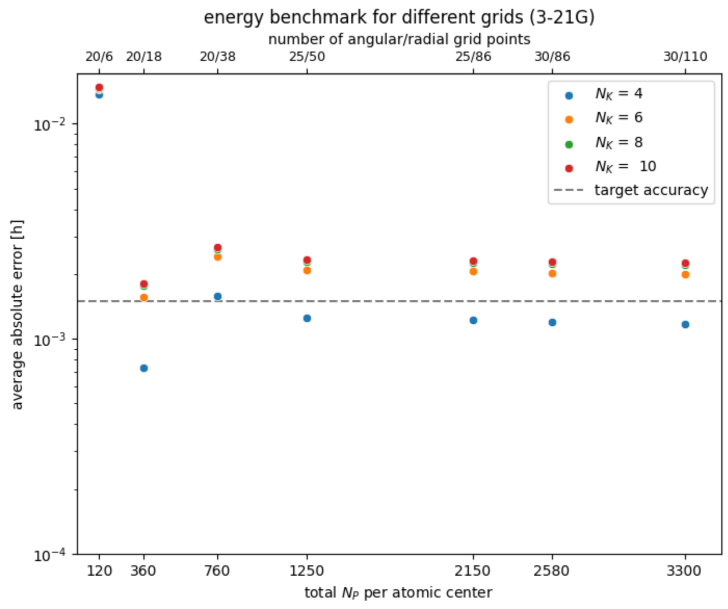


FIGURE 5.4: Average absolute correlation energy correction error of the benchmark set for the 3-21g basis.

energy, reaching the target chemical accuracy. Higher amounts of points increase the energy error. Once again, the smallest 20/6 set fails to reach the target accuracy by a magnitude.

Increasing the number of basis functions in the split-valence series to the 6-31g case results in lowering the overall error of the method, as can be seen in figure 5.5. The

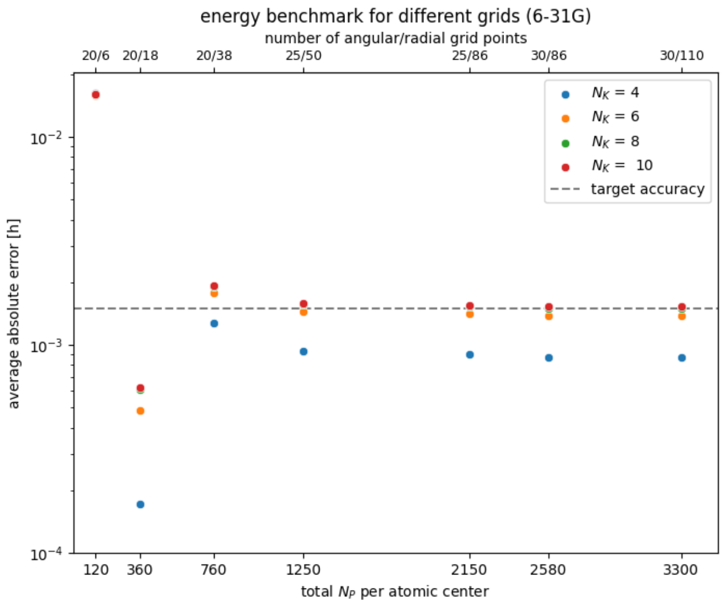


FIGURE 5.5: Average absolute correlation energy correction error of the benchmark set for the 6-31g basis.

trends of the smaller 3-21g basis hold. $N_K = 4$ produces the best reproduction of the energy, with the higher number of quadrature points recovering similar energy values. The target accuracy is once again reached for all combinations but the smallest one, with the 20/18/4 error being a magnitude smaller than the target.

In most cases, $N_K = 4$ reproduced the best estimate of the energy. Figure 5.6 compares the three results for this quadrature. For most combinations, the smallest basis set reproduces its energy the best, with 6-31g performing better than 3-21g. Once again, the 20/18 case falls out of this behavior, with 6-31g performing the best.

It was therefore concluded that the energy was sufficiently recovered for grids of size 20/18 and up for the given basis set. Important for the investigation into the advantages of the Q-MP2 method is the computational performance, which will be covered in the next chapter. To keep a balance between the number of points which will heavily impact the runtime due to the PQ scaling prefactor and a sufficient number of molecular orbitals for the OV part of the scaling the $6-31g$ basis was used together with the 20/38 Becke-weighted grid and $N_K = 4$ one-dimensional quadrature points.

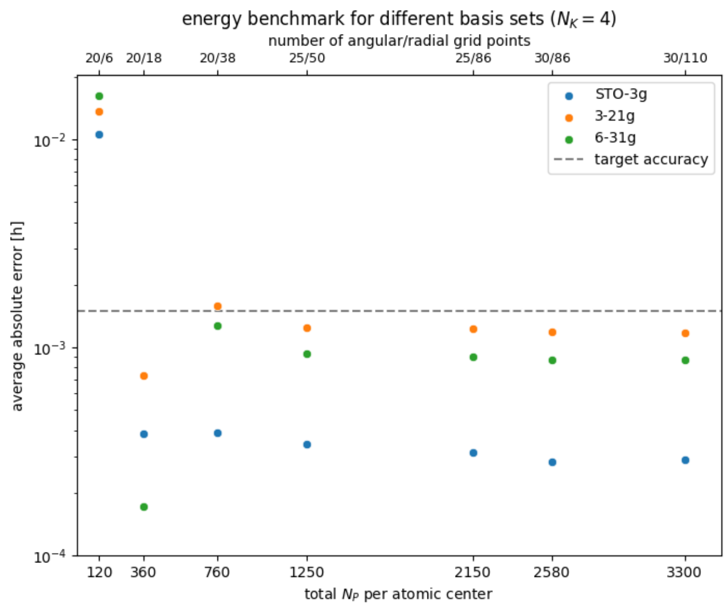


FIGURE 5.6: Average absolute correlation energy correction error of the benchmark set comparing basis sets for $N_K = 4$.

Chapter 6

High-Performance Optimisations

The following chapter will focus on the main topic of this thesis, performance and parallel efficiency of the Q-MP2 method. Section 6.1 will give an introduction to high-performance computing concepts and the parallelism paradigms used in the implementation. Section 6.2 will showcase important implementations of how the parallelism was achieved. Finally, section 6.3 will show and discuss the performance benchmarks.

6.1 High-Performance Concepts

High-performance computing is an area of computer science where programs are executed on large-scale cluster systems. It calls for performant code, both on the individual nodes and the whole computational cluster. Depending on the resources available different strategies have to be employed. To do so, one first has to understand these resources. This section will dive into the single-node optimisation strategies explored and used in the `libqqc` code.

Single-node performance is mostly reliant on two areas: the number of operations which have to be carried out and the number of memory transactions that have to be dealt with.

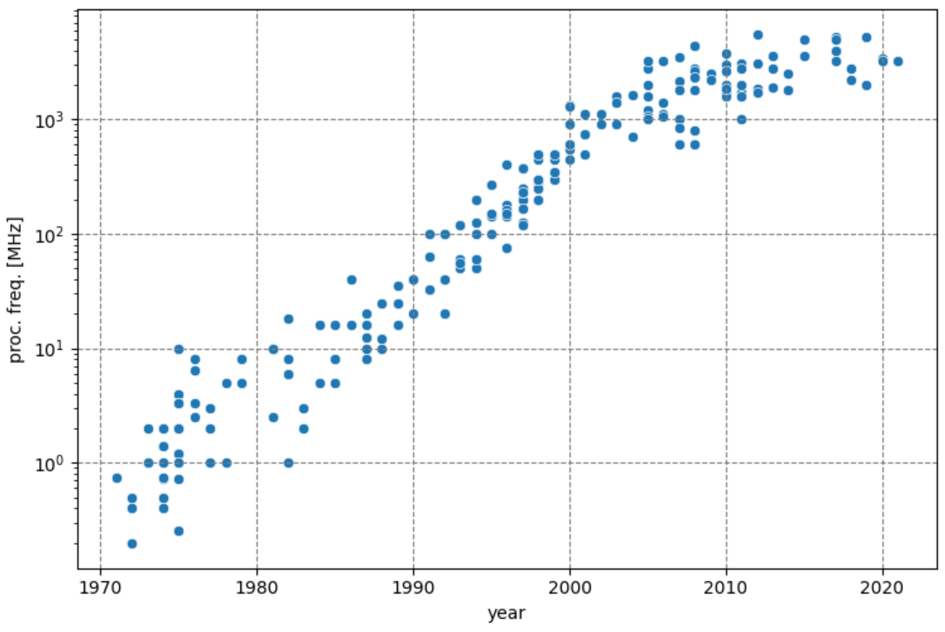


FIGURE 6.1: Historical overview of processor clock speed over the decades.

6.1.1 Processor Performance

Each operation performed by a central processing unit (CPU) takes a certain amount of time to perform. The smallest operation unit is called a cycle. The amount of cycles that the CPU can process in a second gives rise to the frequency of the processor, which is the most common way how performance is advertised. Another often-used number is the number of operations that can be performed on floating point numbers per second (FLOPS).

An increase in the frequency of performed operations used to be the hallmark of the development of CPUs for many years, as can be seen in figure 6.1. To increase it, the density of the underlying transistors of the chip (also called a die) has to be increased. The exponential growth seen in the figure leads to the well-known Moore's Law, which in one of its iterations predicts the doubling of the transistor density per chip every year and with it the increase of frequency of the processor. The development for the more modern CPUs in this graph is more complex: nearly all modern CPUs have a mode in which the speed can be temporarily boosted or throttled depending on demand and temperature. Some of these options can be disabled in the BIOS, however, most clusters which are not specifically used for performance benchmarking, will have them enabled.

Moore's Law started breaking down about two decades ago. The main reason for it is that the energy which is required for the transistor to function has a direct connection to the heat that needs to be dissipated to avoid hardware failure. The usual cooling strategies cannot keep up with this requirement. This is the reason why many modern CPUs try to be much more energy efficient.

There are other reasons, however. The scale at which modern chips are built leads to undesired quantum effects, such as electron tunnelling. This calls for more sophisticated chip designs, which drive the cost of the chip facilities and the production costs up. Which in turn cuts down on profit margins. Today most CPU speed improvements stem from other areas instead.

The main solution for the last two decades to this conundrum was the idea of parallel execution. This is done in multiple ways in modern chips. Modern consumer-grade CPUs can have up to 64 physical cores and modern motherboards, especially in a server-based setting such as a cluster, can have multiple CPUs installed. However, the increase of cores in CPUs does not automatically increase performance.

Programs have to be optimised by software engineers for execution on multiple cores. This is often not straightforward and for many programs not possible, as too much communication has to be carried out between operations. The following sections will delve deeper into the intricacies of parallelising programs for execution across multiple CPU cores.

A different kind of parallelism is sometimes called hyperthreading. Physical cores are split into two logical ones, which share a unit for floating point operations, as well as the other physical resources. The performance increase comes from the idea that if one of the logical cores has to wait for a data transaction, the other can use these resources to operator on already available data. The speed gain of hyperthreading is minimal to non-existent in compute-heavy scenarios, which is why all results presented in this thesis are reported on physical cores (with the threads/tasks bound to them).

Another improvement to the performance of modern CPUs was the introduction of fused multiply-add units. These are components which can deal with operations of the type

$$y \leftarrow a * x + b \tag{6.1}$$

in one combined operation. This type of parallelism is often called single instruction, multiple data (SIMD) or vectorization. Modern vectorization instruction set extensions, such as Intel's AVX, can handle up to 16 single-precision floating point operations, or 8 double-precision floating point operations at once. However, once again, this has many pitfalls. Generally speaking, vectorization relies on the developer. Modern compilers do try to optimise for vectorizations in higher-level optimisation settings, but their results are not guaranteed. Also, once again, not all code can be efficiently vectorized. Conditional heavy code may make vectorization not possible. Data has to be specifically aligned in memory for the vectorization to work. Different CPU manufacturers may implement the vectorization instruction set in different ways. Something that works on one CPU, may not work on another. All in all, the amount of control a developer has in vectorizing their code is limited by many technical challenges.

The peak performance of a modern CPU can be generalised as the speed of the processor times the number of cores times the number of FMA units times the operations per FMA unit possible (depending on single- or double-precision numbers).

$$PP = \frac{f * n_{\text{cores}} * n_{\text{FMA unit}} * n_{\text{operations per FMA}}}{second} \tag{6.2}$$

6.1.2 Memory Performance

A calculation operation can only be performed if the necessary data is available in the CPUs register. To do so, it has to be transferred from higher-level memory. The developer's need for fast memory transactions and the manufacturer's aim for cost reduction have resulted in a memory hierarchy, each with different transaction speeds. Generally, the faster the memory, the closer it is physically to the CPU and the smaller its capacity, as it is more expensive to build. Figure 6.2 gives an overview of such a hierarchy for a multiple-core system.

The closest memory to a CPU core is its registers, which can hold around 125 double-precision numbers on modern hardware. It then has two levels of exclusive cache, where the first level (L1) is split into two parts: one for describing which instructions to use and one for the data on which the instructions will be performed. This level can be typically accessed within a nanosecond, which is

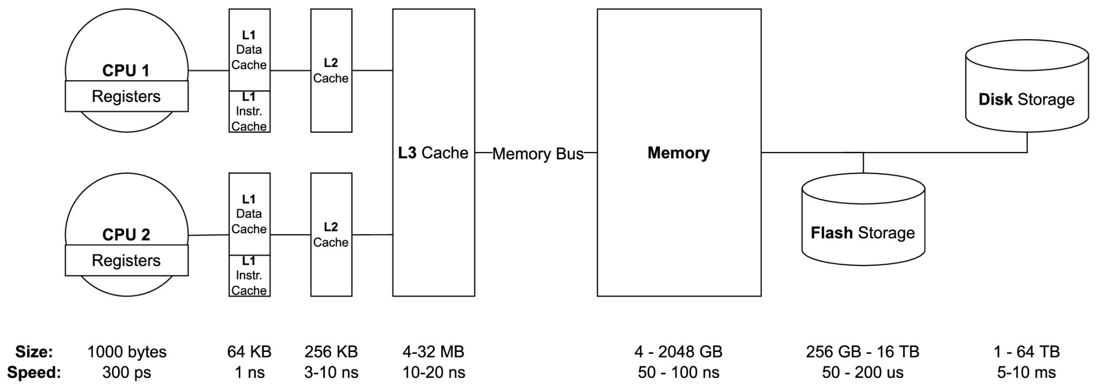

CPU 1
Registers
CPU 2
Registers
L1 Data Cache
L1 Instr. Cache
L2 Cache
L3 Cache
Memory Bus
Memory
Flash Storage
Disk Storage
Size: 1000 bytes 64 KB 256 KB 4-32 MB 4 - 2048 GB 256 GB - 16 TB 1 - 64 TB
Speed: 300 ps 1 ns 3-10 ns 10-20 ns 50 - 100 ns 50 - 200 us 5-10 ms

FIGURE 6.2: Overview of the memory hierarchy of modern CPUs with typical sizes and speeds of commercial desktop hardware and cluster server.

about one magnitude slower than the register. The next step-down performance-wise happens in the third level (L3) cache. It is typically shared among all CPU cores on a board and can be accessed within around 10 nanoseconds, which is two magnitudes slower than the registers. However, its storage is around 4 magnitudes higher than that of the core's registers. It is generally the last level of CPU-only cache. If the data is not within one of these caches they have to be fetched from the machine's memory. There are two types of machine memory. The RAM (random access memory) can hold up to terabytes of data and can be accessed in around 100 nanoseconds. If it has to hold data that is too large, some of it can be outsourced as virtual pages on the main storage together with the other stored data. The two types of storage mediums in modern computers are either flash memory (e.g. SSD, M.3, ...) or physical disk storage. Flash storage can be accessed in around 100 microseconds, whereas disk storage is particularly slow at around 10 milliseconds.

Why is it so important to know the memory hierarchy of a system if one works on high-performance computations? If data is not found in the registers of a CPU core it is fetched from the cache. If it is not found in L1, it will be searched for in L2, then L3, etc. Once found, it is transported towards the CPU through all levels of memory and cache. Therefore, it is paramount that as much of the data operated on is in the cache which is closest to the CPU.

However, CPU memory design does not work on single numbers. When a cache cascade is started data is fetched from higher levels in packages of a fixed length, which are called cache lines. So, for example, a byte which was found in L2 would be transported into L1 together with the physically following 63 bytes in one complete cache line. If the following data is needed next, no transaction is required, which is called a cache hit. If the data is not in these 64 bytes, or any other 64 bytes at the same level, one of the existing cache lines is overwritten with the fetched line from a higher level. This is called a cache miss, and it carries with it a latency depending on where the necessary data resides. Meaning, that a cache miss can, in the worst case, take tens of milliseconds to resolve, as a RAM page has to be loaded in the ram and then transported through all the lower levels.

Cache misses and their latency penalty give rise to one of the most important concepts behind performance optimisation: locality.

6.1.3 Locality

A cache miss carries with it a waiting time until the data has been fetched to the low cache levels. It is therefore paramount to the performance that cache misses are avoided as much as possible. This is done by iterating through the data in the same order as it is physically present (spatial locality) and keeping often-used data as long as possible in the cache (temporal locality).

As an example, see listings 6.1 and 6.2. They differ in the way the two arrays are traversed. The arrays in 6.1 have their fastest changing indices i as the variable for the outer loop and their slower indices j in the inner loop. A good analogy would be thinking of these two arrays as two-dimensional matrices where each row has been saved after another, as shown in figure 6.3.

Listing 6.1: Example of bad spatial locality while multiplying two matrices.

```
for (int i = 0; i < N; i++){
    for (int j = 0; j < N; j++){
        result += matA[i + N * j] * matB[i + N * j];
    }
}
```

Listing 6.2: Example of good spatial locality while multiplying two matrices.

```
for (int i = 0; i < N; i++){
    for (int j = 0; j < N; j++){
        result += matA[j + N * i] * matB[j + N * i];
    }
```

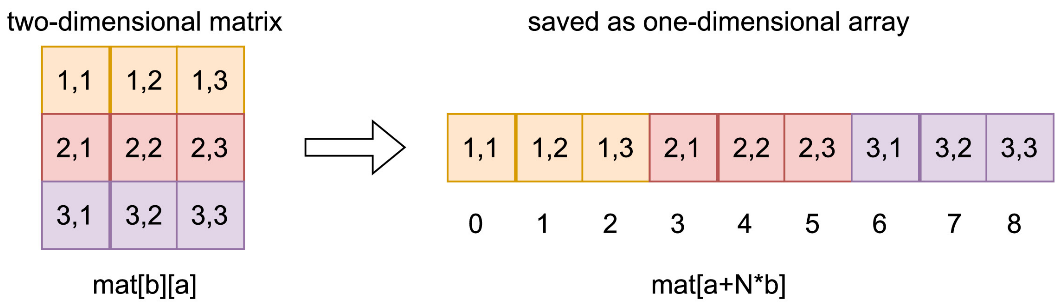


FIGURE 6.3: Example how a row-major matrix is saved in memory.

```
    }
```

In this example, the code in 6.1 would iterate over the array elements $0, 3, 6, 1, 4, 7, 2, 5, 9$ for $N = 3$. If one assumes the cache line to be 3 numbers wide, then for calculating the first elements, the CPU would grab two cache lines of the first three elements $0, 1, 2$ of both matrices. The second iteration multiplies the elements with the one-dimensional index 3 together, resulting in a cache miss and a latency penalty. The second code in 6.2 is better organised and iterates through the elements optimally, only resulting in a cache miss if all data was used.

Compiling these two codes without any optimisation $(-0O)$ shows that the more optimal example is about one magnitude faster in execution (for $N = 1024$).

As mentioned, optimising for temporal locality means trying to keep data as long as possible in low-level caches. As an example, for a classic matrix multiplication $\mathbf{C} = \mathbf{A} * \mathbf{B}$ elements of both matrices are used multiple times across the procedure, and therefore cache misses can be avoided if the calculations which use them are bundled together. This is often achieved by blocking the matrices in smaller sections and running the calculations on these blocks together.

6.1.4 Amdahl's Law

One of the most important rules in program optimisation is called Amdahl's Law. It is used to calculate performance gains that one can get for the overall program by optimising only one specific part.

$$\text{speedup} = \frac{1}{1 - p + \frac{p}{s}} \tag{6.3}$$

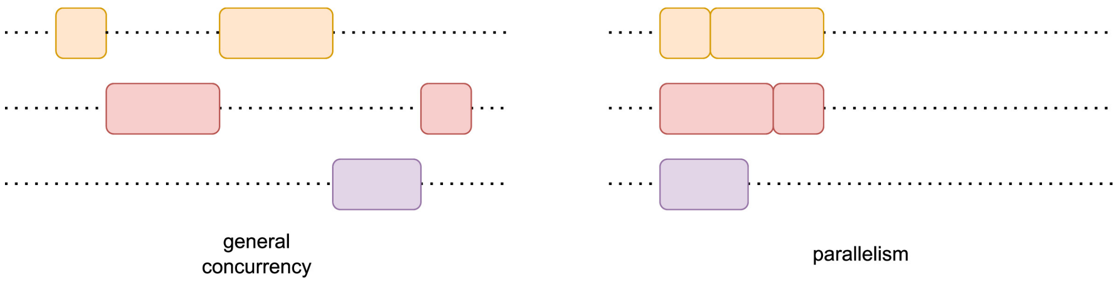


FIGURE 6.4: Difference between the overall concept of concurrency, the execution of multiple programs at the same time and true parallelism, the execution of commands at the same time. Parallelism is a subfield of concurrency.

with p being the fraction of the overall execution time of the part of the program that is going to be optimised and s being the possible speedup of the optimisation. If 90 % of the running time for a program is spent in one part, and a code change results in halving the time necessary for this part, the resulting overall speedup cannot exceed 1.8. If the part that can be optimised takes only 10 % of the execution time of the program but can be sped up 100 times then the overall speedup cannot exceed 1.1.

It is therefore very important to continuously benchmark a program and identify and improve the correct bottlenecks.

6.1.5 Exploiting Parallelism

As has been shown most of the performance improvement of processors over the last decade came from the introduction of multiple cores architectures instead of a substantial increase in clock speed. This is a form of parallelism.
Nearly all computers are capable of some form of multiprogram concurrency, which is the ability to have several programmes running simultaneously. There exist however a difference between general concurrency, as with task switching, and true parallelism. In figure 6.4 this difference is made clear: whereas computers have long been able to sustain the execution of different programs at the same time, switching between the active processes, not all were capable of directly running programs in a parallel execution model.

There are two general types of parallelism. Data-level parallelism is the execution of operations on independent data, whereas task-level parallelism consists of subprograms which can chip away at a larger execution goal independently. From there on multiple subcategories have developed. The first is instruction-level parallelism, which falls under the data paradigm. This category is mainly exploited by the compiler and the chip itself, by using very fundamental pipelining strategies and branch predictions. Pipelining means ordering operations in such a way that they can overlap temporarily (e.g. two memory requests being bundled together). Branch prediction facilities try to guess which execution path the program will take and how those can be optimised. The developer has very little leverage over these concepts. The second is the previously mentioned vector unit architecture of modern CPUs, which is also a data-level parallelism subcategory. They consist of SIMD units, which can perform combined multiply-add commands on vectors of data simultaneously. Graphic processing units, which are a type of accelerator originally used in computer graphics, also fall under this category, as their structure on the lowest level does not differ much from the typical SIMD units found in processors. They will be discussed in a later section. Exploiting vectorization as a developer is generally done by using specific math libraries, such as BLAS (basic linear algebra subprograms), which have been at least partially directly written in optimised assembler code. Modern compilers will also try to generate the vectorized assembler code from higher-level languages (such as C++), and the developer can try to encourage this by the use of specific software architectures. The third level discussed in the sections below is thread-level parallelism. Each program consists of one or multiple processes, which are contained sets of instructions and data using private process states for bookkeeping and their own address space. Threads are substructures in such processes, which share heap data among the threads of the same process. This is illustrated in figure 6.5

Thread-level parallelism has to be added by the developer, it is not automatically generated by the compiler or hardware. It is used to split up the workload while accessing the same data space and is the main parallelism paradigm in symmetric multi-processing (SMP), which will be discussed more below. A different concept is splitting the workload over different processes. This results in resource flexibility, as the processes do not have to stay locally close to each other, introducing however the problem of how to exchange data. To handle this, message-passing libraries are needed, as will be discussed in one of the next sections.

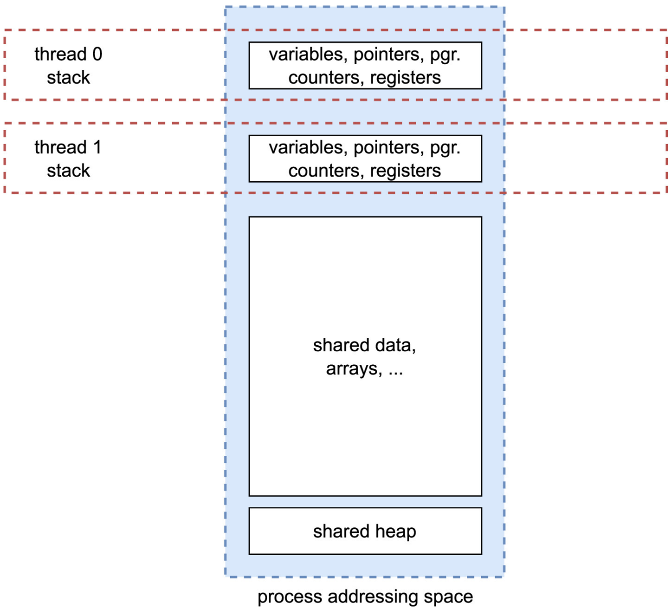


FIGURE 6.5: An overview of the address space of a process and the thread substructure.

6.1.5.1 Shared Memory Parallelism

Modern CPUs generally fall under the symmetric multi-processing (SMP) category. This means that each processor shares the same memory space and therefore each memory transaction takes the same amount of time. As has been said before, modern CPU architecture is multi-core. And for reasons of construction cost, the multi-core architecture results in a memory hierarchy (figure 6.2). This means that when processor A requests memory that lies in the L1 of processor B, or its L1, the memory transactions are not truly equal. One of the last true SMP architectures was the Cray-2 supercomputer released in 1985 and discontinued just 5 years later. However, modern CPUs are still generally considered SMP, as they share a local memory space.

This shared memory allows for a process to split itself into multiple threads, with each thread executing on a different core, but accessing the same memory address space without a need for further communications. This is called shared-memory parallelism. There are many challenges with this form of parallelism. How should the threads be scheduled to the resources? How does one ensure that the data transaction of thread A in $L3$ is being communicated to the copy of the data in the $L2$ of a different processor? These decisions are today handled by the operating system and the hardware and are mainly out of the hands of the developer, so the text will not delve further into them. The author recommends Computer

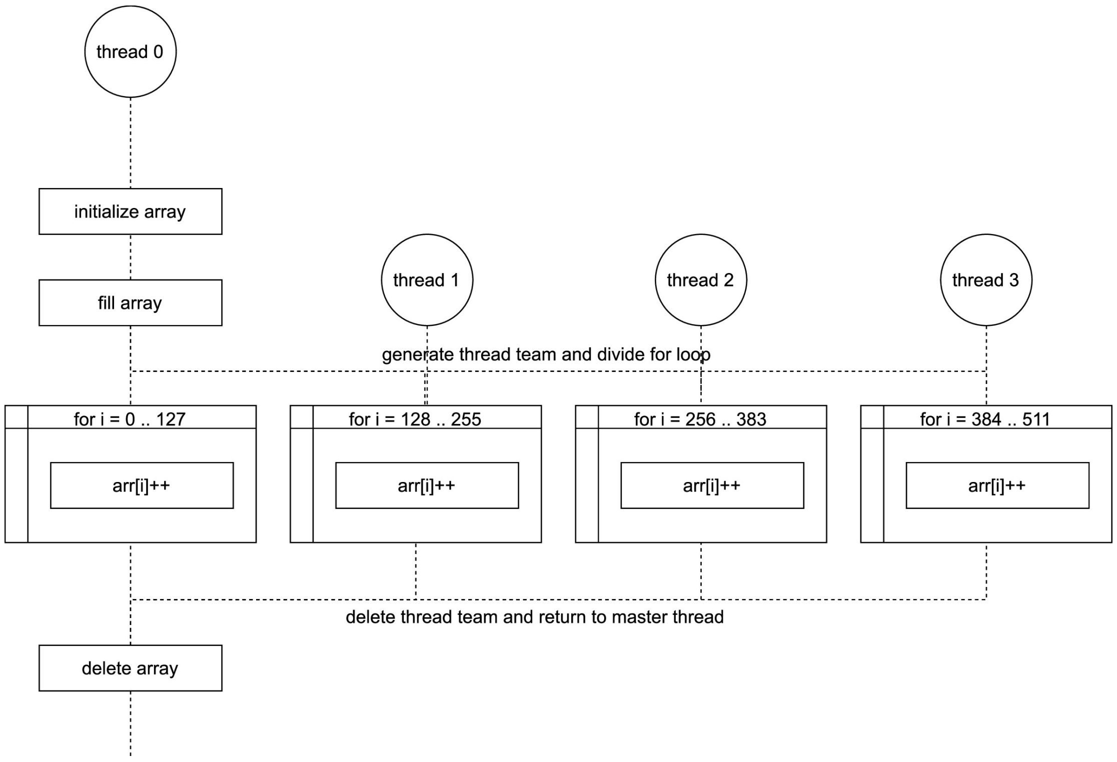


FIGURE 6.6: Fork-join model of `OpenMP`, where the master thread of the process creates the thread team, which split off and joins back after the code execution.

Architecture by Hennessy and Patterson[112] for a very close look at these underlying concepts.

The code presented in this thesis heavily relies on shared memory parallelism on the single-node level. To do so, the high-level API and library `OpenMP`[102] has been used to create the threads and divide the workload of the program.

OpenMP is a set of compiler directives (and some library functions) which allow for the high-level and easy creation and handling of thread teams. The main concept of `OpenMP` is the fork-join model.

An example code is given in listing 6.3 and illustrated in figure 6.6.

LISTING 6.3: Example use of `OpenMP` paradigm to increment 512 double values.

```
double *arr = new double[512];
std::fill_n(arr, 512, 1.0);
// generating the thread team and
// dividing the workload
#pragma omp parallel for
for (int i = 0; i < 512; i++){
    arr[i]++;
}
delete arr;
```

OpenMP works on a set of compiler directives (#pragma omp), which create the threads (`parallel`) and then execute them on a following block. In this example, the block is a for loop. The iterator of the underlying for loop is then divided among these threads and all threads are run in parallel (if resources allow). Dividing the workload among the threads is done by a scheduler and can be set with the `schedule(type, chunk)` clause. The two most used ones are of type static and `dynamic`. Scheduling can be important for performance. The static clause is the default and divides the for loop iterations into blocks of fixed sizes. It performs well if all loop iterations can be finished in about the same timeframe. It can also be used to help with vectorization. In the above example, if the chunk size is set to 4 then the automatic vectorization of the compiler may work better. The dynamic scheduler has more overhead attributed to it. It has a scheduling queue, where each iteration, or each chunk of iterations, is placed. If a thread is done with its workload, it grabs the next chunk from the queue. This is more performant than the static clause if the workload is asymmetrical. There also exists a new SIMD compiler directive with #pragma omp simd. This is meant to help with code vectorisation. However, while all other directives are directly translated by the compiler into their manual threading equivalent (e.g. using `pthreads` on Linux systems), the SIMD directive is merely a suggestion to the compiler and can be arbitrarily ignored by it. The author has found this directive to not work very well at the time of writing and even slow down calculations and therefore choose to omit it and rely on the compiler for vectorisation.

The last concept of `OpenMP` which can have a huge impact on performance is thread pinning. This is the concept of how the different threads are distributed over the available cores. Modern CPUs in a cluster context will have the previously mentioned hyperthreading technology enabled, subscribing multiple logical cores per one physical core resource. As the algorithm in this thesis is very compute intensive the performance boost of subscribing two threads to one physical core would be only a benefit if all other cores are also in use. This can be controlled at run time with the `OpenMP` environment variables such as `OMP_PLACES` to directly specify the logical cores that should be used or `OMP_PROC_BIND` to choose between a set of directives which should be followed. An example of such directives is

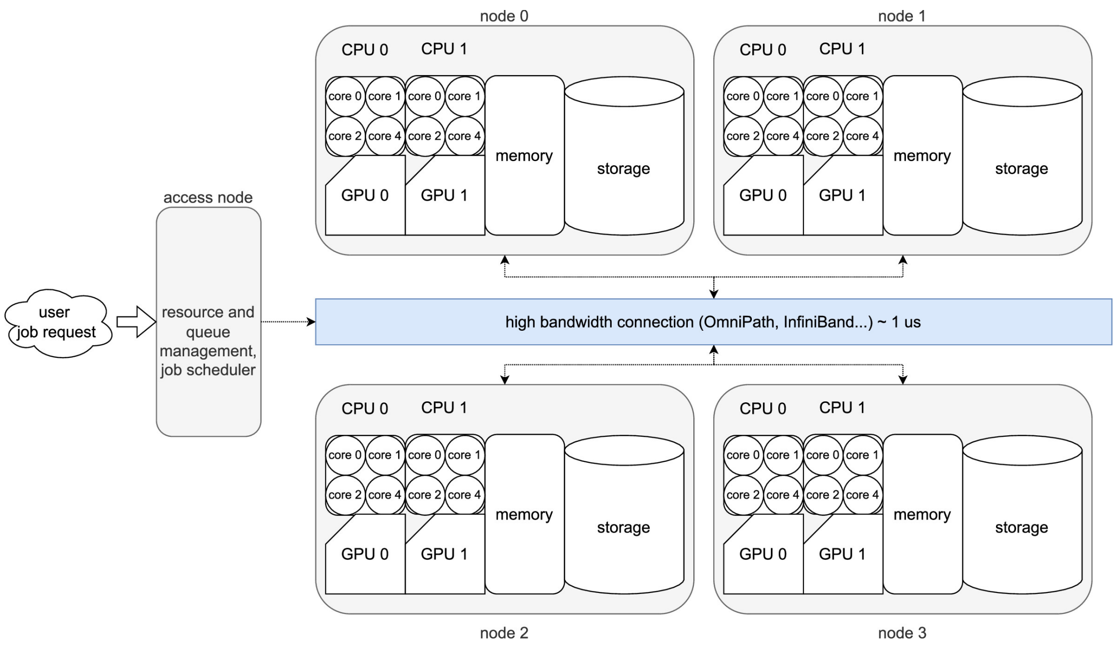


Figure 6.7: Example architecture of cluster system utilising multiple single servers (nodes) connected over a high throughput network interface.

`close`, which tries to keep threads physically close to each other, often resulting in utilising both hyper threads, or `spread`, which tries to maximise the distance of the threads.

6.1.5.2 Distributed Memory Parallelism

If the workload is not distributed over multiple threads of a process, but rather over multiple processes, the advantage of a single address space is missing. If data has to be exchanged or synchronisation is required then a networking interface has to be introduced. The most used one in scientific software is the message-passing interface (`MPI`). `MPI` is a standard for process communication, often used in (physically) distributed systems. This is the case in most cluster-based computer systems, such as the JUSTUS 2 system of the BWgrid initiative used for this thesis. It allows for any type of network communication to facilitate data exchange, be it Omni-Path, Nvidia's InfiniBand technology, or even simple TCP/IP protocols. An example of such a cluster system is given in figure 6.7.

A program using one of the two major implementations of the standard (`OpenMPI`[98] or `MPICH2`[113,114]) is run the same way on every resource instance. The parallelism aspect has to be managed by the developer, as the `MPI` standard only provides communication and supporting utilities. As no shared memory space is available (excluding the shared memory window functionality introduced in recent versions) all data movements have to be managed. Parallelism can be introduced, for example, by identifying each process during runtime and subscribing only its specific workload to it. A short example is given in code listing 6.4, which achieves the same thing as the OpenMP example, incrementing 512 numbers.

Listing 6.4: Example use of the MPI standard.

```
    size_t N = 512;
    //initialising MPI environment
    MPI_Init(NULL, NULL);
    int comm_size;
    //how many overall processes
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    int id;
    //id of this process
    MPI_Comm_rank(MPI_COMM_WORLD, &id);

    //assuming no rest
    size_t n_proc = N / comm_size;

    if (id != 0){
        //local array to increment
        double arr[n_proc];
        std::fill_n(arr, n_proc, 1.0);
        for (int i = 0; i < n_proc; i++){
            arr[i]++;
        }
        //send to master process
        MPI_Gather(arr, n_proc, MPI_DOUBLE, NULL,
                n_proc, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }
    if (id == 0){
        //creating final buffer on master
        double *final_arr = new double[N];
        std::fill_n(final_arr, n_proc, 1.0);

        for (int i = 0; i < (n_proc); i++){
            final_arr[i]++;
        }
        //gather into final arr from all processes
        MPI_Gather(final_arr, n_proc, MPI_DOUBLE, i
                  final_arr, n_proc, MPI_DOUBLE, 0, MPI_COMM_WORLD);

        delete final_arr;
    }

    //closing MPI environment
    MPI_Finalize();
```

As can be seen, utilising the MPI standard is more complex than utilising the parallelism offered by OpenMP, which does not handle distributed systems. First, the MPI environment is initialised, then the number of processes and the id of the current process are fetched. The number of elements to process is calculated (with any rest ignored here). Then parallelism comes into play: The master progress generates the final array and computes its share before gathering all the data from the other processes, which were only working on a small part of the overall workload.

The MPI standard can be used on a single node, but it includes communication overhead. It is most often used in a cluster context. The communication over the network introduces additional latency to the memory transaction which for high-performance InfiniBand networks is around 1 μs, so one magnitude larger than access to RAM. It is therefore important for the performance of a program that communication is kept to a minimum and ideally bundled in as few communication calls as possible.

6.1.5.3 Accelerator-based Parallelism

The last form of parallel computing discussed in this thesis is the use of graphic accelerators. A graphic processing unit (GPU), originally called graphic accelerator, is a separate device which is connected to the PC by a high throughput adapter. While the naming scheme suggests it, GPUs have less in common with standard CPUs and more with wide arrays of SIMD units. They developed as a specialised device to perform the vector multiplications necessary for computer graphics. In the last three decades, the amount of non-graphics software run on a GPU rather than on a CPU has increased dramatically, as more easy-to-use development tools became available. While their development was, and still is, driven by computer graphics applications, mainly the consumer gaming market, they find heavy use in

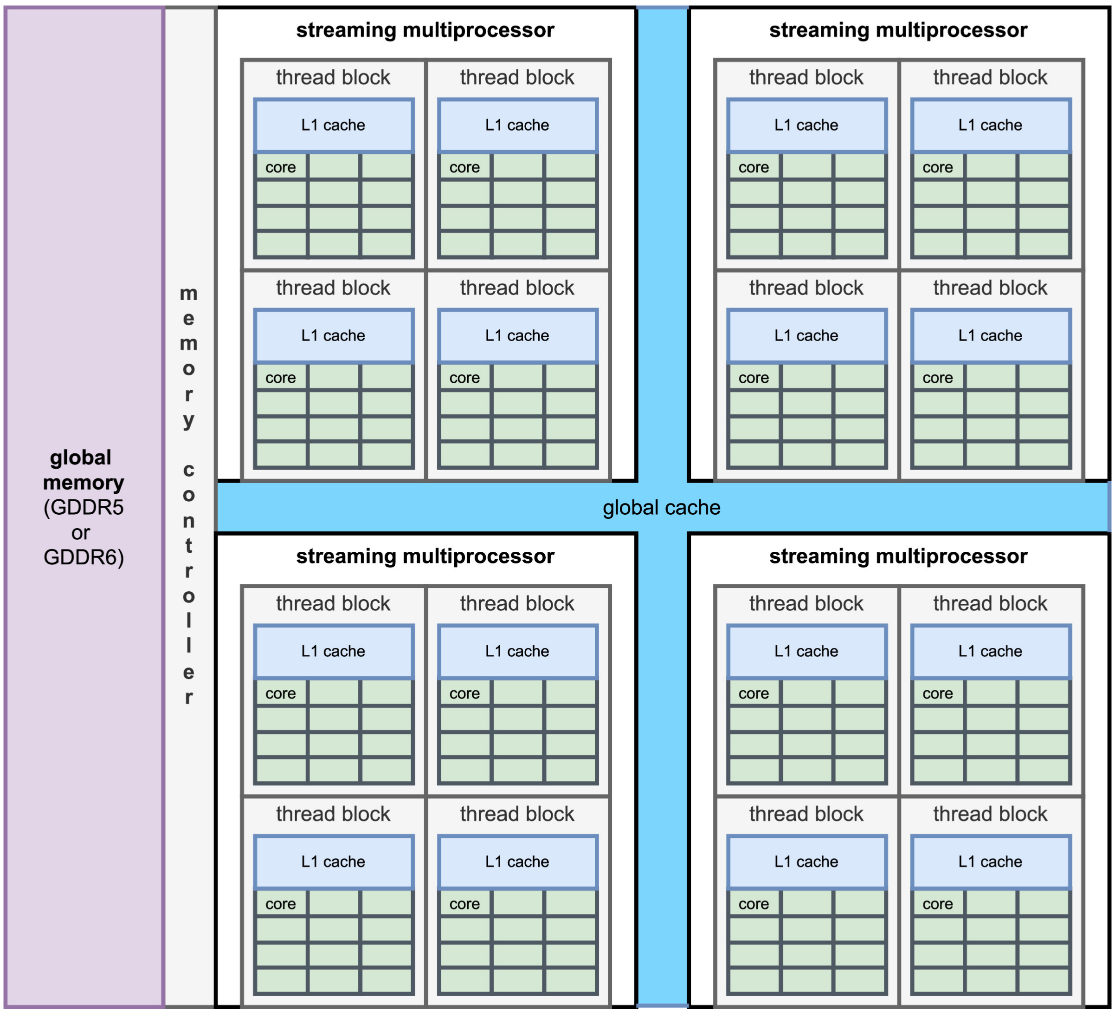


FIGURE 6.8: General architecture of modern graphics processing units.

scientific software where vector operations have to be carried out over a large set of data. They are optimized for data throughput rather than complicated execution paths.

The architecture of a modern GPU is summarized in figure 6.8.

The terminology for the GPU architecture is manufacturer-dependent. The following text will use the terminology generally associated with the CUDA environment of Nvidia's graphics card lineup.

The lowest level of parallelism is called a (CUDA) thread. Each thread executes on a logical ALU (arithmetic logic unit) for SIMD instructions. Multiple threads can be subscribed to the same units, to hide memory latency in case of stalling. The threads are bundled into thread blocks, which have their own instruction cache and can share local memory. This is only the case if the threat blocks are executed on the same streaming multiprocessor, which houses a certain number of CUDA cores. These streaming multiprocessors are the main execution unit of thread-level parallelism in graphics cards. There can be between 2 and 80 streaming multiprocessors on a modern card, with one or multiple thread blocks of up to 512 units. The closest comparison to a CPU core is such a streaming multiprocessor. One of the differences, however, is that the memory hierarchy is much more shallow. There are generally only two memory levels: $L1$ levels for the thread blocks on one streaming processor and a global memory $L2$. However, to reiterate, it is usually not possible to share memory natively between two streaming multiprocessors.

GPUs, therefore, offer two levels of data parallelism: Massive thread-level parallelism and SIMD vectorization. They are extremely useful for performing the same instructions on massive data sets. Thread scheduling and resource management is done at the hardware level and the developer has no access to it.

There are two drawbacks to utilising GPUs. Number one, the data movement from the host (the CPU) to the device (the GPU) has to be managed manually. Additionally, complex execution pathways cannot be handled well on GPUs. There is no sophisticated branch prediction and full threat blocks can stall if one thread has to handle a conditional statement.

The reason why GPUs have seen much more use outside the computer graphics community is the development of high-level APIs and easy-to-use extensions for well-known coding languages. The two most well-known ones are OpenCL, which tries to establish a standard which is manifacturer-independent, and Nvidia's CUDA (compute unified device architecture). CUDA is available as an extension to the C/C++ coding language with its own set of compilers and tools.

Coding using CUDA resembles the usage of `MPI` in the sense that much of the underlying parallelism has to be included by the developer. An example is given in listing 6.5.

Listing 6.5: Example use of the `CUDA` language.

```
//CUDA kernel function
__global__ void inc_array(double *arr, size_t N){
    size_t id = threadidx + blockDimx * blockIdx;
    if (id < N) arr[id]++;
}

int main (int argc, char** argv){
    //allocate data on host
    double *d_arr;
    double *arr = new double[512];
    //suporting function filling arr with 1.0
    std::fill_n(arr, 512, 1.0);

    //allocate data on device
    cudaMalloc((void**) &d_arr, sizeof(double) * 512);
    //copy host data to device
    cudaMemcpy(d_arr, arr, sizeof(double) * 512, ↩
   cudaMemcpyHostToDevice);

    //execute CUDA kernel function in parallel
    inc_array<<<2, 256>>>(d_arr, 512);

    //copy device data to host
    cudaMemcpy(arr, d_arr, sizeof(double) * 512, ↩
   cudaMemcpyDeviceToHost);
    //free device memory
    cudaFree(d_arr);

    delete arr;
    return 0;
}
```

The code listing is once again used to iterate 512 numbers. First, the array on the host device, the CPU side, is initialised together with an address space for the device data. With this address space memory on the device is allocated and the memory is copied to the device. Then the workload is executed. This is done by specifying a kernel function, which is the function consisting of the workload per thread. Parallelism has to be introduced manually. This means in this case calculating the array position for the thread. After the kernel, which is marked by the triple chevrons, has been executed the data is copied from the device to the host and the device data is deleted.

As can be seen, this data movement can be quite complex and cumbersome. Much work is invested today into developing generalised high-level APIs, like `OpenMP` is for shared memory parallelism. The new versions of `OpenMP` compliant compilers support its `target` directive.

With this directive, it is possible to utilise `OpenMP` to parallelise workload and offload it to an accelerator device, such as a GPU. The syntax is quite similar to the standard CPU-shared memory model. An example is given in code listing 6.6.

Listing 6.6: Utilising the `OpenMP` target directive to offload the work to the GPU.

```
double *arr = new double[512];
std::fill_n(arr, 512, 1.0);
// generating the thread team and
// dividing the work load
#pragma omp target teams distribute parallel for map(tofrom:arr)
for (int i = 0; i < 512; i++){
    arr[i]++;
}
delete arr;
```

Here all parallelism is automatically generated. The `target` directive tells the compiler to compile for the established target (e.g. ptx code for Nvidia GPUs), the `teams distribute` generates thread blocks and distributed them over the target device and the data movement is handled by the `map` clause.

While this syntax is much simpler, its underlying technology is still quite new at the time of writing and not many compilers support it fully. The author has tried some implementation work using this directive and stumbled into numerous problems and bugs. Also, the setup of the compiler target device is much more involved and not very easily generalised, especially if one wants to target older generation GPUs (such as the Nvidias 900 series).

6.2 Implementation of Optimisations and Parallelism in `libqqc`

The following sections will cover the performance-specific implementations, focusing on the parallelism-enabling techniques used in `libqqc`.

6.2.1 General Optimisations

The energy transformation step has been covered in section 4.3.4 which dicusses the `loader` module of `libqqc`. It relied on an intermediate-based rewriting of the transformation from the atomic orbital to the molecular orbital space, as can be seen in algorithms 2 and 3. The highest computational scaling step was lowered from $\mathcal{O}(PN^2OV)$ to $\mathcal{O}(PNOV)$. The big-O memory scaling did not change as the intermediate is of lower rank as the full atomic orbital integral (which scales with $\mathcal{O}(PN^2)$), only contributing to the prefactor.

The next important optimisations were presented in section 4.3.6.2. Both the scaling of the P grid integral weights in the constructor, outside of the main work function and the pre-scaling outside of the inner Q grid loop of one part of the orbital doublets in the final quadrature reduced the number of floating point operations needed in the main work loop. The trade-off is once again a higher memory scaling, contributing as a prefactor to the big-O complexity to

$$\mathcal{O}(PN^2 + POV) \leq \mathcal{O}(2PN^2) \tag{6.4}$$

As has also been mentioned while discussing algorithm 5 in section 4.3.6.2, the multiplication symmetry of the main work loop over the KPQ pair was exploited to roughly half the number of PQ iterations, lowering the computational complexity from a full $\mathcal{O}(PQOV)$ to $\mathcal{O}(\frac{1}{2}PQOV)$ (neglecting $K \approx 4$).

The last optimisation was the heavy exploitation of spatial locality in the matrix and vector traversion. Nearly all operations are perfectly cache aligned, iterating through the fastest-changing index of the operations. The only exception to this is the forming of the intermediates $t_1 = \sum_i o_i^P I_{ia}^Q$ and $t_2 = \sum_i \Phi_i^Q c_{ia}^P$. The latter matrix is not traversed correctly, as the forming of the intermediates (which lowers the scaling of these operations) involves iteration over the slower index. This could be remedied by saving a transposed copy of both the unscaled c_{ia}^P and the scaled I_{ia}^Q integral matrix, which would double the big-O memory complexity of the algorithm. It was decided to not use this optimisation, as the higher memory scaling was not deemed acceptable.

6.2.2 Adding Shared-Memory Parallelism: `OpenMP`

The shared-memory parallelism was introduced to the `libqqc` library through the use of the `OpenMP` library, which has been introduced in section 6.1.5.1. All matrix operations have been refactored as cache-optimised for-loops which are parallelised through an `OpenMP` compiler directive construct. An example is given in code listing 6.7.

Listing 6.7: Utilising the `OpenMP` compiler directive to introduce shared-memory parallelism to the library main work loop of the `qmp2_energy` module.

```
#pragma omp parallel for reduction(+:energy) schedule(dynamic) ↩
    default(none)\
     shared(moffset, mnpts_to_proc, m1Dnpts, m3Dnpts, mnocc, mnvirt,↩
     mv1Dwts, mmo, mm1Deps_o, mmv, mm1Deps_v, mc_c, mm1Deps_ov)\
     collapse(2)

     for (size_t k = 0; k < m1Dnpts; k++){
         for (size_t p = moffset; p < moffset+mnpts_to_proc; p++){
```

As this loop construct is ultimately trying to reduce the matrix multiplications to a single energy contribution of the PQ point pair one can use the `reduce` directive to spawn local copies of the energy variable which are later contracted. The scheduler was set to dynamic, as the work of a PQ pair is not linear and increases with later P values. This means that `OpenMP` generates a scheduling queue where each thread takes the next loop iterations after completing its chunk. This helped with the workload balancing. The `dynamic` scheduler produces more overhead, however, so static was generally favoured for other linear workload distributions.

At run time the environment variable `OMP_PROC_BIND=spread` was set to avoid hyper-threading subscription with one `OpenMP` thread subscribed to one physical core.

6.2.3 Adding Distributed-Memory Parallelism: `MPI`

Moving from the shared-memory environment of a single node to the full cluster is not trivial. Distributed-memory systems are more complex to develop, as both the network communication and the synchronisation of the workload overall computing nodes will be important to the final performance.

The `libqqc` library uses the message-passing interface, `MPI`, through the `OpenMPI` library to facilitate node-to-node communication. The host program is tasked with setting up the correct environment. If `OpenMP` is used for node-level parallelisation, then the specific setup in code listing 6.8 is necessary to guarantee thread safety.

Listing 6.8: Correct setup for simultanious use of `MPI` and `OpenMP` to guarantee thread safety.

```
#ifdef _OPENMP
    int provided = 0; // check threading
    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
    if (provided < MPI_THREAD_FUNNELED){
        printf("The supported threading is not sufficient.\
               (less than MPI_THREAD_FUNNELED)\n");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }
#else
    MPI_Init(&argc, &argv);
#endif
```

This setup checks if multi-threading `MPI` calls are supported (necessary for correct behaviour if the program was compiled with `OpenMP`).

Two major steps are currently parallelised over the distributed system, which are illustrated in figure 6.9. The first one is the AO-to-MO transformation in the `loader` module. The grid evaluated Coulomb AO integral $I^P_{\mu\nu}$ transformation is parallelised over the grid slices P. For this, the full AO integral is exchanged in chunks to avoid the maximum message length in earlier versions of the standard. Then only those transformations are carried out which are associated with the P slice for this particular node. As all nodes rely on the full integral data set each node then receives the transformed chunks from the other nodes once.

The second, more complex parallelisation step is the parallelisation of the main work function which calculates the Q-MP2 energy. The main loop of this function consists of the iteration through the K grid and the iteration through the P grid and its $Q \leq P$ counterpart. As the workload is triangular over the PQ grid pair and intermediates are precalculated for every iteration through the Q part, only the outer KP pair is accessible for parallelisation. A problem arises however if a straightforward parallelisation is implemented over the KP pair. PQ Chunks with $Q \leq P$ of same size result in different amounts of work depending on the

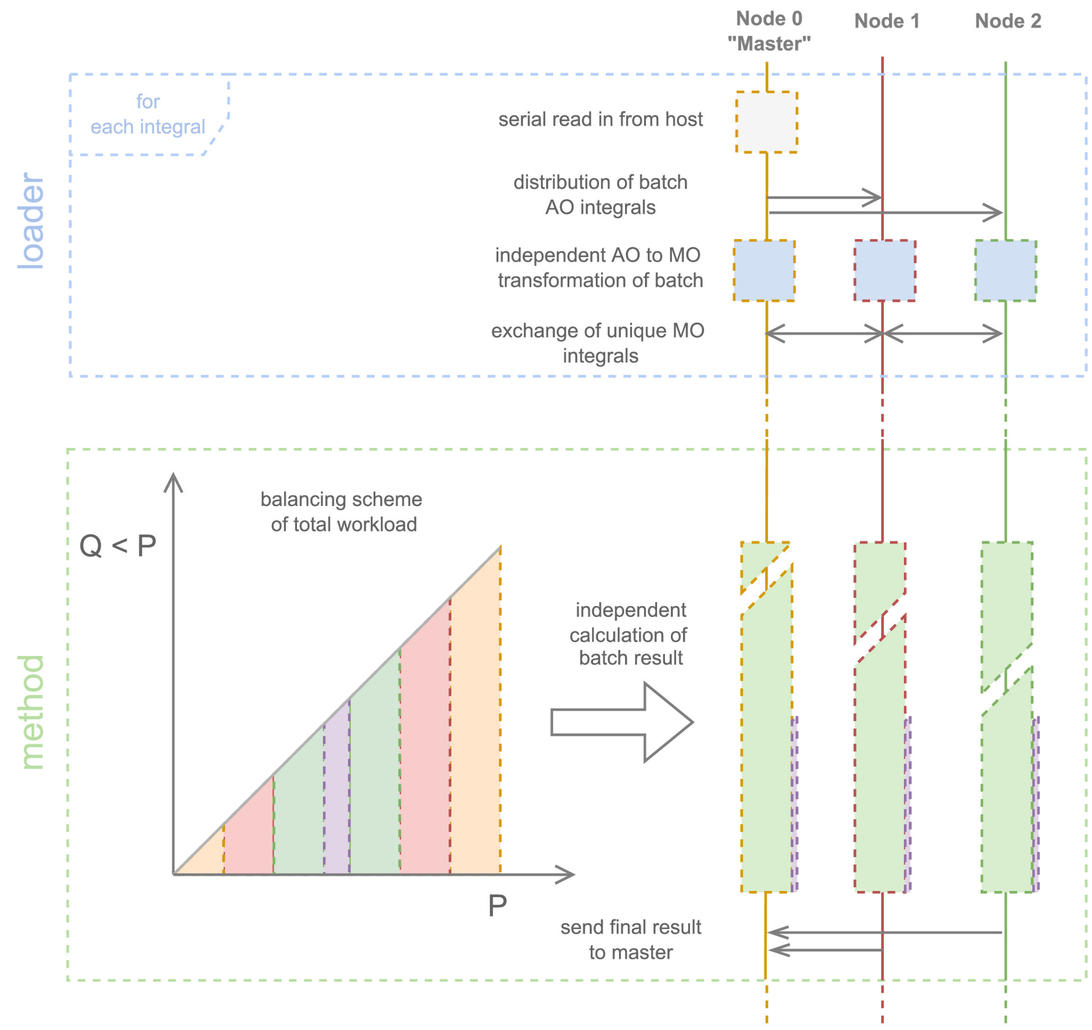


FIGURE 6.9: Overview of the communication and workload distribution of the `MPI` implementation of the `libqqc` library.

position in the P set. This leads to an imbalanced load across the nodes. To avoid this the workload is instead divided up into two times the number of available nodes. Then, each node handles two of these chunks, n and max $- n$. This levels out the workload over the different iterations. This schema is also illustrated in 6.9.

Finally, the `OpenMPI` library is tasked with any exchange of data that is needed on the nodes, e.g. read-in from a file on node A which has to be communicated with node B. These data movements are structured in such a way as to only happen once for each integral in their AO form, with the above-mentioned exception of the Coulomb integral transformation. The only other communication is the trivial exchange of the single double-value energy contribution of the node to the full Q-MP2 energy.

6.2.4 Accelerator Parallelism: OpenMP target

The OpenMP standard supports offloading to accelerator targets, such as the GPU, since its 4.0 release in 2012. However, this support of heterogenous systems continues to evolve and is still in active development and testing.

Seeing as the libqqc library is already fully parallelised through OpenMP the application extension to GPUs was tested. A stand-alone file with the main work loop was created, in which the #pragma omp clause was altered to move the workload to the device, as can be seen in code listing 6.9.

Listing 6.9: Utilisinge the target directive of OpenMP to offload work to the GPU.

```
size_t nthreads = 64;
size_t nteams = npts * tsize/nthreads;
#pragma omp target teams distribute parallel for \
map(tofrom: e_mp2) map(to: v_twts, m_o[0:npts*occ], m_teps_o, m_v↩
    [0:npts*virt], m_teps_v, c_c[0:npts*occ*virt], c_teps) \
num_teams(nteams) thread_limit(nthreads) \
dist_schedule(static, 64) schedule(static, 1) \
reduction(+:e_mp2) collapse(2)
for (int k = 0; k < tsize; k++){
    for (int p = 0; p < npts; p++){
```

The implementation of the standard only supported utilising one GPU through this directive. This was done by calling #pragma omp target. To utilise all streaming multiprocessors the threads have to be bundled into teams distributed over the whole device (teams distribute). Memory movement from and to the device is specified by the map(to:) directive. The number of teams and the limit of threads per team is specified. These map to CUDA's above mentioned grid notation within the triple chevrons. Finally, a scheduler is set for the distribution of the thread teams.

At the time of testing this implementation was then compiled with gcc 9.3.0, which did not fully support the target feature set, on NVidia GPUs. The performance was deemed to be unsatisfactory at the time, giving about a 3 times speedup with a GTX 960 over one 3.7 GHz CPU core. Further testing with the implementation of the newer OpenMP standard is planned.

6.3 Performance Benchmarking

The following chapter will present the current implementation's single-node and multi-node performance.

6.3.1 Computational Details

Four molecular systems of different sizes were chosen for the benchmark. All geometries were optimised using MP2/6-31g level of theory and these geometries were then used for the input quantities. The input quantities (grid evaluated AO integrals, GTOs and atomic orbital energies) were provided by `Q-Chem`[88] 5.2.2 using the 20/38 Becke-weighted grid of the `libdftn` library and exported to disk. The test systems consist of water, methanol, anthracene and porphyrin.

A standalone version of the `libqqc` library was compiled with a minimal host program to minimise any host program-specific overhead. The input quantities for each test system and the minimal host program are available through the `libqqc`[93] GitHub for verification.

The calculations were run on the JUSTUS 2 bwGrid computation cluster with the hardware listed in chapter 4. They were run in the normal cluster environment and not on isolated nodes.

6.3.2 Node-Level Performance

As previously mentioned there are four total variants for the library execution: The main algorithm is expressed either in low-level `C`-style arrays or through the container methods of the `Eigen` library. Furthermore, both variants are then parallelised either through `MPI` tasks alone or through `OpenMP` threads at node-level with `MPI` tasks solely used for communication between the nodes.

A summary of the timings for the different test systems can be found in tables A.8, A.7, A.6 and A.5. Their speedup is shown in figures 6.10 and 6.11.

The absolute timings show the current performance problem of the Q-MP2 method: The PQ prefactor results in a runtime magnitudes slower than that for standard MP2 implementations. This is especially apparent as $PQ >> N^2$ for very small

	variant	nodes	cores	GFLOPs	Perf%
`MPI+OpenMP`	array	1	1	11,33	14,75%
		1	48	629,04	17,06%
		20	48	11523,76	15,63%
	Eigen	1	1	10,76	14,01%
		1	48	520,77	14,13%
		20	48	10269,22	13,93%
`MPI` only	array	1	1	11,33	14,75%
		1	48	373,75	10,14%
		20	48	7378,27	10,01%
	Eigen	1	1	10,76	14,01%
		1	48	351,13	9,52%
		20	48	6927,95	9,40%

Table 6.1: Example GFLOPs performance of a Q-MP2 porphyrin calculation compared to the theoretical performance of the equivalent number of cores of the Intel® Xeon® Gold processor 6252 with advertised 2.1 GHz frequency.

systems, meaning that the overall scaling (e.g. $\mathcal{O}(PQOV)$) is worse than that of the standard MP2 method.

This problem will have to be remedied in the future by grid pruning and integral screening strategies to reduce the effective PQ prefactor. As mentioned in chapter 5, these investigations are already underway.

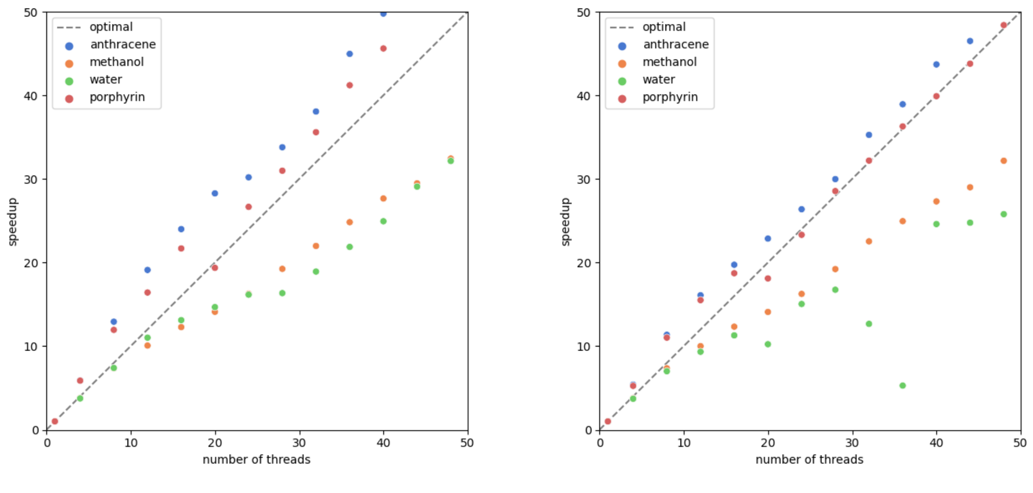
A different problem arises if one compares the absolute performance to the theoretical peak performance of the node, which is listed for one example in table 6.1. Only up to $< 17\%$ of theoretical peak performance is achieved. As a note of caution, theoretical peak performance is not an easily summarised number, as the frequency is lowered and raised dynamically. Still, the general rule of thumb is that $\geq 60\%$ of theoretical peak performance is considered good.

This means that the computation is most likely memory-bound, meaning that the program has to wait a considerable fraction of the running time for memory transactions. Seeing as the spatial locality is already exploited heavily, strategies for an increased temporal locality may increase performance. This could be done in later iterations of the library through blocking strategies of the main work loop.

Comparing the peak performance for the different variants shows that the implementation with low-level C-style arrays and the `MPI+OpenMP` libraries for parallelisation is the most performant.

The saving grace of the implementation is its parallel capabilities. Figures 6.10 and 6.11 show the speedup over the number of cores for the different test systems.

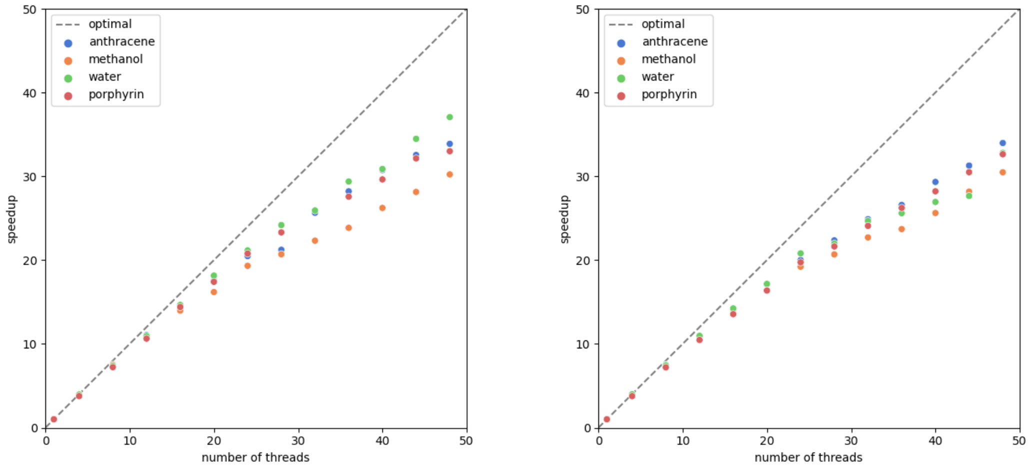
Two main observations can be made: Utilising only the `OpenMPI` library for parallelisation performs worse than having the `OpenMP` library for node-level parallelisation and the `MPI` standard for node-to-node communication. For the first case, all test systems show approximately the same parallel scaling, with significant performance loss after about 12 cores. The mixed variant performs much better. Here, two clear fields can be identified: The smaller systems scale sublinearly but without showing asymptotic slow down at around $\approx 70\%$ speedup. The computational work for the smallest test system, water, is easier affected by artefacts due to the workload of other running processes on the node, which explains the misbehaviour around 30 to 40 cores. The larger systems however show either near-perfect or even superlinear behaviour for this variant. This means that the speedup exceeds 100% of the theoretical best speedup. This can be explained by cache effects. As the chunks of the KP loop get smaller less data movement has to be carried out per core, with more reused data in higher levels of cache, which also leads to faster memory transactions.



(A) using C-style arrays (B) using the `Eigen` implementation

FIGURE 6.10: Speedup of the hybrid `MPI` + `OpenMP` variant of the algorithm.

Despite the bad computational efficiency of the implementation the high degree of parallel efficiency with good exploitation of cache effects show the real advantage of the method.



(A) using C-style arrays (B) using the `Eigen` library

FIGURE 6.11: Speedup of the `MPI` only variant of the algorithm.

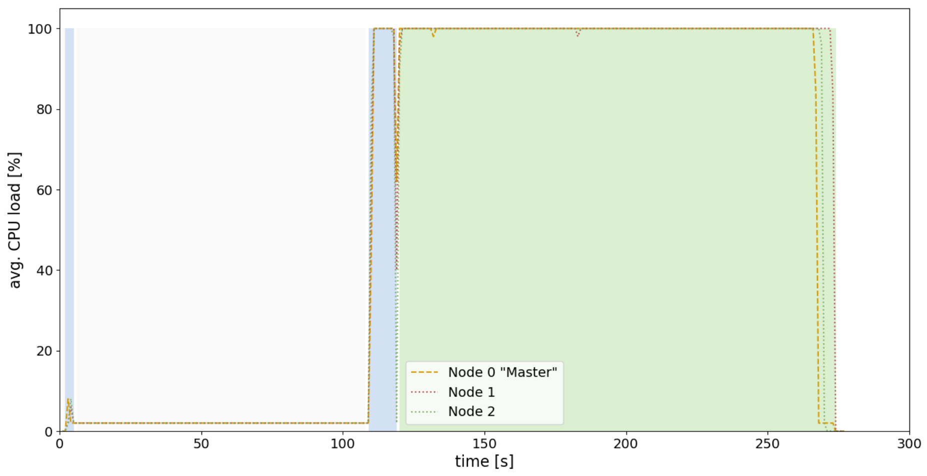


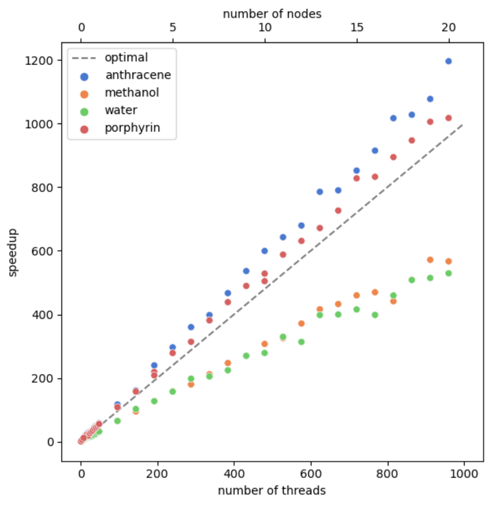
FIGURE 6.12: Workload balancing example for the anthracene system on three nodes.

6.3.3 Multi-Node-Level Performance

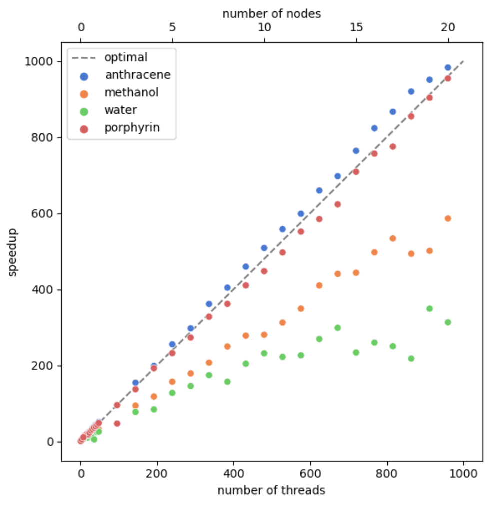
Utilising the `MPI` standard for intra-node communication allows for the use of large distributed cluster systems. Important for the performance of these implementations is the workload balancing across the distributed system. Through the above-mentioned partitioning scheme, a balanced load was achieved. An example of the timings is given in figure 6.12.

After the serial read-in from the disk (grey area), the integrals are exchanged (blue area), and the main work loop is started (green area). The recorded average CPU load for all nodes is consistently at ≈ 100%. The minor difference at the end of the workload is easily explained as the minor load imbalance from different sections of the P space and contributing waiting times for the exchange of the final energy contribution. The workload balancing is therefore deemed satisfactory.

The speedups for up to 20 nodes with 960 physical cores are summarised in figures 6.13 and 6.14.

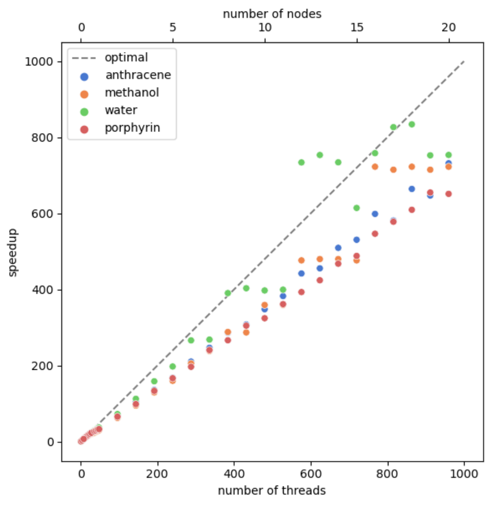


(A) using C-style arrays

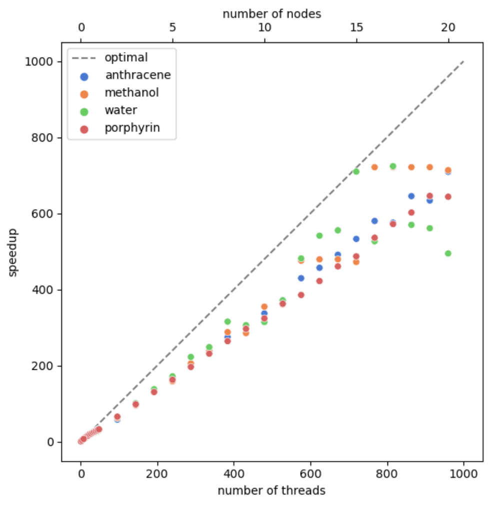


(B) using the `Eigen` implementation

Figure 6.13: Speedup of the hybrid `MPI` + `OpenMP` variant of the algorithm.



(A) using C-style arrays



(B) using the `Eigen` library

Figure 6.14: Speedup of the `MPI` only variant of the algorithm.

The trends of the single-node level hold on the multi-node scale. The parallelisation over only the `MPI` tasks performs more consistently but worse than the mixed variant. The latter split once again into two fields corresponding with the size of the system. The smaller water and methanol systems scale sublinearly, whereas the C-style arrays scale super linearly for the array implementation and linearly for the `Eigen` implementation.

This shows the full potential of the method in the mixed parallelisation scheme implementation. For the full 960 physical cores speedups exceeding 1000 are reached for medium-sized and larger systems.

Chapter 7

Further Work and Outlook

The following chapter will showcase some of the collaborative projects that were undertaken for the Q-MP2 method and the `libqqc` library and provids an outlook into possible future projects. It is separated into a section about optimising the performance of the `libqqc` library and a section about the application of the quadrature scheme to new methods.

7.1 Increasing Performance

As has been mentioned in chapter 6 the single- and multi-node performance is bottlenecked by the large P^2 prefactor of the method, which is memory-bound. To decrease the dimensionality of the P grid and reduce memory-heavy operations some further optimisations can be applied.

7.1.1 Investigation into Optimised Grids

The spatial quadrature grid has the most influence over the accuracy and computational cost of the Q-MP2 method. Optimising this grid is therefore the most important step to further optimise the performance of the method. Investigations into optimised grids were started during the master project of Isabel Vinterbladh[5], which the author supervised.

The project was focused on further implementing grid functionalities in the respective module as reported in chapter 4. The infrastructure for one-, two- and

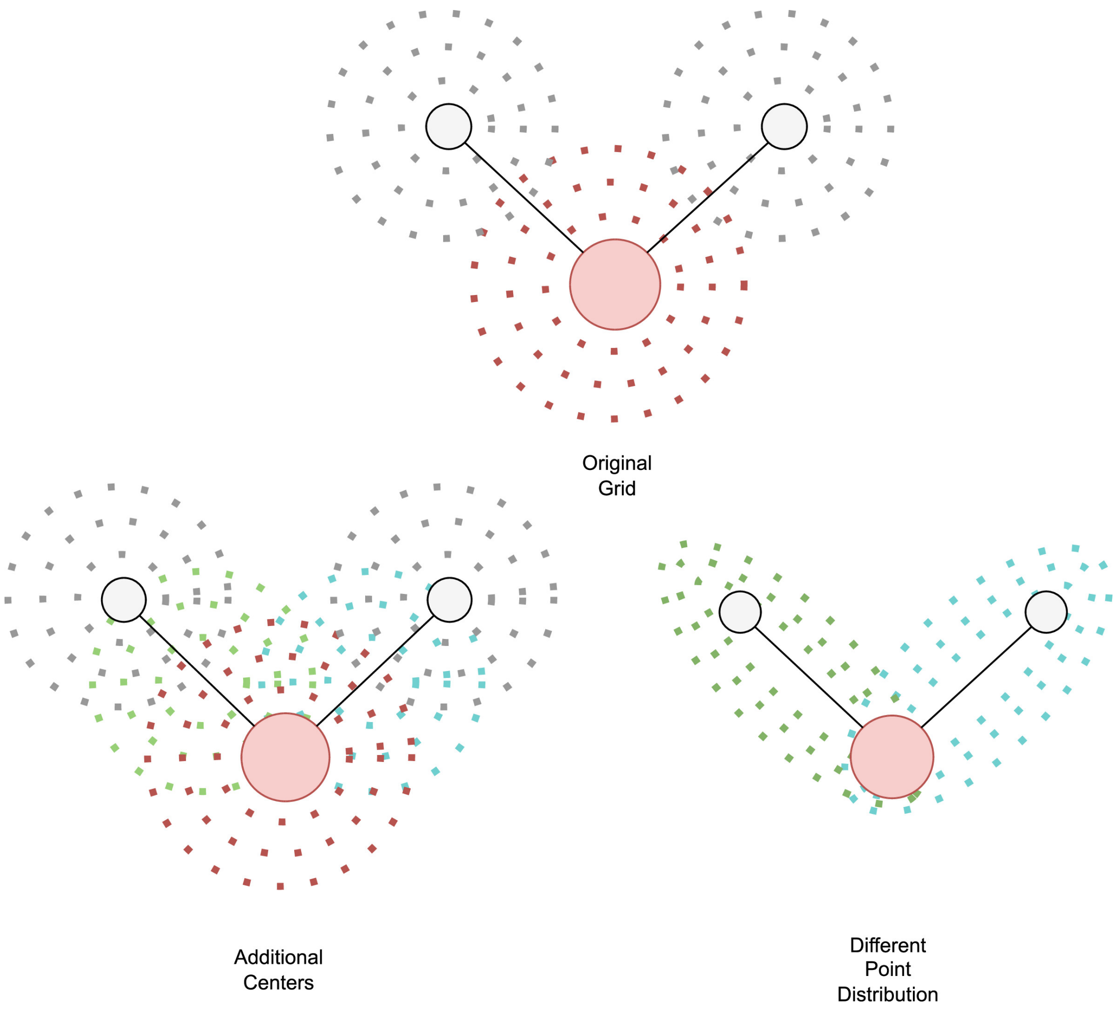


FIGURE 7.1: Different Strategies implemented and tested during the project.

three-dimensional grids was implemented as well as the weighting by Becke[106] and its respective atomic-centred scheme. Additionally, two different grid setups were investigated. For the first one, the grid density was increased by placing pseudo-centres on chemically important features, such as molecular bonds. The second setup changed the grid distribution from a spherical to an ellipsoidal one and centred it on the bonds to approximate the shape of the molecular system. The three molecular grids that were implemented are illustrated on the 2D water case in figure 7.1. Additionally, optimisations in terms of grid cutoffs, screening and scaling (in accordance with Becke's implementation) were implemented and different settings were tested.

The cutoffs tested did not improve the accuracy of the quadrature, the different scaling values however did have an impact on the accuracy and could improve it for the system in question. The *PQ* screening, which identifies relevant pairs by spatial proximity, also resulted in an improvement. However, due to the limited

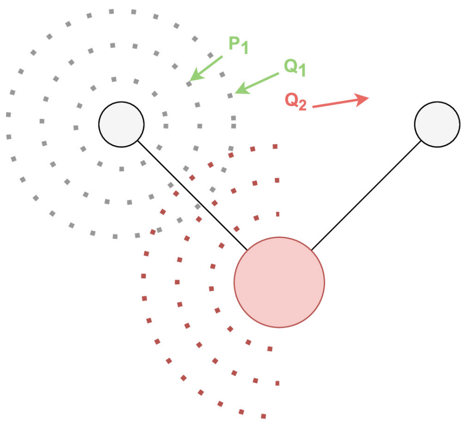


FIGURE 7.2: Exploiting the spatial symmetry of systems to lower the PQ dimensionality.

time of the project, some bugs remained, as convergence to the parent MP2 method was not achieved for larger numbers of quadrature points.

The implementations will be moved to the main branch after further optimisation work and then tested on the benchmark set.

7.1.2 Exploiting Spatial Symmetry

Another way to lower the dimensionality of the P grid is to exploit its spatial symmetry. The atomic-centred setup of the Becke-weighted integration grid allows for more exploitation as the standard molecular symmetry, as different atom centres produce the same grid. If a system is set up correctly, the resulting atomic grids should have the same orientation for each centre. This should allow for the sampling of only a part of the overall space. As an example consider the 2D case for the C_{2v} symmetric 3D H_2O system illustrated in figure 7.2.

The mirror axis results in only half of the point being unique (and only a quarter for the 3D case). The sampling of the full PQ super grid however needs additional support infrastructure. In the figure, both Q_1 and Q_2 are symmetrically identical. Therefore to sample both P_1Q_1 and P_1Q_2 one would have to identify the symmetric equivalent of the virtual Q_2 point, Q_1, and double count the contribution.

The full restructuring of the grid code would then involve:

1. Identifying the symmetry of the grid (and not only the molecular symmetry)

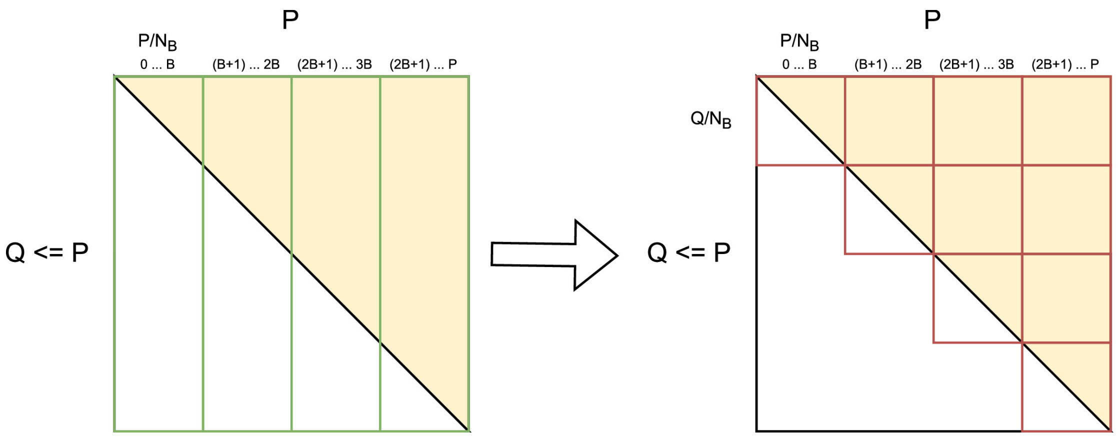


FIGURE 7.3: Example strategy to increase temporal locality in the PQ point pair supermatrix.

Matrix	Memory Scaling at P/Q
ϕ_o^P	$\mathcal{O}(O)$
ϕ_o^Q	$\mathcal{O}(O)$
ϕ_v^P	$\mathcal{O}(V)$
ϕ_V^Q	$\mathcal{O}(V)$
I_{ov}^P	$\mathcal{O}(OV)$
I_{ov}^Q	$\mathcal{O}(OV)$
Total Memory	$\mathcal{O}(2OV + 2O + 2V)$

TABLE 7.1: List of necessary quantities for a PQ pair calculation and their memory contributions.

2. Masking the reducible PQ sampling space
3. Identifying the symmetry-dictated factor for each symmetry-reduced PQ sampling pair and multiplying the contribution with it

7.1.3 Exploiting Temporal Locality

The blocking strategy discussed in section 6.2.3 already exploits the temporal locality of the P quantities to some extent. However, the temporal locality could be improved by the introduction of a further blocking dimension of the PQ pair matrix, as illustrated in figure 7.3.

By forming smaller blocks of the P and Q grid space the relevant quantities can be kept in lower cache levels if the orbital space permits. For a full PQ pair the following matrices are needed:

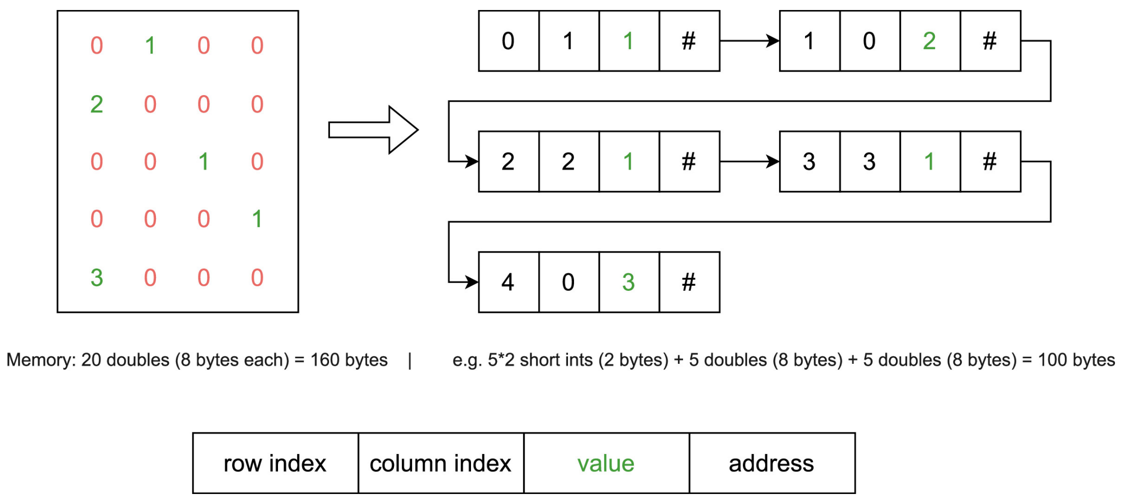

0 1 0 0
2 0 0 0
0 0 1 0
0 0 0 1
3 0 0 0
0 1 1 #
1 0 2 #
2 2 1 #
3 3 1 #
4 0 3 #
Memory: 20 doubles (8 bytes each) = 160 bytes |
e.g. 5*2 short ints (2 bytes) + 5 doubles (8 bytes) + 5 doubles (8 bytes) = 100 bytes
row index
column index
value
address

FIGURE 7.4: Linked list representation example and memory calculation of a sparse matrix.

As long as the total memory for the PQ pair quantities, which is around $(2OV + 2O + 2V) \times 8$ byte, is under the memory limit of a Lx cache a blocking scheme as illustrated in the above figure would lead to a performance increase and allow for higher exploitation of the peak computational performance of the hardware.

For this, the data movement, especially in the distributed memory case, has to be rewritten. This will result in more data transactions. However, if the blocks per execution units can be determined beforehand this additional data movement could be simplified into one transaction with a more fractured memory structure. This would result in a performance hit due to worse spatial locality. If this trade-off results in performance improvement remains to be determined.

7.1.4 Exploiting Matrix Sparsity

Depending on the density of the spatial grid, the spatial grid cutoff and the system in question the resulting matrix quantities can exhibit a high degree of sparsity, meaning a large number of elements are close to 0. The field of linear algebra allows for strategies to approximate sparse matrices with smaller entities and decrease the number of computation operations in matrix-matrix and matrix-vector products.

There are different strategies to save space while storing sparse matrices. These range from saving subblocks of matrix entries which exhibit a lower degree of sparsity to only storing the non-zero elements of the matrix as a tuple in either an array, dictionary or linked list, such as illustrated in figure 7.4.

There are different advantages and disadvantages depending on the chosen representation. Some allow for faster single access of elements but slower sequential element access, others trade element access time for a higher penalty when modifying the matrix.

There are also direct and iterative procedures to efficiently work on sparse matrix-vector products, such as the generalised minimal residual method.

These solvers and different matrix representations are implemented in a wide array of available linear algebra libraries. Depending on the setup and system in question implementing them could speed up the Q-MP2 calculation and allow for larger systems.

7.1.5 Integral Screening

The original Q-MP2-OS paper by Barca et al.[3] heavily relies on integral screening to improve the absolute performance of the algorithm. Three kinds of screening were employed:

PQ-grid screening: The formulation of the Q-MP2-OS algorithm in terms of first-order density matrices allowed for the exploitation of the nearsightedness of these quantities, i.e. their exponential decay in space. PQ pairs are ordered by their distance, and the energy contribution of the nearest pair is calculated. This goes on until $r^{PQ} \leq r^{PQ}_{\text{crit}}$. The critical range was defined as the range of the PQ pair plus the logarithmic user-defined ratio of the energy contribution of this pair, divided by the exponential decay parameter of the density. The latter factor is unknown and had to be manually approximated. This screening lowered the number of relevant PQ pairs from quadratic to linear.

K-grid screening: The number of K grid points was determined by a cutoff strategy, resulting in three to four relevant grid points for the tested systems.

$\mu\nu$ shell-pair screening: The Q-MP2-OS algorithm is expressed in terms of atomic orbital quantities. Direct screening of $\mu\nu$ shell pairs was therefore possible. For this, a cutoff threshold using the Gaussian product prefactors of the basis function was used to decrease the number of relevant shell pairs from quadratic to linear. Furthermore, these relevant shell pairs were additionally screened for their relevancy to the grid point P, reducing the number of overall pairs to $\mathcal{O}(1)$.

$\alpha\nu$ **shell-pair screening:** The final screening was done on the relevant $\alpha\nu$ shell pairs of the $S_{\alpha\nu}^{KP}$ quantities, where the previously discussed exponential decay strategy was used together with traditional shell-pair screening.

As of now, no screening steps are implemented for the Q-MP2 algorithm in `libqqc`. Reducing the number of relevant PQ points would lead to a noticeable speedup of the algorithm. Direct shell-pair screening of AO quantities is not possible, as the algorithm is formulated in terms of molecular orbital contributions. Additionally, the PQ strategy of ordering the quantities in terms of their distance would lead to less sequential memory traversion, lowering the spatial locality of the algorithm. As this algorithm is mostly memory-bound, this step could result in worse performance, especially for smaller and medium-sized systems where one PQ pair matrices do not exhaust the lower-level caches.

7.2 Application to New Methods

The Q-MP2 implementation in the `libqqc` library has been used as a guinea pig for that specific quadrature reformulation. The next step would be to apply this approach to other methods. Especially high-scaling methods would benefit from the lower computational and memory scaling that the quadrature method has to offer.

One project that has already started is the extension of the reformulation to the MP2 equivalent coupled cluster second-order approximation CC2.

7.2.1 Coupled Cluster Method Family

Post-HF methods aim to recover the electron correlation. This is generally done by extending the mathematical space used to describe the system by the introduction of excited determinants, i.e. determinants in which electrons have been promoted from occupied into virtual orbitals. How these excited determinants are included in the description is dependent on the specific method family. For example, in CI a set of excitation operators $\hat{T}$ is used which acts on the ground state wave function to generate the set of excited determinants.

$$\hat{T} = \hat{T}_1 + \hat{T}_2 + \hat{T}_3 + \dots \tag{7.1}$$

Where $\hat{T}_n$ acting upon the ground state determinant generates the nth excited determinant.
Within the coupled-cluster formalism, this operator is not employed directly but enters the wave function expansion using the exponential ansatz

$$|\Phi_{CC}\rangle = e^{\hat{T}} |\Phi_0\rangle \tag{7.2}$$

This transformed wave function can then be used in the Schrödinger Equation

$$\hat{H} e^{\hat{T}} |\Phi_0\rangle = E e^{\hat{T}} |\Phi_0\rangle \tag{7.3}$$

Two problems arise. The power series of the $e^{\hat{T}}$ operator includes all possible excitation of the system following eq. 7.1, where $\hat{T}_1$ acting on the ground state wave function generates the single excited determinant Φ_i^a, etc. In the second quantisation formalism, the $\hat{T}$ operators consist of a string of annihilation and creation operators

$$\hat{T}_1 = \sum_{ia} a_a^\dagger a_i \tag{7.4}$$

$$\hat{T}_2 = \sum_{ijab} a_a^\dagger a_b^\dagger a_j a_i \tag{7.5}$$

Therein, the annihilation operator removes one electron and its corresponding orbital from the system

$$a_p |\Phi_0\rangle = a_p |\phi_p \phi_q \dots \phi_s\rangle = |\phi_q \dots \phi_s\rangle = |\Phi_i^a\rangle \tag{7.6}$$

and the creation orbital adds one electron to a new orbital in the system

$$a_t^\dagger |\Phi_0\rangle = a_t^\dagger |\phi_p \phi_q \dots \phi_s\rangle = a_t^\dagger |\phi_t \phi_p \phi_q \dots \phi_s\rangle = |\Phi_{ij}^{ab}\rangle \tag{7.7}$$

The full derivation using the second quantisation formalism is outside the scope of this work and can be found in the standard literature[115]. One frequently approximation is the truncation of the full $\hat{T}$ operator to only single and double excitations, which is referred to as Coupled-Cluster Singles Doubles (CCSD)[116].

$$\hat{T} = \hat{T}_1 + \hat{T}_2 \tag{7.8}$$

The CCSD wave function generating operator can then be approximated by the power series

$$e^{\hat{T}_1+\hat{T}_2} = 1 + (\hat{T}_1 + \hat{T}_2) + \frac{1}{2}(\hat{T}_1 + \hat{T}_2)^2 + \frac{1}{6}(\hat{T}_1 + \hat{T}_2)^3 + \dots \tag{7.9}$$

$$= 1 + \hat{T}_1 + \frac{1}{2}\hat{T}_1^2 + \hat{T}_2 + \hat{T}_1\hat{T}_2 + \frac{1}{6}\hat{T}_2^2 + \dots \tag{7.10}$$

This series goes on until all excitation for the the N-electron limit have been considered. However, as the reference state Φ_0 is chosen to be a single Slater determinant one can employ Slater's rules, which state that matrix elements of two-particle operators between determinants that differ by more than two orbitals vanish. Projecting the reference wave function on the resulting Schrödinger equation

$$\langle\Phi_0|\hat{H}|\Phi_{CC}\rangle = E\,\langle\Phi_0|\Phi_{CC}\rangle \tag{7.11}$$

leads to an expression for the energy. At most terms up to $\hat{T}^2$ survive. Employing the second quantisation formalism together with Wick's theorem and Slater's rules one arrives at the CCSD energy expression

$$E_{CCSD} = \frac{1}{2}\sum_{ijab} \langle\Phi_0|\hat{H}|\Phi_{ij}^{ab}\rangle\,(t_{ij}^{ab} + t_i^a t_j^b - t_i^b t_j^a) \tag{7.12}$$

That expression already bears structural similarities with the MP2 energy expression, mainly differing in the dependence on the amplitude expression t_i^a and t_{ij}^{ab}. Similarly, projecting an excited state determinant on the equation in this manner leads to an expression for the corresponding amplitude equations.

$$\langle\Phi_i^a|\hat{H}|\Phi_{CC}\rangle = E\,\langle\Phi_i^a|e^{\hat{T}}|\Phi_0\rangle \tag{7.13}$$

$$\langle\Phi_{ij}^{ab}|\hat{H}|\Phi_{CC}\rangle = E\,\langle\Phi_{ij}^{ab}|e^{\hat{T}}|\Phi_0\rangle \tag{7.14}$$

The resulting expressions are non-linear and coupled to the energy. To remedy this the formalism introduces the similarity transformed Hamiltonian by multiplying the inverse of the cluster exponential from the left. The equations then become

$$\langle\Phi_0|e^{-\hat{T}}\hat{H}e^{\hat{T}}|\Phi_0\rangle = E \tag{7.15}$$

$$\langle\Phi_i^a|e^{-\hat{T}}\hat{H}e^{\hat{T}}|\Phi_0\rangle = 0 \tag{7.16}$$

$$\langle\Phi_{ij}^{ab}|e^{-\hat{T}}\hat{H}e^{\hat{T}}|\Phi_0\rangle = 0 \tag{7.17}$$

The amplitude equations are now decoupled from the energy expression. Using the Baker-Campbell-Hausdorff formula[117] one can find a natural termination of the resulting power series for the similarity transformed Hamiltonian $e^{-\hat{T}}\hat{H}e^{\hat{T}}$ which is then used with the second quantisation formalism to arrive at the working equations.

7.2.1.1 Approximated second-order Coupled Cluster: CC2

The working equation expression of the CCSD energy is dependent on the t amplitude equations

$$E_{CCSD} = \sum_{ijab} [2\,\langle ij|ab\rangle - \langle ij|ba\rangle]\,(t_{ij}^{ab} + t_i^a t_j^b) \tag{7.18}$$

However, the single and double amplitude expressions are a set of coupled non-linear expressions. Therefore an iterative procedure has to be employed, where one first constructs t_i^a through a guess, then uses this to construct t_{ij}^{ab}, which is then used to construct a new t_i^a and so on. To remedy this fact one additional approximation can be done. The contributions of the t_{ij}^{ab} terms can be neglected when calculating t_{ij}^{ab}, so that only t_i^a have to be solved iteratively.

This so-called CC2 approximation to the second-order Coupled Cluster energy (CC2) performs comparatively to MP2[118,119] in terms of accuracy and computational cost. Its amplitude equations can be expressed as

$$\begin{aligned}
t_i^a = (&[2\langle ik|ac\rangle - \langle ia|kc\rangle]\, t_k^c + [2\langle ak|cd\rangle - \langle ak|dc\rangle]\, t_i^c t_k^d \\
&- [2\langle il|kc\rangle - \langle ik|lc\rangle]\, t_k^a t_l^c - [2\langle kl|cd\rangle - \langle kl|dc\rangle]\, t_i^c t_k^a t_l^d \\
&- [2\langle ak|cd\rangle - \langle ak|dc\rangle]\, t_{ik}^{cd} - [2\langle il|kc\rangle - \langle ik|lc\rangle]\, t_{kl}^{ac} \\
&+ [2\langle kl|cd\rangle - \langle kl|dc\rangle]\, (2t_{ik}^{ac} - t_{ik}^{ca}) t_l^d \\
&- [2\langle kl|cd\rangle - \langle kl|dc\rangle]\, (t_{kl}^{ac} t_i^c + t_{il}^{cd} t_k^a))\frac{1}{\varepsilon_a - \varepsilon_i}
\end{aligned} \tag{7.19}$$

$$\begin{aligned}
t_{ij}^{ab} = (&\langle ij|ab\rangle + \langle aj|cb\rangle\, t_i^c + \langle ib|ac\rangle\, t_j^c + \langle ij|kb\rangle\, t_k^a - \langle ij|ak\rangle\, t_k^b \\
&+ \langle ab|cd\rangle\, t_i^c t_j^d + \langle ij|kl\rangle\, t_k^a t_l^b - \langle ib|kc\rangle\, t_k^a t_j^c - \langle aj|ck\rangle\, t_i^c t_k^b - \langle ik|ac\rangle\, t_k^b t_j^c \\
&- \langle kj|cb\rangle\, t_k^a t_i^c \\
&+ \langle ic|kl\rangle\, t_k^a t_l^b t_j^c + \langle kj|cl\rangle\, t_k^a t_l^b t_i^c - \langle ak|cd\rangle\, t_i^c t_j^d t_k^b - \langle kb|cd\rangle\, t_i^c t_j^d t_k^a \\
&+ \langle kl|cd\rangle\, t_i^c t_j^d t_k^a t_l^b) \\
&\frac{1}{\varepsilon_a + \varepsilon_b - \varepsilon_i - \varepsilon_j}
\end{aligned} \tag{7.20}$$

where the Einstein summation convention has been implied.

7.2.1.2 Applying the Quadrature: Q-CC2

Because of the similarities with the MP2 expression and the similar performance CC2 was chosen as the first application of the quadrature formalism to a new method. This project was conducted under the supervision of the author by Theofilos Dimitrakopoulos[120] in the course of his final year theaching thesis. Applying the quadrature formalism results in the following working equations of the Q-CC2 method

$$E_{Q-CC2} = \sum_P \omega^P \left[\sum_i \phi_i^P \sum_{jab} \left[\left(2\phi_a^P I_{jb}^P - \phi_b^P I_{ja}^P\right) \left(t_{ij}^{ab} + t_i^a t_j^b\right) \right] \right] \tag{7.21}$$

$$\begin{aligned}
t_i^a = \frac{1}{\varepsilon_a - \varepsilon_i} \sum_P \omega^P (\\
\phi_i^P \left(\sum_{kc} \left[\left(2\phi_a^P I_{kc}^P - \phi_k^P I_{ac}^P\right) t_k^c \right] - \sum_{lck} \left[\left(2\phi_c^P I_{kd}^P - \phi_d^P I_{kc}^P\right) \left(t_k^a t_l^c + t_{kl}^{ac}\right) \right] \right) \\
+ \phi_a^P \sum_{kcd} \left[\left(2\phi_c^P I_{kd}^P - \phi_d^P I_{kc}^P\right) \left(t_i^c t_k^d + t_{ik}^{cd}\right) \right] \\
- \sum_k \left[\phi_k^P \sum_{lcd} \left[\left(2\phi_c^P I_{ld}^P - \phi_d^P I_{lc}^P\right) \left(2t_{ik}^{ac} t_l^d - t_{ik}^{ca} t_l^d - t_i^c t_k^a t_l^d - t_{kl}^{ad} t_i^c + t_{il}^{cd} t_k^a\right) \right] \right])
\end{aligned} \tag{7.22}$$

$$\begin{aligned}
t_{ij}^{ab} = \frac{1}{\varepsilon_a + \varepsilon_b - \varepsilon_i - \varepsilon_j} \sum_P \omega^P (\phi_i^P \phi_a^P I_{jb}^P \\
- \sum_{kc} \left[\phi_i^P \phi_k^P I_{bc}^P t_k^a t_j^c - \phi_i^P \phi_a^P I_{kc}^P t_k^b t_j^c t_i^c t_k^b - \phi_k^P \phi_c^P I_{jb}^P t_k^a t_i^c \right] \\
+ \sum_{dlkc} \left[\phi_k^P \phi_c^P I_{ld}^P t_i^c t_j^d t_k^a t_l^b \right] + \phi_a^P I_{jb}^P \sum_c \left(\phi_c^P t_i^c \right) + \phi_i^P \phi_a^P \sum_c \left(I_{bc}^P t_j^c \right) \\
- \sum_c \phi_c^P \sum_{kd} \left[\phi_a^P I_{kd}^P t_i^c t_j^d t_k^b - \phi_k^P I_{bd}^P t_i^c t_j^d t_k^a \right] - \phi_i^P \sum_k \left[\phi_k^P I_{jb}^P t_k^a + \phi_a^P I_{jk}^P t_k^b \right] \\
- \phi_i^P \sum_k \left[\phi_k^P I_{jb}^P t_k^a + \phi_a^P I_{jk}^P t_k^b \right] + \sum_k \left[\phi_k^P \sum_{lc} \left(\phi_i^P I_{lc}^P t_k^a t_l^b t_j^c + \phi_c^P I_{jl}^P t_k^a t_l^b t_i^c \right) \right])
\end{aligned} \tag{7.23}$$

The Q-CC2 equations show major differences from the Q-MP2 method. The energy expression 7.21 has no energy denominator, a quadrature over the K grid is therefore not needed. Additionally, only integrals on centres of the P quadrature grid exist, so no PQ pairs are necessary. However, the indices remain fully coupled through the amplitude expressions which cannot be resolved. The final scaling for the energy expression is $\mathcal{O}(PO^2V^2)$ compared to the $\mathcal{O}(P^2OV)$ scaling of MP2. The highest-scaling term of Q-CC2 can be found in the last term of eq. 7.22, which scales with $\mathcal{O}(PN^4)$. It is the most costly term as the integral transformation of $I_{\nu\mu}^P$, which is the highest-dimensional integral, only scales with $\mathcal{O}(PN^3)$. In the above derivation, the Q-CC2 method offers no advantage in memory scaling over the CC2 parent method, as the t_{ij}^{ab} amplitudes have to be stored. It is possible to resolve this scaling and only express the method in terms of up to two-dimensional quantities by inserting eq. 7.23 into eq. 7.22 and 7.21.

No full implementation of Q-CC2 has been released in `libqqc`, therefore investigating the parallel capabilities will be done at a later time. The non-linear memory traversion of the amplitude matrices however will result in a higher number of cache misses and a worse performance.

7.2.2 Algebraic Diagrammatic Construction Method Family

So far only ground state methods have been discussed, which often already have (comparatively) low scaling and high-performing variants. wave function-based methods for the description of excited states, however, still suffer from their high scaling and low parallel efficiency.

One such method family is the algebraic diagrammatic construction scheme[121]. Originally developed to determine the poles of the polarisation propagator, which corresponds to the vertical excitation energies of the system, the ADC method family allows for a straightforward, perturbation theory approach to excited states. The original derivation is outside the scope of this work, but an alternative formulation shall be quickly summarised here.

A set of non-orthogonal correlated excited states is generated by letting an excitation operator $\{\hat{C}_n\} = \{a_a^\dagger a_i; a_a^\dagger a_b^\dagger a_j a_i; \dots\}$ act upon the ground state wave function $|\Phi_0\rangle$

$$|\Phi_N^0\rangle = \hat{C}_N \, |\Phi_0\rangle \tag{7.24}$$

These are then orthogonalized using Gram-Schmidt orthogonalization with respect to each other and the ground state, starting from the lowest-lying excitation class. This results in a set of precursor states $|\Phi_l^\#\rangle$. Afterwards, these precursor states are orthogonalized within each excitation class. The resulting states are called the intermediate states[9,44,122,123] $|\Phi_n^{\mathrm{IS}}\rangle$.

The Hamiltonian can then be expressed in the basis of these intermediate states

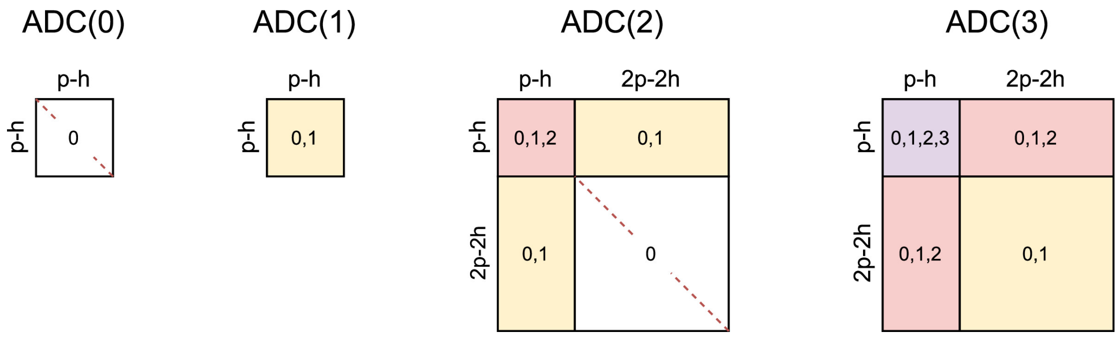


FIGURE 7.5: Schematic view of the ADC matrices for different orders of approximation[9]. The orders of terms contributing to each block are shown.

$$M_{nm} = \langle \Phi_n^{\mathrm{IS}} | \hat{H} - E_0 | \Phi_m^{\mathrm{IS}} \rangle \tag{7.25}$$

By using the Møller-Plesset perturbation expansion of the wave function and the energy expression with this ansatz one arrives at a perturbation-based expression which results in a Hermitian eigenvalue problem of the form

$$\mathbf{MY} = \mathbf{YE} \qquad \mathbf{Y}^{\dagger}\mathbf{Y} = \mathbf{1} \tag{7.26}$$

where $\mathbf{Y}$ is the matrix of amplitude vectors expanding the excited states into the intermediate state basis $\mathbf{Y}_n$ and $\mathbf{E}_n$ is the diagonal matrix of vertical excitation energy eigenvalues.

As a result of the perturbative ground state wave function the resulting Hermitian matrix $\mathbf{M}$ is also expressed as a sum of different perturbation orders

$$\mathbf{M} = \mathbf{M^{(0)}} + \mathbf{M^{(1)}} + \mathbf{M^{(2)}} + \dots \tag{7.27}$$

These orders terminate differently for the single, double, ... excitation block of the overall matrix, depending on the chosen order of the ADC scheme. This results in the matrix structure shown in fig. 7.5 for the examples of the first three orders of the ADC method family for the polarization propagator.

The working equations of the single excitation (particle-hole) block for the second-order ADC scheme for the polarisation propagator (ADC(2)-pp) are given in eq. 7.28.

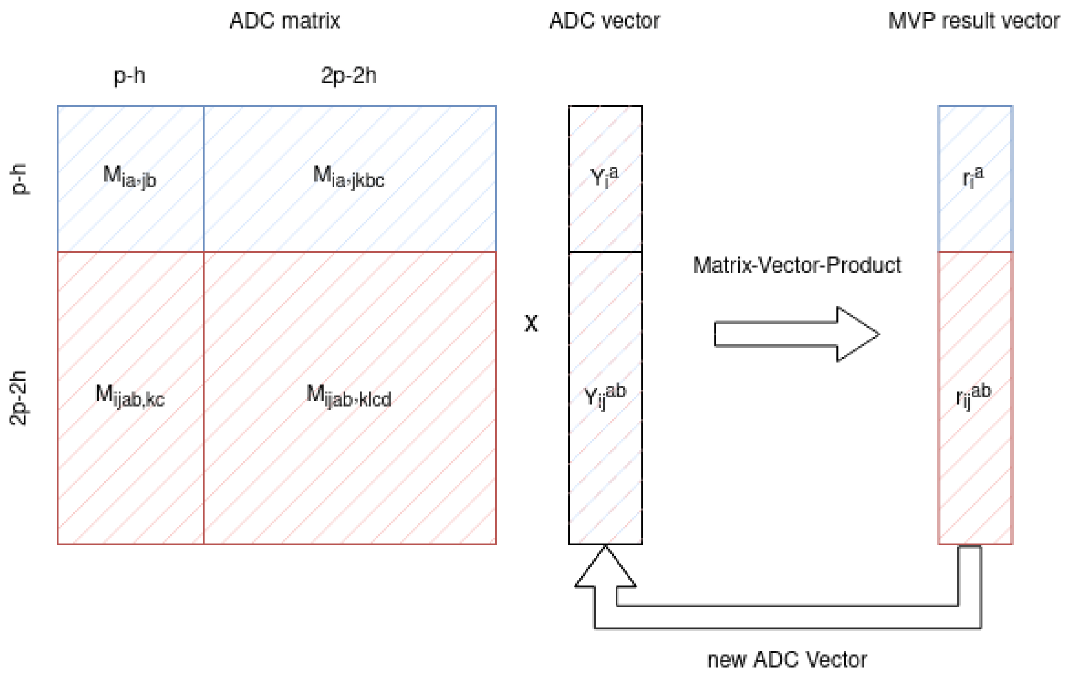


FIGURE 7.6: Matrix-Vector-Product form of the ADC algorithm which is used by the Davidson solver.

$$
\begin{aligned}
M_{ia,jb} =& f_b^a \delta_{ij} - f_j^i \delta_{ab} \\
& - \langle ib||ja\rangle \\
& + \frac{1}{4}\sum_{klc} \langle kl||ac\rangle \, t_{kl}^{bc\,(1)} \delta_{ij} + \frac{1}{4}\sum_{klc} \langle kl||bc\rangle \, t_{kl}^{ac\,(1)} \delta_{ij} \\
& + \frac{1}{4}\sum_{kcd} \langle ik||cd\rangle \, t_{jk}^{cd\,(1)} \delta_{ab} + \frac{1}{4}\sum_{kcd} \langle jk||cd\rangle \, t_{ik}^{cd\,(1)} \delta_{ab} \\
& - \frac{1}{2}\sum_{kc} \langle ik||ac\rangle \, t_{jk}^{bc\,(1)} - \frac{1}{2}\sum_{kc} \langle jk||bc\rangle \, t_{ik}^{ac\,(1)}
\end{aligned}
\tag{7.28}
$$

Diagonalisation of the full matrix leads to the vertical excitation energies as the eigenvalues and the excited state wave functions as eigenfunctions. This is however unfeasible, as the necessary memory for the largest block of the matrix scales with $\mathcal{O}(N^8)$ with respect to the system size.

To remedy this problem and to only calculate the lowest-lying eigenstates a Davidson[124,125] algorithm is used. For this eq. 7.28 can be re-expressed in terms of a matrix-vector product form (figure 7.6), where only the underlying entities have to be stored.

This results in the two vector equations which are iterated until convergence is reached (using r for the new Y). The vector product consists of the following terms

$$\mathbf{M}\begin{pmatrix} r_i^a \\ r_{ij}^{ab} \end{pmatrix} = \begin{pmatrix} r_i^{a,(1)} + r_i^{a,(2)} \\ r_{ij}^{ab,(1)} + r_{ij}^{ab,(2)} \end{pmatrix} = \begin{pmatrix} \sum_{jb} M_{ia,jb} Y_j^b + \sum_{jkbc} M_{ia,jkbc} Y_{jk}^{bc} \\ \sum_{kc} M_{ijab,kc} Y_j^b + \sum_{klcd} M_{ijab,klcd} Y_{kl}^{cd} \end{pmatrix} \tag{7.29}$$

An example for the $r_i^{a,(1)}$ part of the matrix-vector product is given in eq. 7.30.

$$\begin{aligned} r_i^{a,(1)} =& \sum_b Y_i^b f_b^a - \sum_j Y_j^a f_j^i \\ & - \sum_{jb} Y_j^b \langle ib||ja\rangle \\ & + \sum_b Y_i^b \cdot i_1^{ab} \\ & + \sum_j Y_j^a \cdot i_{2,ij} \\ & - \frac{1}{2}\sum_{jkbc} Y_j^b \langle ik||ac\rangle\, t_{jk}^{bc\,(1)} - \frac{1}{2}\sum_{jkbc} Y_j^b \langle jk||bc\rangle\, t_{ik}^{ac(1)} \end{aligned} \tag{7.30}$$

$$i_1^{ab} = \frac{1}{4}\sum_{jkc} \langle jk||ac\rangle\, t_{jk}^{bc\,(1)} + \frac{1}{4}\sum_{jkc} \langle jk||bc\rangle\, t_{jk}^{ac(1)} \tag{7.31}$$

$$i_{2,ij} = \frac{1}{4}\sum_{kbc} \langle ik||bc\rangle\, t_{jk}^{bc\,(1)} + \frac{1}{4}\sum_{kbc} \langle jk||bc\rangle\, t_{ik}^{bc(1)} \tag{7.32}$$

Therein, Y is the ADC vector and $t_{ij}^{ab(1)}$ are the first-order doubles amplitudes from Møller-Plesset theory

$$t_{ij}^{ab(1)} = \frac{\langle ij||ab\rangle}{\varepsilon_a + \varepsilon_b - \varepsilon_i - \varepsilon_j} \tag{7.33}$$

This is done for a set of guess vectors depending on the number of requested lowest-lying states.

The highest-scaling steps for this method are the AO-to-MO transformation of the $\langle pq|rs\rangle$ integrals as well as e.g. the full summation over the intermediate states in the sixth and seventh term of eq. 7.30. A simple implementation of these expressions would have a computational scaling of $\mathcal{O}(N^6)$, although the final implementation of the method scales with $\mathcal{O}(N^5)$.

The ADC(2) equations are made up of antisymmetriesed ERIs $\langle pq||rs\rangle$ and expressions of the form $\langle ij||ab\rangle\, t_{kl}^{cd}$. Employing the Q-MP2-style quadrature scheme one can re-express these underlying quantities as

$$\langle ij|ab\rangle \rightarrow \sum_P \omega^P \left[\left(\phi_i^P\right)\left(\phi_a^P I_{jb}^P\right)\right] \tag{7.34}$$

$$\langle ij||ab\rangle \rightarrow \sum_P \omega^P \left[\left(\phi_i^P\right)\left(\phi_a^P I_{jb}^P - \phi_b^P I_{ja}^P\right)\right] \tag{7.35}$$

$$\frac{\langle ij||ab\rangle}{\varepsilon_a+\varepsilon_b-\varepsilon_i-\varepsilon_j} \rightarrow \sum_K \omega^K \sum_P \omega^P \left[\left(\phi_i^P t_K^{-\varepsilon_i}\right)\left(\phi_a^P t_K^{\varepsilon_a} I_{jb}^P t_K^{\varepsilon_b-\varepsilon_j-1} - \phi_b^P t_K^{\varepsilon_b} I_{ja}^P t_K^{\varepsilon_a-\varepsilon_j-1}\right)\right] \tag{7.36}$$

and

$$\begin{aligned}
\langle ij||ab\rangle\, t_{kl}^{cd} \rightarrow &\sum_P \omega^P \left[\left(\phi_i^P\right)\left(\phi_a^P I_{jb}^P - \phi_b^P I_{ja}^P\right)\right] \qquad (7.37)\\
&\sum_K \omega^K \sum_Q \omega^Q \left[\left(\phi_k^Q t_K^{-\varepsilon_k}\right)\left(\phi_c^Q t_K^{\varepsilon_c} I_{ld}^Q t_K^{\varepsilon_d-\varepsilon_l-1} - \phi_d^Q t_K^{\varepsilon_d} I_{lc}^Q t_K^{\varepsilon_c-\varepsilon_l-1}\right)\right]\\
&= \sum_K \omega^K \sum_Q \omega^Q \sum_P \omega^P \Bigg[\left(\phi_i^P \phi_k^Q t_K^{-\varepsilon_k}\right)\\
&\Bigg(\phi_a^P I_{jb}^P \phi_c^Q t_K^{\varepsilon_c} I_{ld}^Q t_K^{\varepsilon_d-\varepsilon_l-1} - \phi_a^P I_{jb}^P \phi_d^Q t_K^{\varepsilon_d} I_{lc}^Q t_K^{\varepsilon_c-\varepsilon_l-1}\\
&- \phi_b^P I_{ja}^P \phi_c^Q t_K^{\varepsilon_c} I_{ld}^Q t_K^{\varepsilon_d-\varepsilon_l-1} + \phi_b^P I_{ja}^P \phi_d^Q t_K^{\varepsilon_d} I_{lc}^Q t_K^{\varepsilon_c-\varepsilon_l-1}\Bigg)\Bigg] \qquad (7.38)
\end{aligned}$$

Inserting eq. 7.35, 7.36 and 7.38 into 7.30 and utilising the symmetry $\langle ik||bc\rangle = \langle ki||cb\rangle$ changes the scaling to $\mathcal{O}(P^2OV^2)$. The full equations are given in eq. 7.39 - 7.41.

$$
\begin{aligned}
r_i^{a,(1)} \approx & \sum_b Y_i^b f_b^a - \sum_j Y_j^a f_j^i \\
& - \sum_P \omega^P \left[\left(\phi_i^P\right) \left(\sum_{jb} Y_j^b \phi_j^P I_{ba}^P - \phi_a^P \sum_{jb} Y_j^b I_{bj}^P \right) \right] \\
& + \sum_b Y_i^b \cdot i_1^{ab} \\
& + \sum_j Y_j^a \cdot i_{2,ij} \\
& - \frac{1}{2} \sum_K \omega^K \sum_Q \omega^Q \sum_P \omega^P \Bigg[\sum_k \left(\phi_k^P \phi_k^Q t_K^{-\varepsilon_k} \right) \\
& \Bigg(I_{ia}^P \sum_c \left(\phi_c^P \phi_c^Q t_K^{\varepsilon_c} \right) \sum_{jb} \left(Y_j^b I_{jb}^Q t_K^{\varepsilon_b - \varepsilon_j - 1} \right) - I_{ia}^P \sum_j \left(\sum_b \left(Y_j^b \phi_b^Q t_K^{\varepsilon_b} \right) \sum_c \left(\phi_c^P I_{jc}^Q t_K^{\varepsilon_c - \varepsilon_j - 1} \right) \right) \\
& - \phi_a^P \sum_c \left(I_{ic}^P \phi_c^Q t_K^{\varepsilon_c} \right) \sum_{jb} \left(Y_j^b I_{jb}^Q t_K^{\varepsilon_b - \varepsilon_j - 1} \right) + \phi_a^P \sum_j \left(\sum_b \left(Y_j^b \phi_b^Q t_K^{\varepsilon_b} \right) \sum_c \left(I_{ic}^P I_{jc}^Q t_K^{\varepsilon_c - \varepsilon_j - 1} \right) \right) \Bigg) \Bigg] \\
& - \frac{1}{2} \sum_K \omega^K \sum_Q \omega^Q \sum_P \omega^P \Bigg[\sum_k \left(\phi_k^P \phi_k^Q t_K^{-\varepsilon_k} \right) \\
& \Bigg(I_{ia}^Q t_K^{\varepsilon_a - \varepsilon_i - 1} \sum_c \left(\phi_c^P \phi_c^Q t_K^{\varepsilon_c} \right) \sum_{jb} \left(Y_j^b I_{jb}^P \right) - \phi_a^Q t_K^{\varepsilon_a} \sum_c \left(\phi_c^P I_{ic}^Q t_K^{\varepsilon_c - \varepsilon_i - 1} \right) \sum_{jb} \left(Y_j^b I_{jb}^P \right) \\
& - I_{ia}^Q t_K^{\varepsilon_a - \varepsilon_i - 1} \sum_j \left(\sum_b \left(Y_j^b \phi_b^P \right) \sum_c \left(I_{jc}^P \phi_c^Q t_K^{\varepsilon_c} \right) \right) \\
& + \phi_a^Q t_K^{\varepsilon_a} \sum_j \left(\sum_b \left(Y_j^b \phi_b^P \right) \sum_c \left(I_{jc}^P I_{ic}^Q t_K^{\varepsilon_c - \varepsilon_i - 1} \right) \right) \Bigg) \Bigg]
\end{aligned}
\tag{7.39}
$$

$$
\begin{aligned}
i_1^{ab} =& \frac{1}{4}\sum_K \omega^K \sum_Q \omega^Q \sum_P \omega^P \Bigg[\sum_k \left(\phi_k^P \phi_k^Q t_K^{-\varepsilon_k} \right) \\
&\left(\sum_j \left(I_{ja}^P I_{jb}^Q t_K^{\varepsilon_b - \varepsilon_j - 1} \right) \sum_c \left(\phi_c^P \phi_c^Q t_K^{\varepsilon_c} \right) - \phi_b^Q t_K^{\varepsilon_b} \sum_j \left(I_{ja}^P \sum_c \left(\phi_c^P I_{jc}^Q t_K^{\varepsilon_c - \varepsilon_j - 1} \right) \right) \right. \\
&\left. - \phi_a^P \sum_j \left(\sum_c \left(I_{jc}^P \phi_c^Q t_K^{\varepsilon_c} \right) I_{jb}^Q t_K^{\varepsilon_b - \varepsilon_j - 1} \right) + \phi_a^P \phi_b^Q t_K^{\varepsilon_b} \sum_{jc} \left(I_{jc}^P I_{jc}^Q t_K^{\varepsilon_c - \varepsilon_j - 1} \right) \right) \Bigg] \\
&+ \frac{1}{4}\sum_K \omega^K \sum_Q \omega^Q \sum_P \omega^P \Bigg[\sum_k \left(\phi_k^P \phi_k^Q t_K^{-\varepsilon_k} \right) \\
&\left(\sum_j \left(I_{jb}^P I_{ja}^Q t_K^{\varepsilon_a - \varepsilon_j - 1} \right) \sum_c \left(\phi_c^P \phi_c^Q t_K^{\varepsilon_c} \right) - \phi_a^Q t_K^{\varepsilon_a} \sum_j \left(I_{jb}^P \sum_c \left(\phi_c^P I_{jc}^Q t_K^{\varepsilon_c - \varepsilon_j - 1} \right) \right) \right. \\
&\left. - \phi_b^P \sum_j \left(\sum_c \left(I_{jc}^P \phi_c^Q t_K^{\varepsilon_c} \right) I_{ja}^Q t_K^{\varepsilon_a - \varepsilon_j - 1} \right) + \phi_b^P \phi_a^Q t_K^{\varepsilon_a} \sum_{jc} \left(I_{jc}^P I_{jc}^Q t_K^{\varepsilon_c - \varepsilon_j - 1} \right) \right) \Bigg]
\end{aligned}
\tag{7.40}
$$

$$
\begin{aligned}
i_{2,ij} =& \frac{1}{4}\sum_K \omega^K \sum_Q \omega^Q \sum_P \omega^P \Bigg[\sum_k \left(\phi_k^P \phi_k^Q t_K^{-\varepsilon_k} \right) \\
&\left(\sum_b \left(I_{ib}^P I_{jb}^Q t_K^{\varepsilon_b - \varepsilon_j - 1} \right) \sum_c \left(\phi_c^P \phi_c^Q t_K^{\varepsilon_c} \right) - \sum_b \left(I_{ib}^P \phi_b^Q t_K^{\varepsilon_b} \right) \sum_c \left(\phi_c^P I_{jc}^Q t_K^{\varepsilon_c - \varepsilon_j - 1} \right) \right. \\
&\left. - \sum_b \left(\phi_b^P I_{jb}^Q t_K^{\varepsilon_b - \varepsilon_j - 1} \right) \left(\sum_c I_{ic}^P \phi_c^Q t_K^{\varepsilon_c} \right) + \sum_b \left(\phi_b^P \phi_b^Q t_K^{\varepsilon_b} \right) \sum_c \left(I_{ic}^P I_{jc}^Q t_K^{\varepsilon_c - \varepsilon_j - 1} \right) \right) \Bigg] \\
&+ \frac{1}{4}\sum_K \omega^K \sum_Q \omega^Q \sum_P \omega^P \Bigg[\sum_k \left(\phi_k^P \phi_k^Q t_K^{-\varepsilon_k} \right) \\
&\left(\sum_b \left(I_{jb}^P I_{ib}^Q t_K^{\varepsilon_b - \varepsilon_i - 1} \right) \sum_c \left(\phi_c^P \phi_c^Q t_K^{\varepsilon_c} \right) - \sum_b \left(I_{jb}^P \phi_b^Q t_K^{\varepsilon_b} \right) \sum_c \left(\phi_c^P I_{ic}^Q t_K^{\varepsilon_c - \varepsilon_i - 1} \right) \right. \\
&\left. - \sum_b \left(\phi_b^P I_{ib}^Q t_K^{\varepsilon_b - \varepsilon_i - 1} \right) \sum_c \left(I_{jc}^P \phi_c^Q t_K^{\varepsilon_c} \right) + \sum_b \left(\phi_b^P \phi_b^Q t_K^{\varepsilon_b} \right) \sum_c \left(I_{jc}^P I_{ic}^Q t_K^{\varepsilon_c - \varepsilon_i - 1} \right) \right) \Bigg]
\end{aligned}
\tag{7.41}
$$

The largest remaining quantity is the I_{pq}^P integral, which lowers the AO-to-MO transformation step scaling to $\mathcal{O}(PN^3)$, which is the most expensive step in terms of system size. The highest-scaling memory object is r_{ijab}, meaning that memory still scales with $\mathcal{O}(N^4)$, however with a lower prefactor as the need for storing $\langle \nu\mu | \lambda\sigma \rangle$ is removed. A Schur complement ansatz can then be used to fold the doubles space into the single space, lowering the memory scaling to $\mathcal{O}(PN^2)$, dominated by the I_{pq}^{P} integral.

$$\mathbf{M}\vec{r} = \begin{pmatrix} \sum_{jb} M_{ia,jb} Y_j^b + \sum_{jkbc} M_{ia,jkbc} Y_{jk}^{bc} \\ \sum_{kc} M_{ijab,kc} Y_k^c + \sum_{klcd} M_{ijab,klcd} Y_{kl}^{cd} \end{pmatrix} = \begin{pmatrix} \lambda Y_i^a \\ \lambda Y_{ij}^{ab} \end{pmatrix} \tag{7.42}$$

$$Y_{ij}^{ab} = \sum_{kc} M_{ijab,kc} Y_k^c \left(\sum_{mnef} [\lambda \mathbf{1} - M_{ijab,mnef}] \right)^{-1} \tag{7.43}$$

$$\begin{aligned} r_i^a &= \sum_{jb} M_{ia,jb} Y_j^b + \sum_{jkbc} M_{ia,jkbc} Y_{jk}^{bc} & (7.44) \\ &= \sum_{jb} M_{ia,jb} Y_j^b + \sum_{jkbc} M_{ia,jkbc} \left(\sum_{ld} M_{jkbc,ld} Y_l^d \left(\sum_{mnef} [\lambda \mathbf{1} - M_{jkbc,mnef}] \right)^{-1} \right) & (7.45) \end{aligned}$$

Solving eq. 7.42 for Y_{ij}^{ab} one arrives at eq. 7.43. This eliminates the necessity of storing r_{ij}^{ab} by plugging this equation into the equation for r_i^a, eq. 7.44, which leads to eq. 7.43. Now only r_i^a has to be stored and iterated over. The double part of the amplitude can be recovered through eq. 7.43.

The calculation of r_{ia} and r_{ijab} has the chance of becoming embarrassingly parallel, with a parallelisation scheme very close to the Q-MP2 algorithm. The only difference is that after the calculation of r_{ia} and r_{ijab} both have to be communicated to the other nodes for the Davidson algorithm. Alternatively, only r_{ia} has to be communicated if the above-mentioned ansatz is used.

In summary, with the provided equations a fully parallel, high-performance Q-ADC(2) calculation is within reach.

Chapter 8

Conclusion

Modern (super-)computer clusters achieve their peta- and now exascale performance through the use of millions of parallel execution units, such as CPU or GPU cores. To utilise these resources fully a high degree of parallel efficiency is necessary for an algorithm, as well as a low level of data communication. Achieving these properties for common wave function-based methods is not a trivial task and many present implementations of post-HF method are therefore limited in the system size they can treat not because of lacking resources, but because of lacking implementation. Additionally, their high computational and memory scaling limits their applicability, which often stems from the need to store, transform and manipulate two- and more-electron repulsion integrals. The constant search for low-scaling methods must also be a search for formulations which show a high parallel efficiency to utilise modern computer architecture to its fullest potential.
The Møller-Plesset perturbation theory is a common ground state method used daily by computational chemists for the treatment of small to large systems. It is most often used in its second-order form, which scales computationally with $\mathcal{O}(N^5)$ and with a memory scaling of $\mathcal{O}(N^4)$ with N being the number of atomic orbitals of the system. Its memory scaling comes from storing the atomic orbital representation of the two-electron integral $\langle \mu\nu|\lambda\omega\rangle$ and its computational scaling from the need to transform this quantity into its molecular orbital representation. The calculation of the MP2 correlation energy correction has a computational scaling of $\mathcal{O}(O^2V^2)$, as all molecular orbital indices are coupled through the energy denominator.
This thesis presented an ansatz based on the Laplace transformation of the energy

denominator[1,51,52], which effectively decouples the orbital indices, as well as the reformulation of the four indices ERI into lower-ranking two- and three-dimensional entities[2–4] through numerical quadrature. The quadrature form of the full MP2 energy expression, Q-MP2[6], has a lower computational scaling ($\mathcal{O}(PQOV)$) and lower memory scaling ($\mathcal{O}(POV)$) as its parent method. The scaling is dominated by the P, Q prefactors, which are the number of spatial quadrature grid points. Furthermore, the quadrature scheme is embarrassingly parallel, as all PQ quadrature pairs are fully independent. The accuracy of this method in reproducing the energies of the parent MP2 method was shown to be dependent on the number and nature of atomic centres, as well as of the employed integration grids. The latter was found to be of the most importance.
The Q-MP2 method was implemented into a new open-source `C++` library, `libqqc`[93]. This library was developed to be portable, modular, and follow sustainable coding guidelines with automatic continuous integration and documentation. The energy calculation is provided in two variants, one utilising the `Eigen`[105] library and one written in terms of low-level `C` arrays. The implementation focuses on high parallel efficiency, either as a `MPI`, (single-node only) `OpenMP` or hybrid `OpenMP+MPI` variant.
This implementation was then used to investigate the influence of the different integration grids on the accuracy of the Q-MP2 method. For this, the `libqqc` library was interfaced with `Q-Chem`[88], which provided the underlying quantities and the Becke-weighted integration grid commonly used by DFT calculations. A specialized integration grid module was later added to the implementation[5]. The grid benchmark was performed for seven small-sized systems with 28 different integration grid setups and 3 different basis sets. Both the absolute and relative energy errors were calculated, as the numerical quadrature only recovers the parent method energy proportionally. For all but the smallest spatial grids, the relative energy error is between 0.33 % and 1.5 % with the absolute error being the same magnitude as the target chemical accuracy or one order of magnitude lower. The energy error increased when going from STO-3G to 3-21G, but got lowered by increasing the Pople basis set to 6-31G. A one-dimensional integration grid of $k = 4$ was deemed to be sufficient and therefore dropped from the formal scaling. However, the chosen one-dimensional quadrature scheme seemed to be misbehaved for the Laplace transformed integral. The spatial integration followed expectations which produced lower relative errors with a denser integration grid. There appears to be a specific error cancellation with 20/18 (radial, angular) grids,

that lowered the error more than expected. As all but the lowest grid setting was shown to provide the target accuracy, the lowest well-behaved value, 20/38, was chosen for future runs.
The parallel efficiency was tested for all combinations of the two implementations and the three parallelisation schemes. To increase efficiency, a minimal communication scheme was employed as well as a scheme to balance the computational load between the nodes. The different tests were done on up to 960 physical cores and on up to 20 nodes on the `JUSTUS 2` computer cluster. The hybrid `OpenMP+MPI` variant with the low-level C arrays was shown to have the best single-node performance. The performance however was lacking overall, only reaching up to ≈ 17 % of the theoretical peak. This shows that the algorithm is memory-bound and can most likely be increased in the future by better exploitation of data locality and sparsity. However, parallel efficiency on the single-node level was above ≈ 70 %, with the larger systems achieving superlinear speedups in the hybrid variant. This is most likely the result of better utilisation of faster, low-lying caches at a higher number of employed cores. This effect was not seen on any of the small examples, as their data already resides mostly in the fast caches.
An example of the efficiency of the load-balancing scheme was shown for three nodes. Multi-node performance of up to 960 cores/20 nodes followed the trend of the single-node performance. For the hybrid variant, the larger systems were once again shown to exhibit linear to superlinear parallel efficiency, with the smallest examples showing sublinear performance. The variant utilising only `MPI` for parallelization shows a more consistent behaviour, with a parallel efficiency of over 70 %.
Future work will focus on increasing the absolute performance by optimising the integration grids and lowering the number of integration grid points. This project has already been started through the master thesis of Isabel Vinterbladh[5]. Additional performance increases may come from further integral screening techniques and the better utilisation of temporal locality and exploitation of the sparsity of the underlying matrix entities.

The quadrature scheme presented in this thesis was extended to new methods. The quadrature form of the second-order approximation of the coupled cluster method (Q-CC2) was derived in cooperation with Theofilos Dimitrakopoulos[120], however, the decoupling of the indices was not possible due to the explicit coupling of the amplitude terms in the energy expression.

Finally, the quadrature scheme was extended to the particle-hole matrix-vector-products of the second-order excited-state algebraic diagrammatic construction scheme for the polarisation propagator in its ISR formalism. Here the theoretical computational scaling of the parent method is $\mathcal{O}(N^5)$, stemming from both the terms and the AO-to-MO transformation of the two-electron ERIs, and the memory scaling is $\mathcal{O}(N^4)$. This calculation has to be repeated multiple times for each requested excited state and has a low parallel efficiency in its current implementation. The newly derived Q-ADC(2) equations have a theoretical scaling of $\mathcal{O}(P^2OV^2)$ and the largest stored entity needs $\mathcal{O}(O^2V^2)$ memory. The latter problem, however, can be addressed by folding the doubles space into the singles space, decreasing the memory scaling to $\mathcal{O}(PN^2)$. A future implementation is expected to have higher absolute single-node performance compared to Q-MP2, as more computational work is done per memory operation. Its parallel efficiency is expected to be similar to the current Q-MP2 implementation, as the partial calculations of the matrix-vector-products remain fully independent and the only additional communication is the exchange of the partial MVPs between nodes.

A highly parallel efficient and lower scaling Q-ADC(2) method is therefore within reach.

Bibliography

[1] M. Häser, J. Almlöf, *The Journal of Chemical Physics* **1992**, *96*, 489–494.

[2] K. Ishimura, S. Ten-no, *Theoretical Chemistry Accounts* **2011**, *130*, 317–321.

[3] G. M. Barca, S. C. McKenzie, N. J. Bloomfield, A. T. Gilbert, P. M. Gill, *Journal of Chemical Theory and Computation* **2020**, *16*, 1568–1577.

[4] N. J. Bloomfield, *Jovian Moller-Plesset Perturbation Theory: A low-scaling and massively parallelizable approach*, **2016**.

[5] I. Vinterbladh, *Optimising grids for HPC quantum chemical methods*, **2022**.

[6] B. Thomitzni, *Implementation of the JOVE Algorithm for MP2 Correlation Energies*, **2018**.

[7] Y. Shao, Z. Gan, E. Epifanovsky, A. T. Gilbert, M. Wormit, J. Kussmann, A. W. Lange, A. Behn, J. Deng, X. Feng *et al.*, *Molecular Physics* **2015**, *113*, 184–215.

[8] E. Epifanovsky, M. Wormit, T. Kuś, A. Landau, D. Zuev, K. Khistyaev, P. Manohar, I. Kaliman, A. Dreuw, A. I. Krylov, *New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations*, **2013**.

[9] A. Dreuw, M. Wormit, *Wiley Interdisciplinary Reviews: Computational Molecular Science* **2015**, *5*, 82–95.

[10] M. Zheng, J. Zhao, C. Cui, Z. Fu, X. Li, X. Liu, X. Ding, X. Tan, F. Li, X. Luo *et al.*, *Medicinal research reviews* **2018**, *38*, 914–950.

[11] K. Houk, F. Liu, *Accounts of chemical research* **2017**, *50*, 539–543.

[12] I. S. Moreira, P. A. Fernandes, M. J. Ramos, *Journal of computational chemistry* **2010**, *31*, 317–342.

[13] G. B. Goh, N. O. Hodas, A. Vishnu, *Journal of computational chemistry* **2017**, *38*, 1291–1307.

[14] C. N. Cavasotto, M. G. Aucar, N. S. Adler, *International Journal of Quantum Chemistry* **2019**, *119*, e25678.

[15] M. Hassanzadeganroudsari, A. H. Ahmadi, N. Rashidi, M. K. Hossain, A. Habib, V. Apostolopoulos, *Biologics* **2021**, *1*, 111–128.

[16] A. Chakraborty, S. Kunnikuruvan, M. Dixit, D. T. Major, *Israel Journal of Chemistry* **2020**, *60*, 850–862.

[17] B. Craig, C.-K. Skylaris, T. Schoetz, C. P. de León, *Energy Reports* **2020**, *6*, 198–208.

[18] A. Urban, D.-H. Seo, G. Ceder, *npj Computational Materials* **2016**, *2*, 1–13.

[19] S. Ding, X. Yu, Z.-F. Ma, X. Yuan, *Journal of Materials Chemistry A* **2021**, *9*, 8160–8194.

[20] Y. Cui, P. Zhu, X. Liao, Y. Chen, *Journal of Materials Chemistry C* **2020**, *8*, 15920–15939.

[21] B. G. Sumpter, V. Meunier, *Journal of Polymer Science Part B: Polymer Physics* **2012**, *50*, 1071–1089.

[22] N. Agnihotri, *Journal of Photochemistry and Photobiology C: Photochemistry Reviews* **2014**, *18*, 18–31.

[23] F. A. Angel, M. B. Camarada, I. A. Jessop, *Critical Reviews in Solid State and Materials Sciences* **2022**, 1–28.

[24] W. Wang, O. Donini, C. M. Reyes, P. A. Kollman, *Annual review of biophysics and biomolecular structure* **2001**, *30*, 211.

[25] O. Guvench, A. D. MacKerell, *Molecular modeling of proteins* **2008**, 63–88.

[26] J. W. Ponder, D. A. Case, *Advances in protein chemistry* **2003**, *66*, 27–85.

[27] I. M. Padilla Espinosa, T. D. Jacobs, A. Martini, *Journal of chemical theory and computation* **2021**, *17*, 4486–4498.

[28] H. Heinz, T.-J. Lin, R. Kishore Mishra, F. S. Emami, *Langmuir* **2013**, *29*, 1754–1765.

[29] K. Choudhary, B. DeCost, F. Tavazza, *Physical review materials* **2018**, *2*, 083801.

[30] E. Schrödinger, *Annalen der Physik* **1926**, *385*, 437–490.

[31] C. Møller, M. S. Plesset, *Physical Review* **1934**, *46*, 618.

[32] R. Cammi, B. Mennucci, J. Tomasi, *Computational chemistry: reviews of current trends* **2003**, 1–79.

[33] C. Amovilli, V. Barone, R. Cammi, E. Cancès, M. Cossi, B. Mennucci, C. S. Pomelli, J. Tomasi, *Advances in Quantum Chemistry* **1998**, *32*, 227–261.

[34] B. Mennucci, *Wiley Interdisciplinary Reviews: Computational Molecular Science* **2012**, *2*, 386–404.

[35] S. Ten-no, J. Noga, *Wiley Interdisciplinary Reviews: Computational Molecular Science* **2012**, *2*, 114–125.

[36] T. B. Adler, G. Knizia, H.-J. Werner, *The Journal of chemical physics* **2007**, *127*, 221106.

[37] G. Knizia, T. B. Adler, H.-J. Werner, *The Journal of chemical physics* **2009**, *130*, 054104.

[38] W. Klopper, C. C. Samson, *The Journal of chemical physics* **2002**, *116*, 6397–6410.

[39] H.-J. Werner, T. B. Adler, F. R. Manby, *The Journal of chemical physics* **2007**, *126*, 164102.

[40] L. S. Cederbaum, W. Domcke, J. Schirmer, *Phys. Rev. A* **1980**, *22*, 206–222.

[41] A. Barth, L. Cederbaum, *Physical Review A* **1981**, *23*, 1038.

[42] A. Barth, J. Schirmer, *Journal of Physics B: Atomic and Molecular Physics (1968-1987)* **1985**, *18*, 867.

[43] J. Wenzel, A. Holzer, M. Wormit, A. Dreuw, *The Journal of Chemical Physics* **2015**, *142*, 214104.

[44] M. Wormit, D. R. Rehn, P. H. Harbach, J. Wenzel, C. M. Krauter, E. Epifanovsky, A. Dreuw, *Molecular Physics* **2014**, *112*, 774–784.

[45] J. Wenzel, M. Wormit, A. Dreuw, *Journal of Computational Chemistry* **2014**, *35*, 1900–1915.

[46] D. A. Matthews, *Molecular Physics* **2020**, *118*, e1771448.

[47] R. A. Kendall, H. A. Früchtl, *Theoretical Chemistry Accounts* **1997**, *97*, 158–163.

[48] I. Røeggen, T. Johansen, *The Journal of chemical physics* **2008**, *128*, 194107.

[49] M. Feyereisen, G. Fitzgerald, A. Komornicki, *Chemical Physics Letters* **1993**, *208*, 359–363.

[50] A. P. Rendell, T. J. Lee, *The Journal of chemical physics* **1994**, *101*, 400–408.

[51] J. Almlöf, *Chemical physics letters* **1991**, *181*, 319–320.

[52] M. Häser, *Theoretica Chimica Acta* **1993**, *87*, 147–173.

[53] S. Y. Willow, K. S. Kim, S. Hirata, *The Journal of Chemical Physics* **2012**, *137*, 204122.

[54] A. E. Doran, S. Hirata, *Journal of Chemical Theory and Computation* **2016**, *12*, 4821–4832.

[55] J. Shalf, *Philosophical Transactions of the Royal Society A* **2020**, *378*, 20190061.

[56] K. Rupp, S. Selberherr, *IEEE Transactions on Semiconductor Manufacturing* **2010**, *24*, 1–4.

[57] J. R. Powell, *Proceedings of the IEEE* **2008**, *96*, 1247–1248.

[58] A. Smith, N. James, *2022 IEEE Hot Chips 34 Symposium (HCS)*, **2022**, pp. 1–23.

[59] A. Szabo, N. S. Ostlund, *Modern quantum chemistry: introduction to advanced electronic structure theory*, Courier Corporation, **2012**.

[60] P. A. M. Dirac, *The principles of quantum mechanics*, Oxford university press, **1981**.

[61] J.-P. Antoine, *Journal of Mathematical Physics* **1969**, *10*, 53–69.

[62] R. Rannacher, *Vorlesungsskriptum SS 2005* **2006**.

[63] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, E. Teller, *The Journal of Chemical Physics* **1953**, *21*, 1087–1092.

[64] W. K. Hastings, *Biometrika* **1970**, *57*, 97–109.

[65] A. Kolmogorov, *Dokl. Akad. Nauk. SSR*, **1954**, pp. 2–3.

[66] J. Möser, *Nachr. Akad. Wiss. Göttingen II* **1962**, 1–20.

[67] V. I. Arnold, *Russ. Math. Surv* **1963**, *18*, 9–36.

[68] M. Born, R. Oppenheimer, *Annalen der Physik* **1927**, *389*, 457–484.

[69] S. Matsika, *Chemical Reviews* **2021**, *121*, 9407–9449.

[70] W. Domcke, D. Yarkony, H. Köppel, *Conical intersections: electronic structure, dynamics & spectroscopy*, *Vol. 15*, World Scientific, **2004**.

[71] W. Domcke, D. R. Yarkony, H. Köppel, *Conical intersections: theory, computation and experiment*, *Vol. 17*, World Scientific, **2011**.

[72] W. Pauli, *Einführung und Originaltexte* **1925**, 229.

[73] J. S. Binkley, J. A. Pople, W. J. Hehre, *Journal of the American Chemical Society* **1980**, *102*, 939–947.

[74] M. S. Gordon, J. S. Binkley, J. A. Pople, W. J. Pietro, W. J. Hehre, *Journal of the American Chemical Society* **1982**, *104*, 2797–2803.

[75] W. J. Hehre, R. Ditchfield, J. A. Pople, *The Journal of Chemical Physics* **1972**, *56*, 2257–2261.

[76] T. H. Dunning Jr, *The Journal of Chemical Physics* **1989**, *90*, 1007–1023.

[77] A. Schäfer, H. Horn, R. Ahlrichs, *The Journal of Chemical Physics* **1992**, *97*, 2571–2577.

[78] D. R. Hartree, *Mathematical Proceedings of the Cambridge Philosophical Society*, **1928**, pp. 89–110.

[79] V. Fock, *Zeitschrift für Physik* **1930**, *61*, 126–148.

[80] C. C. J. Roothaan, *Reviews of modern physics* **1951**, *23*, 69.

[81] G. Hall, *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* **1951**, *205*, 541–552.

[82] D. Feller, E. D. Glendening, D. E. Woon, M. W. Feyereisen, *The Journal of chemical physics* **1995**, *103*, 3526–3542.

[83] M. W. Feyereisen, D. Feller, D. A. Dixon, *The Journal of Physical Chemistry* **1996**, *100*, 2993–2997.

[84] C. Hättig, A. Hellweg, A. Köhn, *Physical Chemistry Chemical Physics* **2006**, *8*, 1159–1169.

[85] S. A. Maurer, L. Clin, C. Ochsenfeld, *The Journal of chemical physics* **2014**, *140*, 224112.

[86] H.-J. Werner, F. R. Manby, P. J. Knowles, *The Journal of Chemical Physics* **2003**, *118*, 8149–8160.

[87] N. H. Beebe, J. Linderberg, *International Journal of Quantum Chemistry* **1977**, *12*, 683–705.

[88] Y. Shao, Z. Gan, E. Epifanovsky, A. T. Gilbert, M. Wormit, J. Kussmann, A. W. Lange, A. Behn, J. Deng, X. Feng *et al.*, *Molecular Physics* **2015**, *113*, 184–215.

[89] Y. Ohtsuka, S. Nagase, *Chemical Physics Letters* **2008**, *463*, 431–434.

[90] F. Petruzielo, A. Holmes, H. J. Changlani, M. Nightingale, C. Umrigar, *Physical review letters* **2012**, *109*, 230201.

[91] D. Cleland, G. H. Booth, A. Alavi, *The Journal of chemical physics* **2010**, *132*, 041103.

[92] T. J. Martinez, E. A. Carter, *The Journal of chemical physics* **1994**, *100*, 3631–3638.

[93] B. Thomitzni, *libqqc is a library for high performance evaluation of quantum chemistry methods through quadrature schemes*, **2022**, `https://github.com/BenTho-Uni/libqqc`.

[94] bwHPC WIKI, *JUSTUS 2 Hardware*, **2023**, `https://wiki.bwhpc.de/e/JUSTUS2/Hardware`.

[95] I. Free Software Foundation, *GNU Compiler Collection*, **2023**, `https://gcc.gnu.org/`.

[96] Kitware, *CMake*, **2023**, `https://cmake.org/`.

[97] W. Hoffman, K. Martin, *Dr. Dobb's Journal: Software Tools for the Professional Programmer* **2003**, *28*, 40–43.

[98] T. O. M. Project, *Open MPI: Open Source High Performance Computing*, **2023**, `https://www.open-mpi.org/`.

[99] R. L. Graham, T. S. Woodall, J. M. Squyres, *International Conference on Parallel Processing and Applied Mathematics*, **2006**, pp. 228–239.

[100] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, A. Lumsdaine, *2006 IEEE International Conference on Cluster Computing*, **2006**, pp. 1–9.

[101] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, **2004**, pp. 97–104.

[102] OpenMP, *OpenMP*, **2023**, `https://www.openmp.org/`.

[103] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, J. McDonald, *Parallel programming in OpenMP*, Morgan kaufmann, **2001**.

[104] L. Dagum, R. Menon, *IEEE computational science and engineering* **1998**, *5*, 46–55.

[105] G. Guennebaud, B. Jacob *et al.*, *Eigen v3*, http://eigen.tuxfamily.org, **2010**.

[106] A. D. Becke, *The Journal of Chemical Physics* **1988**, *88*, 2547–2553.

[107] A. Bowyer, *The computer journal* **1981**, *24*, 162–166.

[108] D. F. Watson, *The computer journal* **1981**, *24*, 167–172.

[109] S. Rebay, *Journal of computational physics* **1993**, *106*, 125–138.

[110] S. Sloan, G. Houlsby, *Advances in Engineering Software (1978)* **1984**, *6*, 192–197.

[111] L. Keegan, D. Kempf, B. Thomitzni, *Performance Engineering Example: libjove*, **2021**, `https://github.com/ssciwr/jove-performance`.

[112] J. L. Hennessy, D. A. Patterson, *Computer architecture: a quantitative approach*, Elsevier, **2011**.

[113] MPICH, *MPICH | High-Performance Portable MPI*, `https://www.mpich.org/about/collaborators/`.

[114] W. Gropp, *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, **2002**, pp. 7–7.

[115] T. D. Crawford, H. F. Schaefer III, *Reviews in computational chemistry* **2007**, *14*, 33–136.

[116] G. D. Purvis III, R. J. Bartlett, *The Journal of Chemical Physics* **1982**, *76*, 1910–1918.

[117] R. S. Strichartz, *Journal of Functional Analysis* **1987**, *72*, 320–345.

[118] O. Christiansen, H. Koch, P. Jørgensen, *Chemical Physics Letters* **1995**, *243*, 409–418.

[119] C. Hättig, A. Köhn, *The Journal of chemical physics* **2002**, *117*, 6939–6951.

[120] T. Dimitrakopoulos, *Quadratur der approximativen Coupled Cluster Methode zweiter Ordnung*, **2021**.

[121] J. Schirmer, *Physical Review A* **1982**, *26*, 2395.

[122] A. Trofimov, I. Krivdina, J. Weller, J. Schirmer, *Chemical physics* **2006**, *329*, 1–10.

[123] J. Schirmer, A. Trofimov, *The Journal of chemical physics* **2004**, *120*, 11449–11464.

[124] E. R. Davidson in *Methods in computational molecular physics*, Springer, **1983**, pp. 95–113.

[125] E. R. Davidson, *Computer Physics Communications* **1989**, *53*, 49–60.

Appendix A

Supporting Information: Q-MP2 Benchmarks

A.1 Energy Benchmark

Table containing the energy benchmark of the Q-MP2 method as presented in section 3.3.2.

System	Basis	N_P	N_O	N_V	ΔE[Ha]	ΔE_{rel}
H_2	STO-3G	1,520	1	1	$2.36 \cdot 10^{-6}$	0.02%
	3-21g	1,520	1	3	$2.85 \cdot 10^{-5}$	0.16%
	6-31g	1,520	1	3	$7.99 \cdot 10^{-6}$	0.05%
H_2O	STO-3G	2,280	5	2	$6.71 \cdot 10^{-5}$	0.16%
	3-21g	2,280	5	8	$1.02 \cdot 10^{-3}$	0.83%
	6-31g	2,280	5	8	$9.29 \cdot 10^{-4}$	0.72%
CO	STO-3G	1,520	7	3	$3.25 \cdot 10^{-4}$	0.22%
	3-21g	1,520	7	11	$2.37 \cdot 10^{-3}$	1.10%
	6-31g	1,520	7	11	$1.75 \cdot 10^{-3}$	0.80%
CH_4	STO-3G	3,800	5	4	$4.55 \cdot 10^{-4}$	0.80%
	3-21g	3,800	5	12	$2.08 \cdot 10^{-3}$	2.10%
	6-31g	3,800	5	12	$1.46 \cdot 10^{-3}$	1.47%
H_2CO	STO-3G	3,040	8	4	$7.93 \cdot 10^{-4}$	0.63%
	3-21g	3,040	8	14	$2.22 \cdot 10^{-3}$	1.00%
	6-31g	3,040	8	14	$1.89 \cdot 10^{-3}$	0.84%
H_3COH	STO-3G	4,560	9	5	$4.67 \cdot 10^{-4}$	0.50%
	3-21g	4,560	9	17	$1.82 \cdot 10^{-3}$	0.86%
	6-31g	4,560	9	17	$1.61 \cdot 10^{-3}$	0.74%
NH_3	STO-3G	3,040	5	3	$2.38 \cdot 10^{-4}$	0.44%
	3-21g	3,040	5	10	$8.87 \cdot 10^{-4}$	0.79%
	6-31g	3,040	5	10	$7.18 \cdot 10^{-4}$	0.62%
Anthracene	STO-3G	18,240	47	33	$7.35 \cdot 10^{-3}$	0.87%
	3-21g	18,240	47	99	$2.15 \cdot 10^{-2}$	1.76%
	6-31g	18,240	47	99	$1.97 \cdot 10^{-2}$	1.61%
Porphyrine	STO-3G	28,880	81	53	$7.80 \cdot 10^{-3}$	0.51%
	3-21g	28,880	81	163		%
	6-31g	28,880	81	163	$2.49 \cdot 10^{-2}$	1.11%

Table A.1: Correlation energy errors of the Q-MP2 method for smaller and medium size molecules using different basis sets and $20, 38$ molecular Becke weighted grid points. Energy difference against the parent MP2 method in Hartree.

A.2 Grid Benchmark

Basis	N_R	N_L	N_K	ΔE_{avr}	$\sigma(\Delta E_{avr})$	ΔE_{avr}^{rel}	$\sigma(\Delta E_{avr}^{rel})$
STO-3G	20	6	4	$-1.06 \cdot 10^{-2}$	$9.33 \cdot 10^{-3}$	11.39%	11.39%
			6	$-1.06 \cdot 10^{-2}$	$9.36 \cdot 10^{-3}$	11.46%	11.38%
			8	$-1.06 \cdot 10^{-2}$	$9.36 \cdot 10^{-3}$	11.47%	11.38%
			10	$-1.06 \cdot 10^{-2}$	$9.36 \cdot 10^{-3}$	11.47%	11.38%
		18	4	$3.83 \cdot 10^{-4}$	$5.84 \cdot 10^{-4}$	−0.52%	1.03%
			6	$2.10 \cdot 10^{-4}$	$5.14 \cdot 10^{-4}$	−0.36%	0.87%
			8	$2.01 \cdot 10^{-4}$	$5.15 \cdot 10^{-4}$	−0.36%	0.87%
			10	$2.00 \cdot 10^{-4}$	$5.15 \cdot 10^{-4}$	−0.35%	0.87%
		38	4	$-3.91 \cdot 10^{-4}$	$2.47 \cdot 10^{-4}$	0.46%	0.24%
			6	$-4.56 \cdot 10^{-4}$	$2.81 \cdot 10^{-4}$	0.53%	0.24%
			8	$-4.65 \cdot 10^{-4}$	$2.88 \cdot 10^{-4}$	0.54%	0.24%
			10	$-4.66 \cdot 10^{-4}$	$2.89 \cdot 10^{-4}$	0.54%	0.24%
	25	50	4	$-3.42 \cdot 10^{-4}$	$2.26 \cdot 10^{-4}$	0.46%	0.42%
			6	$-4.08 \cdot 10^{-4}$	$2.24 \cdot 10^{-4}$	0.54%	0.40%
			8	$-4.17 \cdot 10^{-4}$	$2.24 \cdot 10^{-4}$	0.54%	0.40%
			10	$-4.18 \cdot 10^{-4}$	$2.24 \cdot 10^{-4}$	0.55%	0.40%
		86	4	$-3.12 \cdot 10^{-4}$	$1.70 \cdot 10^{-4}$	0.39%	0.26%
			6	$-3.77 \cdot 10^{-4}$	$1.88 \cdot 10^{-4}$	0.46%	0.25%
			8	$-3.86 \cdot 10^{-4}$	$1.91 \cdot 10^{-4}$	0.47%	0.24%
			10	$-3.87 \cdot 10^{-4}$	$1.92 \cdot 10^{-4}$	0.47%	0.24%
	30	86	4	$-2.81 \cdot 10^{-4}$	$1.59 \cdot 10^{-4}$	0.35%	0.25%
			6	$-3.47 \cdot 10^{-4}$	$1.75 \cdot 10^{-4}$	0.42%	0.23%
			8	$-3.56 \cdot 10^{-4}$	$1.78 \cdot 10^{-4}$	0.43%	0.23%
			10	$-3.57 \cdot 10^{-4}$	$1.78 \cdot 10^{-4}$	0.43%	0.23%
		110	4	$-2.89 \cdot 10^{-4}$	$1.57 \cdot 10^{-4}$	0.36%	0.24%
			6	$-3.54 \cdot 10^{-4}$	$1.73 \cdot 10^{-4}$	0.43%	0.23%
			8	$-3.63 \cdot 10^{-4}$	$1.77 \cdot 10^{-4}$	0.44%	0.22%
			10	$-3.64 \cdot 10^{-4}$	$1.77 \cdot 10^{-4}$	0.44%	0.22%

TABLE A.2: Average error (in Hartree) and average relative error (in %) of the Q-MP2 correlation energy compared to the parent MP2 method for the STO-3G basis set.

Basis	N_R	N_L	N_K	ΔE_{avr}	$\sigma(\Delta E_{avr})$	ΔE_{avr}^{rel}	$\sigma(\Delta E_{avr}^{rel})$
3-21g	20	6	4	$-1.37 \cdot 10^{-2}$	$1.29 \cdot 10^{-2}$	7.63%	7.11%
			6	$-1.45 \cdot 10^{-2}$	$1.33 \cdot 10^{-2}$	8.12%	7.18%
			8	$-1.47 \cdot 10^{-2}$	$1.35 \cdot 10^{-2}$	8.23%	7.19%
			10	$-1.47 \cdot 10^{-2}$	$1.35 \cdot 10^{-2}$	8.25%	7.19%
		18	4	$-7.30 \cdot 10^{-4}$	$1.44 \cdot 10^{-3}$	0.98%	1.35%
			6	$-1.56 \cdot 10^{-3}$	$1.24 \cdot 10^{-3}$	1.43%	1.11%
			8	$-1.77 \cdot 10^{-3}$	$1.21 \cdot 10^{-3}$	1.54%	1.04%
			10	$-1.81 \cdot 10^{-3}$	$1.20 \cdot 10^{-3}$	1.56%	1.02%
		38	4	$-1.58 \cdot 10^{-3}$	$9.22 \cdot 10^{-4}$	0.95%	0.72%
			6	$-2.42 \cdot 10^{-3}$	$1.41 \cdot 10^{-3}$	1.41%	0.83%
			8	$-2.62 \cdot 10^{-3}$	$1.55 \cdot 10^{-3}$	1.52%	0.87%
			10	$-2.66 \cdot 10^{-3}$	$1.58 \cdot 10^{-3}$	1.54%	0.87%
	25	50	4	$-1.25 \cdot 10^{-3}$	$7.21 \cdot 10^{-4}$	0.78%	0.53%
			6	$-2.09 \cdot 10^{-3}$	$1.20 \cdot 10^{-3}$	1.23%	0.63%
			8	$-2.29 \cdot 10^{-3}$	$1.34 \cdot 10^{-3}$	1.34%	0.66%
			10	$-2.33 \cdot 10^{-3}$	$1.37 \cdot 10^{-3}$	1.36%	0.67%
		86	4	$-1.23 \cdot 10^{-3}$	$7.21 \cdot 10^{-4}$	0.75%	0.46%
			6	$-2.06 \cdot 10^{-3}$	$1.23 \cdot 10^{-3}$	1.20%	0.59%
			8	$-2.27 \cdot 10^{-3}$	$1.37 \cdot 10^{-3}$	1.59%	0.11%
			10	$-2.31 \cdot 10^{-3}$	$1.40 \cdot 10^{-3}$	1.33%	0.65%
	30	86	4	$-1.19 \cdot 10^{-3}$	$7.13 \cdot 10^{-4}$	0.72%	0.45%
			6	$-2.03 \cdot 10^{-3}$	$1.21 \cdot 10^{-3}$	1.17%	0.59%
			8	$-2.23 \cdot 10^{-3}$	$1.36 \cdot 10^{-3}$	1.28%	0.63%
			10	$-2.27 \cdot 10^{-3}$	$1.39 \cdot 10^{-3}$	1.30%	0.64%
		110	4	$-1.17 \cdot 10^{-3}$	$7.06 \cdot 10^{-4}$	0.71%	0.46%
			6	$-2.01 \cdot 10^{-3}$	$1.20 \cdot 10^{-3}$	1.16%	0.59%
			8	$-2.22 \cdot 10^{-3}$	$1.35 \cdot 10^{-3}$	1.27%	0.63%
			10	$-2.25 \cdot 10^{-3}$	$1.38 \cdot 10^{-3}$	1.29%	0.64%

TABLE A.3: Average error (in Hartree) and average relative error (in %) of the Q-MP2 correlation energy compared to the parent MP2 method for the 3-21g basis set.

Basis	N_R	N_L	N_K	ΔE_{avr}	$\sigma(\Delta E_{avr})$	ΔE^{rel}_{avr}	$\sigma(\Delta E^{rel}_{avr})$
6-31g	20	6	4	$-1.63 \cdot 10^{-2}$	$1.16 \cdot 10^{-2}$	9.41%	6.53%
			6	$-1.59 \cdot 10^{-2}$	$1.31 \cdot 10^{-2}$	8.73%	7.82%
			8	$-1.61 \cdot 10^{-2}$	$1.31 \cdot 10^{-2}$	8.79%	7.83%
			10	$-1.61 \cdot 10^{-2}$	$1.31 \cdot 10^{-2}$	8.80%	7.83%
		18	4	$-1.71 \cdot 10^{-4}$	$1.59 \cdot 10^{-3}$	0.33%	1.31%
			6	$-4.84 \cdot 10^{-4}$	$1.31 \cdot 10^{-3}$	0.68%	1.29%
			8	$-6.06 \cdot 10^{-4}$	$1.26 \cdot 10^{-3}$	0.75%	1.24%
			10	$-6.27 \cdot 10^{-4}$	$1.25 \cdot 10^{-3}$	0.76%	1.24%
		38	4	$-1.27 \cdot 10^{-3}$	$7.09 \cdot 10^{-4}$	0.75%	0.48%
			6	$-1.78 \cdot 10^{-3}$	$1.03 \cdot 10^{-3}$	1.03%	0.57%
			8	$-1.91 \cdot 10^{-3}$	$1.11 \cdot 10^{-3}$	1.09%	0.59%
			10	$-1.93 \cdot 10^{-3}$	$1.13 \cdot 10^{-3}$	1.10%	0.59%
	25	50	4	$-9.34 \cdot 10^{-4}$	$5.73 \cdot 10^{-4}$	0.58%	0.47%
			6	$-1.45 \cdot 10^{-3}$	$8.35 \cdot 10^{-4}$	0.86%	0.52%
			8	$-1.57 \cdot 10^{-3}$	$9.08 \cdot 10^{-4}$	0.92%	0.53%
			10	$-1.59 \cdot 10^{-3}$	$9.22 \cdot 10^{-4}$	0.93%	0.53%
		86	4	$-8.97 \cdot 10^{-4}$	$5.56 \cdot 10^{-4}$	0.53%	0.37%
			6	$-1.41 \cdot 10^{-3}$	$8.57 \cdot 10^{-4}$	0.81%	0.44%
			8	$-1.53 \cdot 10^{-3}$	$9.38 \cdot 10^{-4}$	0.87%	0.46%
			10	$-1.55 \cdot 10^{-3}$	$9.53 \cdot 10^{-4}$	0.88%	0.46%
	30	86	4	$-8.70 \cdot 10^{-4}$	$5.44 \cdot 10^{-4}$	0.52%	0.36%
			6	$-1.38 \cdot 10^{-3}$	$8.43 \cdot 10^{-4}$	0.79%	0.43%
			8	$-1.50 \cdot 10^{-3}$	$9.23 \cdot 10^{-4}$	0.85%	0.45%
			10	$-1.52 \cdot 10^{-3}$	$9.37 \cdot 10^{-4}$	0.86%	0.45%
		110	4	$-8.68 \cdot 10^{-4}$	$5.38 \cdot 10^{-4}$	0.52%	0.36%
			6	$-1.38 \cdot 10^{-3}$	$8.37 \cdot 10^{-4}$	0.79%	0.43%
			8	$-1.50 \cdot 10^{-3}$	$9.17 \cdot 10^{-4}$	0.85%	0.45%
			10	$-1.52 \cdot 10^{-3}$	$9.32 \cdot 10^{-4}$	0.86%	0.45%

TABLE A.4: Average error (in Hartree) and average relative error (in %) of the Q-MP2 correlation energy compared to the parent MP2 method for the 6-31g basis set.

A.3 Performance Benchmark

cores/tasks	nodes		$t_{\text{array, hybrid}}$	$t_{\text{array, mpi}}$	$t_{\text{eigen, hybrid}}$	$t_{\text{eigen, mpi}}$
1	1	1	$4.30 \cdot 10^7$	$4.30 \cdot 10^7$	$4.53 \cdot 10^7$	$4.53 \cdot 10^7$
4	1	4	$7.33 \cdot 10^6$	$1.14 \cdot 10^7$	$8.63 \cdot 10^6$	$1.20 \cdot 10^7$
8	1	8	$3.60 \cdot 10^6$	$5.96 \cdot 10^6$	$4.11 \cdot 10^6$	$6.29 \cdot 10^6$
12	1	12	$2.62 \cdot 10^6$	$4.05 \cdot 10^6$	$2.92 \cdot 10^6$	$4.33 \cdot 10^6$
16	1	16	$1.98 \cdot 10^6$	$2.99 \cdot 10^6$	$2.42 \cdot 10^6$	$3.34 \cdot 10^6$
20	1	20	$2.22 \cdot 10^6$	$2.47 \cdot 10^6$	$2.50 \cdot 10^6$	$2.77 \cdot 10^6$
24	1	24	$1.61 \cdot 10^6$	$2.07 \cdot 10^6$	$1.94 \cdot 10^6$	$2.29 \cdot 10^6$
28	1	28	$1.39 \cdot 10^6$	$1.85 \cdot 10^6$	$1.59 \cdot 10^6$	$2.09 \cdot 10^6$
32	1	32	$1.21 \cdot 10^6$		$1.41 \cdot 10^6$	$1.88 \cdot 10^6$
36	1	36	$1.04 \cdot 10^6$	$1.56 \cdot 10^6$	$1.25 \cdot 10^6$	$1.73 \cdot 10^6$
40	1	40	$9.43 \cdot 10^5$	$1.45 \cdot 10^6$	$1.14 \cdot 10^6$	$1.61 \cdot 10^6$
44	1	44	$8.53 \cdot 10^5$	$1.34 \cdot 10^6$	$1.03 \cdot 10^6$	$1.49 \cdot 10^6$
48	1	48	$7.67 \cdot 10^5$	$1.30 \cdot 10^6$	$9.36 \cdot 10^5$	$1.39 \cdot 10^6$
48	2	96	$3.96 \cdot 10^5$	$6.53 \cdot 10^5$	$4.74 \cdot 10^5$	$6.89 \cdot 10^5$
48	3	144	$2.73 \cdot 10^5$	$4.35 \cdot 10^5$	$3.30 \cdot 10^5$	$4.61 \cdot 10^5$
48	4	192	$1.96 \cdot 10^5$	$3.22 \cdot 10^5$	$2.35 \cdot 10^5$	$3.47 \cdot 10^5$
48	5	240	$1.54 \cdot 10^5$	$2.58 \cdot 10^5$	$1.95 \cdot 10^5$	$2.78 \cdot 10^5$
48	6	288	$1.37 \cdot 10^5$	$2.19 \cdot 10^5$	$1.66 \cdot 10^5$	$2.31 \cdot 10^5$
48	7	336	$1.13 \cdot 10^5$	$1.79 \cdot 10^5$	$1.38 \cdot 10^5$	$1.96 \cdot 10^5$
48	8	384	$9.81 \cdot 10^4$	$1.61 \cdot 10^5$	$1.25 \cdot 10^5$	$1.71 \cdot 10^5$
48	9	432	$8.79 \cdot 10^4$	$1.41 \cdot 10^5$	$1.10 \cdot 10^5$	$1.53 \cdot 10^5$
48	10	480	$8.14 \cdot 10^4$	$1.33 \cdot 10^5$	$1.01 \cdot 10^5$	$1.40 \cdot 10^5$
48	11	528	$7.32 \cdot 10^4$	$1.19 \cdot 10^5$	$9.11 \cdot 10^4$	$1.25 \cdot 10^5$
48	12	576	$6.82 \cdot 10^4$	$1.09 \cdot 10^5$	$8.21 \cdot 10^4$	$1.17 \cdot 10^5$
48	13	624	$6.41 \cdot 10^4$	$1.01 \cdot 10^5$	$7.75 \cdot 10^4$	$1.07 \cdot 10^5$
48	14	672	$5.92 \cdot 10^4$	$9.20 \cdot 10^4$	$7.26 \cdot 10^4$	$9.83 \cdot 10^4$
48	15	720	$5.20 \cdot 10^4$	$8.81 \cdot 10^4$	$6.39 \cdot 10^4$	$9.30 \cdot 10^4$
48	16	768	$5.17 \cdot 10^4$	$7.87 \cdot 10^4$	$5.99 \cdot 10^4$	$8.45 \cdot 10^4$
48	17	816	$4.81 \cdot 10^4$	$7.44 \cdot 10^4$	$5.84 \cdot 10^4$	$7.92 \cdot 10^4$
48	18	864	$4.54 \cdot 10^4$	$7.06 \cdot 10^4$	$5.30 \cdot 10^4$	$7.52 \cdot 10^4$
48	19	912	$4.28 \cdot 10^4$	$6.57 \cdot 10^4$	$5.01 \cdot 10^4$	$7.01 \cdot 10^4$
48	20	960	$4.23 \cdot 10^4$	$6.60 \cdot 10^4$	$4.75 \cdot 10^4$	$7.03 \cdot 10^4$

TABLE A.5: Main workload timings of Q-MP2 for the porphyrine system in ms on the JUSTUS2 cluster system.

cores/tasks	nodes		$t_{\text{array, hybrid}}$	$t_{\text{array, mpi}}$	$t_{\text{eigen, hybrid}}$	$t_{\text{eigen, mpi}}$
1	1	1	$2.66 \cdot 10^7$	$2.66 \cdot 10^7$	$2.94 \cdot 10^7$	$2.94 \cdot 10^7$
4	1	4	$4.50 \cdot 10^6$	$6.83 \cdot 10^6$	$5.44 \cdot 10^6$	$7.55 \cdot 10^6$
8	1	8	$2.06 \cdot 10^6$	$3.58 \cdot 10^6$	$2.59 \cdot 10^6$	$3.91 \cdot 10^6$
12	1	12	$1.39 \cdot 10^6$	$2.40 \cdot 10^6$	$1.82 \cdot 10^6$	$2.70 \cdot 10^6$
16	1	16	$1.11 \cdot 10^6$	$1.87 \cdot 10^6$	$1.49 \cdot 10^6$	$2.07 \cdot 10^6$
20	1	20	$9.40 \cdot 10^5$	$1.51 \cdot 10^6$	$1.28 \cdot 10^6$	$1.71 \cdot 10^6$
24	1	24	$8.80 \cdot 10^5$	$1.30 \cdot 10^6$	$1.11 \cdot 10^6$	$1.47 \cdot 10^6$
28	1	28	$7.86 \cdot 10^5$	$1.25 \cdot 10^6$	$9.79 \cdot 10^5$	$1.31 \cdot 10^6$
32	1	32	$6.98 \cdot 10^5$	$1.04 \cdot 10^6$	$8.32 \cdot 10^5$	$1.18 \cdot 10^6$
36	1	36	$5.91 \cdot 10^5$	$9.42 \cdot 10^5$	$7.54 \cdot 10^5$	$1.10 \cdot 10^6$
40	1	40	$5.34 \cdot 10^5$	$8.64 \cdot 10^5$	$6.72 \cdot 10^5$	$1.00 \cdot 10^6$
44	1	44	$5.05 \cdot 10^5$	$8.17 \cdot 10^5$	$6.31 \cdot 10^5$	$9.38 \cdot 10^5$
48	1	48	$4.52 \cdot 10^5$	$7.84 \cdot 10^5$	$5.75 \cdot 10^5$	$8.64 \cdot 10^5$
48	2	96	$2.27 \cdot 10^5$	$3.84 \cdot 10^5$		$5.06 \cdot 10^5$
48	3	144	$1.65 \cdot 10^5$	$2.53 \cdot 10^5$	$1.90 \cdot 10^5$	$2.91 \cdot 10^5$
48	4	192	$1.11 \cdot 10^5$	$1.95 \cdot 10^5$	$1.48 \cdot 10^5$	$2.14 \cdot 10^5$
48	5	240	$8.96 \cdot 10^4$	$1.58 \cdot 10^5$	$1.15 \cdot 10^5$	$1.76 \cdot 10^5$
48	6	288	$7.38 \cdot 10^4$	$1.26 \cdot 10^5$	$9.87 \cdot 10^4$	$1.43 \cdot 10^5$
48	7	336	$6.67 \cdot 10^4$	$1.08 \cdot 10^5$	$8.12 \cdot 10^4$	$1.24 \cdot 10^5$
48	8	384	$5.69 \cdot 10^4$	$9.28 \cdot 10^4$	$7.26 \cdot 10^4$	$1.07 \cdot 10^5$
48	9	432	$4.95 \cdot 10^4$	$8.63 \cdot 10^4$	$6.39 \cdot 10^4$	$9.67 \cdot 10^4$
48	10	480	$4.43 \cdot 10^4$	$7.63 \cdot 10^4$	$5.77 \cdot 10^4$	$8.71 \cdot 10^4$
48	11	528	$4.13 \cdot 10^4$	$6.94 \cdot 10^4$	$5.26 \cdot 10^4$	$7.88 \cdot 10^4$
48	12	576	$3.91 \cdot 10^4$	$6.01 \cdot 10^4$	$4.90 \cdot 10^4$	$6.83 \cdot 10^4$
48	13	624	$3.38 \cdot 10^4$	$5.83 \cdot 10^4$	$4.45 \cdot 10^4$	$6.42 \cdot 10^4$
48	14	672	$3.36 \cdot 10^4$	$5.22 \cdot 10^4$	$4.21 \cdot 10^4$	$5.97 \cdot 10^4$
48	15	720	$3.12 \cdot 10^4$	$5.01 \cdot 10^4$	$3.84 \cdot 10^4$	$5.51 \cdot 10^4$
48	16	768	$2.90 \cdot 10^4$	$4.44 \cdot 10^4$	$3.57 \cdot 10^4$	$5.06 \cdot 10^4$
48	17	816	$2.61 \cdot 10^4$	$4.57 \cdot 10^4$	$3.39 \cdot 10^4$	$5.10 \cdot 10^4$
48	18	864	$2.59 \cdot 10^4$	$4.00 \cdot 10^4$	$3.19 \cdot 10^4$	$4.55 \cdot 10^4$
48	19	912	$2.47 \cdot 10^4$	$4.11 \cdot 10^4$	$3.09 \cdot 10^4$	$4.63 \cdot 10^4$
48	20	960	$2.22 \cdot 10^4$	$3.63 \cdot 10^4$	$2.99 \cdot 10^4$	$4.14 \cdot 10^4$

TABLE A.6: Main workload timings of Q-MP2 for the anthracene system in ms on the JUSTUS2 cluster system.

cores/tasks	nodes		$t_{\text{array, hybrid}}$	$t_{\text{array, mpi}}$	$t_{\text{eigen, hybrid}}$	$t_{\text{eigen, mpi}}$
1	1	1	$6.58 \cdot 10^4$	$6.58 \cdot 10^4$	$6.57 \cdot 10^4$	$6.57 \cdot 10^4$
4	1	4	$1.73 \cdot 10^4$	$1.65 \cdot 10^4$	$1.74 \cdot 10^4$	$1.65 \cdot 10^4$
8	1	8	$8.94 \cdot 10^3$	$8.74 \cdot 10^3$	$8.93 \cdot 10^3$	$8.83 \cdot 10^3$
12	1	12	$6.53 \cdot 10^3$	$6.05 \cdot 10^3$	$6.57 \cdot 10^3$	$6.04 \cdot 10^3$
16	1	16	$5.36 \cdot 10^3$	$4.71 \cdot 10^3$	$5.33 \cdot 10^3$	$4.66 \cdot 10^3$
20	1	20	$4.66 \cdot 10^3$	$4.06 \cdot 10^3$	$4.66 \cdot 10^3$	$4.00 \cdot 10^3$
24	1	24	$4.04 \cdot 10^3$	$3.41 \cdot 10^3$	$4.04 \cdot 10^3$	$3.42 \cdot 10^3$
28	1	28	$3.42 \cdot 10^3$	$3.18 \cdot 10^3$	$3.42 \cdot 10^3$	$3.18 \cdot 10^3$
32	1	32	$2.99 \cdot 10^3$	$2.95 \cdot 10^3$	$2.91 \cdot 10^3$	$2.89 \cdot 10^3$
36	1	36	$2.65 \cdot 10^3$	$2.76 \cdot 10^3$	$2.63 \cdot 10^3$	$2.77 \cdot 10^3$
40	1	40	$2.38 \cdot 10^3$	$2.51 \cdot 10^3$	$2.40 \cdot 10^3$	$2.56 \cdot 10^3$
44	1	44	$2.23 \cdot 10^3$	$2.34 \cdot 10^3$	$2.26 \cdot 10^3$	$2.33 \cdot 10^3$
48	1	48	$2.03 \cdot 10^3$	$2.18 \cdot 10^3$	$2.04 \cdot 10^3$	$2.15 \cdot 10^3$
48	2	96	$1.02 \cdot 10^3$	$1.05 \cdot 10^3$		$1.06 \cdot 10^3$
48	3	144	$6.93 \cdot 10^2$	$6.92 \cdot 10^2$	$6.97 \cdot 10^2$	$6.86 \cdot 10^2$
48	4	192	$5.16 \cdot 10^2$	$5.08 \cdot 10^2$	$5.56 \cdot 10^2$	$5.03 \cdot 10^2$
48	5	240	$4.17 \cdot 10^2$	$4.11 \cdot 10^2$	$4.18 \cdot 10^2$	$4.14 \cdot 10^2$
48	6	288	$3.65 \cdot 10^2$	$3.20 \cdot 10^2$	$3.67 \cdot 10^2$	$3.19 \cdot 10^2$
48	7	336	$3.10 \cdot 10^2$	$2.76 \cdot 10^2$	$3.17 \cdot 10^2$	$2.75 \cdot 10^2$
48	8	384	$2.66 \cdot 10^2$	$2.28 \cdot 10^2$	$2.63 \cdot 10^2$	$2.28 \cdot 10^2$
48	9	432	$2.44 \cdot 10^2$	$2.29 \cdot 10^2$	$2.36 \cdot 10^2$	$2.30 \cdot 10^2$
48	10	480	$2.14 \cdot 10^2$	$1.83 \cdot 10^2$	$2.34 \cdot 10^2$	$1.85 \cdot 10^2$
48	11	528	$2.02 \cdot 10^2$	$1.83 \cdot 10^2$	$2.10 \cdot 10^2$	$1.82 \cdot 10^2$
48	12	576	$1.77 \cdot 10^2$	$1.38 \cdot 10^2$	$1.88 \cdot 10^2$	$1.38 \cdot 10^2$
48	13	624	$1.58 \cdot 10^2$	$1.37 \cdot 10^2$	$1.60 \cdot 10^2$	$1.37 \cdot 10^2$
48	14	672	$1.52 \cdot 10^2$	$1.37 \cdot 10^2$	$1.49 \cdot 10^2$	$1.37 \cdot 10^2$
48	15	720	$1.43 \cdot 10^2$	$1.38 \cdot 10^2$	$1.48 \cdot 10^2$	$1.39 \cdot 10^2$
48	16	768	$1.40 \cdot 10^2$	$9.10 \cdot 10^1$	$1.32 \cdot 10^2$	$9.10 \cdot 10^1$
48	17	816	$1.49 \cdot 10^2$	$9.20 \cdot 10^1$	$1.23 \cdot 10^2$	$9.10 \cdot 10^1$
48	18	864	$1.29 \cdot 10^2$	$9.10 \cdot 10^1$	$1.33 \cdot 10^2$	$9.10 \cdot 10^1$
48	19	912	$1.15 \cdot 10^2$	$9.20 \cdot 10^1$	$1.31 \cdot 10^2$	$9.10 \cdot 10^1$
48	20	960	$1.16 \cdot 10^2$	$9.10 \cdot 10^1$	$1.12 \cdot 10^2$	$9.20 \cdot 10^1$

Table A.7: Main workload timings of Q-MP2 for the H_3COH system in ms on the JUSTUS2 cluster system.

cores/tasks	nodes		$t_{\text{array, hybrid}}$	$t_{\text{array, mpi}}$	$t_{\text{eigen, hybrid}}$	$t_{\text{eigen, mpi}}$
1	1	1	$7.54 \cdot 10^2$	$7.54 \cdot 10^2$	$1.34 \cdot 10^3$	$1.34 \cdot 10^3$
4	1	4	$2.01 \cdot 10^2$	$1.91 \cdot 10^2$	$3.63 \cdot 10^2$	$3.38 \cdot 10^2$
8	1	8	$1.02 \cdot 10^2$	$1.02 \cdot 10^2$	$1.92 \cdot 10^2$	$1.81 \cdot 10^2$
12	1	12	$6.84 \cdot 10^1$	$6.89 \cdot 10^1$	$1.44 \cdot 10^2$	$1.23 \cdot 10^2$
16	1	16	$5.75 \cdot 10^1$	$5.15 \cdot 10^1$	$1.19 \cdot 10^2$	$9.44 \cdot 10^1$
20	1	20	$5.14 \cdot 10^1$	$4.15 \cdot 10^1$	$1.31 \cdot 10^2$	$7.84 \cdot 10^1$
24	1	24	$4.67 \cdot 10^1$	$3.56 \cdot 10^1$	$8.94 \cdot 10^1$	$6.46 \cdot 10^1$
28	1	28	$4.61 \cdot 10^1$	$3.12 \cdot 10^1$	$8.03 \cdot 10^1$	$6.12 \cdot 10^1$
32	1	32	$3.98 \cdot 10^1$	$2.91 \cdot 10^1$	$1.06 \cdot 10^2$	$5.45 \cdot 10^1$
36	1	36	$3.45 \cdot 10^1$	$2.57 \cdot 10^1$	$2.54 \cdot 10^2$	$5.25 \cdot 10^1$
40	1	40	$3.02 \cdot 10^1$	$2.44 \cdot 10^1$	$5.46 \cdot 10^1$	$4.99 \cdot 10^1$
44	1	44	$2.59 \cdot 10^1$	$2.19 \cdot 10^1$	$5.43 \cdot 10^1$	$4.86 \cdot 10^1$
48	1	48	$2.35 \cdot 10^1$	$2.03 \cdot 10^1$	$5.21 \cdot 10^1$	$4.10 \cdot 10^1$
48	2	96	$1.16 \cdot 10^1$	$1.04 \cdot 10^1$		$2.04 \cdot 10^1$
48	3	144	$7.34 \cdot 10^0$	$6.69 \cdot 10^0$	$1.74 \cdot 10^1$	$1.33 \cdot 10^1$
48	4	192	$5.92 \cdot 10^0$	$4.75 \cdot 10^0$	$1.60 \cdot 10^1$	$9.75 \cdot 10^0$
48	5	240	$4.79 \cdot 10^0$	$3.81 \cdot 10^0$	$1.05 \cdot 10^1$	$7.83 \cdot 10^0$
48	6	288	$3.80 \cdot 10^0$	$2.83 \cdot 10^0$	$9.23 \cdot 10^0$	$6.03 \cdot 10^0$
48	7	336	$3.68 \cdot 10^0$	$2.81 \cdot 10^0$	$7.72 \cdot 10^0$	$5.41 \cdot 10^0$
48	8	384	$3.36 \cdot 10^0$	$1.93 \cdot 10^0$	$8.56 \cdot 10^0$	$4.26 \cdot 10^0$
48	9	432	$2.79 \cdot 10^0$	$1.87 \cdot 10^0$	$6.58 \cdot 10^0$	$4.39 \cdot 10^0$
48	10	480	$2.70 \cdot 10^0$	$1.90 \cdot 10^0$	$5.80 \cdot 10^0$	$4.27 \cdot 10^0$
48	11	528	$2.29 \cdot 10^0$	$1.89 \cdot 10^0$	$6.05 \cdot 10^0$	$3.62 \cdot 10^0$
48	12	576	$2.40 \cdot 10^0$	$1.03 \cdot 10^0$	$5.94 \cdot 10^0$	$2.79 \cdot 10^0$
48	13	624	$1.89 \cdot 10^0$	$1.00 \cdot 10^0$	$4.99 \cdot 10^0$	$2.48 \cdot 10^0$
48	14	672	$1.89 \cdot 10^0$	$1.03 \cdot 10^0$	$4.50 \cdot 10^0$	$2.42 \cdot 10^0$
48	15	720	$1.81 \cdot 10^0$	$1.23 \cdot 10^0$	$5.75 \cdot 10^0$	$1.89 \cdot 10^0$
48	16	768	$1.89 \cdot 10^0$	$9.94 \cdot 10^{-1}$	$5.17 \cdot 10^0$	$2.55 \cdot 10^0$
48	17	816	$1.64 \cdot 10^0$	$9.12 \cdot 10^{-1}$	$5.37 \cdot 10^0$	$1.86 \cdot 10^0$
48	18	864	$1.48 \cdot 10^0$	$9.04 \cdot 10^{-1}$	$6.17 \cdot 10^0$	$2.36 \cdot 10^0$
48	19	912	$1.46 \cdot 10^0$	$1.00 \cdot 10^0$	$3.85 \cdot 10^0$	$2.40 \cdot 10^0$
48	20	960	$1.42 \cdot 10^0$	$1.00 \cdot 10^0$	$4.29 \cdot 10^0$	$2.72 \cdot 10^0$

TABLE A.8: Main workload timings of Q-MP2 for the H_2O system in ms on the JUSTUS2 cluster system.

Appendix B

Additional Code Listings

B.1 Implementation

LISTING B.1: Example `GitHub` action executing the CI pipeline

```
name: CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: configure
      run: ./configure
    - name: make
      run: cd build && make
    - name: ctest
      run: cd build && ctest -V
```

LISTING B.2: Example run output for the anthracene $(6-31G)$ test system, using the array variant and the hybrid OpenMP+MPI parallelization scheme. The run was executed on 19 nodes.

```
/home/hd/hd_hd/hd_tb381/workspace/libqqc/runs/scan_hybrid
n1405
Fr 15. Apr 15:46:20 CEST 2022
#!/bin/bash
#SBATCH -J libqqc_hybrid_20x48
#SBATCH -n20 --cpus-per-task=48 -t12:00:00
#SBATCH --mem=200G
#SBATCH --job-name=libqqc_hybrid_20x48
#SBATCH --output=libqqc_hybrid_20x48_por-%j.out
#SBATCH --error=libqqc_hybrid_20x48_por-%j.err
pwd; hostname; date
cat run_20x48.sh
module load compiler/gnu
module load mpi/openmpi
cd ../../host_example/.
mpic++ -fopenmp -O3 -ffast-math -march=native main_mpi.cpp ../lib/↩
    libqqc.a -o 20x48.out
OMP_NUM_THREADS=48 srun --cpus-per-task=$OMP_NUM_THREADS ./20x48.↩
    out
rm 20x48.out
Timing Loader_qmp2_from_file:: loader :CPU: 0 ms ; Wall: 0 ms
Timings AoToMo CGTO: CPU: 1740 ms ; Wall: 831 ms
    -- Loading in: CPU: 744 ms ; Wall: 745 ms
    -- Distribute Batch: CPU: 8 ms ; Wall: 7 ms
    -- Tranforming Batch: CPU: 398 ms ; Wall: 9 ms
    -- Distribute Batch Results: CPU: 311 ms ; Wall: 39 ms

Timings AoToMo Coulomb Integral: CPU: 172604 ms ; Wall: 108470 ms
    -- Loading in: CPU: 105441 ms ; Wall: 105563 ms
    -- Distribute Batch: CPU: 601 ms ; Wall: 602 ms
    -- Transforming Batch: CPU: 65355 ms ; Wall: 1366 ms
    -- Distribute Batch Results: CPU: 1129 ms ; Wall: 857 ms

Timing Vault_qmp2:: vault : CPU: 174743 ms ; Wall: 109366 ms
Timing Do_qmp2:: qmp2 : CPU: 1061861 ms ; Wall: 23351 ms
 -----------------------------------------------------------------
|                                                                 |
|                 ** Quadrature Calculation through libqqc **     |
|                                                                 |
|+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|
| * Author(s): Benjamin Thomitzni                                    |
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|
|                              Programm Details                      |
|--------------------------------------------------------------------|
| * library vers.      v0.1                                          |
|     -- Loader vers.  v0.1                                          |
|     -- Vault vers.   v0.1                                          |
|     -- Do_qmp2 vers. v0.1                                          |
|--------------------------------------------------------------------|
| * OpenMP enabled                                                   |
|     -- threads:   48                                               |
| * MPI enabled                                                      |
|     -- processes: 20                                               |
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|
|                             Calculation Details                    |
|--------------------------------------------------------------------|
| * Method:              Q-MP(2)                                     |
|--------------------------------------------------------------------|
| * grid points (3D):    18240                                       |
| * grid points (1D):    16                                          |
|--------------------------------------------------------------------|
| * atomic orbitals:     146                                         |
| * molecular orbitals:  146                                         |
|     -- occupied:       47                                          |
|     -- virtual:        99                                          |
|--------------------------------------------------------------------|
|                                   Results                          |
|--------------------------------------------------------------------|
| * Ground State Energy Correction (eV): -1.24223                    |
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|
|                             Calculation Timings                    |
|--------------------------------------------------------------------|
| Timing Qmp2_energy::compute : CPU: 1060257 ms ; Wall: 22236 ms     |
|     -- Setting up calculation: CPU: 805 ms ; Wall: 17 ms           |
|     -- Calc. Batch 1/2: CPU: 11612 ms ; Wall: 242 ms               |
|     -- Calc. Batch 2/2: CPU: 1047840 ms ; Wall: 21976 ms           |
| Gather result fr. nodes : CPU: 1348 ms ; Wall: 1100 ms             |
|--------------------------------------------------------------------|
|             if no timings are printed, choose higher print level|
|__________________________________________________________________|
```

Listing B.3: Example call to load in the AO integral data `Q-Chem`

```
//initializing memory on heap due to size
double* coul_ao = new double[npts * nao * nao];
libqints::basis_1e1c_cgto<double> basis = libqints::qchem::aobasis.↩
    b1;
std::vector< libqints::ftype_multipole > v_pts;
for (int i = 0; i < npts; i++){
    libqints::ftype_multipole f_mult;
    f_mult.x = grid.get_mpts()[i * 3 + 0];
    f_mult.y = grid.get_mpts()[i * 3 + 1];
    f_mult.z = grid.get_mpts()[i * 3 + 2];
    f_mult.k = 0;
    v_pts.push_back(f_mult);
}//for i
libqints::basis_1e1c_multipole bm (v_pts);
double mom[npts];
for (int i = 0; i < npts; i++) mom[i] = 1;
libqints::array_view<double> av_mom((double*) &mom[0], npts);
libqints::dev_omp dev;
dev.init(256UL * 1024 * 1024);
libqints::array_view<double> av_vmul(coul_ao, npts*nao*nao);
//This is the digester, which we grabbed from libfock and modified
//it for out integral. This gives us back the evaluated AO ↩
    integrals
qqc_vmul<double>(basis, bm, dev).compute(av_mom, av_vmul);
```

Listing B.4: Example fail state of the testing suite automatically executed by each push to the main branch of the repository.

```
    Test project /home/bentho/workspace/libqqc/build
    Start 1: LIBQQC_LOADER_TESTS
1/5 Test #1: LIBQQC_LOADER_TESTS .............   Passed 0.25 sec
    Start 2: LIBQQC_GRIDS_TESTS
2/5 Test #2: LIBQQC_GRIDS_TESTS ..............   Passed 0.00 sec
    Start 3: LIBQQC_METHODS_TESTS
3/5 Test #3: LIBQQC_METHODS_TESTS ............   Passed 2.96 sec
    Start 4: LIBQQC_UTILS_TESTS
4/5 Test #4: LIBQQC_UTILS_TESTS ..............***Failed 0.54 sec
Performing tests for libqqc/utils ...

Testing utils/ttimer.h, Tclock::
    Testing Tclock::get_mrun()         ... passed
    Testing Tclock::stop_clock()       ... passed
    Testing Tclock::start_clock()       ... passed
    Testing Tclock::wall_time()         ... passed
    Testing Tclock::cpu_time()          ... passed
    Testing Tclock::print_time()        ... failed
... some failed

Testing utils/ttimer.h, Ttimer::
    Testing Ttimer::get_clocks()        ... passed
    Testing Ttimer::start_new_clock()   ... passed
    Testing Ttimer::stop_clock()        ... passed
    Testing Ttimer::stop_all_clocks()   ... passed
    Testing Ttimer::print_clocks()      ... failed
    Testing Ttimer::print_all_clocks()  ... failed
... some failed

Testing utils/load_from_file.h,
    Testing load_dim_from_file()        ... passed
    Testing load_array_from_file()      ... passed
... all passed

    Start 5: LIBQQC_VAULTS_TESTS
5/5 Test #5: LIBQQC_VAULTS_TESTS .............   Passed 0.00 sec

80% tests passed, 1 tests failed out of 5

Total Test time (real) =   3.76 sec

The following tests FAILED:
      4 - LIBQQC_UTILS_TESTS (Failed)
Errors while running CTest
```