Goals of Quantum Chemical Method Development

- To be able to calculate 'things' (energies, properties) that could not be calculated before or on systems that were not accessible before
- To develop a better (more accurate, more elegant, more compact, more transparent, ...) theory for a known property.
- Develop new approximations to known equations
- To obtain the same number faster than before
- To obtain an approximate number faster and in 'improved scaling' than before
Scaling Laws

A quantum chemical algorithm can be characterized by it’s scaling behavior:

- Scaling with respect to system size (#(Atoms), #(Basis functions),…)
- Scaling with respect to basis set (Size, Angular momentum,…)

A scaling law can be written as:

\[ T = a N^b \]

- \( T \) : Time taken by algorithm
- \( a \) : ‘Prefactor’
- \( b \) : Scaling Exponent

Optimizing an algorithm:
- Bring down the prefactor
- Bring down the scaling

Holy grail: \( Linear scaling \) with a small prefactor
Figuring out the Scaling Law

General:

**Dimensionality of target quantity** $\times$ **Scaling of loops required to obtain it**

Example:

$$\psi_p (r) = \sum_{\mu} c_{\mu p} \varphi_p (r)$$

- Number of occupied *and* virtual MOs is proportional to system size
- Number of AOs is proportional to system size

$$\left( \mu \nu \mid \kappa \tau \right)$$  Number of AOs integrals proportional to $N^4$ ($O(N^4)$)

$$\left( i a \mid j b \right) = \sum_{\mu} \sum_{\nu} \sum_{\kappa} \sum_{\tau} c_{\mu i} c_{\nu a} c_{\kappa j} c_{\tau b} \left( \mu \nu \mid \kappa \tau \right)$$

$O(N^4) \quad O(N) \quad O(N) \quad O(N) \quad O(N) \rightarrow O(N^4)$

$O(N^8)$
Prefactor vs Scaling

For many applications nonlinear scaling with a small prefecture is the preferred choice.

In developing reduced scaling algorithms one shoots for *early crossover*. 

---

crossover point
In general, the workflow of a quantum chemical algorithm contains many steps (e.g. localization, integral transformation, equation solution, perturbative correction, …),

Each step will have its own scaling law
### Profile your Program!

<table>
<thead>
<tr>
<th>Task</th>
<th>Time</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total execution time</td>
<td>153019.575 sec</td>
<td></td>
</tr>
<tr>
<td>Localization of occupied MO's</td>
<td>7516.449 sec</td>
<td>4.9%</td>
</tr>
<tr>
<td>Fock Matrix Formation</td>
<td>11392.614 sec</td>
<td>7.4%</td>
</tr>
<tr>
<td><strong>First Half Transformation</strong></td>
<td>37824.285 sec</td>
<td>24.7%</td>
</tr>
<tr>
<td>RI-PNO integral transformation</td>
<td>17832.376 sec</td>
<td>11.7%</td>
</tr>
<tr>
<td>Initial Guess</td>
<td>5376.961 sec</td>
<td>3.5%</td>
</tr>
<tr>
<td>DIIS Solver</td>
<td>8855.850 sec</td>
<td>5.8%</td>
</tr>
<tr>
<td>State Vector Update</td>
<td>1.744 sec</td>
<td>0.0%</td>
</tr>
<tr>
<td>Sigma-vector construction</td>
<td>8177.969 sec</td>
<td>5.3%</td>
</tr>
<tr>
<td>\langle 0</td>
<td>H</td>
<td>D \rangle</td>
</tr>
<tr>
<td>\langle 0</td>
<td>H</td>
<td>S \rangle</td>
</tr>
<tr>
<td>\langle D</td>
<td>H</td>
<td>D \rangle(0-ext)</td>
</tr>
<tr>
<td>\langle D</td>
<td>H</td>
<td>D \rangle(2-ext Fock)</td>
</tr>
<tr>
<td>\langle D</td>
<td>H</td>
<td>D \rangle(2-ext)</td>
</tr>
<tr>
<td>\langle D</td>
<td>H</td>
<td>D \rangle(4-ext)</td>
</tr>
<tr>
<td>\langle D</td>
<td>H</td>
<td>D \rangle(4-ext-corr)</td>
</tr>
<tr>
<td>CCSD doubles correction</td>
<td>33.534 sec</td>
<td>0.4% of sigma</td>
</tr>
<tr>
<td>\langle S</td>
<td>H</td>
<td>S \rangle</td>
</tr>
<tr>
<td>\langle S</td>
<td>H</td>
<td>D \rangle(1-ext)</td>
</tr>
<tr>
<td>\langle D</td>
<td>H</td>
<td>S \rangle(1-ext)</td>
</tr>
<tr>
<td>\langle S</td>
<td>H</td>
<td>D \rangle(3-ext)</td>
</tr>
<tr>
<td>CCSD singles correction</td>
<td>0.108 sec</td>
<td>0.0% of sigma</td>
</tr>
<tr>
<td>Fock-dressing</td>
<td>1541.152 sec</td>
<td>18.8% of sigma</td>
</tr>
<tr>
<td>Singles amplitudes</td>
<td>15.255 sec</td>
<td>0.2% of sigma</td>
</tr>
<tr>
<td>(ik</td>
<td>jl)-dressing</td>
<td>441.823 sec</td>
</tr>
<tr>
<td>(ij</td>
<td>ab),(ia</td>
<td>jb)-dressing</td>
</tr>
<tr>
<td>Pair energies</td>
<td>1.235 sec</td>
<td>0.0% of sigma</td>
</tr>
<tr>
<td><strong>Total Time for the density</strong></td>
<td>632.934 sec</td>
<td>0.4% of ALL</td>
</tr>
<tr>
<td><strong>Total Time for computing (T)</strong></td>
<td>32529.433 sec</td>
<td>21.3% of ALL</td>
</tr>
</tbody>
</table>
Prelude:

A little information on Computers
Growth in clock rate of microprocessors. Between 1978 and 1986, the clock rate improved less than 15% per year while performance improved by 25% per year. During the “renaissance period” of 52% performance improvement per year between 1986 and 2003, clock rates shot up almost 40% per year. Since then, the clock rate has been nearly flat, growing at less than 1% per year, while single processor performance improved at less than 22% per year.
So far consistent with Moore’s law (processor performance doubles every 12-24 months)

Optimistic estimates claim that Moore’s law can be fulfilled until ~2020-2030

Physical limits of miniturization will ultimately be reached
Consequence’s of Moore’s Law

Paradigm Change:

Requires explicit parallelization by the programmer!

“From this historical perspective, it’s startling that the whole IT industry has bet its future that programmers will finally successfully switch to explicitly parallel programming”

(Patterson, Hennessy: The Hardware/Software Interface, 2009)
Amdahl’s Law of Diminishing returns

\[
S(N) = \frac{1}{(1 - P) + \frac{P}{N}}.
\]

P: Parallel portion of code
N: Number of Processors
Disk access is very slow
Mem. to CPU transfer is slow

Algorithms need to carefully balance I/O and memory operations, not just minimize FLOP count.
Development Guidelines for Quantum Chemistry

„Getting Exactly the same number faster“
Who are you developing for?

✓ For yourself because you want to check out some ideas  Everything is ok!

✓ Just for a paper, but not to be used later  Mostly anything is ok!

✓ For your boss because you want to get a Ph.D.  ... depends on your boss

✓ For a program package that is supposed to be long lived

  ... it needs to be well documented (in english)
  ... don’t try to be funny!
  ... Write the FM. (so that users can avoid reading the FM)
  ... Make sure it compiles on any platform
  ... Minimize the dependence on elements that are outside your control
  ... put effort into making it as efficient as possible
The Do’s and Don’t’s of Programming: Overview

Some rules for scientific programming that are relevant for obtaining high performance:

- Avoid short, nested Loops
- Avoid Multidimensional Arrays
- Access arrays in "Unit Stride"
- Avoid indirect addressing
- Make use of matrix multiplications and BLAS
- Make use of LAPACK
- Move redundant work out of the inner loops
- Minimize disk I/O, do it in larger chunks and do it as far 'outside' as possible
- Watch out of Load Balancing in parallel programming
Instruction Pipelines and Logic

Ideal: The CPU has preloaded a ‘pipeline’ of instructions and the data required to perform the next operations is in the CACHE.

A logical instruction whose outcome can not be predicted at compile time brings the CPU and CACHE out of the ‘groove’.

```
if (x<y)
  z=x+y
else
  n=n+1
  xp= sin(2*yp)
end
```

GOOD: x,y,z are in the CACHE, performance is optimal.

BAD: xp, yp and n are not in the CACHE. The pipeline must be cleared and a slow memory operation (MOP) is performed to get this data.

careful optimization avoids logical decisions in time critical parts of the program.
Unit Stride Access

The CACHE has a finite size that is rather small. If one loads an array into the CACHE that is larger than the CACHE size, one should avoid 'jumping' around in the array but rather only access consecutive positions in the array (unit stride access).

**Example**: Say, the CACHE holds 1024 array elements and we want to add up the elements of an array y that contains 2048 elements.

**Good:**
\[
\begin{align*}
x &= 0 \\
\text{for } (i=0; i<2048; i++) \quad x &= x + y[i]
\end{align*}
\]

- The compiler can optimize well: load the first 1024 elements of y and the next 1024 elements. Performs optimally without any 'CACHE misses'.

**Bad:**
\[
\begin{align*}
x &= 0 \\
\text{for } (i=0; i<2048; i++) \quad x &= x + y[yorder[i]] \\
\text{or for } (i=0; i<2048; i++) \quad x &= x + y[i] - y[N-i-1]
\end{align*}
\]

Two problems:
- yorder[i] may be anything in the range 0..2047 for any i and hence we may have to reload y into the CACHE multiple times
- We use 'indirect addressing'. There is no way for the compiler to know the value of yorder[i] and hence after each addition we have to look again which element of y we need next.
Libraries: The only ones you really need

Relying on third party software that may or may not be maintained in long term or may or may not be portable between platforms can be dangerous! There are three you likely cannot avoid:

1. **BLAS (Basic Linear Algebra System)**
   a) **Level 1**: Vector/Vector operations
   b) **Level 2**: Matrix/Vector operations
   c) **Level 3**: Matrix/Matrix operations

2. **LAPACK (Linear Algebra Package)**
   Linear algebra routines (Diagonalization, Linear equation systems, Cholesky decomposition, singular value decomposition, …)

3. **MPI (Message Passing Interface)**
   Low level routines for parallelization using a distributed memory paradigm

*These are highly efficient, standardized and portable libraries.*

(In ORCA, we nevertheless have put one software layer above them in order to have no direct calls to third party software whatsoever)
Example: The power of BLAS

Let us look at two 'innocent' matrix multiplications:

\[ C = AB \quad C_{ij} = \sum_k A_{ik} B_{kj} \]

\[ C = AB^T \quad C_{ij} = \sum_k A_{ik} B_{jk} \]

Which we can program as follows:

```plaintext
loop i = 1 ... N
  loop j = 1 ... N
    x=0.0;
    loop k = 1 ... N
      x=x+A(i,k)*B(k,j); or x=x+A(i,k)*B(j,k)
    end loop k
    C(i,j)=x;
  end loop j
end loop i
```
Example: The power of BLAS (II)

For two densely filled essentially random, square matrices $A$ and $B$ with $N=2750$

<table>
<thead>
<tr>
<th>directly programmed</th>
<th>BLAS (dgemm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C = AB$</td>
<td>99</td>
</tr>
<tr>
<td>$C = AB^T$</td>
<td>11</td>
</tr>
<tr>
<td>$C = A^T B$</td>
<td>104</td>
</tr>
</tbody>
</table>

Why that?

- The matrices are arranged row-wise in contiguous memory places. Hence $A(i,k)$ is accessing the matrix in unit stride while $A(k,i)$ is not!
- Huge (factor 10!) performance penalty!
- Even worse would be to have rows scattered somewhere in the main memory (e.g. Numerical Recipes matrix routines in C)

$A =$

Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz
Example: The power of LAPACK

Example: 3000x3000 matrix

<table>
<thead>
<tr>
<th></th>
<th>Hand written</th>
<th>Intel-MKL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Diagonalization</strong></td>
<td>28.1 sec</td>
<td>dsyevr 5.3 sec</td>
</tr>
<tr>
<td><strong>Cholesky decomposition</strong></td>
<td>2.4 sec</td>
<td>dpotrf 0.2 sec</td>
</tr>
<tr>
<td><strong>Singular value decomposition</strong></td>
<td>315.0 sec</td>
<td>dgesvd 21.7 sec</td>
</tr>
</tbody>
</table>
Example: Loop Unrolling

Time critical routines should not contain logic and should not contain nested loops. The process of eliminating short loops in favor of hand optimized, explicit code is called 'Loop unrolling'.

Example: Calculation of integrals using the McMurchie/Davidson method

In the MD method, molecular integrals can be very elegantly calculated using an expansion of the Gaussian product in a Gaussian Hermite basis

Cartesian Gaussian on center A: \[ G_{ab\alpha}^A = (x - X_A)^a(y - Y_A)^b(z - Z_A)^c \exp(-\alpha r_A^2) \]

Repulsion integral in MD:

\[
(G_{abc;\alpha}^A G_{a'b'c';\beta}^B | G_{def;\gamma}^C G_{d'e'f';\delta}^D) = \sum_{t=0}^{a+a'} \sum_{u=0}^{b+b'} \sum_{v=0}^{c+c'} E_t^{AB} E_u^{AB} \sum_{t'=0}^{d+d'} \sum_{u'=0}^{e+e'} \sum_{v'=0}^{f+f'} (-1)^{t'+u'+v'} E_{t'}^{CD} E_{u'}^{CD} E_{v'}^{CD} R_{t+t',u+u',v+v'}
\]

const

Expansion of \( G^A G^B \) in Hermite basis

Expansion of \( G^C G^D \) in Hermite basis

Integrals in Hermite basis
Example: Short Loops and Multidimensional Arrays

Pseudocode for a general MD integral routine

Calculate Array EAB
Calculate Array ECD
Calculate Array R

loop ixyz over Cartesian components of A
  loop jxyz over Cartesian components of B
    loop kxyz over Cartesian components of C
      loop lxyz over Cartesian components of D
        x=0
        loop t =0..a+a’
          loop u =0..b+b’
            loop v =0..c+c’
              loop v’ =0..f+f’
                loop t’ =0..d+d’
                  loop u’ =0..e+e’
                    x=x+ EAB[x][a][a’][t] *EAB[y][b][b’][u] *EAB[z][c][c’][v] *ECD[x][d][d’][t’] *ECD[y][e][e’][u’] *ECD[z][f][f’][v’] *R[t+t’][u+u’][v+v’] *(-1)^t’+u’+v’
                    end loops t’,u’,v’
                  end loops t,u,v
                end loops t’
              end loops v
            end loops u
          end loops v’
        end loops t
      end loops k
    end loops j
  end loops k
end loops i

ELREP[ixyz][jxyz][kxyz][lxyz]=x

10 nested loops!
For s and p functions these run basically from 0 to 1

recursive formulas. Nested loops of length ~l_A+l_B (or l_C+l_D)
Example: Short Loops and Multidimensional Arrays

**Alternative**: For low angular momenta create hand optimized routines and store integrals in linearized arrays

```c
Calc_ssss()
ab  = a+b
cd  = c+d
abcd = ab+cd;
pprim = 4.0*ab*cd*sqrt(abcd);
SR  = Kab*Kcd/pprim;
PQX  = (PX-QX);
PQY  = (PY-QY);
PQZ  = (PZ-QZ);
RPQ2 = PQX*PQX+PQY*PQY+PQZ*PQZ;
W    = ab*cd/abcd;
RT   = W*RPQ2;
Calc_F_Function(F)
ELREP[0]= F[0]*SR;
```

```c
Calc_sssp()
ab  = a+b
cd  = c+d
abcd = ab+cd;
pprim = 4.0*ab*cd*sqrt(abcd);
SR  = Kab*Kcd/pprim;
PQX  = (PX-QX);
PQY  = (PY-QY);
PQZ  = (PZ-QZ);
RPQ2 = PQX*PQX+PQY*PQY+PQZ*PQZ;
W    = ab*cd/abcd;
RT   = W*RPQ2;
Calc_F_Function(F)
t1   = W/cd*F[1];
ELREP[0]= (QDZ*F[0]+PQZ*t1)*SR;
ELREP[1]= (QDX*F[0]+PQX*t1)*SR;
ELREP[2]= (QDY*F[0]+PQY*t1)*SR;
```

**NO** logic, **NO** short loops ➞ The compiler can optimize this code most efficiently

➢ Efficient modern integral libraries (e.g. libint) make use of machine generated, highly unrolled code
Numerical Example

<table>
<thead>
<tr>
<th></th>
<th>unoptimized code</th>
<th>unrolled code</th>
<th>libint</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ss</td>
<td>ss) (10^7 times)</td>
<td>1.8</td>
<td>1.2</td>
<td>0.7</td>
</tr>
<tr>
<td>(pp</td>
<td>pp) (10^6 times)</td>
<td>8.3</td>
<td>2.6</td>
<td>0.4</td>
</tr>
<tr>
<td>(dd</td>
<td>dd) (10^4 times)</td>
<td>4.1</td>
<td>0.4</td>
<td>0.1</td>
</tr>
<tr>
<td>(ff</td>
<td>ff) (10^3 times)</td>
<td>9.1</td>
<td>0.5</td>
<td>0.2</td>
</tr>
</tbody>
</table>

„to a large extend the efficiency of a computer code is a result of the care taken during the implementation stage and not due to the particular method selected for implementation.“ — Roland Lindh
Transformation to Spherical Harmonics

Molecular integrals are usually first calculated over Cartesian Gaussian functions and then transformed to spherical harmonics

\[
\left( G^l_a m_a G^l_b m_b | G^l_c m_c G^l_d m_d \right) = \sum_p \sum_q \sum_r \sum_s d^l_p m_a d^l_q m_b d^l_r m_c d^l_s m_d \left( G^x_p y_p z_p G^x_q y_q z_q | G^x_r y_r z_r G^x_s y_s z_s \right)
\]

- many nested, short loops, many zero’s in the d-coefficients
- Reasonable compilers manage to detect this situation and produce well optimized code

```c++
c++ -O3 -funroll-loops ...```
**Straightforward code**

Cart2Slm(SRC, DST)

  loop i in xyz_b
    loop j in xyz_c
      loop k in xyz_d
        loop m in slm_a
          x=0
          loop l in xyz_a
            x=x+SRC[l,i,j,k]*d[m,l]
          end loop l
        end loop
        TEMP(m,i,j,k)=x
      end loops i,j,k
    end loops
  end loops
end subroutine

e etc for the other three indices
to fill target array DST

end subroutine

**Unrolled, optimized code**

Cart2Slm_4_f(SRC, DST)

  loop i in xyz_a
    loop j in xyz_b
      loop k in xyz_c
        DST[ 0+ 7*(k+dim3*(j+dim2*i))]=   +0.258198889747161153*SRC[ 2+10*(k+dim3*(j+dim2*i))] 
          -0.387298334620741647*SRC[ 4+10*(k+dim3*(j+dim2*i))] 
          -0.387298334620741647*SRC[ 6+10*(k+dim3*(j+dim2*i))] 
        DST[ 1+ 7*(k+dim3*(j+dim2*i))]=   -0.158113883008418971*SRC[ 0+10*(k+dim3*(j+dim2*i))] 
          -0.158113883008418943*SRC[ 5+10*(k+dim3*(j+dim2*i))] 
          +0.632455532033675771*SRC[ 7+10*(k+dim3*(j+dim2*i))] 
        ... 
    end loops
end subroutine
Design of an algorithm: FLOP count

In the early days of algorithm design, developers were carefully minimizing the number of floating point operations (FLOPs) required to accomplish a given task.

Example: Partial integral transformation \((\mu \nu | \kappa \tau) \rightarrow (ia | jb)\)

\(i,j=\) occupied MOs \((#=O)\), \(a,b,\) unoccupied MOs \((#=V)\), \(\mu,\nu,\kappa,\tau=\) basis functions \((#=B)\)

\[
\psi_p(r) = \sum_{\mu} c_{\mu p} \varphi_p(r)
\]

Naive:

\[
(ia | jb) = \sum_{\mu} \sum_{\nu} \sum_{\kappa} \sum_{\tau} c_{\mu i} c_{\nu a} c_{\kappa j} c_{\tau b} (\mu \nu | \kappa \tau) \quad FLOPS = B^4 O^2 V^2
\]

\(O(N^8)\) scaling

Must be possible to do better than that.
FLOP Count: Partial Integral transformation

**Algorithm A: occupied indices first**

\[(i\nu | \kappa \tau) = \sum_{\mu} c_{\mu i} (\mu \nu | \kappa \tau) \quad (B^4 O) \quad 3125\]

\[(i\nu | j\tau) = \sum_{\kappa} c_{\kappa j} (i\nu | \kappa \tau) \quad (O^2 B^3) \quad 312\]

\[(i a | j\tau) = \sum_{\nu} c_{\nu a} (i\nu | j\tau) \quad (O^2 V B^2) \quad 281\]

\[(i a | j b) = \sum_{\tau} c_{\tau b} (i a | j \tau) \quad (O^2 V^2 B) \quad 253\]

**Algorithm B: virtual indices first**

\[(\mu a | \kappa \tau) = \sum_{\nu} c_{\nu a} (\mu \nu | \kappa \tau) \quad (B^4 V) \quad 28215\]

\[(\mu a | \nu b) = \sum_{\tau} c_{\tau b} (\mu a | \kappa \tau) \quad (V^2 B^3) \quad 25312\]

\[(i a | \nu b) = \sum_{\mu} c_{\mu i} (\mu \nu | \nu b) \quad (O V^2 B^2) \quad 2531\]

\[(i a | j b) = \sum_{\nu} c_{\nu j} (i a | \nu b) \quad (O^2 V^2 B) \quad 253\]

**Four O(N^5) steps**

ratio of FLOP counts:

\[
\frac{\#(FLOPS)_A}{\#(FLOPS)_B} = \frac{O}{V} \frac{(2B^3 - V^3)}{(B^2 + 3B^2V - 3BV^2 + V^3)} < 1
\]

Always transform the index first that offers the largest data reduction!

Example: GFLOPS for B=500, O=50, V=450

0.07
FLOP count versus Performance

In order to capitalize on the efficiency of the BLAS routines, it is sometimes advantageous to sacrifice optimal FLOP count.

Example: Integral direct partial integral transformation for MP2

\[
E_{MP2} = -\frac{1}{4} \sum_{i,j,a,b} \frac{[(ia \mid jb) - (ib \mid ja)]^2}{\epsilon_a + \epsilon_b - \epsilon_i - \epsilon_j}
\]

Key step: integral transformation

\[
(ia \mid jb) = \sum_{\mu} \sum_{\nu} \sum_{\kappa} \sum_{\tau} c_{\mu i} c_{\nu a} c_{\kappa j} c_{\tau b} (\mu \nu \mid \kappa \tau)
\]
FLOP count optimized algorithm

1. Loop over batches of occupied MOs:
   - Loop over p = 1..NBas
     - Loop over q = 1..p
       - Loop over r = 1..p
         - Loop over s = 1..r|q
           - Calculate(pq|rs)
           - Loop over i = 1..Nocc (in ibatch)
             - \( ITMP[p, q, r, i] += C_{occ}[s, i] \times (pq|rs) \) and non-redundant permutations of indices
             - End loop i in ibatch
           - End loop s
         - End loop r
       - End loop q
     - End loop p
   - End loops over p, q, r, s

2. Transformation of 2nd index:
   - Loop over i = 1..Nocc (in ibatch)
     - Loop over j = 1..i
       - Loop over q = 1..NBas
         - \( JTMP[p, j, r, i] += C_{occ}[q, j] \times ITMP[p, q, r, i] \)
       - End loop q
     - End loop j
   - End loop i

3. Transformation of 3rd index:
   - Loop over i = 1..Nocc (in ibatch)
     - Loop over j = 1..i
       - Loop over p over AO’s
         - Loop over b = 1..NVirt
           - Loop over r over AO’s
             - \( ATMP(p, b) += C[r, b] \times JTMP[p, j, r, i] \)
           - End loops over r, b, p
         - End loop over AO’s
       - End loop over p
     - End loop over AO’s
   - End loop over i

4. Transformation of 4th index:
   - Loop over a = 1..Nvrt
     - Loop over b = 1..Nvrt
       - Loop over p over AO’s
         - \( KIJ[a, b] += C[p, a] \times ATMP[p, b] \)
       - End loops over p, a, b
     - End loop over AO’s
   - End loop over AO’s
   - Evaluate MP2 amplitudes and pair energy
     - End loops over i, j
   - End loop over i
   - End loop over ibatch

Full eightfold permutation symmetry used

Have to be able to store \( N_{Basis}^3 \) integrals for each occupied MO. Hence need batches of occupied MOs.
BLAS optimized algorithm

\[
\text{loop } p=1..\text{NBas} \\
\quad \text{loop } r=1..p \\
\quad \quad \text{loop } q=1..\text{NBas} \\
\quad \quad \quad \text{loop } s=1..\text{NBas} \\
\quad \quad \quad \quad \text{calculate } K[p,r](q,s) = (pq|rs) \\
\quad \quad \text{end loop } q,s \\
\quad \textbf{Perform transformation} \ K[p,r](i,j) = (C_{\text{occ}}^T K[p,r] C_{\text{occ}})_{ij} \\
\quad \text{Write matrix } K[p,r] \text{ to disk} \\
\text{end loops } p,r \\
\text{Resort Integrals } K[p,r](i,j) \text{ to give } K[i,j](p,r) \ (i<=j) \\
\text{Loop } i=1..\text{Nocc} \\
\quad \text{loop } j=1..i \\
\quad \quad \text{Read integrals } K[i,j](p,r) \\
\quad \quad \textbf{Perform transformation} \ K[i,j](a,b) = (C_{\text{virt}}^T K[i,j] C_{\text{virt}})_{ab} \\
\quad \quad \text{Calculate MP2 amplitudes } T[i,j](a,b) \\
\quad \quad \text{Calculate MP2 pair energy } e(i,j) \\
\quad \quad \text{Sum up MP2 correlation energy} \\
\text{end loops } i,j
\]

We only use one out of eightfold permutational symmetry, which means that we generate the integrals effectively 4 times.

Two BLAS level 3 multiplications in the rate determining step.

We only use one permutational symmetry here too, which means we store 4 times too many integrals.

Awkward: Lots of I/O

Two BLAS level 3 multiplications.
Hexane
def2-TZVP (258 basis functions)
4 GB main memory used

Performance Test (I)

FLOP optimized algorithm
(1 batches necessary)
108 sec

BLAS optimized algorithm
79.8 sec

TOTAL TIME for half transformation: 79.4 sec
AO-integral generation : 71.5 sec
Half transformation : 5.7 sec
K-integral sorting : 0.4 sec
Performance Test (II)

FLOP optimized algorithm
(25 batches necessary)

BLAS optimized algorithm

(total) many hours

1732 sec

TOTAL TIME for half transformation: 1697.0 sec
AO-integral generation : 1078.9 sec
Half transformation : 354.0 sec
K-integral sorting : 60.4 sec

Diclophenac
def2-TZVP (667 basis functions)
4 GB main memory used
Example: Factorization in Coupled Cluster

The scaling of an algorithm can sometimes be reduced through factorization. This happens if intermediates can be defined that only depend on a subset of the summation indices. In this case, the summations can be carried out in two steps:

Look at one nonlinear term in the CCSD equations:

$$\sigma_{ab}^{ij} \leftarrow \sum_{kl} \sum_{cd} \langle kl || cd \rangle t_{cd}^{ij} t_{ab}^{kl}$$

$O(N^8)$ scaling

- 4 target indices
- 4 summation indices
- … But any quantity depends on only 2 target indices at a time
- Must be able to re-arrange loops more cleverly

Two possibilities:

$$\sigma_{ab}^{ij} \leftarrow \sum_{kl} t_{ab}^{kl} \sum_{cd} \langle kl || cd \rangle t_{cd}^{ij}$$

$$\sigma_{ab}^{ij} \leftarrow \sum_{cd} t_{cd}^{ij} \sum_{kl} t_{ab}^{kl} \langle kl || cd \rangle$$

or

$$\sigma_{ab}^{ij} \leftarrow \sum_{cd} t_{cd}^{ij} \sum_{kl} t_{ab}^{kl} \langle kl || cd \rangle$$

$$\sigma_{ab}^{ij} \leftarrow \sum_{kl} t_{ab}^{kl} \sum_{cd} \langle kl || cd \rangle t_{cd}^{ij}$$
Example: Factorization in Coupled Cluster

Algorithm 1  $N_{\text{occ}}^2$  

\[ \sigma_{ab}^{ij} \leftarrow \sum_{kl} t_{ab}^{kl} \sum_{cd} \langle kl || cd \rangle t_{cd}^{ij} \]

\[ X_{kl}^{ij} = \sum_{cd} \langle kl || cd \rangle t_{cd}^{ij} \]

$N_{\text{occ}}^4$ Storage  

$N_{\text{occ}}^4 N_{\text{virt}}^2$ FLOPS

Algorithm 2  $N_{\text{virt}}^2$

\[ \sigma_{ab}^{ij} \leftarrow \sum_{kl} t_{ab}^{kl} X_{kl}^{ij} \]

$N_{\text{occ}}^4 N_{\text{virt}}^2$ FLOPS

\[ Y_{cd}^{ab} = \sum_{kl} t_{ab}^{kl} \langle kl || cd \rangle \]

$N_{\text{virt}}^4$ Storage  

$N_{\text{occ}}^2 N_{\text{virt}}^4$ FLOPS

\[ \sigma_{ab}^{ij} \leftarrow \sum_{cd} t_{cd}^{ij} Y_{cd}^{ab} \]

$N_{\text{occ}}^2 N_{\text{virt}}^4$ FLOPS

$O(N^6)$ scaling

\[ 2x N_{\text{occ}}^4 N_{\text{virt}}^2 \text{ FLOPS} \]

\[ N_{\text{occ}}^4 \text{ Storage} \]

\[ O(N^6) \text{ scaling} \]

\[ 2x N_{\text{occ}}^2 N_{\text{virt}}^4 \text{ FLOPS} \]

\[ N_{\text{virt}}^4 \text{ Storage} \]

\[ 2x N_{\text{occ}}^2 N_{\text{virt}}^4 \text{ FLOPS} \]

\[ N_{\text{virt}}^4 \text{ Storage} \]

\[ \text{much better and much less Storage!} \]
Substantial performance gains can be realized by choosing intermediates wisely such that redundant work is move out of the inner loops.

**Example:** Integrate integral evaluation as early as possible into the target quantities. For the Coulomb matrix, Ahmadi & Almlöf suggested:

\[
J_{\mu\nu} = \sum_{\kappa\tau} P_{\kappa\tau} (\mu\nu \mid \kappa\tau)
\]

\[
= \sum_{\kappa\tau} P_{\kappa\tau} \sum_{tuvw} E_{tuvw}^{\mu\nu} \sum_{t'u'v'} (-1)^{t'+u'+v'} E_{t'u'v'}^{\kappa\tau} R_{t+t',u+u',v+v'}
\]

\[
= \sum_{tuvw} E_{tuvw}^{\mu\nu} \sum_{t'u'v'} R_{t+t',u+u',v+v'} \sum_{\kappa\tau} (-1)^{t'+u'+v'} P_{\kappa\tau} E_{t'u'v'}^{\kappa\tau}
\]

\[
= \sum_{tuvw} E_{tuvw}^{\mu\nu} \sum_{t'u'v'} P_{t'u'v'} \sum_{\kappa\tau} (-1)^{t'+u'+v'} E_{t'u'v'}^{\kappa\tau} P_{\kappa\tau}
\]

When we calculate the integrals one by one, we repeated re-calculate this quantity $N^2$ times although it is independent of $\mu, \nu$. Likewise:

Transformation to spherical harmonics

Hermite basis density

Hermite to $S_{lm}$ transformation

Hermite basis repulsion
## Performance example

<table>
<thead>
<tr>
<th>Coulomb term (sec) (20-builds)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Traditional treatment</strong></td>
</tr>
<tr>
<td>5796 sec</td>
</tr>
<tr>
<td><strong>Split-J algorithm</strong></td>
</tr>
<tr>
<td>2834 sec</td>
</tr>
</tbody>
</table>

- Ahmadi-Almlöf
- Head-Gordon J-engine

Identical numerical result, same scaling, but significant speedup realized through thoughtful structuring of the entire computational process.
Example: I/O Heavy Algorithms

The I/O system is the slowest part of your computer!
➢ Use it as little as possible
➢ Move its usage as far outside in the loop structure as reasonably possible
➢ Avoid reading small chunks of data

Example: Integral symmetrization in EOM-CCSD

Loop \( i=1..N_{occ} \)
  loop \( a=1..N_{vir} \)
    Write NULL matrix \( K^a \) into buffer IABC
  end loop a
  loop \( a=1..N_{vir} \)
    Read matrix \( K^a(b,c) = (ib|ac) \) from IABC
    loop \( b=1..N_{vir} \)
      Read matrix \( K^b(c,d) = (ic|bd) \) from IABC
      loop \( c=1..N_{vir} \)
        \( K^b(a,c)=+K^b(a,c)+K^a(b,c) \);
      end loop c
    end loop b
  end loop a
end loop i

Loop \( i=1..N_{occ} \)
  Initialize buffer \( K^b \) for all \( b \)
  loop \( a=1..N_{vir} \)
    read matrix \( K^a(b,c) \) from IABC
    loop \( b=1..N_{vir} \)
      loop \( c=1..N_{vir} \)
        \( K^b(a,c)=+K^b(a,c)+K^a(b,c) \);
      end loop c
    end loop b
  end loop a
  Write entire buffer \( K^b \) into IABC
end loop i

SAME operation count!
Factor 200 performance difference!!
Automatic Code Generation
Problems with Method Development

Conclusions:

- The technicalities of development occupy most of our time
- Humans make mistakes, Debugging takes a lot of time
- The human brain can only deal with so much complexity. Beyond it is hopeless

➡ We need programming tools that take us directly from the Ansatz (our idea) to efficient, production level code
➡ **Automatic Code Generation**
Code Generation Tools

✓ Janssen & Schaefer, ROCCSD, pioneering work 1991
✓ Tensor contraction engine in NWChem, various CC (Hirata, Auer & Co)
✓ Diagram based arbitrary order CC/MRCC (Kallay)
✓ Gecco Internally contracted MRCC (Köhn)
✓ Genetic algorithm based code generator, MRCC (Hanrath)
✓ Automatic code generator, FIC-MRCI (Knizia, Werner)
✓ MREOM-CC (Huntington, Nooijen)
✓ General active space EOM CC (Kong, Demel, Shamsundar, Nooijen)
✓ Bagel/Smith CASPT2 gradient, (Shiozaki)
✓ Yanai, Saitow, DMRG-CASPT2, various contracted variants
✓ ACES III programming 'super-language' (Deumens, Bartlett & Co)
✓ Cyclops (Solomonik)
✓ Tiled Arrays (Valeev)
✓ .... many others
Simple & Straightforward Equation Generation

Any Ansatz (single- or multi-reference) that can be formulated in terms of 2nd quantization, quickly leads to expectation values of the form

\[ \langle \Psi_0 | E_n E_p \ldots E_r | \Psi_0 \rangle, \quad E^q_p = a^+_{q\beta} a_{p\beta} + a^+_{q\alpha} a_{p\alpha}. \]

Or, in terms of elementary spin-orbital operators:

\[ \langle \Psi_0 | a^n_m a^q_p \ldots a^s_r | \Psi_0 \rangle, \]

If the orbital space is divided in internal (i,j,k,l), active (t,u,v,w) and virtual (a,b,c,d), the important commutation relations apply:

\[ \left[ E^q_p, E^s_r \right] = E^s_p \delta_{qr} - E^q_r \delta_{ps}, \]

Thus:

\[ E^p_i | \Psi_0 \rangle = 2\delta_{ip} | \Psi_0 \rangle, \quad \langle \Psi_0 | E^i_p = 2\delta_{ip} \langle \Psi_0 |, \]

\[ E^p_a | \Psi_0 \rangle = 0, \quad \langle \Psi_0 | E^a_p = 0, \]
Equation Generation

Strategy:

✓ Use the commutation relation to change the order of operators
✓ Move lower internal labels to the right
✓ Move upper internal labels to the left
✓ Move lower external labels to the right
✓ Move upper external labels to the left

→ Creates 0’s, Kronecker deltas and ‘pre-densities’ (MR case)

Awkward by hand, easy for a computer

Issues: ✓ redundant terms are generated
✓ terms that cancel each other are generated
✓ Equivalent terms may have inequivalent labels
✓ ...

Post-processing required

\[
\gamma_{lw...x}^{uw...y} = \langle \Psi_0 \mid E^u_l E^w_v ... E^y_x \mid \Psi_0 \rangle.
\]
Code Generation Chain

1. Equation Generator:
   ✓ Takes the Ansatz and generates equations
   ✓ Identifies identical, redundant and cancelling terms
   ✓ brings all labels into a 'canonical form'

2. Factorizer
   ✓ Identifies possible intermediates
   ✓ Finds the best possible intermediates and contraction order
   ✓ Finds common intermediates in different terms
   ✓ Ensures that all terms have their correct formal scaling

3. Code generator
   ✓ Writes code for a specific electronic structure package
   ✓ Recognizes patterns/contractions for which highly optimized code exists
   ✓ Ensures that all terms have their correct formal scaling
   ✓ Ensures minimal I/O and maximal use of BLAS
   ✓ Generates parallel code, code for specific machines, ....
Realization of a Code generation chain (AGE)

**Wavefunction input**
The definition of wavefunction

**matelem**
Produces blocks of Hamiltonian matrix and other quantities as a string of excitation operators

**gen+process**
Applies commutation rules to obtain the working equations

**canonicalizer**
Exploit the tensor symmetry to reduce the number of equations

**tau-apply**
Applies a pre-defined set of τ intermediates

**factorizer**
Factorize contractions of more than two tensors
Introduce intermediates.

**sumint**
Finds summation intermediates

**shuffle**
Change ordering of indexes in tensor to reduce I/O cost and allow BLAS operations

**memoryopt**
Change the order of contractions to minimize the storage cost

**autogen**
Turn the equations into the code, applying BLAS for matrix-matrix operations and specialized routines for recognized types of contractions.
Cost model

In order to find the best possible intermediates and factorization, we need to have a prediction how long each contraction should take.

![Graph showing the relationship between cost model time estimation and wall time for different methods including RHF CISD, ROHF CISD, and FIC-MRCI(6,6).]
# Efficiency: Example

Naphtalene, CISD, no symmetry used

| CISD  | Total (s) | $\langle D|H|D \rangle$ 4ext | $\langle D|H|D \rangle$ 2ext | $\langle S|H|D \rangle$ 3ext |
|-------|-----------|------------------|------------------|------------------|
|      | $N_{\text{virt}}$ | native | AGE | native | AGE | native | AGE | native | AGE |
| Basis set |           |          |      |        |      |        |      |        |      |
| SVP   | 146       | 121     | 235  | 31     | 30   | 67     | 128  | 11     | 32  |
| TZVP  | 204       | 330     | 640  | 113    | 117  | 164    | 332  | 33     | 89  |
| TZVPP | 388       | 2850    | 4950 | 1471   | 1526 | 1053   | 2080 | 244    | 646 |

Hand code:

$$\sigma^{ij} \leftarrow \sum_k \left\{ 2C^{ik} - (C^{ik})^\dagger \right\} \left\{ K^{kj} - \frac{1}{2} J^{kj} \right\} - \frac{1}{2} \left\{ (C^{ik})^\dagger J^{kj} \right\} - \left\{ (C^{ik})^\dagger J^{kj} \right\}^\dagger + \left\{ K^{ik} - \frac{1}{2} J^{ik} \right\} \left\{ 2C^{kj} - (C^{kj})^\dagger \right\} - \frac{1}{2} \left\{ J^{ik} (C^{kj})^\dagger \right\} - \left\{ J^{ik} (C^{kj})^\dagger \right\}^\dagger \right. .$$

4 dgemm/k

Generated code:

$$\sigma^{ij} \leftarrow \sum_k -J^{ik} C^{kj} - C^{kj} J^{ik} - J^{kj} C^{ik} - C^{ik} J^{kj} - C^{kj} K^{ij} - K^{ik} C^{jk} + 2K^{ik} C^{jk} + 2C^{ik} K^{kj}$$

8 dgemm/k
Complexity: Example

Fully internal contracted MRCI (or MRCC, also CASPT2/NEVPT2) works with contracted functions in the first-order interacting space (FOIS)

\[
\Phi_{ij}^{\text{ta}} = E_{ij}^{\text{ta}} \Psi_0 = \sum I C_I^{(\text{CASSCF})} E_{ij}^{\text{ta}} \Phi_I^{(\text{CAS})}
\]

✓ 10 Excitation classes  -> 100 Blocks of matrix elements
✓ Not orthogonal
✓ Not linearly independent
➡ Extremely complicated matrix elements
➡ 1945 equations including up to four body density
➡ Factorized into 3674 equations
➡ Removed 1222 redundant intermediates

➡ Nearly hopeless to program by hand. Readily done with code generator as a matter of hours (perhaps days)
Influence of the choice of projection manifolds in the CASPT2 implementation

Takeshi Yanai, Yuki Kurashige, Masaaki Saitow, Jakub Chalupský, Roland Lindh & Per-Åke Malmqvist

Pages 2077-2085 | Received 12 Oct 2016, Accepted 01 Dec 2016, Published online: 27 Dec 2016

... found a (small) bug in the hand coded version of the CASPT2 method

COMMUNICATION: Automatic code generation enables nuclear gradient computations for fully internally contracted multireference theory

Matthew K. MacLeod and Toru Shiozaki
Department of Chemistry, Northwestern University, 2145 Sheridan Rd., Evanston, Illinois 60208, USA

(Received 11 January 2015; accepted 27 January 2015; published online 5 February 2015)

... Fully automated, large scale nuclear gradient for CASPT2. Optimizations of metalloporphyrins

Analytical Gradient Theory for Strongly Contracted (SC) and Partially Contracted (PC) N-Electron Valence State Perturbation Theory (NEVPT2)

Jae Woo Park

J. Chem. Theory Comput., Just Accepted Manuscript • DOI: 10.1021/acs.jctc.9b00762 • Publication Date (Web): 10 Sep 2019
Code generation: Summary

✓ Code generation enables the implementation of ‘impossibly complicated’ methods
✓ Code generation reduces development times from years to hours/days
✓ Code generation can produce code for specific hardware, thus ensuring optimal performance
✓ Code generation can ensure that all parts of the code have consistent quality
✓ Once the code generation chain produces correct results, it is extremely reliable (e.g. a small bug was identified in the original CASPT2 code in 2015, CASPT2 is from 1990!)

➡ Code generation will play an important part in future quantum chemistry
➡ Generated code can be made almost as efficient as the best hand optimized code
➡ In the future we keep just a wavefunction Ansatz in the source code repository and generate the code during compile time. Any improvement in the code generation chain is the immediately applied to all parts of the program.