European Summer School of Quantum Chemistry

2024 Torre Normanna Sicily

Lectures 1 & 2: Algorithm Design

Frank Neese

MAX-PLANCK-GESELLSCHAFT

MPI für Kohlenforschung

Kaiser-Wilhelm Platz 1

45470 Mülheim an der Ruhr

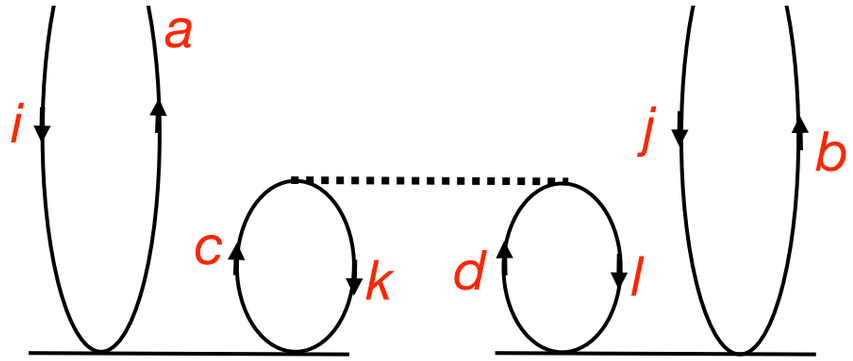
You have an Equation, what now?

See coupled cluster lectures!

Let us assume that you have derived an equation, e.g. using Wick's theorem

$$= \frac{1}{4}\sum_{pqrs}\sum_{kc}\langle pq||rs\rangle t_k^c \left(\{ a_i^\dagger a_j^\dagger a_b a_a a_p^\dagger a_q^\dagger a_s a_r a_c^\dagger a_k \} + \right.$$

Or diagrams …



i,j,k,l=occupied
a,b,c,d=virtual

$$\sigma_{ab}^{ij} \leftarrow P(ij)P(ab)\sum_{kl}\sum_{cd}\langle kl||cd\rangle t_{ac}^{ik} t_{db}^{lj}$$

You are charged (or simply want) to implement that. What do you do?

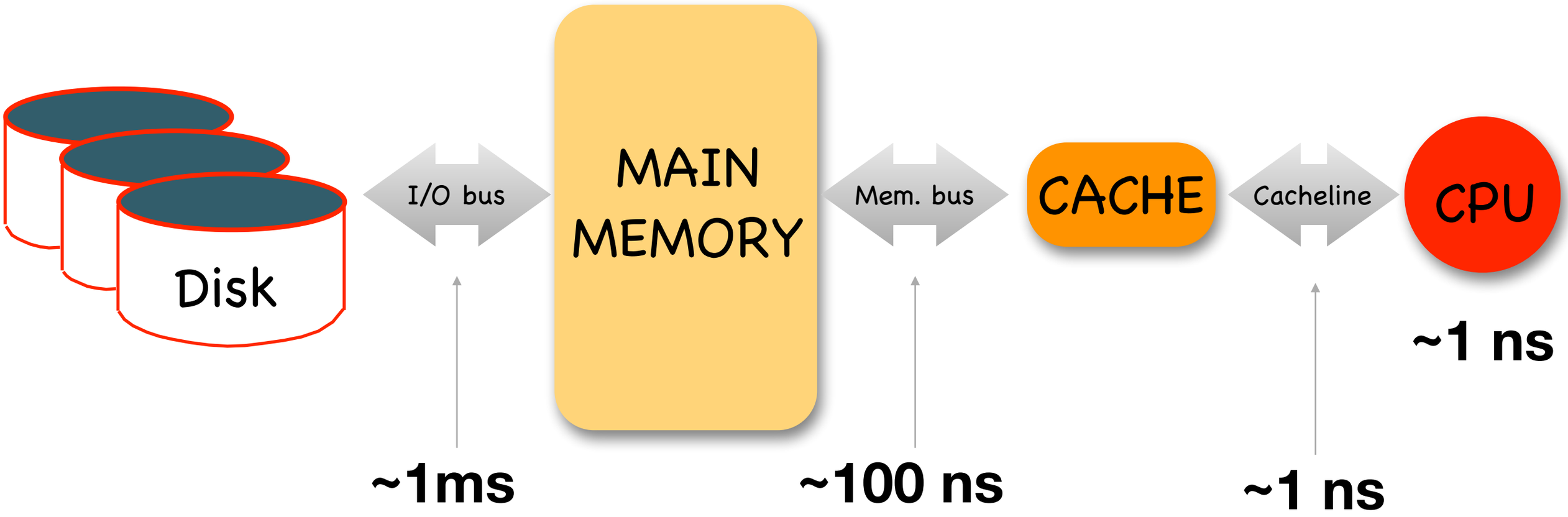
Goals of Quantum Chemical Method Development

- ✓ 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**

Prelude

A little information on Computers

Computer Architecture and Algorithm Design



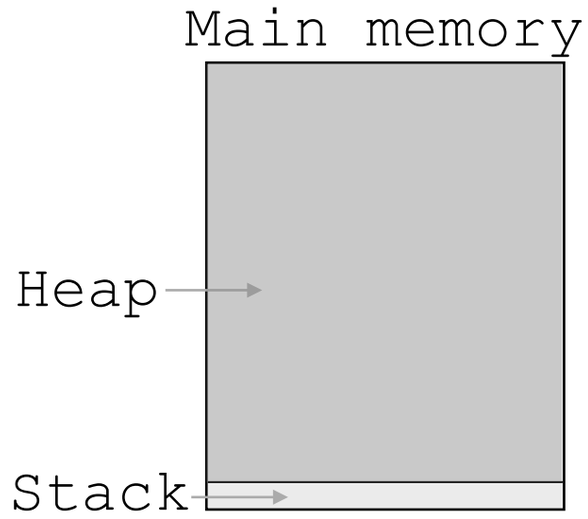
- ✓ Disk access is very slow
- ✓ Memory to CPU transfer is slow

Algorithms need to carefully balance I/O and memory operations, not *just* minimize FLOP count

Main Memory: Stack vs Heap

Stack: Fixed Size Reserved Main Memory region of static variables

Heap: Variable Size Main Memory region of dynamic variables




```
void SignOfLife()
{
  int i;
  double x[1024];
  double *y = new double[1024];
  printf(„This is SignOfLife - still hanging in there!“);
  delete[] y;
};
```

Memory allocated on the Stack

Memory allocated on the Heap

Release memory allocated on the heap

NOTE

- memory allocated on the Stack will be released automatically when the variable goes out of scope.
- Memory allocated on the Heap you need to de-allocate yourself or you are created a memory leak (very common mistake)

The Stack can Overflow

- ✓ The Stack is usually small (e.g. default on a Mac is 8MB)
  - ➡ If you are putting a lot of data on the stack if may overflow
  - ➡ Core dump / Segmentation fault

```
void SignOfLife()
{
  double x[1024];
  printf("Sign of Life – still hanging in there!\n");
  fflush(stdout);
};
```

```
frankneese@MacBook-Pro-von-Frank 2024 - ESQC-24 % ./stack
Sign of Life – still hanging in there!
```

```
void SignOfLife()
{
  double x[1024*1024];
…
```

```
frankneese@MacBook-Pro-von-Frank 2024 - ESQC-24 % ./stack
zsh: segmentation fault  ./stack
```

- ✓ The Heap can overflow too of course. In this case there might be chance to recover using „exception handlers“ (e.g. C++ **`try`** … **`catch`** ….)

Side Remark: Call by Reference vs Call by Value

An argument **passed by value:**

- creates the copy of the variable on the stack
- unchanged after the function call is done
- Potentially creates overhead from copying or stack overflows

```
void foo(int i){ i=i+1; };

int i=1;  foo(i); printf("i=%d");
```

The printout will be **i=1**

An argument **passed by reference:**

- What is passed is a pointer to the storage location of the variable (Heap or Stack)
- Potentially changed after the function call is done
- Creates no overhead

```
void foo(int &i){ i=i+1; };

int i=1;  foo(i); printf("i=%d");
```

The printout will be **i=2**

Summary: Hardware components

Hard-disk:
- Potentially large (assumed infinite in the „von Neumann machine“ or the abstract „Turing machine“
- Very slow compared to the rest. Avoid as much as you can!

Main memory:
- Small Stack (static), large Heap (dynamic)
- Relatively slow. It takes time to dig data out from there.

Bus:
- Transfer from main memory to CPU/Cache
- Often rate limiting in actual calculations

Cache:
- Fast memory directly attached to the CPU
- Relatively small. You want all the data the is being processed there

Register:
- Central part of the CPU that holds the data or instructions that are being processed next

CPU:
- Runs instructions in sequence (e.g. GHz means nsec for individual instructions) - make sure it does not run idle!

PART 1

Before we talk about how to compute things efficiently:

„The greatest performance gains are coming from the calculations that you <u>don't</u> do at all“

3 Ways to Avoid Unproductive Computation

1. Use of symmetry

Integrals have selection rules, e.g. in $(\mu\nu|\kappa\tau)$

The direct product $\Gamma(\mu)\otimes\Gamma(\nu)\otimes\Gamma(\kappa)\otimes\Gamma(\tau)$

Must contain the **totally symmetric irrep**, provided μ, ν, κ, τ

Are adapted to the irreps of the point group

2. Use of permutation symmetry

Integrals have permutation symmetry that *usually* should be used

$$(\mu\nu|\kappa\tau) = (\mu\nu|\tau\kappa) = (\nu\mu|\kappa\tau) = (\nu\mu|\tau\kappa)$$

$$= (\kappa\tau|\mu\nu) = (\tau\kappa|\mu\nu) = (\kappa\tau|\nu\mu) = (\tau\kappa|\nu\mu)$$

3. Avoid terms that are (near) zero or factors that are (near) one

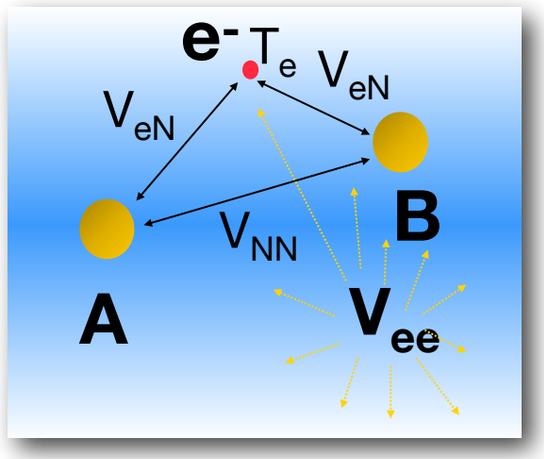
… here is where the art & science of thresholding starts!

Avoid small numbers but make sure errors don’t add up!

Self Consistent Field

$$\mathbf{F}(\mathbf{c})\mathbf{c}_i = \varepsilon_i \mathbf{S}\mathbf{c}_i$$

$$S_{\mu\nu} = \langle \mu \mid \nu \rangle$$



"Mean Field"

Hartree-Fock

$$P_{\mu\nu} = \sum_i c_{\mu i} c_{\nu i}$$

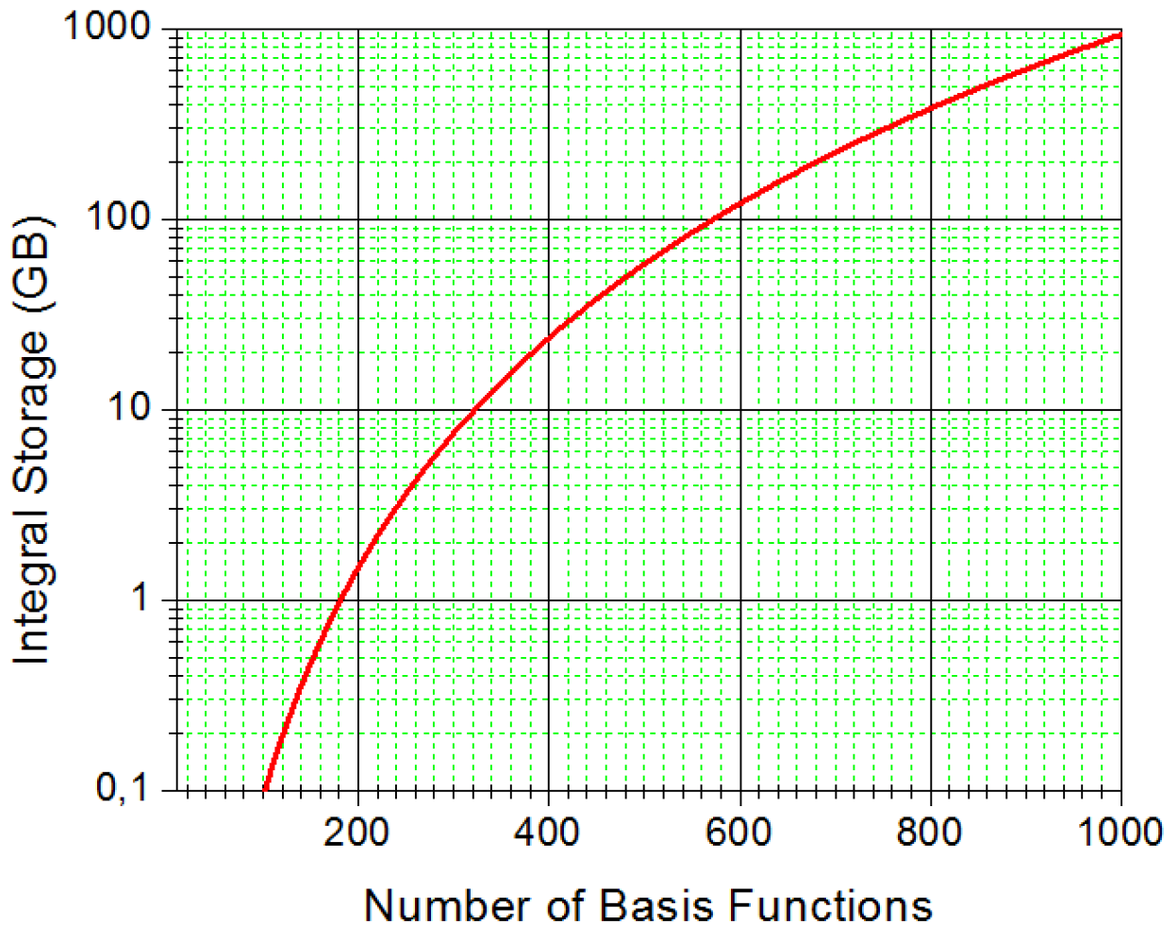
$$h_{\mu\nu} = \left\langle \mu \left| -\tfrac{1}{2}\nabla^2 - \sum\nolimits_A Z_A r_A^{-1} \right| \nu \right\rangle$$

$$F_{\mu\nu} = h_{\mu\nu} + \sum_{\kappa\tau} P_{\kappa\tau}(\mu\nu \mid \kappa\tau) - \frac{c_x}{2}\sum_{\kappa\tau} P_{\kappa\tau}(\mu\kappa \mid \nu\tau) + \int \mu(\mathbf{r}) \underbrace{\frac{\delta E_{XC}[\rho]}{\delta\rho(\mathbf{r})}}_{V_{XC}(\mathbf{r})} \nu(\mathbf{r})\, d\mathbf{r}$$

One Electron **Coulomb** **HF Exchange** **XC Potential**

Almlöf's Revolutionary Proposals

For decades progress in quantum chemistry was prevented by the **O(N^4)** of two-electron integrals.



Even if the integrals can be stored for a 1000 basis function calculation, the I/O penalty is huge and the CPU remains largely idle while waiting for data to arrive from the hard drive

The integral bottleneck was finally overcome by Almlöfs revolutionary proposals

1. **Do NOT store integrals. Recalculate when needed (direct SCF)**
2. **Split the calculation of the Coulomb and exchange terms and use the most efficient approximation for each rather than use the same integrals for both.**

Let's take a look at Electron Repulsion Integrals

Look at an ERI: $$(\mu\nu|\kappa\tau) = \int\int \frac{\mu(\boldsymbol{r}_1)\nu(\boldsymbol{r}_1)\kappa(\boldsymbol{r}_2)\tau(\boldsymbol{r}_2)}{|\boldsymbol{r}_1 - \boldsymbol{r}_2|} d\boldsymbol{r}_1 d\boldsymbol{r}_2$$

This can be viewed as the **electrostatic interaction of two smeared out charge distributions:**

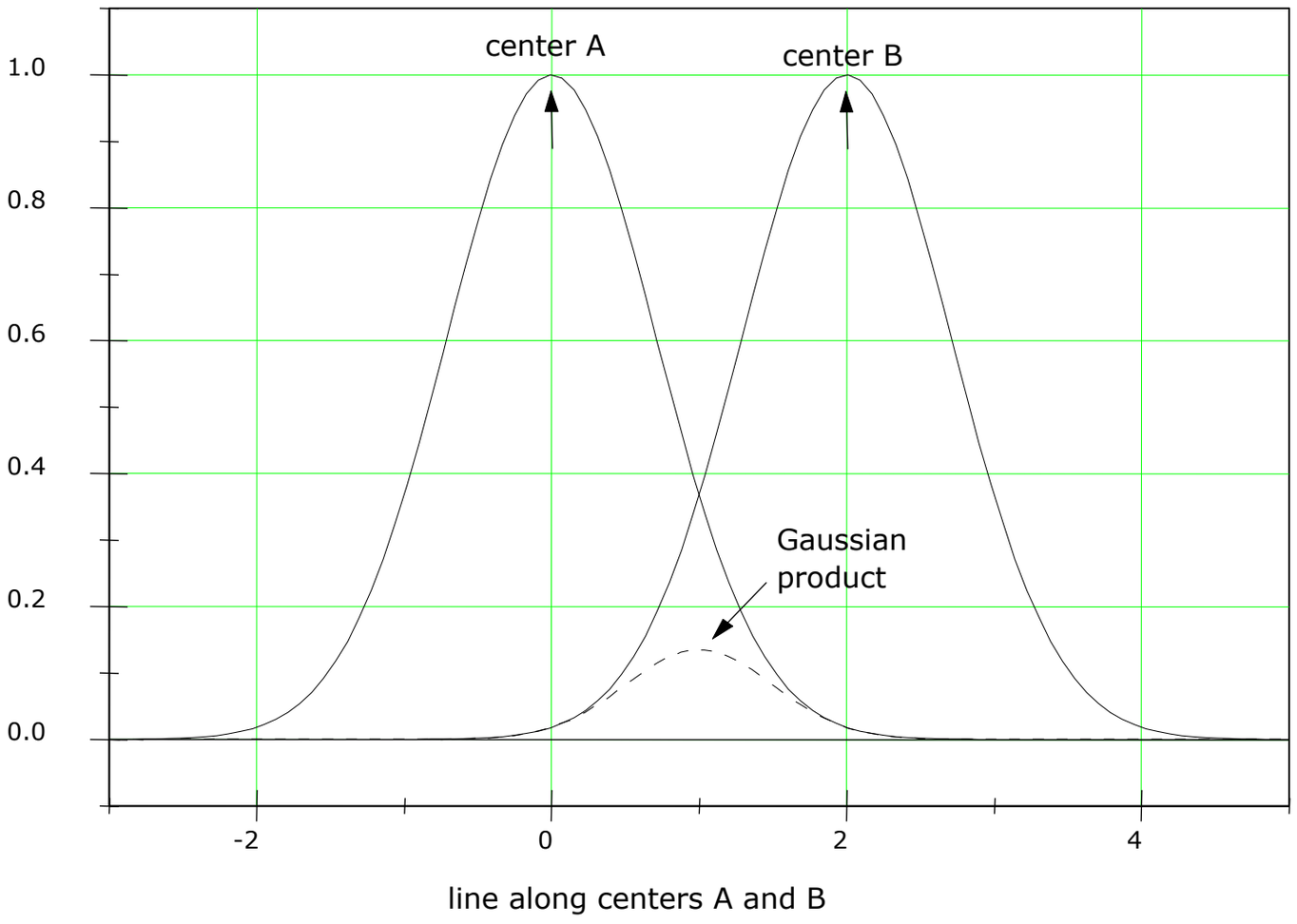
$$\rho_{\mu\nu}(\boldsymbol{r}_1) = \mu(\boldsymbol{r}_1)\nu(\boldsymbol{r}_1)$$

$$\rho_{\kappa\tau}(\boldsymbol{r}_2) = \kappa(\boldsymbol{r}_2)\tau(\boldsymbol{r}_2)$$

And it is advantageous to take the basis functions themselves as Gaussians:

$$\mu_A(\boldsymbol{r}) = S_{l_\mu m_\mu}(\boldsymbol{r} - \boldsymbol{R_A}) N_\mu \sum_k d_{k\mu} \exp\left(-\alpha_k r_A^2\right)$$

Negligible Integrals: Gaussian Product Theorem



$$\exp(-\alpha r_A^2)\exp(-\beta r_B^2)$$
$$= \underbrace{\exp(-qQ^2)}_{K_{AB}}\exp(-pr_P^2)$$

$$p = \alpha + \beta$$
$$q = \alpha\beta \,/\, p$$
$$\mathbf{Q} = \mathbf{R}_A - \mathbf{R}_B$$
$$\mathbf{r}_P = \tfrac{1}{p}(\alpha\mathbf{R}_A + \beta\mathbf{R}_B)$$

- In a large system there are only *O*(N) ‚significant' Gaussian products.
- The should be precomputed and stored as a list (e.g. cut-off $K_{AB} >= T_{cut}$)
- The significant bra- and ket-products interact via the 1/r operator (never small!).
- ***There are O(N²) non-negligible integrals***

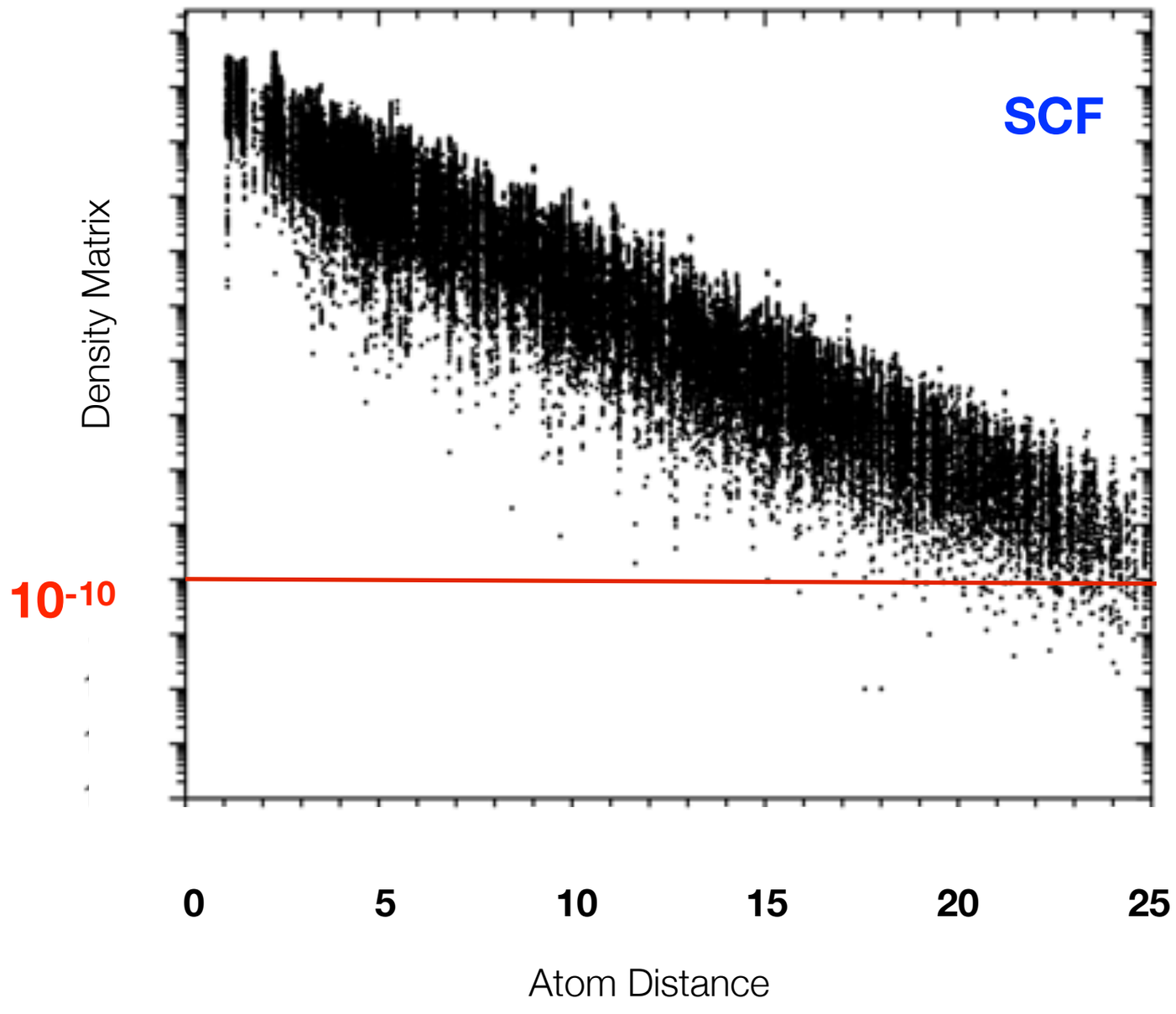
The principle of „Direct SCF“

$$F_{\mu\nu} = h_{\mu\nu} + G_{\mu\nu} \quad G_{\mu\nu} = \sum_{\kappa\tau} P_{\kappa\tau} \left[(\mu\nu \mid \kappa\tau) - (\mu\kappa \mid \nu\tau) \right]$$

```
G=0
loop μ
     loop ν≤μ
          loop κ
               loop τ≤κ (μν≤κτ)
                    test= IntegralEstimate(μ,ν,κ,τ)
                          *max(P(κ,τ) ,P(ν,τ) ,...)
                    if (test<Thresh) skip
                    else
                      Calculate (μν|κτ)
                      add G(μ,ν)+=P(κ,τ)(μν|κτ)  (Coulomb)
                      add G(μ,κ)-=P(ν,τ)(μν|κτ)  (Exchange)
                      (and permutations of indices)
                    end (else)
end loops μ,ν, κ,
```

- Only contributions >= Thresh go into the Fock matrix
- Better than testing for small integrals alone since P can be large

Kohn's Conjecture and the Density Matrix



For an insulator (finite HOMO-LUMO gap), the density matrix decays roughly exponentially with distance

(what we mean by that is the distance between the atoms the basis functions are attached to)

$$P_{\mu_A \nu_B}$$

- The decay is exponential, but slow. 10^{-10} is only reached at 20-25 Angström!
- Nevertheless, in insulators, there are only *O*(N) significant density matrix elements

Intrinsic Scaling of Coulomb and Exchange

Assuming exponential decay of the density, Almlöf realized that the intrinsic scaling of the Coulomb and exchange terms is different:

Coulomb:

$$J_{\mu_A\nu_B} = \sum_{\kappa\tau} \underbrace{P_{\kappa_C\tau_D}}_{\propto \exp(-const\times R_{CD})} \underbrace{\left(\underbrace{\mu_A\nu_B}_{\propto \exp(-const\times R_{AB})} \middle| \underbrace{\kappa_C\tau_D}_{\propto \exp(-const\times R_{CD})} \right)}_{\exp(-const\times(R_{AB}+R_{CD}+R_{CD}))} \Rightarrow O(N^2)$$

Exchange:

$$K_{\mu_A\nu_B} = \sum_{\kappa\tau} \underbrace{P_{\kappa_C\tau_D}}_{\propto \exp(-const\times R_{CD})} \underbrace{\left(\underbrace{\mu_A\kappa_C}_{\propto \exp(-const\times R_{AC})} \middle| \underbrace{\nu_B\tau_D}_{\propto \exp(-const\times R_{BD})} \right)}_{\exp(const\times(R_{AC}+R_{BD}+R_{CD}))} \Rightarrow O(N)$$

Conclusion: Use the most efficient way to calculate or approximation each term separately!

Integral Estimates (I): Almlöf's estimate

In order to not decide that we do not calculate an integral, we need an estimate for it

Look at an ERI: $$(\mu\nu|\kappa\tau) = \int\int \frac{\mu(\boldsymbol{r}_1)\nu(\boldsymbol{r}_1)\kappa(\boldsymbol{r}_2)\tau(\boldsymbol{r}_2)}{|\boldsymbol{r}_1 - \boldsymbol{r}_2|} d\boldsymbol{r}_1 d\boldsymbol{r}_2$$

Let us preted for a moment that r_{12}^{-1} is not there. Then:

$$\int\int \mu(\boldsymbol{r}_1)\nu(\boldsymbol{r}_1)\kappa(\boldsymbol{r}_2)\tau(\boldsymbol{r}_2) d\boldsymbol{r}_1 d\boldsymbol{r}_2 = \int \mu(\boldsymbol{r}_1)\nu(\boldsymbol{r}_1) d\boldsymbol{r}_1 \int \kappa(\boldsymbol{r}_2)\tau(\boldsymbol{r}_2) d\boldsymbol{r}_2 = S_{\mu\nu}S_{\kappa\tau}$$

Now assume that the bra- and ket distributions are centered at

$$\boldsymbol{R}_{\mu\nu} = \langle\mu|\boldsymbol{r}|\nu\rangle \quad \boldsymbol{R}_{\kappa\tau} = \langle\kappa|\boldsymbol{r}|\tau\rangle$$

Now set $r_{12}^{-1} \approx \left|\boldsymbol{R}_{\mu\nu} - \boldsymbol{R}_{\kappa\tau}\right|^{-1} \equiv R^{-1}$

And arrive at $$(\mu\nu|\kappa\tau) \approx \frac{S_{\mu\nu}S_{\kappa\tau}}{R}$$

Estimate $(\mu\nu|\kappa\tau)_{est-I} = max\left(\left|\frac{S_{\mu\nu}S_{\kappa\tau}}{R}\right|\right)$

(Over the members of the 4 shells)

- ➡ NOT a rigorous upper bound
- ➡ Reasonably cheap to compute
- ➡ Does take the R dependence into account to some extent

Integral Estimates (II): Ahlrich's estimate

Häser and Ahlrichs used the Schwartz inequality to show:

$$(\mu\nu|\kappa\tau) \leq \sqrt{(\mu\nu|\mu\nu)}\sqrt{(\kappa\tau|\kappa\tau)} = Q_{\mu\nu}Q_{\kappa\tau}$$

Estimate $(\mu\nu|\kappa\tau)_{est-II} = max(Q_{\mu\nu}Q_{\kappa\tau})$

(Over the members of the 4 shells)

- Is a rigorous upper bound
- Is cheap to compute
- Does NOT depend on R and hence will strongly overestimate integrals with well separated bra and ket distributions

Integral Estimates (III): Multipole Estimate

Let us take two expansion points $\boldsymbol{R}_{\mu\nu} = \langle\mu|\boldsymbol{r}|\nu\rangle \quad \boldsymbol{R}_{\kappa\tau} = \langle\kappa|\boldsymbol{r}|\tau\rangle$

And express the two charge distributions in terms of their (real, spherical) multipoles:

$$M_{LM}^{\mu\nu} = \sqrt{\frac{4\pi}{2L+1}}\langle\mu|r^L S_{LM}|\nu\rangle$$

For one-center charge distributions $L = |l_\mu - l_\nu| \ldots l_\mu + l_\nu$

Assuming the two local coordinate systems are aligned and the charge distributions are not overlapping, the **bipolar expansion** yields:

$$(\mu\nu|\kappa\tau)_{multipole} = \sum_{L_1}\sum_{L_2}\sum_{m=-l_<}^{l_<} f_{L_1L_2m}\frac{M_{L_1m}^{\mu\nu}M_{L_2m}^{\kappa\tau}}{R^{L_1+L_2+1}}$$

$$f_{L_1L_2m} = \frac{(-1)^{L_2+|m|}}{\sqrt{(L_1+L_2)!\,(L_1+m)!\,(L_2+m)!\,(L_1-|m|)!\,(L_2-|m|)!}}$$

Bühler, RJ, Hirschfelder, JO Bipolar Expansion of Coulombic Potentials, *Phys. Rev.*, **1951**, 83, 628-633

The multipole formula **becomes fully accurate** (at least 16 digits) once the charge distributions don't overlap.

✓ Evaluating the **multipole formula exactly** is too costly - the estimate may become as expensive or more expensive than the actual integral calculation

✓ For the purpose of pre-screening, one should only be interested in the **lowest multipole** interaction, because it is the one that covers the longest distances:

Estimate $$(\mu\nu|\kappa\tau)_{est-III} = max\left(\left|f_{L_1^{min}L_2^{min}m}\right|_{max}\frac{\left|M^{\mu\nu}_{L_1^{min}m}\right|\left|M^{\kappa\tau}_{L_2^{min}m}\right|}{R^{L_1^{min}+L_2^{min}+1}}\right)$$

➡ Will break down for overlapping charge distributions overlap

➡ Not cheap to compute

➡ Misses higher order multipole contributions.

➡ NOT an upper bound, i.e. Will perhaps dramatically *underestimate* the integral @medium R

Integral Estimates (IV): The „QQR“ and „CSAM“

- Lambrecht and Ochsenfeld *J. Chem. Phys.*, **2005**, 123, 184102 derived rigorous upper bounds on the basis of the multipole expansion (too expensive in practice)

- Maurer, Ochsenfeld et al. *J. Chem. Phys.*, **2012**, 136, 144107 realized that higher multipoles can be simulated by the Schwartz integral and proposed the „QQR“ estimate:

Estimate $(\mu\nu|\kappa\tau)_{est-IV} = max\left(\frac{Q_{\mu\nu}Q_{\kappa\tau}}{R - ext_{\mu\nu} - ext_{\kappa\tau}}\right)$

The ***extent*** of a charge distribution is defined by:

$$ext_{\mu\nu} = \sqrt{\frac{2}{\alpha+\beta}}\, erf^{-1}(1-\tau)$$

$$\alpha, \beta = exponents, \tau \approx 10^{-4} - 10^{-6}, erf^{-1} = inverse\ error\ function$$

➡ Thompson and Ochsenfeld et al. *J. Chem. Phys.*, **2017**, 147, 144101 further tweaked the QQR by realizing that the distance dependence can be simplified

$$(\mu\nu|\kappa\tau) \leq \sqrt{(\mu\mu|\kappa\kappa)}\sqrt{(\nu\nu|\tau\tau)} = T_{\mu\kappa}T_{\nu\tau}$$

$$(\mu\nu|\kappa\tau) \leq \sqrt{(\mu\mu|\tau\tau)}\sqrt{(\nu\nu|\kappa\kappa)} = T_{\mu\tau}T_{\nu\kappa}$$

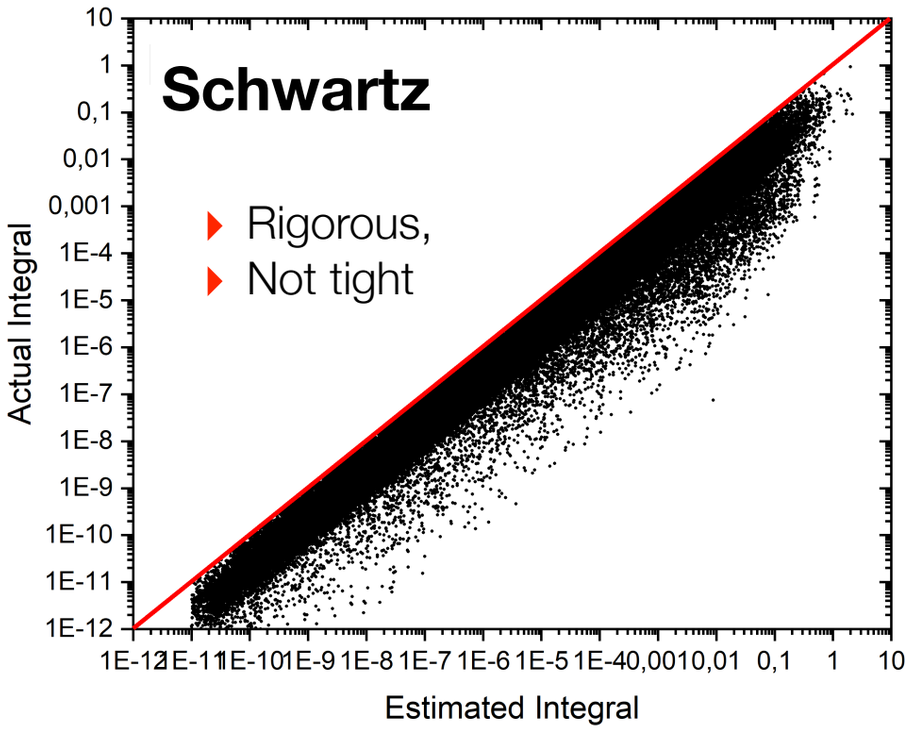
Which features the distance dependence of the interacting bra/ket distributions.

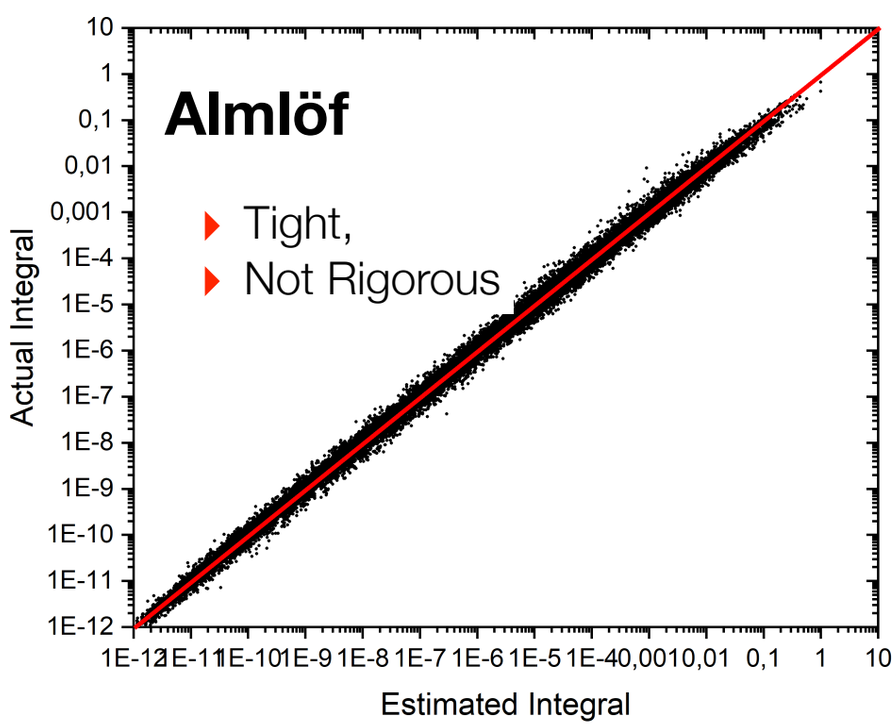
Defining: $$\tilde{T}_{\mu\kappa} = \frac{T_{\mu\kappa}}{\sqrt{Q_{\mu\mu}Q_{\kappa\kappa}}}$$

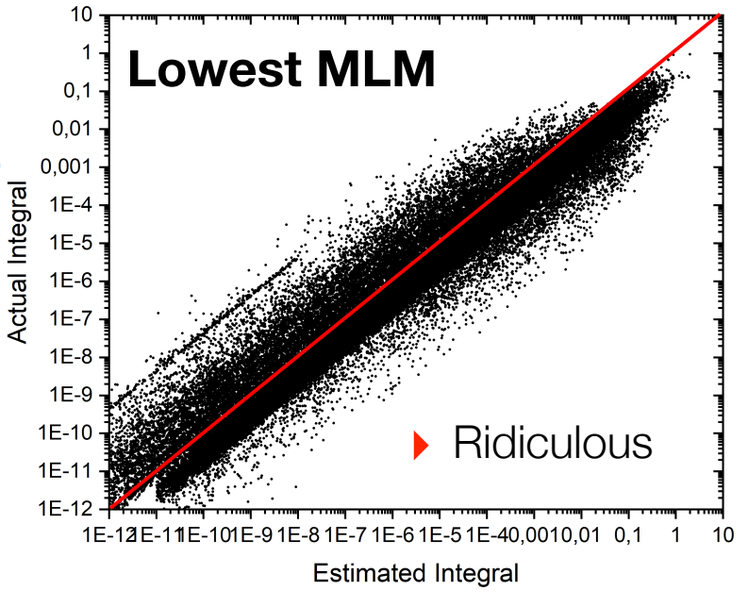
Gives the final (CSAM) estimate:

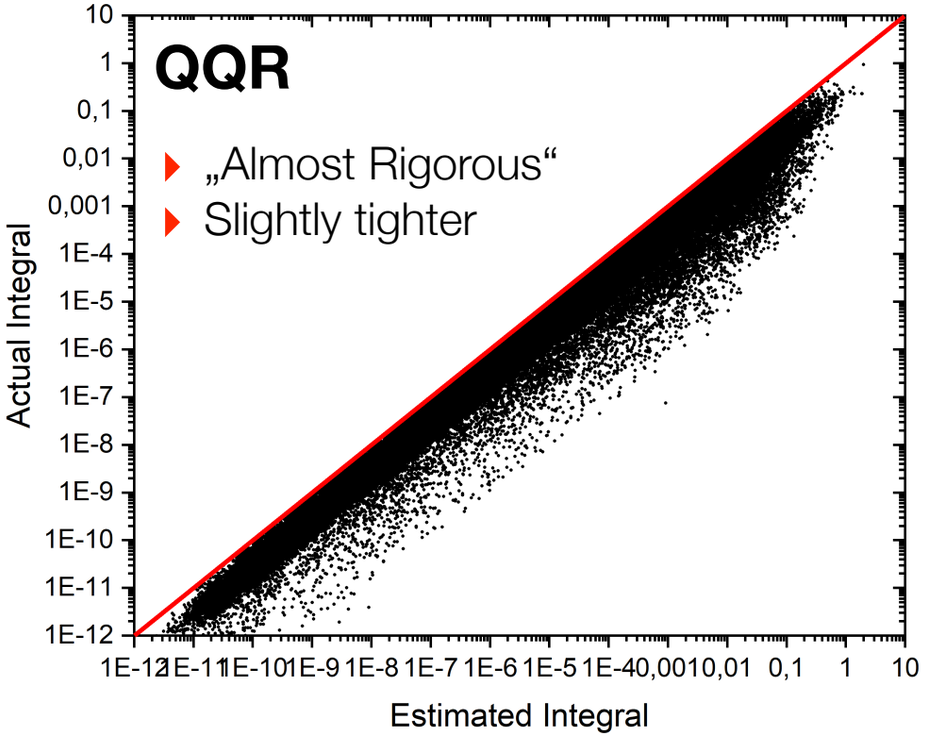
Estimate $$(\mu\nu|\kappa\tau)_{est-V} = Q_{\mu\nu}Q_{\kappa\tau}max\left(\tilde{T}_{\mu\kappa}\tilde{T}_{\nu\tau}, \tilde{T}_{\mu\tau}\tilde{T}_{\nu\kappa}\right)$$

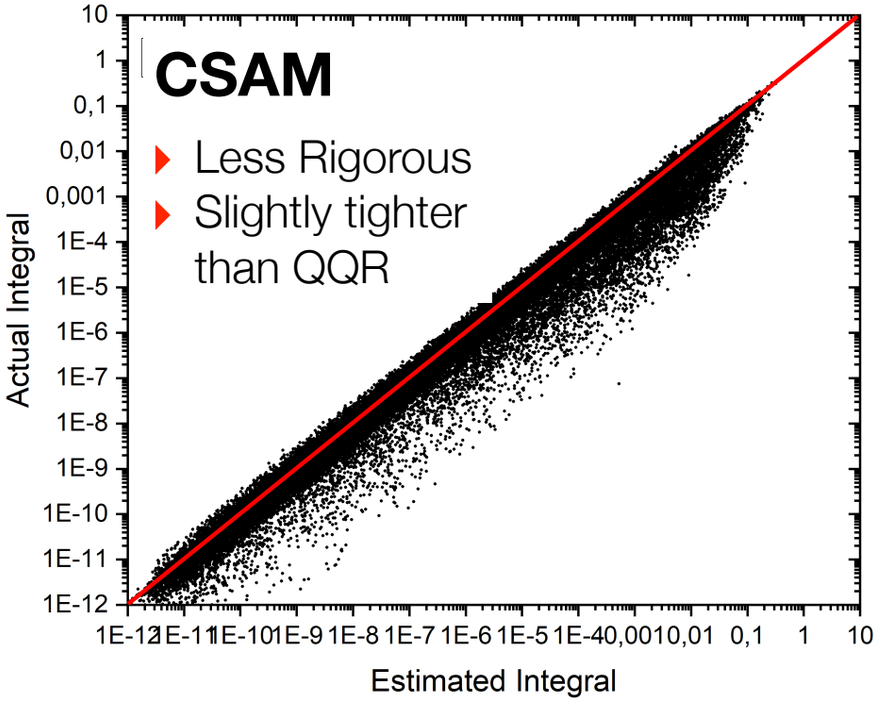
Comparison of Estimates





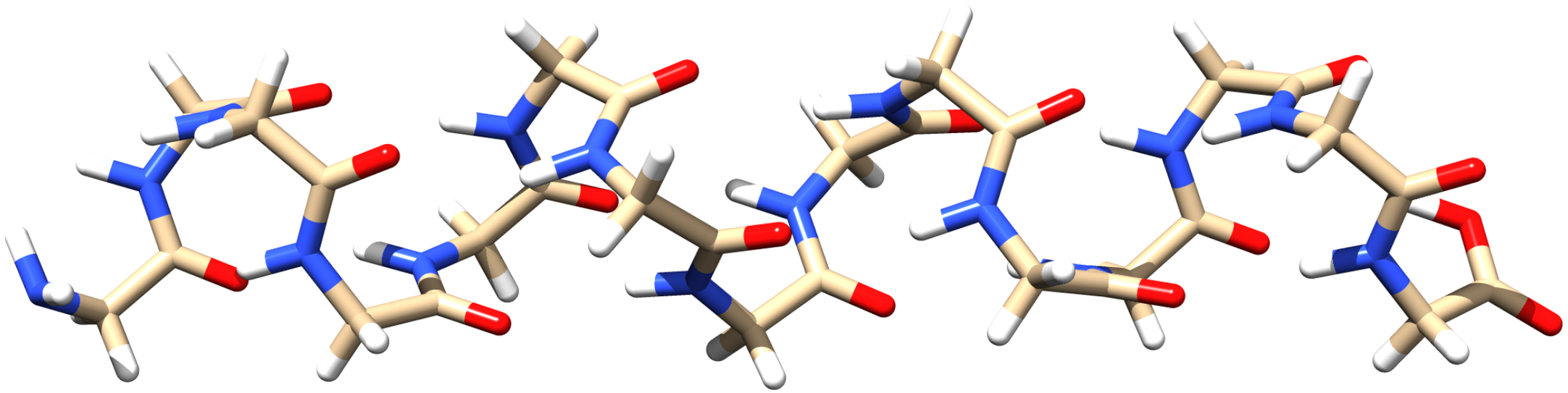






$(gly)_3$ / def2-SV(P)

Performance in practice: $(Gly)_{15}$/def2-SVP



	#(Cycles)	Energy (Eh)	#(Fock time/sec)	
Schwartz	11	-3175.706180418709	1822	
QQR	11	-3175.706179146775	1659	(9%)
CSAM	11	-3175.706177491313	1561	(15%)
Almloef	Wild divergence			

… 0.3 Microhartree loss of accuracy for 15% performance gain
(Will be more for larger systems)

Pre-screening: Wrapping up

- ✓ The best way to speed up a computation is to not do it :-)
- ✓ Identifiying near zero's is and art & science that is not done even after 30+ years
- ✓ In skipping small contributions:
  - It is good but not strictly necessary to have rigorous upper bounds
  - Numerical stability must never be sacrificed

Always remember:

> - **Computing a bad number fast is useless because it is still a bad number**
> - **First the approximation has to meet a specified accuracy goal, then it *can* be fast**

PART 2

How to compute things you cannot avoid efficiently

Chapter 1:

Scaling Laws and Their Impact on Algorithms

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:

$$T = aN^b$$

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 **x** **Scaling of loops required to obtain it**

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

➡ The number of occupied *and* virtual MOs is proportional to system size

➡ The Number of AOs is proportional to system size

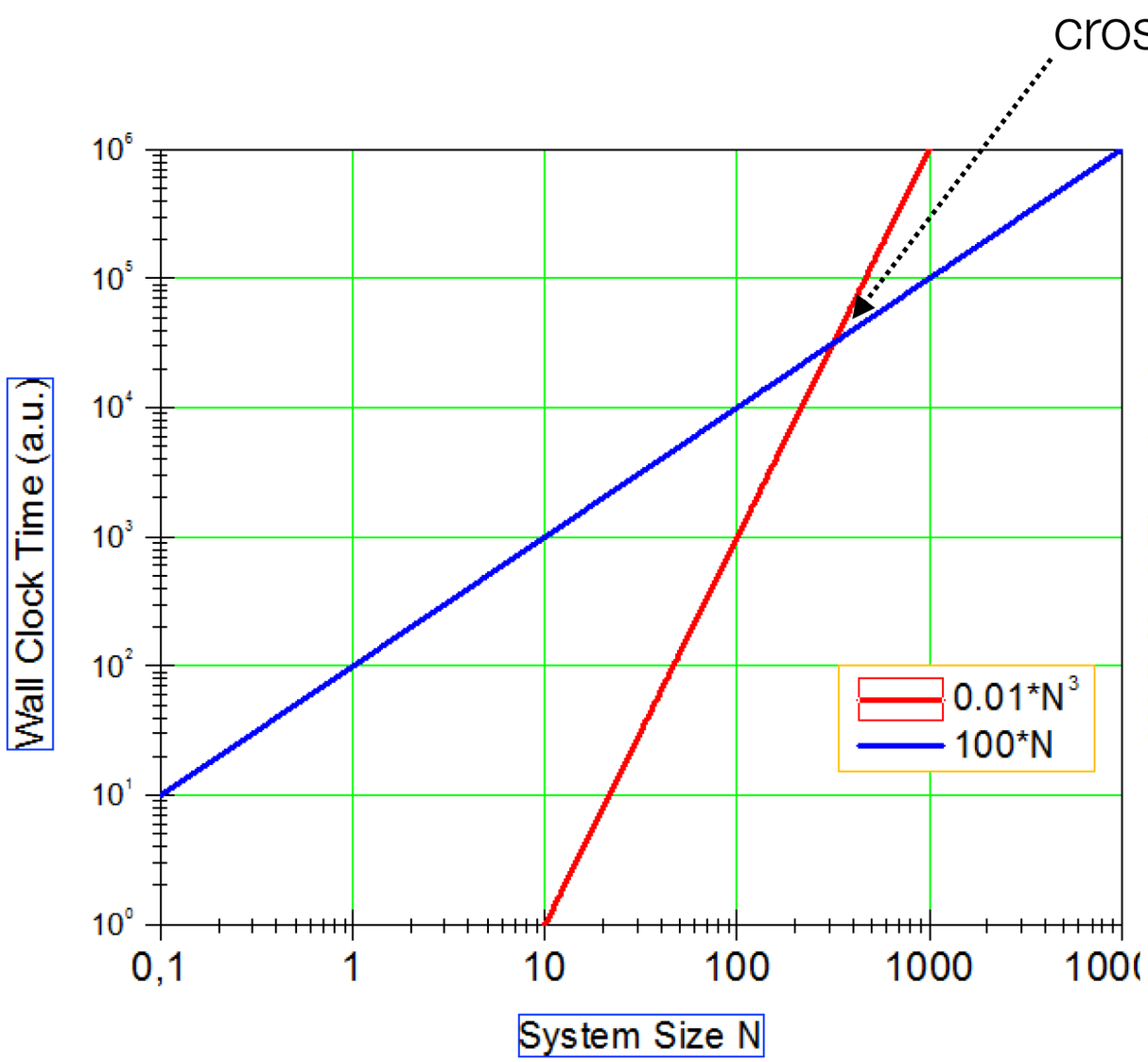
$(\mu\nu \mid \kappa\tau)$ Number of AOs integrals proportional to N^4 ($O(N^4)$)

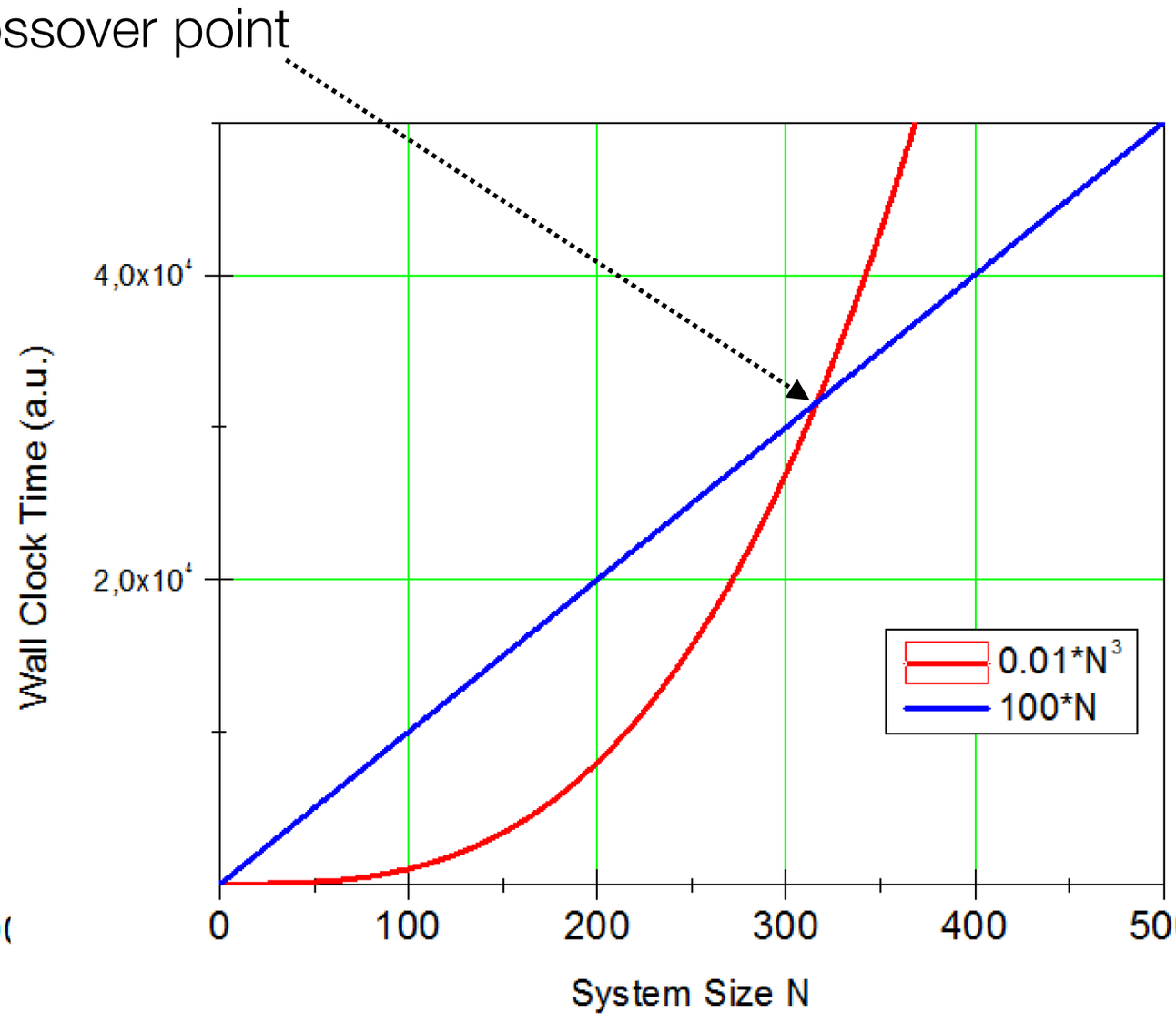
$$(ia \mid jb) = \sum_\mu \sum_\nu \sum_\kappa \sum_\tau c_{\mu i} c_{\nu a} c_{\kappa j} c_{\tau b} (\mu\nu \mid \kappa\tau)$$

$O(N^4)$ $O(N)$ $O(N)$ $O(N)$ $O(N)$ $\longrightarrow$ **$O(N^4)$**

$O(N^8)$

Prefactor vs Scaling



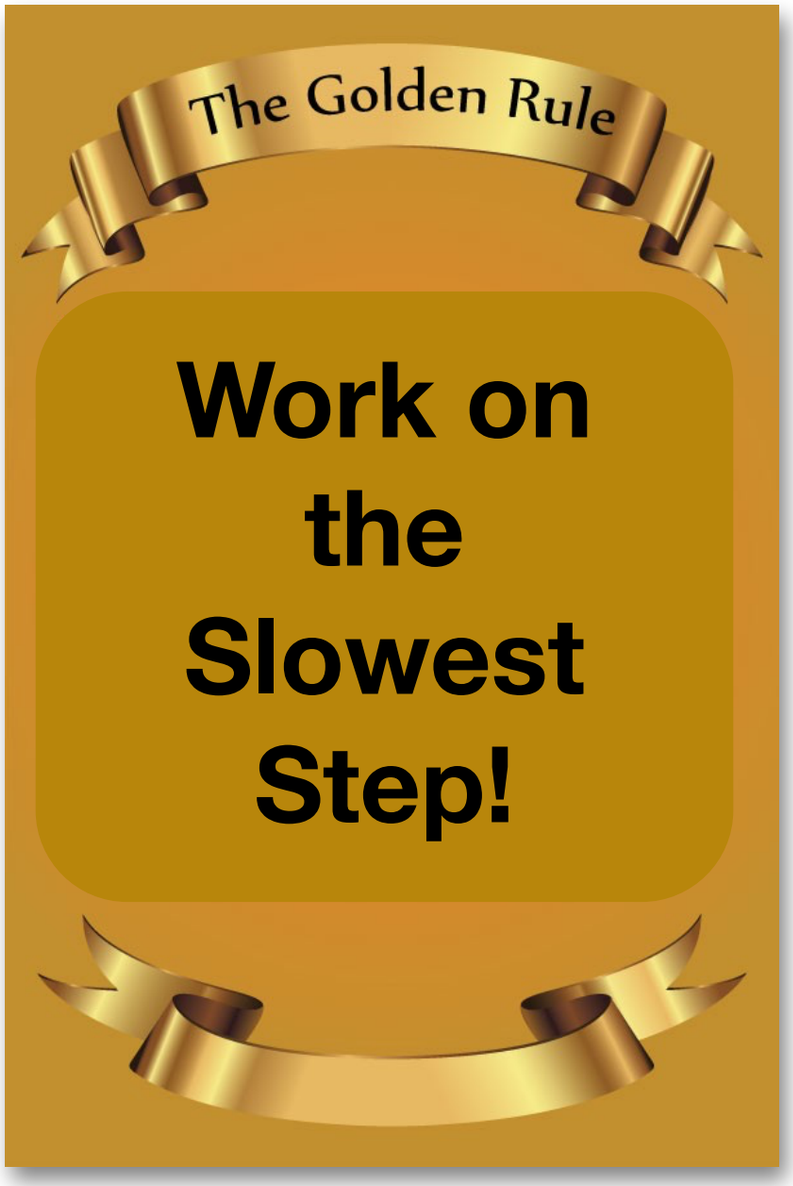


For many applications nonlinear scaling with a small prefecture is the preferred choice

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

Golden Law of Development

- ✓ 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!

```
Total execution time                ...  153019.575 sec

Localization of occupied MO's       ...    7516.449 sec (  4.9%)
Fock Matrix Formation               ...   11392.614 sec (  7.4%)
First Half Transformation           ...   37824.285 sec ( 24.7%)
RI-PNO integral transformation      ...   17832.376 sec ( 11.7%)
Initial Guess                       ...    5376.961 sec (  3.5%)
DIIS Solver                         ...    8855.850 sec (  5.8%)
State Vector Update                 ...       1.744 sec (  0.0%)
Sigma-vector construction           ...    8177.969 sec (  5.3%)
  <0|H|D>                           ...       0.072 sec ( 0.0% of sigma)
  <0|H|S>                           ...       0.003 sec ( 0.0% of sigma)
  <D|H|D>(0-ext)                    ...     575.591 sec ( 7.0% of sigma)
  <D|H|D>(2-ext Fock)               ...       1.921 sec ( 0.0% of sigma)
  <D|H|D>(2-ext)                    ...    1512.608 sec ( 18.5% of sigma)
  <D|H|D>(4-ext)                    ...     684.157 sec ( 8.4% of sigma)
  <D|H|D>(4-ext-corr)               ...    2880.920 sec ( 35.2% of sigma)
  CCSD doubles correction           ...      33.534 sec (  0.4% of sigma)
  <S|H|S>                           ...      78.695 sec ( 1.0% of sigma)
  <S|H|D>(1-ext)                    ...      79.135 sec ( 1.0% of sigma)
  <D|H|S>(1-ext)                    ...       5.117 sec ( 0.1% of sigma)
  <S|H|D>(3-ext)                    ...      28.949 sec ( 0.4% of sigma)
  CCSD singles correction           ...       0.108 sec (  0.0% of sigma)
  Fock-dressing                     ...    1541.152 sec ( 18.8% of sigma)
  Singles amplitudes                ...      15.255 sec (  0.2% of sigma)
  (ik|jl)-dressing                  ...     441.823 sec (  5.4% of sigma)
  (ij|ab),(ia|jb)-dressing          ...     213.171 sec (  2.6% of sigma)
  Pair energies                     ...       1.235 sec (  0.0% of sigma)
Total Time for the density          ...     632.934 sec (  0.4% of ALL)
Total Time for computing (T)        ...   32529.433 sec ( 21.3% of ALL)
```

This is worth your while!

How much can you gain from optimizing these steps?

Chapter 2

Writing Efficient Programs

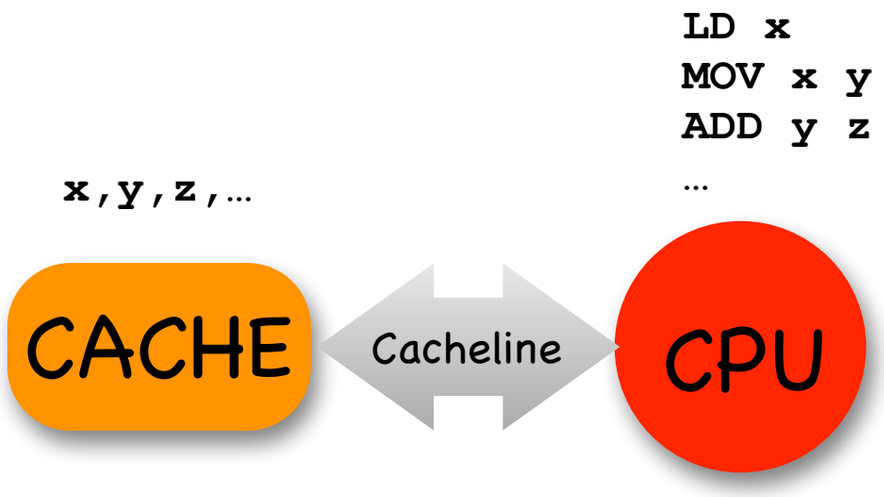
The Do's and Don't's of Programming: Overview

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

Chapter 2.1

Unit stride and avoiding short loops

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:

```
x=0
for (i=0;i<2048;i++) x=x+y[i]
```

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

Bad:

```
x=0
for (i=0;i<2048;i++) x=x+y[yorder[i]]
```

or

```
for (i=0;i<2048;i++) x=x+y[i]-y[N-i-1]
```

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.

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: $G^{A}_{abc;\alpha} = (x - X_A)^a (y - Y_A)^b (z - Z_A)^c \exp(-\alpha r_A^2)$

Repulsion integral in MD:

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

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

} *recursive formulas. Nested loops of length ~l_A+l_B (or l_C+l_D)*

```
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']*(-1)t'+u'+v'
                  *R[t+t'][u+u'][v+v']
            end loops t',u',v'
         end loops t,u,v
         ELREP[ixyz][jxyz][kxyz][lxyz]=x
end loops i,j,k,lxyz
```

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

Example: Short Loops and Multidimensional Arrays

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

```
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;
```

```
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

	unoptimized code	unrolled code	libint	speedup
(ss|ss) (10^7 times)	**1.8**	**1.2**	**0.7**	**(3x)**
(pp|pp) (10^6 times)	**8.3**	**2.6**	**0.4**	**(21x)**
(dd|dd) (10^4 times)	**4.1**	**0.4**	**0.1**	**(41x)**
(ff|ff) (10^3 times)	**9.1**	**0.5**	**0.2**	**(45x)**

„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

A Real Life Example

In the theory of multipole interactions, There occurs the **multipole interaction tensor:**

$$T_{LM,L'M'}(\boldsymbol{R}_P - \boldsymbol{R}_Q) = (-1)^{L'} I_{L+L'}^{M+M'*}(\boldsymbol{R}_P - \boldsymbol{R}_Q)$$

$$= (-1)^{L'} \sqrt{\frac{4\pi(L+M)!\,(L-M)!}{(2L+1)}} R_{PQ}{}^{-L-1} Y_{L+L'}^{M+M'*}(\theta_{PQ}, \phi_{PQ})$$

(**R**P, **R**Q are the multipole expansion centers, we need all l=0..L, l'=0..L')

Let us write a subroutine for that - easy, we only have to evaluate some spherical harmonics, right?

... Better version

```
void MultipoleInteractionTensor_ILMreal_Fame(double *PQ, double &RPQ, int LmaxBra, int LmaxKet, TRMatrix &T){

  //Load all required ILM at once
  Load_AllIlm_stddef(LmaxBra+LmaxKet,X,Y,Z,R,ILM.p);  no redundant computation

  // Start looping over angular momenta
 int64 LM1=0;
  double *TLM1 = &(T(0,0));
  for (int64 L1=0;L1<=LmaxBra;L1++){
    for (int64 M1=-L1;M1<0;M1++){
```

Correct loop order for unit stride access (Bra indices)

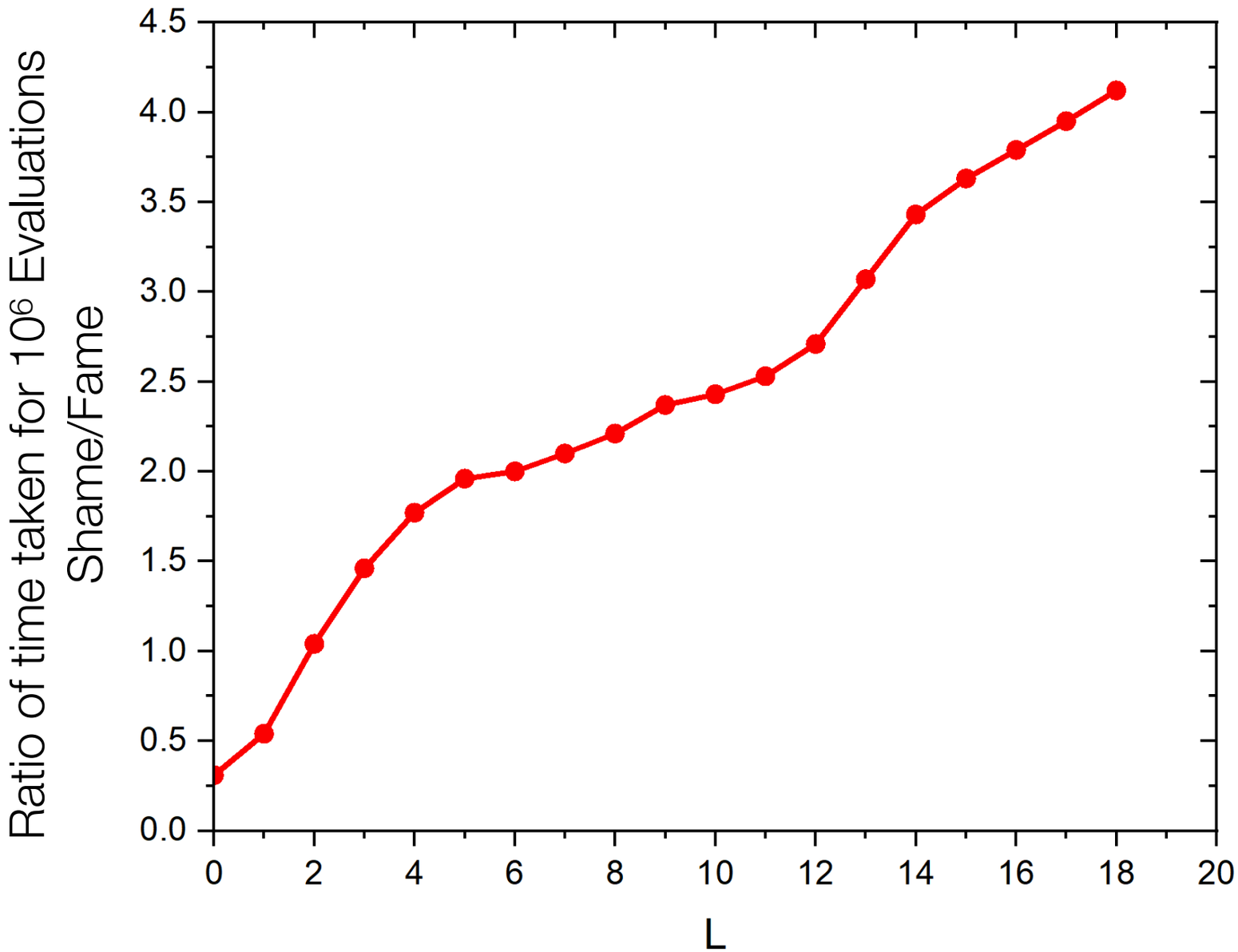
```
int64 LM2 = 0;
for (int64 L2=0;L2<=LmaxKet; L2++){
 // Pointer to correct ILM vector
  double *ILre  = &(ILM(L12,0     ));
  double phase1 =  (L2%2==0)? 1.0 : -1.0;
    // Now we have to distinguish four cases to form Trygve's real interaction tensor
  for (int64 M2=-L2;M2<0;M2++){
    // Common prefactor
    double PreFac    = phase1 * 2.0;
    double ILM_P_re  = ILre[M12Paddr];  // Real part      of I( L1+L2, M1+M2)
    double ILM_M_re  = ILre[M12Maddr];  // Real part      of I( L1+L2, M1-M2)
    TLM1[LM2]        =  PreFac* (-ILM_P_re + phase2 * ILM_M_re);
    LM2++;
    ////////////////////////
  }// M2
  ... three more loops like this for the three other cases
```

Correct loop order for unit stride access (Ket indices)

No silly array access for phase factors

No Logic in the innermost loop

MultipoleInteractionTensor Shame vs Fame



➡ The version of the same function that has memory optimized unit stride access performs a **FACTOR of 2-4** faster for non-trivial angular momenta

Chapter 2.2

Making the most out of using External Libraries

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:

$$\mathbf{C} = \mathbf{AB} \quad C_{ij} = \sum_k A_{ik} B_{kj}$$

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

Which we can program as follows:

```
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)

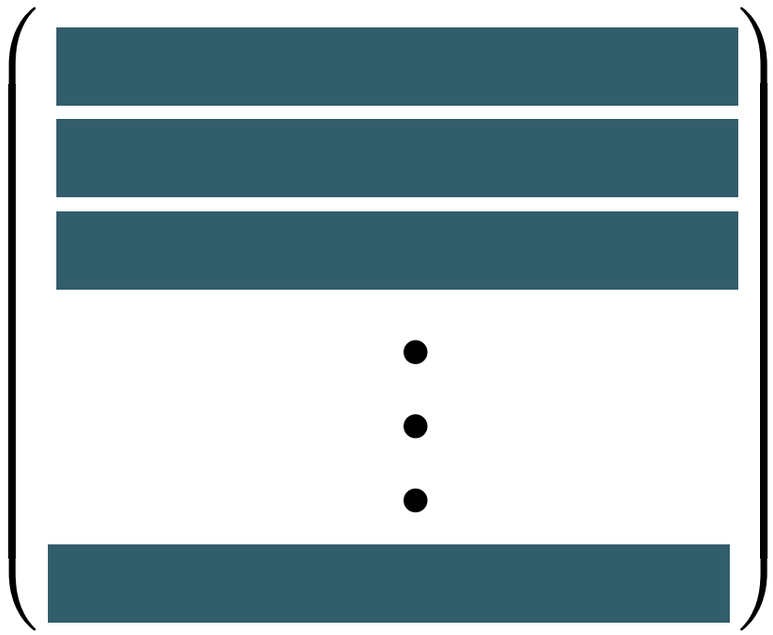
For two densely filled essentially random, square matrices A and B with N=2750

		directly programmed	BLAS (dgemm)
$\mathbf{C} = \mathbf{AB}$	:	99	1.7
$\mathbf{C} = \mathbf{AB}^T$	:	11	1.7
$\mathbf{C} = \mathbf{A}^T\mathbf{B}$	:	104	1.7

USE BLAS LEVEL 3 (DGEMM) WHENEVER YOU CAN!)

Why that?

$\mathbf{A} =$



- ✓ The matrices are arrange 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)

Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz

Example: The power of LAPACK

Example: 3000x3000 matrix

	Hand written	Intel-MKL	
Diagonalization	**28.1 sec**	`dsyevr` **5.3 sec**	**~5x**
Cholesky decomposition	**2.4 sec**	`dpotrf` **0.2 sec**	**~12x**
Singular value decomposition	**315.0 sec**	`dgesvd` **21.7 sec**	**~25x**

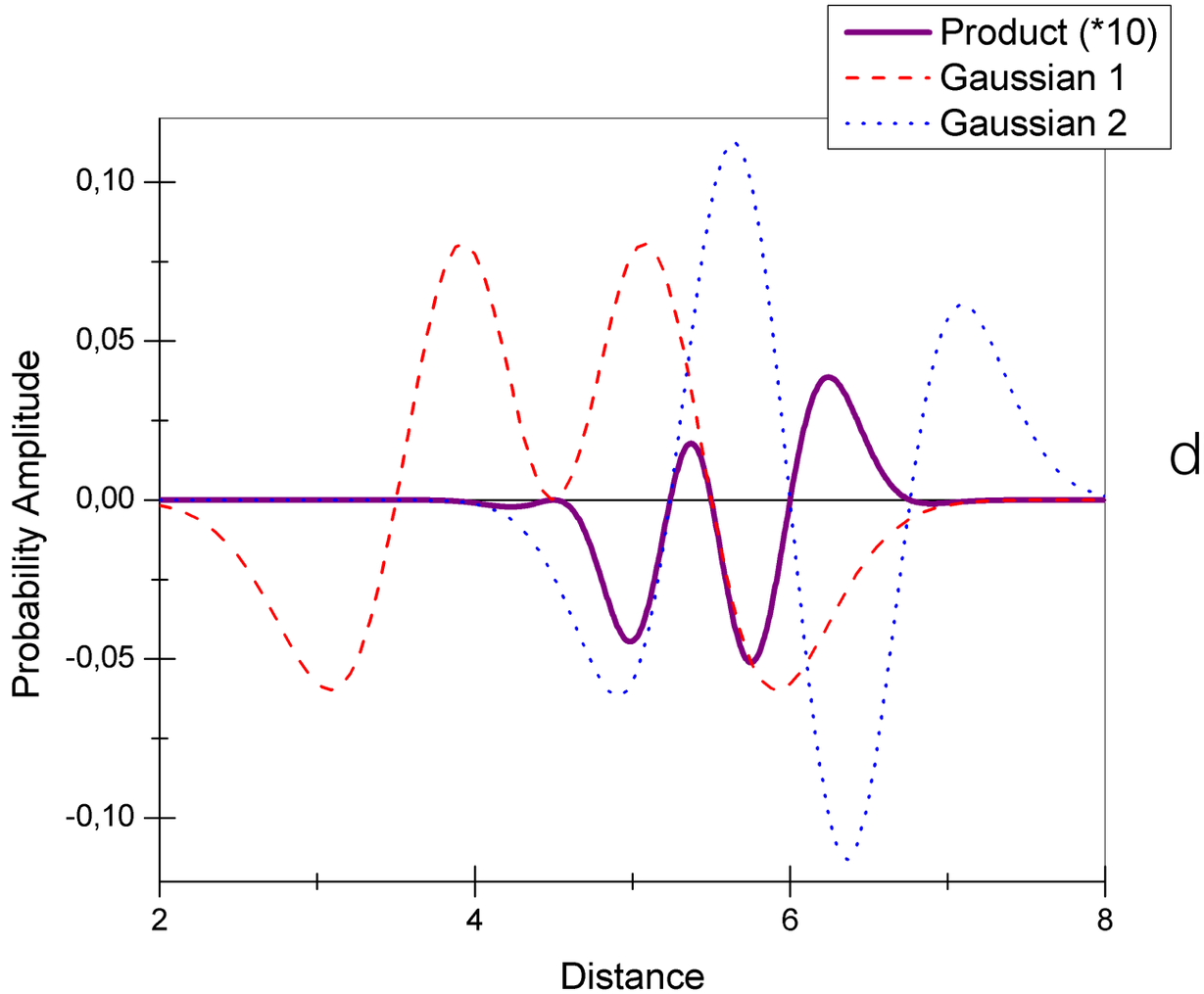
Intel(R) Core(TM) i7-4810MQ CPU @ 2.80GHz

Application: Integrals over Gaussians

The form of a Gaussian $$\varphi_\mu^A(\mathbf{r}) = N_{abc}(x - X_A)^{a_\mu}(y - Y_A)^{b_\mu}(z - Z_A)^{c_\mu}\exp(-\alpha_\mu r_A^2)$$

Multiplying two Gaussians

$$\varphi_\mu^A(\mathbf{r})\varphi_\nu^B(\mathbf{r}) = N_\mu N_\nu K_{AB}\underbrace{(x - X_A)^{a_\mu}(y - Y_A)^{b_\mu}(z - Z_A)^{c_\mu}(x - X_B)^{a_\nu}(y - Y_B)^{b_\nu}(z - Z_B)^{c_\nu}}_{\textbf{VERY NASTY!}}\underbrace{\exp(-\alpha_P r_P^2)}_{\textbf{Nice!}}$$



difficult to integrate in 3 dimensions

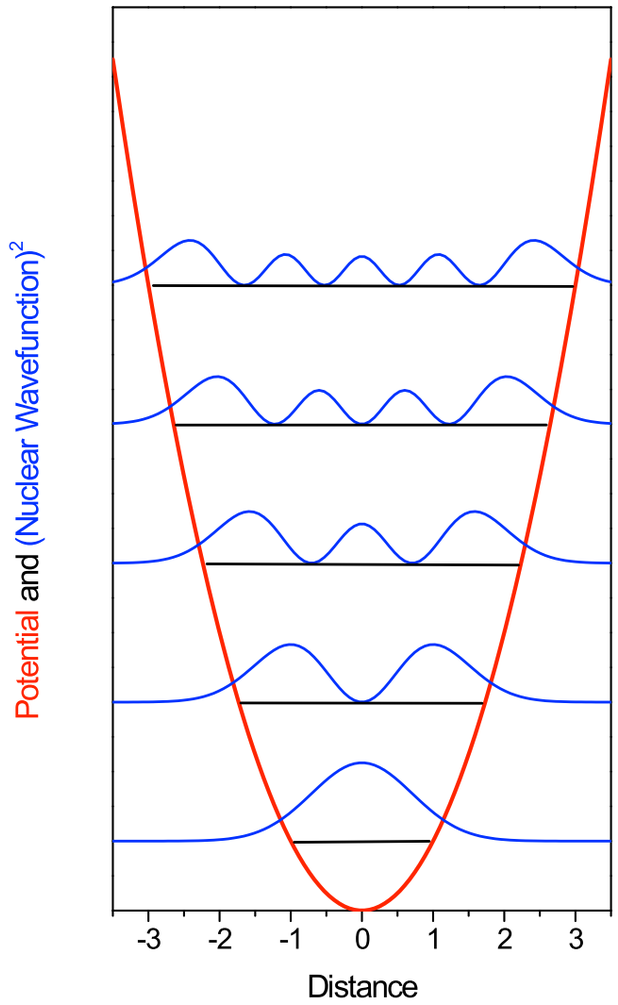
The McMurchie Davidson Method (I)

If we have a nasty polynomial, we can expand it in terms of nice polynomials

➡ Orthonormal harmonic oscillator eigenfunctions (**Hermite polynomials**)

$$H_n(x) = (-1)^n \exp(x^2) \frac{d^n}{dx^n} \exp(x^2)$$

$$H_0(x) = 1$$
$$H_1(x) = 2x$$
$$H_2(x) = 4x^2 - 2$$
$$H_3(x) = 8x^3 - 12x$$



In one dimension:

$$(x - X_A)^{a_\mu} (x - X_B)^{a_\nu} = \sum_{t=0}^{a_\mu + a_\nu} E_t^{a_\mu a_\nu} H_t (x - X_P)$$

Recursion relation

$$E_t^{i+1,j} = \frac{1}{2p} E_{t-1}^{ij} + (X_P - X_A) E_t^{ij} + (t+1) E_{t+1}^{ij}$$

$$E_0^{00} = 1$$

(a) McMurchie, L.E.; Davidson, E. (**1978**) J. Comp. Phys. 26,218

(b) Helgaker, T.; Raylor, P.R. (**1995**) in: Yarkony (Ed.) Modern Electronic Structure Theory, World Scientific, 725ff

McMurchie-Davidson in Matrix Form

Think of $E_t^{a_\mu a_\nu} E_u^{b_\mu b_\nu} E_v^{c_\mu c_\nu}$ $E_{t'}^{a_\kappa a_\tau} E_{u'}^{b_\kappa b_\tau} E_{v'}^{c_\kappa c_\tau}$ $R_{t+t',u+u',v+v'}$

As matrices $E^{bra}(tuv,\mu\nu)$ $E^{ket}(t'u'v',\kappa\tau)$ $R(tuv,t'u'v')$

Then we have: $(\mu^A \nu^B \mid \kappa^C \tau^D) = (\mathbf{E}^{bra} \mathbf{R} \mathbf{E}^{ket})_{\mu\nu,\kappa\tau}$

Obviously, not all *tuv* combinations occur for each member of the shell pair

#(tuv) combinations as a function of angular momenta

L_μ / L_ν	0	1	2	3	4
0	1	4	10	20	35
1		10	20	35	56
2			35	56	84
3				84	120
4					165

Advantage of Factorisation

E-Matrix $E\left(tuv,\mu\nu\right)$ (Pre-computed; linear scaling) $O(L^5)$

$L^3 \quad L^2$

R-Matrix $R\left(tuv,t'u'v'\right)$ (Generated on the fly) $O(L^6)$

$L^3 \quad L^3$

RE-product $T(tuv,\kappa\tau)=\sum_{t'u'v'} R(tuv,t'u'v')E\left(t'u'v',\kappa\tau\right)$ $O(L^8)$

$L^3 \; L^2 \qquad L^2 \quad L^3 \quad L^3 \quad L^2$

TE-product $I(\mu\nu,\kappa\tau)=\sum_{tuv} E\left(tuv,\mu\nu\right)T(tuv,\kappa\tau)$ $O(L^7)$

$L^2 \; L^2 \qquad L^3 \; L^2 \quad L^3 \; L^2$

➡ **Never worse than *O*(L⁸) which is better than O(L¹⁰) in the original MD**

FN The SHARK integral generation and digestion system, *J. Comp. Chem.*, **2022**, 1-16 (DOI: 10.1002/jcc.26942)

SHARK vs Libint: „in vitro“

Contraction Depth

Angular Momentum

*I**D*	1	2	3	4	5	6
0	1.371	1.724	2.073	2.268	2.347	2.430
1	1.100	1.003	1.052	1.169	1.157	1.174
2	1.199	0.765	0.778	0.793	0.802	0.791
3	5.471	3.628	3.610	3.355	3.787	3.406
4	8.758	5.284	5.210	5.229	5.264	5.227
5	9.792	5.589	4.409	4.360	4.655	4.788
6	18.567	5.912	4.852	4.704	4.751	4.604
7	31.497	6.749	5.360	5.227	5.412	5.031

Notes: The numbers in the table give tm(Libint)/tm(SHARK), where tm(X) is the time taken by algorithm X to calculate the indicated number of integral batches.

<1 Libint is faster

>1 SHARK is faster

FN The SHARK integral generation and digestion system, *J. Comp. Chem.*, **2022**, 1-16 (DOI: 10.1002/jcc.26942)

Chapter 2.3

Finding Algorithms with Minimal FLoating Point Operations

(… and whether this is the ultimate goal)

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 $(\mu\nu \mid \kappa\tau) \rightarrow (ia \mid jb)$

i,j= occupied MOs (#=O), a,b, unoccupied MOs (#=V), μ,ν,κ,τ=basis functions (#=B)

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

Naive: $$(ia \mid jb) = \sum_\mu \sum_\nu \sum_\kappa \sum_\tau c_{\mu i} c_{\nu a} c_{\kappa j} c_{\tau b} (\mu\nu \mid \kappa\tau) \qquad FLOPS = B^4 O^2 V^2$$

O(N⁸) scaling

Must be possible to do better than that

FLOP Count: Partial Integral transformation

Algorithm A: occupied indices first

$$(i\nu \mid \kappa\tau) = \sum_{\mu} c_{\mu i}(\mu\nu \mid \kappa\tau) \qquad (B^4O) \qquad \textbf{3125}$$

$$(i\nu \mid j\tau) = \sum_{\kappa} c_{\kappa j}(i\nu \mid \kappa\tau) \qquad (O^2B^3) \qquad \textbf{312}$$

$$(ia \mid j\tau) = \sum_{\nu} c_{\nu a}(i\nu \mid j\tau) \qquad (O^2VB^2) \qquad \textbf{281}$$

$$(ia \mid jb) = \sum_{\tau} c_{\tau b}(ia \mid j\tau) \qquad (O^2V^2B) \qquad \textbf{253}$$

Algorithm B: virtual indices first

$$(\mu a \mid \kappa\tau) = \sum_{\nu} c_{\nu a}(\mu\nu \mid \kappa\tau) \qquad (B^4V) \qquad \textbf{28215}$$

$$(\mu a \mid \nu b) = \sum_{\tau} c_{\tau b}(\mu a \mid \kappa\tau) \qquad (V^2B^3) \qquad \textbf{25312}$$

$$(ia \mid \nu b) = \sum_{\mu} c_{\mu i}(\mu a \mid \nu b) \qquad (OV^2B^2) \qquad \textbf{2531}$$

$$(ia \mid jb) = \sum_{\nu} c_{\nu j}(ia \mid \nu b) \qquad (O^2V^2B) \qquad \textbf{253}$$

Four O(N⁵) steps

ratio of FLOP counts:

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

0.07

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

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

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

$$E_{MP2} = -\frac{1}{4}\sum_{i,j,a,b}\frac{\left[(ia \mid jb) - (ib \mid ja)\right]^2}{\varepsilon_a + \varepsilon_b - \varepsilon_i - \varepsilon_j}$$

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

Full eightfold permutation symmetry used

have to be able to store N_{Bas}^3 integrals for each occupied MO. Hence need batches of occupied MOs

Transformation of 2nd index

Transformation of 3rd index

Transformation of 4th index

```
loop ibatch over batches of occupied MOs
   loop p=1..NBas
     loop q=1…p
       loop r=1…p
         loop s=1..r|q
            Calculate(pq|rs)
            loop i=1..Nocc (in ibatch)
            ITMP[p,q,r,i]+= Cocc[s,i]*(pq|rs) and non-redundant permutations of indices
         end i in ibatch
    end loops p,q,r,s
    loop p=1..NBas
        loop r=1..NBas
          loop i=1,…Nocc (in ibatch)
             loop j=1..i
              loop q=1..NBas
                 JTMP[p,j,r,i]+= Cocc[q,j]*ITMP[p,q,r,i]
              end loop q
    end loops j,i,r,p
    loop i=1..Nocc (in ibatch)
      loop j=1..i
        loop p over AO's
          loop b=1..NVirt
            loop r over AO's
              ATMP(p,b)+=C[r,b]*JTMP[p,j,r,i]
        end loops r,b,p
        loop a=1..Nvirt
          loop b=1..Nvirt
            loop p over AO's
              KIJ[a,b]+= C[p,a]*ATMP[p,b]
        end loops p,a,b
        Evaluate MP2 amplitudes and pair energy
    end loops i,j
       end loop i
end loop ibatch
```

BLAS optimized algorithm

```
loop p=1..NBas
  loop r= 1..p
    loop q=1..NBas
      loop s=1..NBas
          calculate K[p,r](q,s)= (pq|rs)
    end loop q,s
    Perform transformation K[p,r](i,j)=(Cocc^T*K[p,r]*Cocc)ij
    Write matrix K[p,r] to disk
end loops p,r
Resort Integrals K[p,r](i,j) to give K[i,j](p,r)   (i<=j)
Loop i= 1..Nocc
  loop j=1..i
    Read integrals K[i,j](p,r)
    Perform transformation K[i,j](a,b)=(Cvirt^T*K[i,j]*Cvirt)ab
    Calculate MP2 amplitudes T[i,j](a,b)
    Calculate MP2 pair energy e(i,j)
    Sum up MP2 correlation energy
end loops i,j
```

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

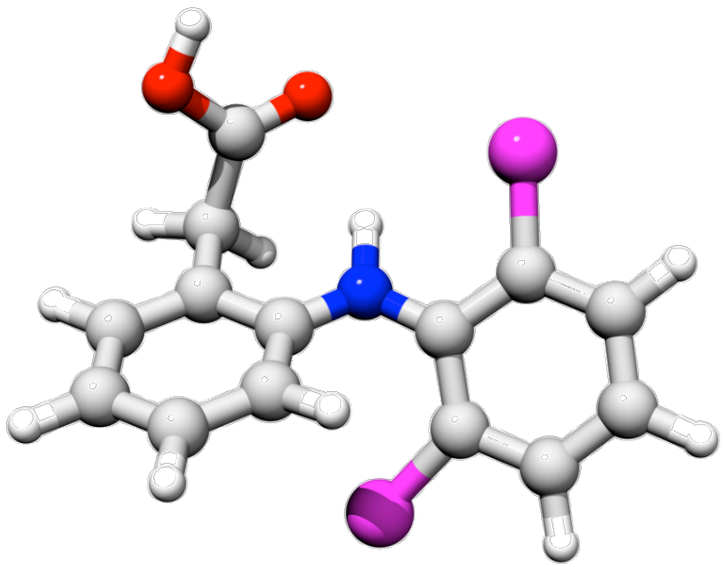
Two BLAS level 3 multiplications in the rate determining step

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

Performance Test



Diclophenac

def2-TZVP (667 basis functions)

4 GB main memory used

FLOP optimized algorithm

(25 batches necessary)

>100,000 sec

BLAS optimized algorithm

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
```

Chapter 2.4

Using Factorizations and Finding the ones with the best FLOP count

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:

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

O(N⁸) 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:

$$\sigma_{ab}^{ij} \leftarrow \sum_{kl} t_{ab}^{kl} \underbrace{\sum_{cd}\left\langle kl \,||\, cd \right\rangle t_{cd}^{ij}}_{X_{kl}^{ij}} \quad \textbf{or} \quad \sigma_{ab}^{ij} \leftarrow \sum_{cd} t_{cd}^{ij} \underbrace{\sum_{kl} t_{ab}^{kl} \left\langle kl \,||\, cd \right\rangle}_{Y_{cd}^{ab}}$$

Example: Factorization in Coupled Cluster

$$\sigma_{ab}^{ij} \leftarrow \sum_{kl} t_{ab}^{kl} \underbrace{\sum_{cd} \left\langle kl \,||\, cd \right\rangle t_{cd}^{ij}}_{X_{kl}^{ij}} \quad : \quad X_{kl}^{ij} = \sum_{cd} \left\langle kl \,||\, cd \right\rangle t_{cd}^{ij}$$

N_{occ}^4 Storage
$N_{occ}^4 N_{virt}^2$ FLOPS

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

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

O(N⁶) scaling

*2x**$N_{occ}^4 N_{virt}^2$** FLOPS
N_{occ}^4 *Storage*

$$\sigma_{ab}^{ij} \leftarrow \sum_{cd} t_{cd}^{ij} \underbrace{\sum_{kl} t_{ab}^{kl} \left\langle kl \,||\, cd \right\rangle}_{Y_{cd}^{ab}} \quad : \quad Y_{cd}^{ab} = \sum_{kl} t_{ab}^{kl} \left\langle kl \,||\, cd \right\rangle$$

N_{virt}^4 Storage
$N_{occ}^2 N_{virt}^4$ FLOPS

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

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

O(N⁶) scaling

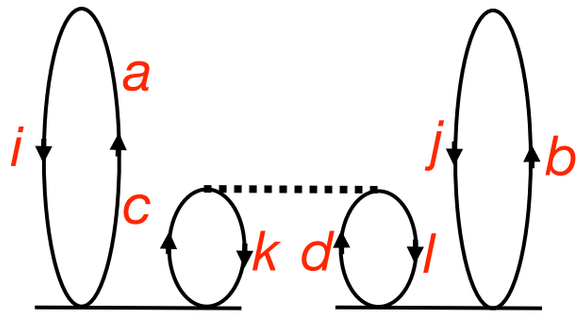
*2x**$N_{occ}^2 N_{virt}^4$** FLOPS
N_{virt}^4 *Storage*

Algorithm 1 ***N_{occ}^2*** ***<— — MUCH better and MUCH less Storage!***
— — — — — = — — — FLOPS <<1
Algorithm 2 ***N_{virt}^2***

Let us return to our initial question!

We had:

i,j,k,l=occupied
a,b,c,d=virtual



$$\sigma_{ab}^{ij} \leftarrow P(ij)P(ab)\sum_{kl}\sum_{cd}\langle kl||cd\rangle t_{ac}^{ik}t_{db}^{lj}$$

$O(N^4)$ $O(N^4)$ $\rightarrow O(N^8)$?

STEP 1: What is your data and how do you store what?

How many occupied (n_o) and virtual (n_v) orbitals do I have?

- Say n_o=50, n_v=400, then $(n_o*n_v)^{2\sim3\ GB}$ storage, $(n_v)^4$=190 GB
- Probably need to store that on disk and retrieve in portions

STEP 2: Rewrite the equations in matrix form

Integrals: $\langle kl||cd\rangle = (kc|ld) - (kd|lc) \equiv K^{kl}(c,d) - K^{kl}(d,c)$

Amplitudes: $t_{ab}^{ij} = T^{ij}(a,b)$

- Series of matrices orderd by internal label pairs
- **Always look for matrices and vectors!**

$$\widetilde{K}^{kl}(c,d) \equiv K^{kl}(c,d) - K^{kl}(d,c)$$

$$\boldsymbol{\sigma}^{ij} \leftarrow P(ij)P(ab)\sum_{kl}\boldsymbol{T}^{ik}\widetilde{\boldsymbol{K}}^{kl}\boldsymbol{T}^{lj}$$

How it is REALLY *NOT* done

```
Get array SIGMA(i,j,a,b)
Get array T(i,j,a,b)
Get array KS(i,j,a,b)

 Loop over i>=j                O(N2)
   Loop over a,b                  O(N2)
     Loop over k,l                  O(N2)
       Loop over c,d                  O(N2)

          SIGMA(i,ja,b,)+= T(i,k,a,c)
                          *T(j,l,d,b)
                          *KS(k,l,c,d)


       End c,d
     End k,l
   End a,b
End i,j
```

Overall $O(N^8)$
With heavy
Memory
demands
and no
BLAS

How it is *ALSO NOT* done

```
Loop over pairs i>=j                                    O(N2)
  Get matrix SIGMA(i,j)
  Loop over k,l                                         O(N2)
    Get matrix T(i,k)
    Get matrix T(j,l)
    Get matrix KS(k,l) (KS=K-squiggle)
    Form intermediate X= KS(k,l)*T(j,l)                 O(N3)
    Add to SIGMA(i,j)+= X(k,j)*T(j,l)                   O(N3)
  End k,l
  Store matrix SIGMA(i,j)
End i,j
```

Overall $O(N^7)$
With heavy
I/O

How it IS done

```
Loop over pairs kl                    O(N2)
  Get matrix KS(k,l)  (KS=K-squiggle)
  Loop over j                         O(N)
    Get matrix T(j,l)
    Form intermediate X(k,j)= KS(k,l)*T(j,l)
    Store X(k,j)                                  O(N3)
  End j
End kl
Loop over pairs i>=j                  O(N2)
  Get matrix SIGMA(i,j)
  Loop over k                         O(N)
    Get matrix T(i,k)
    Get matrix X(k,j)
    Add to SIGMA(i,j)+= X(k,j)*T(j,l)          O(N3)
  End k
  Store matrix SIGMA(i,j)
End i,j
```

BLAS matrix x matrix

BLAS matrix x matrix

(Let's drop P(ij)P(ab) for the moment to keep things simple)

Overall $O(N^6)$

Overall $O(N^6)$

Chapter 2.5

Precompute what you can afford to avoid redundant re-computation

Precomputed quantities

Example: shell pair data

```
Loop ish>= jsh
    Calculate KIJ=di*dj*exp(-ai*aj/(ai+aj)*R_AB^2)
    Calculate P = 1/(ai+aj)*(ai*R_A+aj*R_B)
    Loop ksh, lsh(ish==ksh?jsh:ksh)
        Calculate KKL=dk*dl*exp(-ak*al/(ak+al)*R_CD^2)
        Calculate Q = 1/(ak+al)*(ak*R_C+al*R_D)
        Calculate (IJ|KL) {P,Q,KAB,KCD,…}
        …
```

Highly redundant since independent of ish,jsh, A or B!

Better: Precompute shell pair data AND screen for negligible shell pairs

```
Loop ish>= jsh
   Calculate KIJ=di*dj*exp(-ai*aj/(ai+aj)*R_AB^2)
   if |KIJ|<TCut then reject shell pair
   Calculate P = 1/(ai+aj)*(ai*R_A+aj*R_B)
   Store KIJ, P in memory or on disk
```

Move Work out of the Inner Loops: Split-J

Choosing intermediates wisely such that redundant work is move out of the inner loops helps performance

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

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

When we calculate the integrals one by one, we repeatedly re-calculate this quantity N^2 times although it is independent of μ,ν.Likewise: Transformation to spherical harmonics

$$= \sum_{\kappa\tau} P_{\kappa\tau} \underbrace{\sum_{tuv} E^{\mu\nu}_{tuv}}_{independent\ of\ \kappa\tau} \sum_{t'u'v'} (-1)^{t'+u'+v'} E^{\kappa\tau}_{t'u'v'} R_{t+t',u+u',v+v'}$$

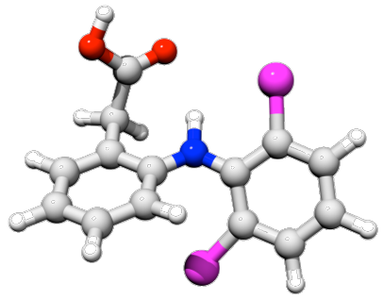
$$= \sum_{tuv} E^{\mu\nu}_{tuv} \sum_{t'u'v'} R_{t+t',u+u',v+v'} \underbrace{\sum_{\kappa\tau} (-1)^{t'+u'+v'} P_{\kappa\tau} E^{\kappa\tau}_{t'u'v'}}_{\equiv P_{t'u'v'}\ independent\ of\ \mu\nu,tuv}$$

$$= \sum_{tuv} E^{\mu\nu}_{tuv} \sum_{t'u'v'} P_{t'u'v'} R_{t+t',u+u',v+v'}$$

Hermite to S_{lm} Transformation | **Hermite basis density** | **Hermite basis repulsion**

FN *J. Comp. Chem.* **2003**, *24*, 1740-1747; FN *J. Comp. Chem.*, **2022**, 1-16

Performance example



def2-TZVP=667 BFs

	Coulomb term (sec) (20-builds)
Traditional treatment	**5796 sec**
Split-J algorithm =Ahmadi-Almlöf =Head-Gordon J-engine	**2834 sec**

Identical numerical result, same scaling, but significant speedup realized through thoughtful structuring of the entire computational process

Chapter 2.6

Be careful with Input/Output

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

6641 sec

```
Loop i=1…Nocc
  loop a=1..Nvir
   Write NULL matrix K^ia into buffer IABC
  end loop a
  loop a=1..Nvir
    Read matrix K^ia(b,c) = (ib|ac) from IABC
    loop b=1..Nvir
      Read matrix K^ib(c,d) = (ic|bd) from IABC
      loop c=1..Nvir
        K^ib(a,c)=+K^ib(a,c)+K^ia(b,c);
      end loop c
      Store matrix K^ib in IABC
    end loop b
  end loop a
end loop i
```

31 sec

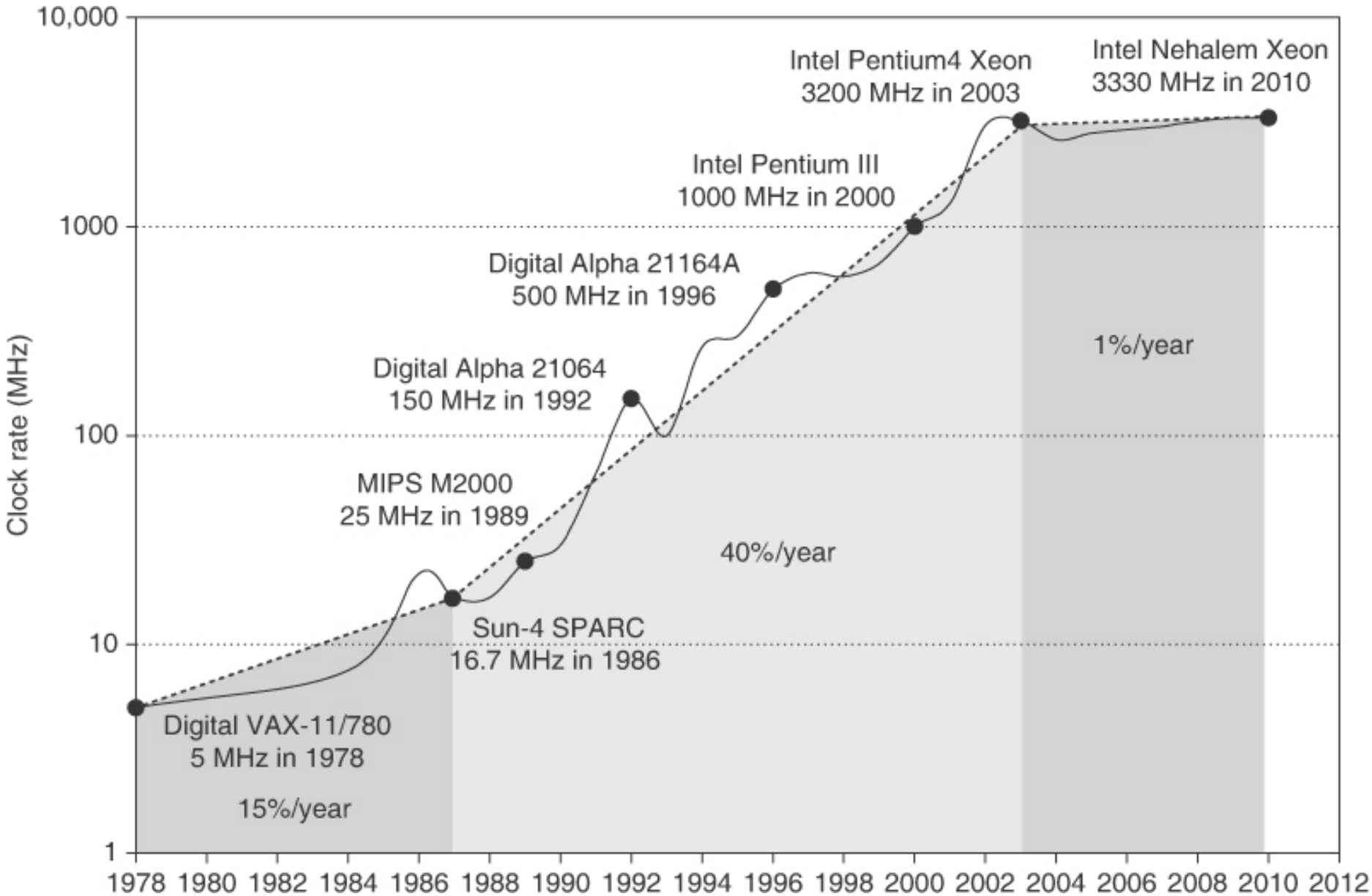
```
Loop i=1..Nocc
  Initialize buffer K^ib for all b
  loop a=1..Nvir
    read matrix Kia(b,c) from IABC
    loop b=1..Nvir
      loop c=1..Nvir
        K^ib(a,c)+=Kia(b,c);
      end loop c
    end loop b
   end loop a
   Write entire buffer K^ib into IABC
end loop i
```

SAME operation count!
Factor 200 performance difference!!

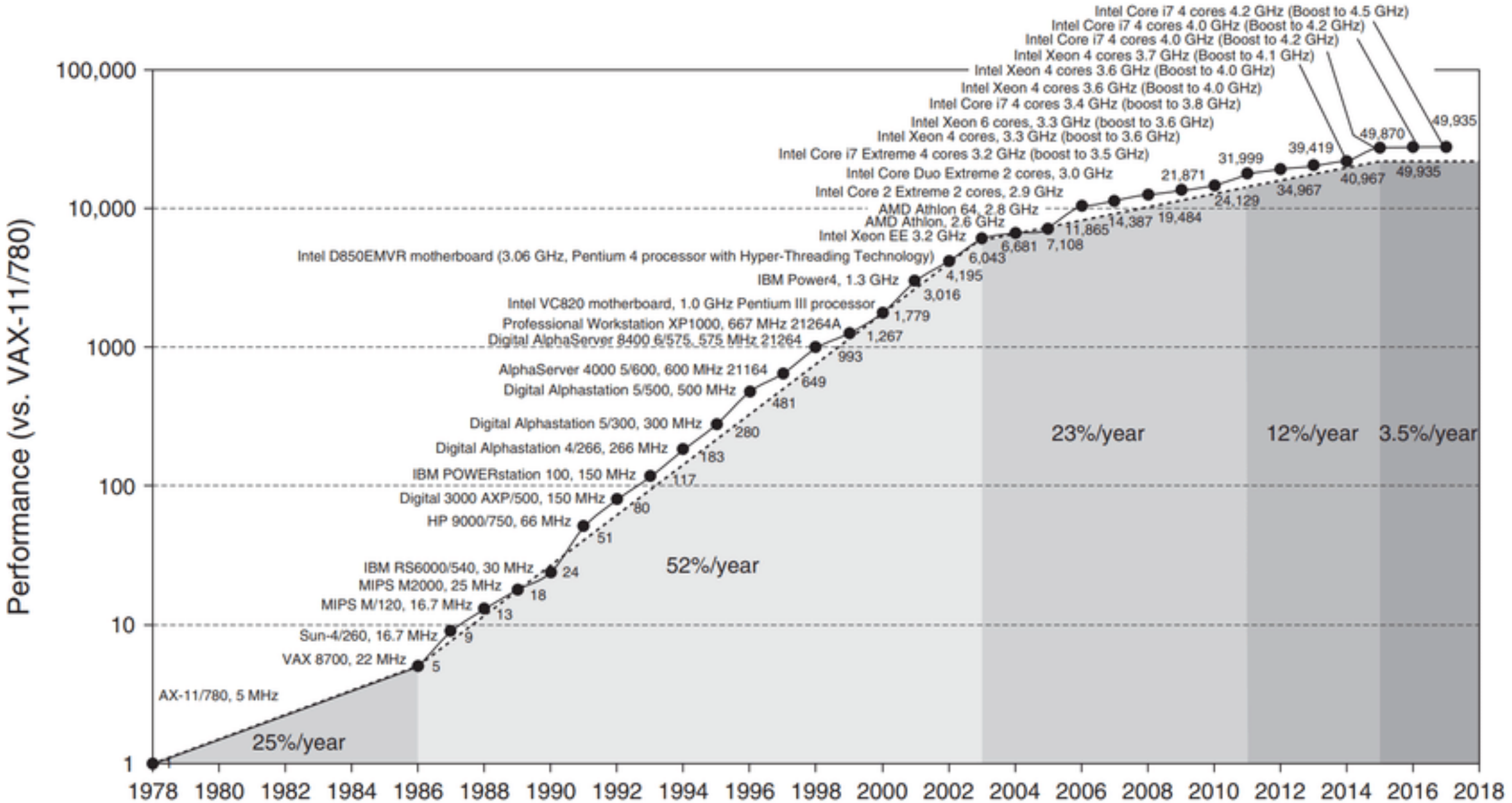
Chapter 2.7

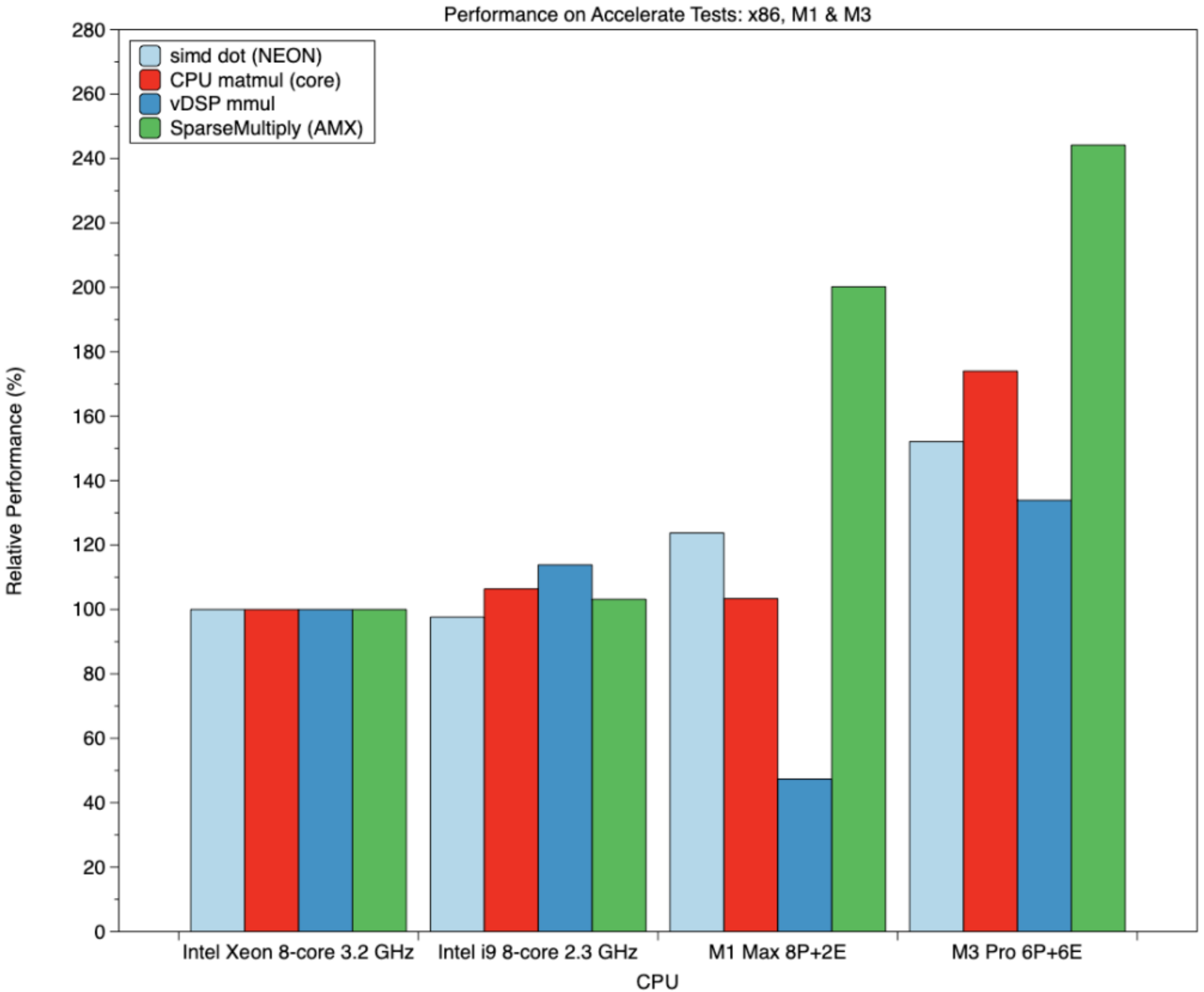
Parallelization in a nutshell

Single CPU Clockspeed / Single Thread Performance



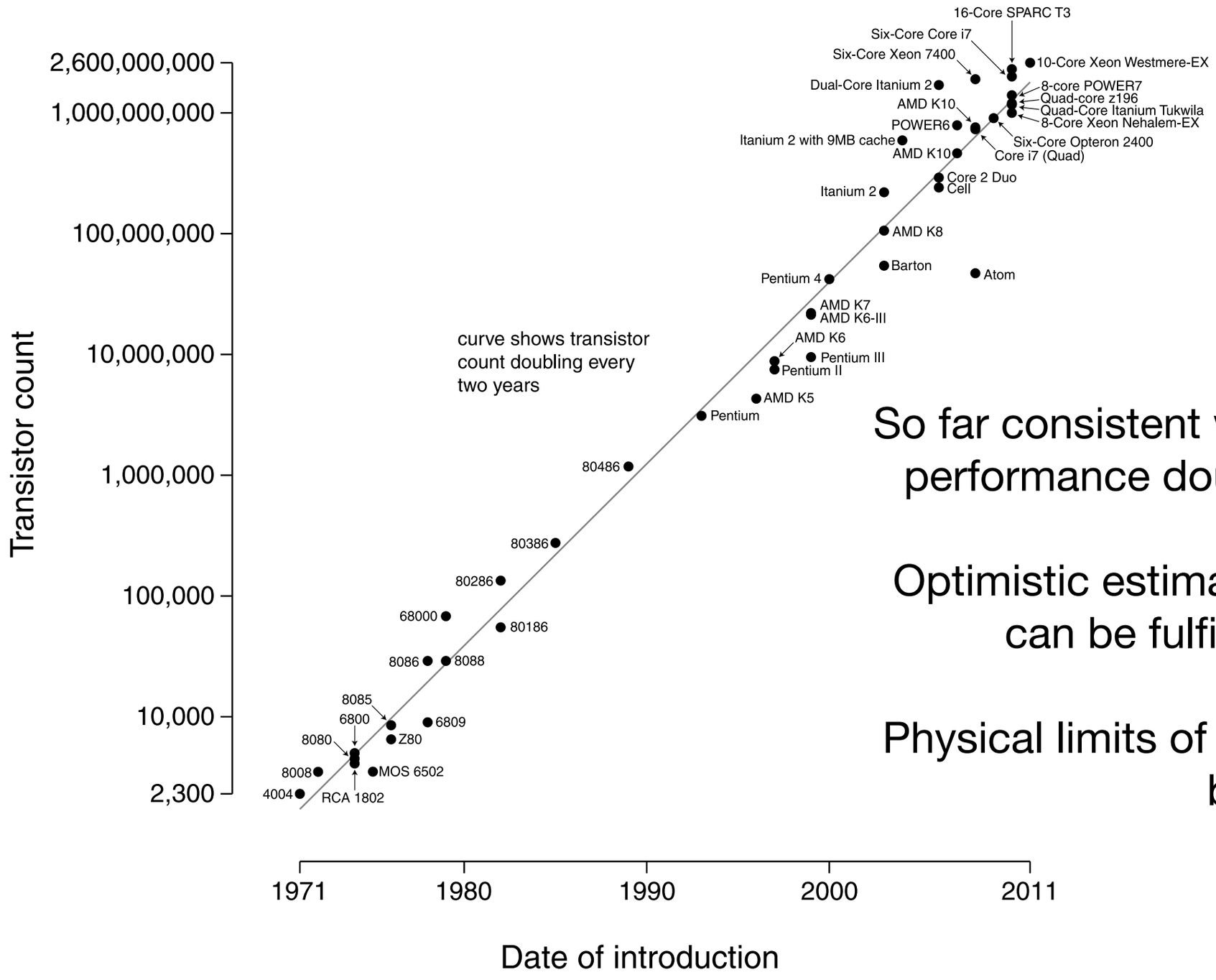
Growth in clock rate of microprocessors. Between 1978 and 1986, the clock rate improved less than 15% per year while performance improved by 25% per year. During the "renaissance period" of 52% performance improvement per year between 1986 and 2003, clock rates shot up almost 40% per year. Since then, 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.


Performance (vs. VAX-11/780)
100,000
10,000
1000
100
10
1
1978
1980
1982
1984
1986
1988
1990
1992
1994
1996
1998
2000
2002
2004
2006
2008
2010
2012
2014
2016
2018
Intel Core i7 4 cores 4.2 GHz (Boost to 4.5 GHz)
Intel Core i7 4 cores 4.0 GHz (Boost to 4.2 GHz)
Intel Core i7 4 cores 4.0 GHz (Boost to 4.2 GHz)
Intel Xeon 4 cores 3.7 GHz (Boost to 4.1 GHz)
Intel Xeon 4 cores 3.6 GHz (Boost to 4.0 GHz)
Intel Xeon 4 cores 3.6 GHz (Boost to 4.0 GHz)
Intel Core i7 4 cores 3.4 GHz (boost to 3.8 GHz)
Intel Xeon 6 cores, 3.3 GHz (boost to 3.6 GHz)
Intel Xeon 4 cores, 3.3 GHz (boost to 3.6 GHz)
Intel Core i7 Extreme 4 cores 3.2 GHz (boost to 3.5 GHz)
Intel Core Duo Extreme 2 cores, 3.0 GHz
Intel Core 2 Extreme 2 cores, 2.9 GHz
AMD Athlon 64, 2.8 GHz
AMD Athlon, 2.6 GHz
Intel Xeon EE 3.2 GHz
Intel D850EMVR motherboard (3.06 GHz, Pentium 4 processor with Hyper-Threading Technology)
IBM Power4, 1.3 GHz
Intel VC820 motherboard, 1.0 GHz Pentium III processor
Professional Workstation XP1000, 667 MHz 21264A
Digital AlphaServer 8400 6/575, 575 MHz 21264
AlphaServer 4000 5/600, 600 MHz 21164
Digital Alphastation 5/500, 500 MHz
Digital Alphastation 5/300, 300 MHz
Digital Alphastation 4/266, 266 MHz
IBM POWERstation 100, 150 MHz
Digital 3000 AXP/500, 150 MHz
HP 9000/750, 66 MHz
IBM RS6000/540, 30 MHz
MIPS M2000, 25 MHz
MIPS M/120, 16.7 MHz
Sun-4/260, 16.7 MHz
VAX 8700, 22 MHz
AX-11/780, 5 MHz
49,935
49,870
39,419
31,999
21,871
24,129
34,967
40,967
49,935
19,484
14,387
11,865
7,108
6,681
6,043
4,195
3,016
1,779
1,267
993
649
481
280
183
117
80
51
24
18
13
9
5
1
25%/year
52%/year
23%/year
12%/year
3.5%/year


Performance on Accelerate Tests: x86, M1 & M3
simd dot (NEON)
CPU matmul (core)
vDSP mmul
SparseMultiply (AMX)
Relative Performance (%)
280
260
240
220
200
180
160
140
120
100
80
60
40
20
0
Intel Xeon 8-core 3.2 GHz
Intel i9 8-core 2.3 GHz
M1 Max 8P+2E
M3 Pro 6P+6E
CPU

Performance: Moore's Law

Microprocessor Transistor Counts 1971-2011 & Moore's Law



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:

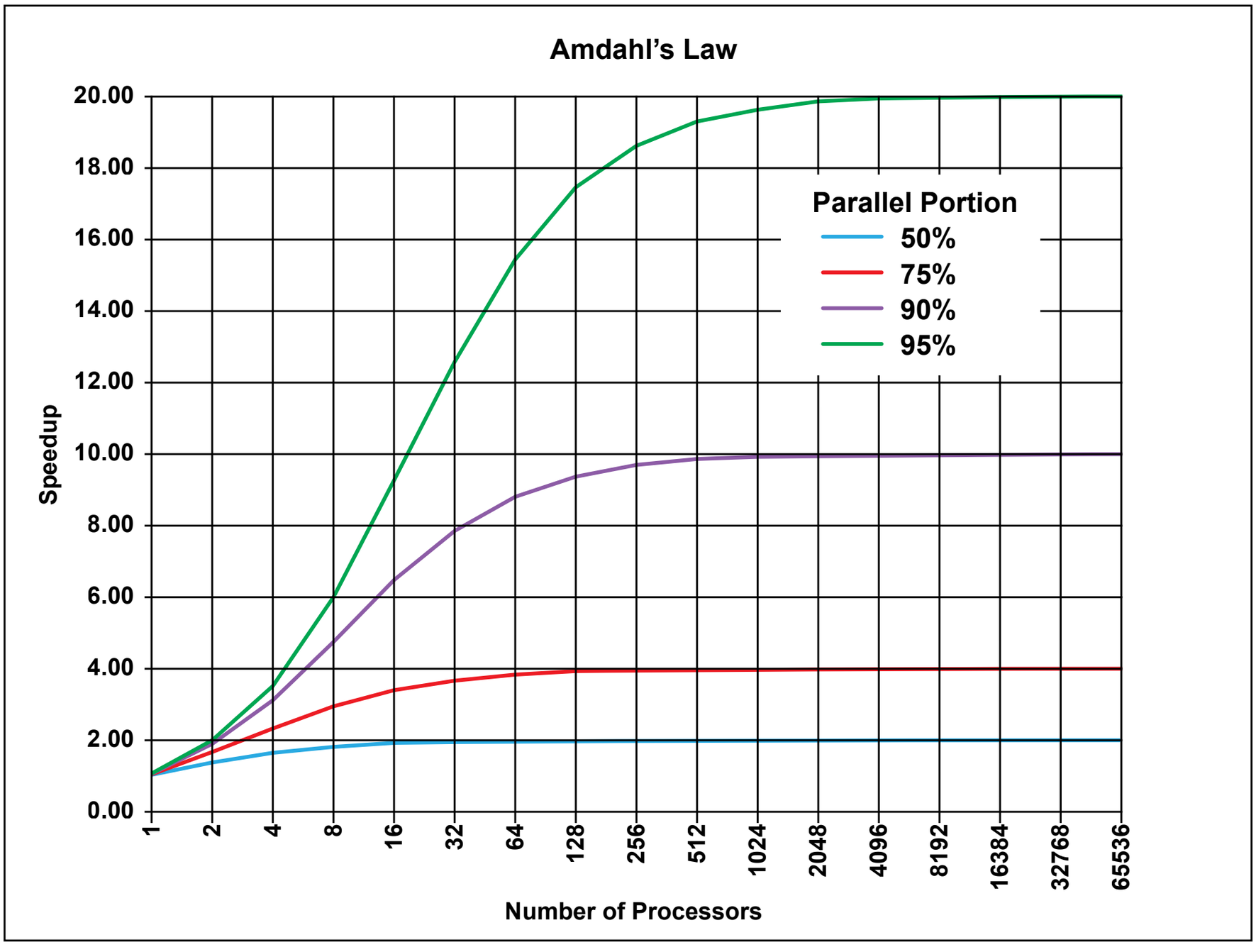
Requires explicit parallelization by the programmer!

"*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



Speedup:

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

P: Parallel portion of code
N: Number of Processors

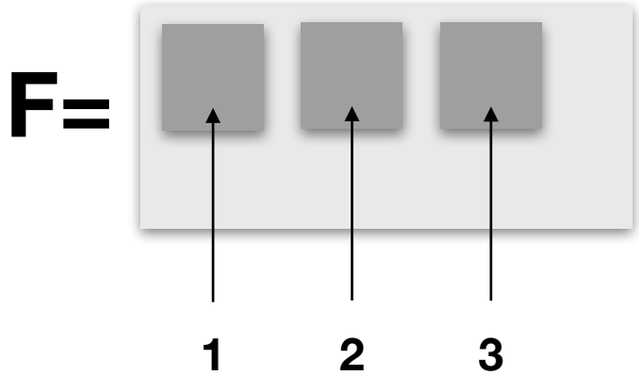
Parallelization in a Nutshell

Principle idea: let a number of processors, say *n*, work on parts of the computational problem in parallel and combine sub results into the final result.

Ideal Scenario: The problem breaks down perfectly and the time required to solve the problem is 1/*n*.

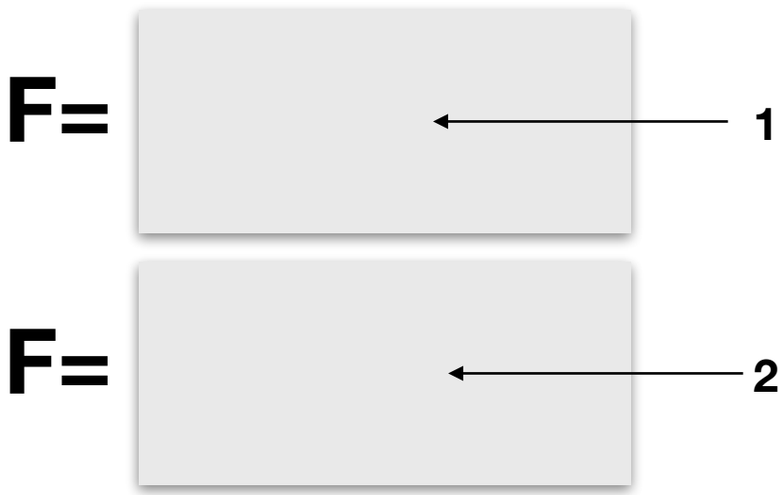
Shared Memory Models:

- Open MP, POSIX threads
- Efficient use of resources, no memory replication
- Difficult to debug large programs
- Can only be used on one machine with common memory



Message Passing Models

- Communication via messages between processes
- Choice between replicated and distributed memory
- Distributed memory difficult to implement efficiently
- Can be used between machines



Hybrid Models:

- Threads + MPI
- Combines shared memory on one machine with message passing between machines
- Adaptation into official standards is slow

Parallelization

Parallelization is of vital importance in modern high-performance computing, yet a LOT can go wrong here! We can only scratch the surface of this complex subject.

A few rules:

1. Each process should have roughly the same amount of work to do (**Load Balancing**).

2. Do the parallelization as far ‚**outside**‘ as possible (e.g. distribute the *outer*most loop).

3. Excessive **communication** (e.g. sending large chunks of data) between processes should be avoided as much as possible.

4. **Synchronization** should not happen inside time critical loops and there should be as little of it as possible.

5. **I/O in parallel applications** is difficult if several processes access the same file.

Load Balancing Example

Integral calculation: The time required to calculate a given integral batch is a complicated function of angular momenta, contraction depth, orbital exponents and pre-screening efficiency

- Load balancing difficult to guarantee
- Random distribution of batches among processors.
- Uneveness will average out in the limit of many batches

Local to process

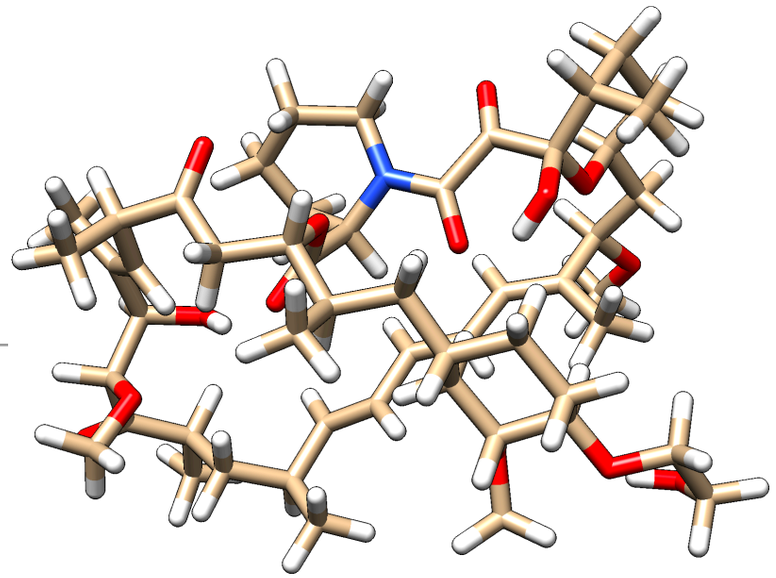
```
Target F=0

loop i=1..N,i+=1          loop i=1..N,i+=NProcs

      Calculate something

      Sum into Local F on processor

 End loop
MPI-AllSum(F)
MPI-AllSum(F)
```

Skip over i's not „assigned" to this process

Make sure all processes are finished before doing anything else

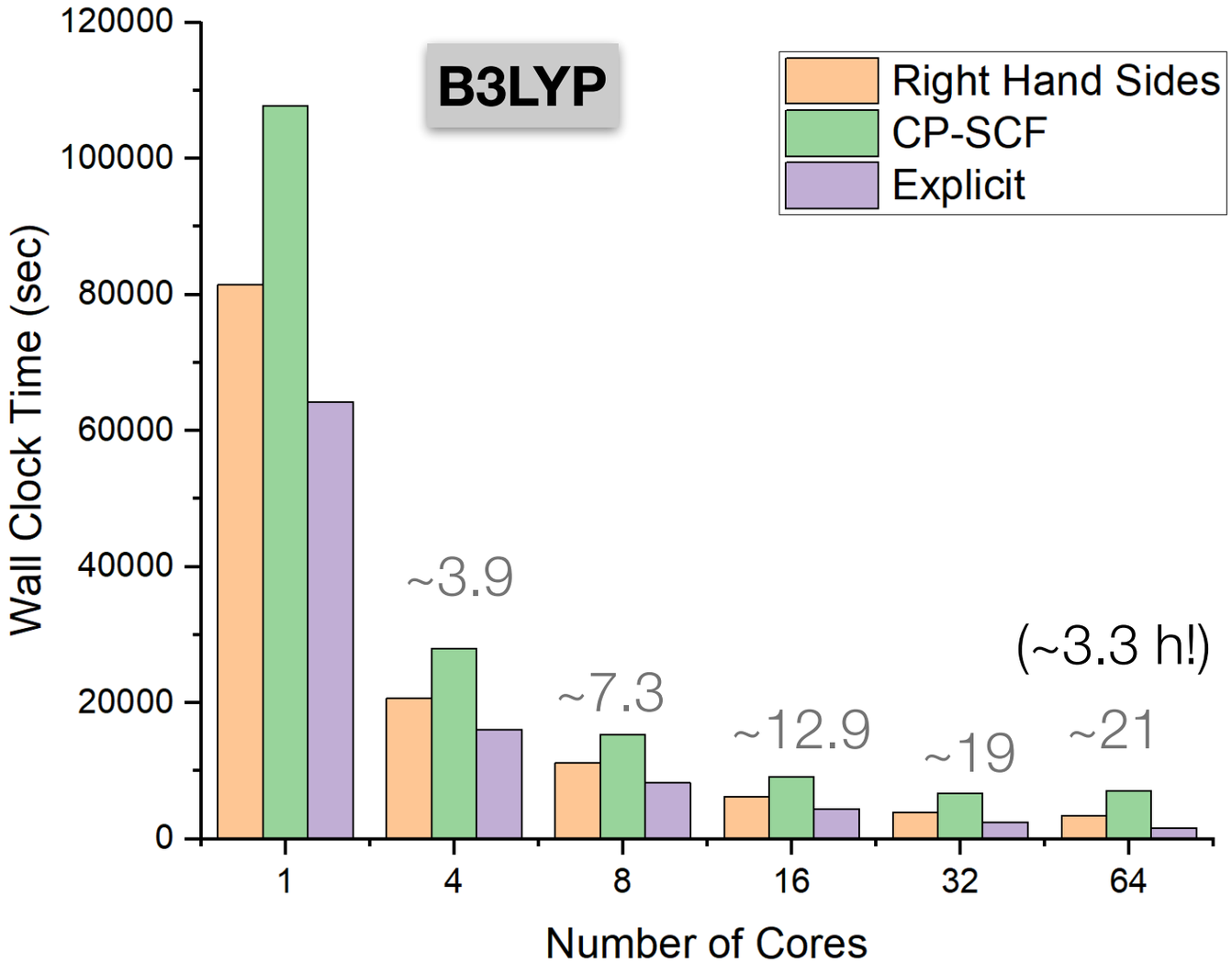
Sum up results obtained at individual processes

Parallelization and Ahmdal's Law



Everolimus
151 atoms,
def2-tzvp (2606 basis functions)

Memory: 8 GB /core

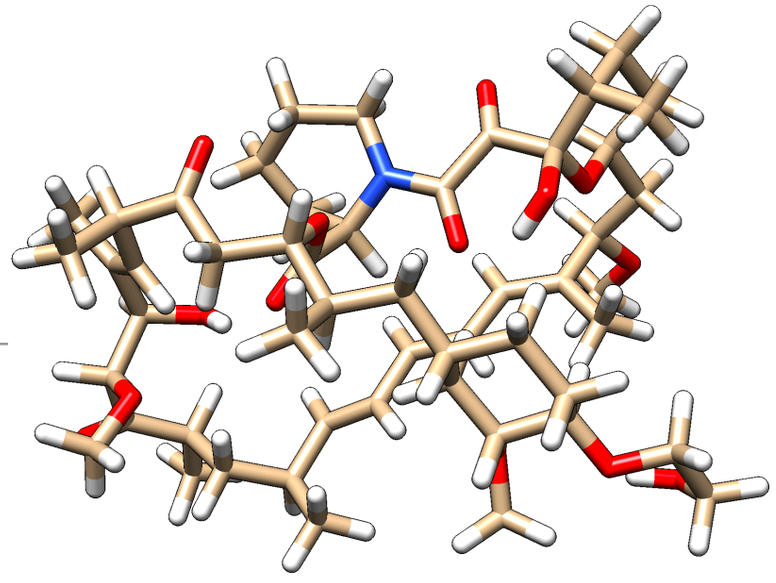


Group Parallelization

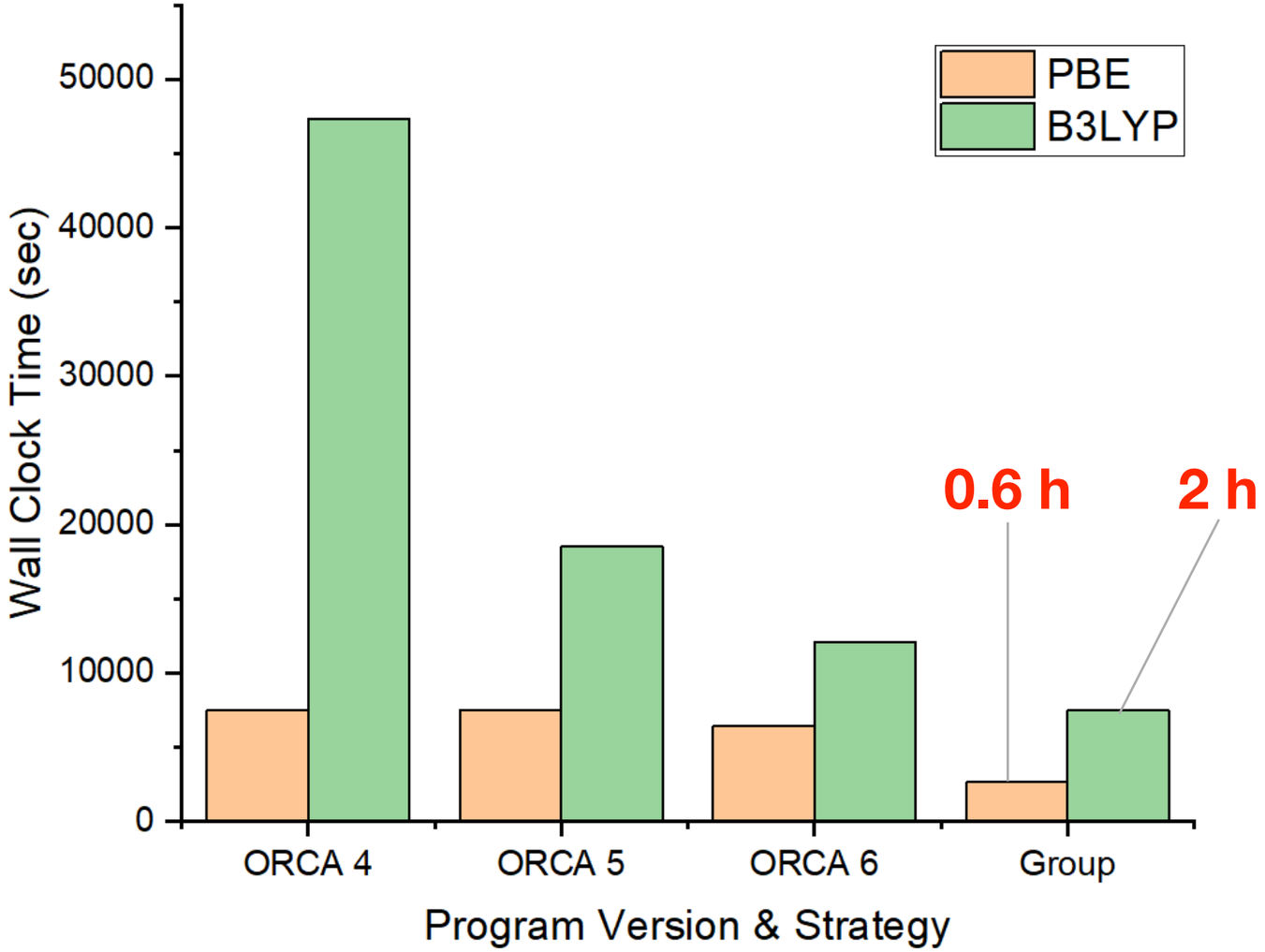
Idea:

Divide the processes into groups that scale well (e.g. up to 8) and then parallelize independently over the groups

! PAL(8x8)



Everolimus
151 atoms,
def2-tzvp (2606 basis functions)



Up to a factor of four faster than „bulk“ parallelization

Chapter 3

Some Useful Programming Techniques for Writing „good“ Software

Prelude: Who are you writing code 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

Good Programming Habits

- ✓ Show some **respect** for the work that went into the package:
  - Communicate with the team and dont' go off on lonely tangents
  - Be respectful of the code organization and try to fit in
  - Do not rewrite code of others without prior conversation
- ✓ Write lots of **comments** (in english!) including references to papers.
- ✓ Use the **existing infrastructure** of the package
- ✓ **Debug** carefully before checking into the main branch. Delete debug code
- ✓ Provide plenty of **test jobs** with reference results.
- ✓ **Profile** and **optimize** the code carefully
- ✓ Be **feature complete** (e.g. not just closed shell)
- ✓ Program as **simple as possible** and only as fancy as necessary. Do not "show off"
- ✓ Do not write 50000 line functions - break it down, be **modular**, make it **reusable**
- ✓ Use logical, recognizable **file names**

... bad habits are the opposite of everything on the list

Chapter 3.1

Using Recursion

Recursion

Recursion can be a very effective way to arrive at compact, elegant code.

1. Define the starting point of the recursion
2. Define the recursive conditions strategy
3. Define the termination conditions

Easy example: the factorial number: $n! = \prod_{i=1}^{n} i$ $\qquad 0! = 1$

Straightforward linear programming:

```
int fac_n = 1;
for (int i=1;i<=n;i++) fac_n = fac_n*i;
```

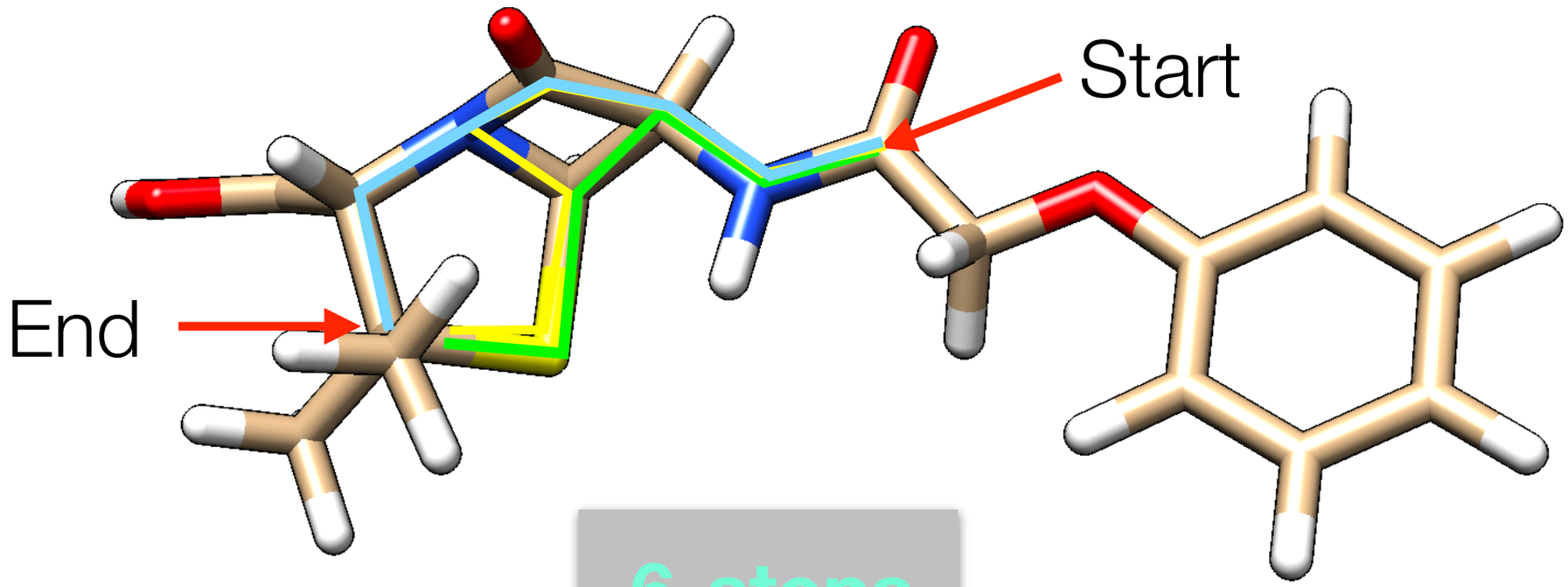
Recursive programming:

must be passed by value!

```
int factorial(int n){
  if (n>0) return n*factorial(n-1);
  else return 1;
};
```

termination condition

Recursion: Finding the shortest path from A to B



6-steps

5-steps

6-steps

```
// ------------------------------------------------------------------------------------------
// FN 08/18
// Find a pathway between A and B (if it exists)
//
// ON INPUT  A             - the first atom one is looking for
//           B             - the target atom for the pathway
//           XAB           - the list of connectivities XAB(A,k) - k'th atom connected to A; k<NAB(A)
//           NAB           - the number of connectivities for each atom NAB(A)=number of bonds at A
//           ActualLength  - the recursion depth
//           MaxLength     - the maximal length of the path that is allowed
//           abortAtLength - abort the search if a path of this length or shorter has been found
// ON OUTPUT Length        - the length of the pathway found
// ------------------------------------------------------------------------------------------
void GEO_FindPathway(int A, int B, TIMatrix &XAB, TIVector &NAB, int ActualLength, int &Length, int &MaxLength, int abortAtLength)
{
  // Abort if a path with abortAtLength is found
  if (Length <= abortAtLength && Length > 0) return;

  // Abort, if no path is found or we have exceeded the allowed maximal length
  if (ActualLength> MaxLength) return;

  // The number of connections made by atom A
  int NA= NAB(A);
  for (int k=0;k<NA;k++){
     // C is the actual k'th atom connected to A
     int C= XAB(A,k);
     // If we have found our target atom, we stop the recursion
     if (C==B){
       // first pathway found
       if (Length<0){
         Length= ActualLength;
       }
       // if we had one before, check whether this one
       // is shorter. We take the shortest
       else{
         if (ActualLength<Length) Length=ActualLength;
       }
       // There can be no path shorter than one bond.
       if (Length==1) return;
       // In any other case, we keep recursing
       GEO_FindPathway(C,B,XAB,NAB,ActualLength+1,Length,MaxLength, abortAtLength);
       // .... but stop here, since no other atom in that list
       // can match B
       return;
     }
     // and otherwise continue the recursion with the actual atom along the pathway
     else{
       GEO_FindPathway(C,B,XAB,NAB,ActualLength+1,Length,MaxLength, abortAtLength);
     }
  }
}
```

Start condition

Check whether actual walk is shorter

Termination

Recursion. New origin of search is C

Recursion. New origin of search is C

Recursion: Setting up ORMAS CI Spaces

ORMAS (Occupation Restricted Multiple Active Spaces)

GAS (Generalized Active Space)

- ✓ The orbital space is divides into N subspaces
- ✓ Subspace K can have Kmin ... Kmax electrons
- ✓ Subspace K has NORBK orbitals
- ✓ Inside each subspace a CAS(NELK,NORBK) is formed
- ✓ All combinations of subspaces that give the correct NEL are wanted

```
/* ------------------------------------------------------------------------
 * FN 08/2024
 *
 *  Recursive function to figure out the combination of subspace lists that lead
 *  to the correct number of electrons in the active space
 *
 *  ON INPUT  ORMAS         - the information about the subspaces
 *            NELTARGET     - target number of electrons
 *            ACTNEL        - actual number of electron during recursion
 *            ACTLEVEL      - actual level of recursion (e.g. LEVEL=subspace level)
 *            ACTCOMBO      - actual combination
 *            JustCount     - only count combinations or also store them
 *
 *  ON OUTPUT NCOMBINATIONS - number of combinations that lead to the correct NEL
 *            COMBO         - if we store, the combinations themselves
 *  ------------------------------------------------------------------------
 */
void ORMAS_FindSubSpaceCombos(TIVector &ORMAS,
                              int       NELTARGET,
                              int       ACTNEL,
                              int       ACTLEVEL,
                              TIVector &ACTCOMBO,
                              int      &NCOMBOS,
                              TIMatrix &COMBO){
  int64 NLEVELS= ORMAS(0);
  int64 norb   = ORMAS(1+3*ACTLEVEL+0);
  int64 nmin   = ORMAS(1+3*ACTLEVEL+1);
  int64 nmax   = ORMAS(1+3*ACTLEVEL+2);
  TIVector TCOMBO;
  TCOMBO.CopyVec(ACTCOMBO);
  // Loop over all electron numbers of this level
  int ACTNEL0= ACTNEL;
  for (int64 n=nmin; n<=nmax; n++){
    // Remember where we are at this level
    TCOMBO(ACTLEVEL)=n;
    // current number of electrons
    ACTNEL= ACTNEL0 + n;
    // if that is already too large we can skip
    if (ACTNEL>NELTARGET) break;
    // We arrived at the bottom level: see what we have
    if (ACTLEVEL == NLEVELS -1){
      // That is the correct number of electrons
      if (ACTNEL==NELTARGET){
        for (int64 i=0;i<NLEVELS;i++){
          COMBO(NCOMBOS,i)= TCOMBO(i);
        };
        NCOMBOS++;
       } else{
        // nothing to do: number of electrons is incorrect
      }
    }
    // ------------------------------------------------
    // NOT bottom level: recurse to the next level
    // ------------------------------------------------
    else{
      ORMAS_FindSubSpaceCombos(ORMAS,
                               NELTARGET,
                               ACTNEL,
                               ACTLEVEL+1,
                               TCOMBO,
                               NCOMBOS,
                               COMBO,
                               JustCount);
    };
  };
};
```

Reached last subspace

Correct number of electrons found

Recursion to next next subspace

```
ORMAS(14: 6  10 12 / 2 0 4 / 50 0 2)
```

no of electrons (total)

Subspace 1 (domos) 6 MOs, 0-2 holes

Subspace 2 (active) any occupation

Subspace 3 (virtuals) 50 MOs 0-2 electrons

Subspace CAS lists

```
Subspace   1:  10 electrons in   6 orbitals => 21 configurations
Subspace   1:  11 electrons in   6 orbitals => 6 configurations
Subspace   1:  12 electrons in   6 orbitals => 1 configurations
Subspace   2:   0 electrons in   2 orbitals => 1 configurations
Subspace   2:   1 electrons in   2 orbitals => 2 configurations
Subspace   2:   2 electrons in   2 orbitals => 3 configurations
Subspace   2:   3 electrons in   2 orbitals => 2 configurations
Subspace   2:   4 electrons in   2 orbitals => 1 configurations
Subspace   3:   0 electrons in  50 orbitals => 1 configurations
Subspace   3:   1 electrons in  50 orbitals => 50 configurations
Subspace   3:   2 electrons in  50 orbitals => 1275 configurations
```

Subspace combinations

```
Number of valid subspace combinations = 9

COMBO   1:  10   2   2  =>    80325 CFGs
COMBO   2:  10   3   1  =>     2100 CFGs
COMBO   3:  10   4   0  =>       21 CFGs
COMBO   4:  11   1   2  =>    15300 CFGs
COMBO   5:  11   2   1  =>      900 CFGs
COMBO   6:  11   3   0  =>       12 CFGs
COMBO   7:  12   0   2  =>     1275 CFGs
COMBO   8:  12   1   1  =>      100 CFGs
COMBO   9:  12   2   0  =>        3 CFGs
```

Chapter 3.1

Object Oriented Programming

Object Oriented Programming

In object oriented programming you celebrate the unity of code and data by creating **classes**, that contain

- Data that are private to the object
- Functions that work on these data

➡ MUCH safer than having global data that is passed around the program

➡ MUCH easier to build up and administrate complicated data structures

```
class TGaussianShell{
  private:
  int nprim;        // number of primitives
  int l;            // angular momentum
  int ofs;          // position in the basis function list
  double *a, *d;    // exponents, contraction coefficients
  public:
  // Constructor and Destructor
  TGaussianShell(){
     nprim =0; l=0; ofs=0; a=nullptr; d=nullptr;
  };
  ~TGaussianShell(){
     if (a!=nullptr){ delete[] a; a=nullptr;
     if (d!=nullptr){ delete[] d; d=nullptr;
    };
  // Setters and Getters
  int GetNPrim(){ return nprim; };
  double *GetA(){ return a; };
  ...
  // Productive functions
  void Copy(int xnprim, int xl, int xofs, double *xa, double *xd){
          nprim= xnprim; .... };
  void Copy(TGaussianShell &SH){ Copy(SH.GetNPrim(),SH.GetL(),SH.GetOfs(),SH.GetA(), SH.GetD());};
  void Normalize();
  void Store(FILE *f);
  void Read(FILE *f);
  void Print();
};
```

data

Constructor and Destructor

Access to data

Productive functions

```
class TGaussianAtom{
  private:
  int lmaxA;       // Highest L for this atom
  int *NShells;    // Number of shells in each angular momentum
  TGaussianShell **Shells;
 public:
  TGaussianAtom(){ ... };   // initialize
  ~TGaussianAtom(){ ....};  // delete data
  // Getters and Setters
  TGaussianAtom *GetShell(int l, int ish){ return &(Shells[l][ish]); };
  // Productive functions
  void GetMemory(int xlmaxA, int *xNShells;){
    lmaxA= xlmaxA;   NShells= new int[lmaxA+1];
    for (int l=0;l<=lmaxA;i++) NShells[l]= xNShells[l];
    Shells= new TGaussianShell *[lmaxA+1];
    for (int l=0; l<=lmaxA; l++) Shells[l]=new TGaussianShell[NShells[l];
  };
  void Normalize(){
    for (int l=0;l<=lmaxA; l++)
      for (int ish=0; ish<NShells[l];ish++) Shells[l][ish].Normalize();
  };
  void Copy(TGaussianAtom &GA){
    GetMemory( GA.GetLmax(), GA.GetNShells() );
    for (int l=0;l<=lmaxA; l++)
      for (int ish=0; ish<NShells[l];ish++)
        Shells[l][ish].Copy(*GA.GetShell(l, ish));
  };
  void Store(FILE *f);
  void Read(FILE *f);
  void Print();
};
```

```
class TSegmentedBasisSet{
  private:
  int NAtoms;              // Number of atoms
  TGaussianAtom *Gauss; // the actual Gaussians for each atom
  public:
  TSegmentedBasisSet(){ ... };   // initialize
  ~TSegmentedBasisSet(){ ....};  // delete data
  // Productive functions
  void GetMemory(int xNAtoms){
    NAtoms= xNAtoms;
    Gauss= new TGaussianAtom[NAtoms];
  };
  void SetAtomBasis(int A,TGaussianAtom &GA){
    if (xAtom>=0 and xAtom<NAtoms) Gauss[A].Copy(GA);
    else <throw exception>;
  };
   void Normalize(){
     for (int A=0;A<NAtoms; A++) Gauss[A].Normalize();
   };
   void CalcOverlap(TSymmetricMatrix &S);
   void Read(FILE *f);
   void Print();
};
```

Inheritance and Virtual Functions

- ✓ Very often, you have a bunch of tasks to do that have something in common and they are embedded in larger tasks, where they perform one specific action.
- ✓ In order to help you streamlining such situations, C++ let's you design „virtual functions"
- ✓ Example:

```
/* ------------------------------------------------------
 * First we define an abstract task that is supposed to
 * do something. What it is we leave unspecified
 *
 * This is the content of the virtual function DoSomething
 * NOTE: the "=0" tells the compiler that any "child" of
 *       the class AbstractTask MUST overload this function
 * ------------------------------------------------------
 */
class AbstractTask{
public:
  virtual void DoSomething(const char* ) = 0;
};
```

Making Virtual Functions Concrete

```
/* ------------------------------------------------
 * Now we define one concrete task.
 *
 * The concrete task is an "heir" of Abstract task and
 * overloads the virtual function "DoSomething" which is
 * will do something concrete.
 *
 * NOTE: the virtual function here is declared with "=0"
 * ------------------------------------------------
 */
class WindowCleaner : public AbstractTask{
public:
  virtual void DoSomething(const char *house);
};

void WindowCleaner::DoSomething(const char *house)
{
  printf("Cleaning Windows in house %8s\n",house);
}

/* ------------------------------------------------
 *
 * Another concrete task
 *
 * ------------------------------------------------
 */
class FloorCleaner : public AbstractTask{
public:
  virtual void DoSomething(const char *house);
};

void FloorCleaner::DoSomething(const char *house)
{
  printf("Cleaning floors in house %8s\n",house);
}
```

Incorporating virtual functions

```
/* ------------------------------------------------
 * Here is how to use it:
 * We have a function that is in need of performaing a
 * certain task, and pass the concrete task onto it
 * ------------------------------------------------
 */
void CleaningForce(AbstractTask &MyTask)
{
  int NHouses = 5;
  const char *House[5] = {"Miller","Smith","Jones","Mayer","Trump"};

  for (int i=0;i<NHouses; i++){
    printf("Now working on house %d, the home of %s\n",i+1,House[i]);
    printf(" task is -> ");
    MyTask.DoSomething(House[i]);
    printf("\n");
  };
};
```

How it looks in practice

```
int main(int argc, char **argv)
{
  // These are our "workers" - the ones who do something concrete
  WindowCleaner W;
  FloorCleaner  F;
  JewelThief    J;

  // Now we can drive the performance of concrete tasks easily
  printf("First cleaning task\n");
  printf("--------------------\n");
  CleaningForce(W);
  printf("Second cleaning task\n");
  printf("--------------------\n");
  CleaningForce(F);
  printf("Third cleaning task\n");
  printf("--------------------\n");
  CleaningForce(J);

  // But this will work too
  AbstractTask  *X;

  printf("Dynamic cleaning task\n");
  printf("----------------------\n");
  X= new JewelThief;
  CleaningForce(*X);
}
```

Program output

```
First cleaning task
-------------------
Now working on house 1, the home of Miller
 task is -> Cleaning Windows in house   Miller

Now working on house 2, the home of Smith
 task is -> Cleaning Windows in house    Smith

Now working on house 3, the home of Jones
 task is -> Cleaning Windows in house    Jones

Now working on house 4, the home of Mayer
 task is -> Cleaning Windows in house    Mayer

Now working on house 5, the home of Trump
 task is -> Cleaning Windows in house    Trump

Second cleaning task
--------------------
Now working on house 1, the home of Miller
 task is -> Cleaning floors in house   Miller

Now working on house 2, the home of Smith
 task is -> Cleaning floors in house    Smith

Now working on house 3, the home of Jones
 task is -> Cleaning floors in house    Jones

Now working on house 4, the home of Mayer
 task is -> Cleaning floors in house    Mayer

Now working on house 5, the home of Trump
 task is -> Cleaning floors in house    Trump
```

```
Third cleaning task
-------------------
Now working on house 1, the home of Miller
 task is -> Stealing jewelery in house   Miller while the other idiots are cleaning

Now working on house 2, the home of Smith
 task is -> Stealing jewelery in house    Smith while the other idiots are cleaning

Now working on house 3, the home of Jones
 task is -> Stealing jewelery in house    Jones while the other idiots are cleaning

Now working on house 4, the home of Mayer
 task is -> Stealing jewelery in house    Mayer while the other idiots are cleaning

Now working on house 5, the home of Trump
 task is -> Stealing jewelery in house    Trump while the other idiots are cleaning
 ... he he, at least he deserves it
```

Example: Calculating One-Electron Integrals

```
SUBROUTINE OneElectronLoop(TIntegralKernel   &K,
                           TIntegralConsumer &C)

BEGIN
  Loop ish over shells
    Loop jsh<=ish over shells
      Compute or get la, lb, P, PA, PB
      ICART=0
      For iprim=0..nprim-1
        CALL K.PrimitiveIntegrals(IPRIM)
        Sum ICART += da*db*IPRIM
      end
      Tranform ICART to Spherical harm. ILM
        CALL C.IntegralConsumer(ILM)
    End jsh
  End ish
END
```

calculates integrals for one primitive pair

Does something with one batch of integrals

Only one such loop (and one for general contraction) in the entire program package! Covers ALL one-electron integrals

Example of an Integral Kernel

```
// Calculation of the overlap integral
void TOverlapIntegral::CalcPrimitiveIntegrals(int64 la, int64 lb, double a, double b,
                                              double R2, double *P, double *PA, double *PB,
                                              Tensor<5, double> &ET, Tensor<5, double> &RT,
                                              Tensor<3, double> &INTS)
{
  auto E      = ET.SubTensor<4>({0},{0});
  // Compute the constant SAB
  double ab  = a+b;
  double ABI = 0.5/ab;
  double SAB = pow(2.0 * M_PI * ABI, 1.5) * exp(-2.0 * ABI * a * b * R2);
  // Compute the auxiliary array E
  E_Function(la, lb, ABI, SAB, PA, PB,  E);
  // Loop over Cartesian components and make S
  int64 dimi= CDIM(la);
  int64 dimj= CDIM(lb);
  for (int64 i = 0; i < dimi; i++) {
    int64 l1 = GTO_xyz[la][i][0];
    int64 m1 = GTO_xyz[la][i][1];
    int64 n1 = GTO_xyz[la][i][2];
    for (int64 j = 0; j < dimj; j++) {
      int64 l2 = GTO_xyz[lb][j][0];
      int64 m2 = GTO_xyz[lb][j][1];
      int64 n2 = GTO_xyz[lb][j][2];
      INTS(0, i, j) = E(0,l1,l2,0) * E(1,m1,m2,0) * E(2,n1,n2,0);
    }; // j
  }; // i
};
```

Example of an Integral Consumer

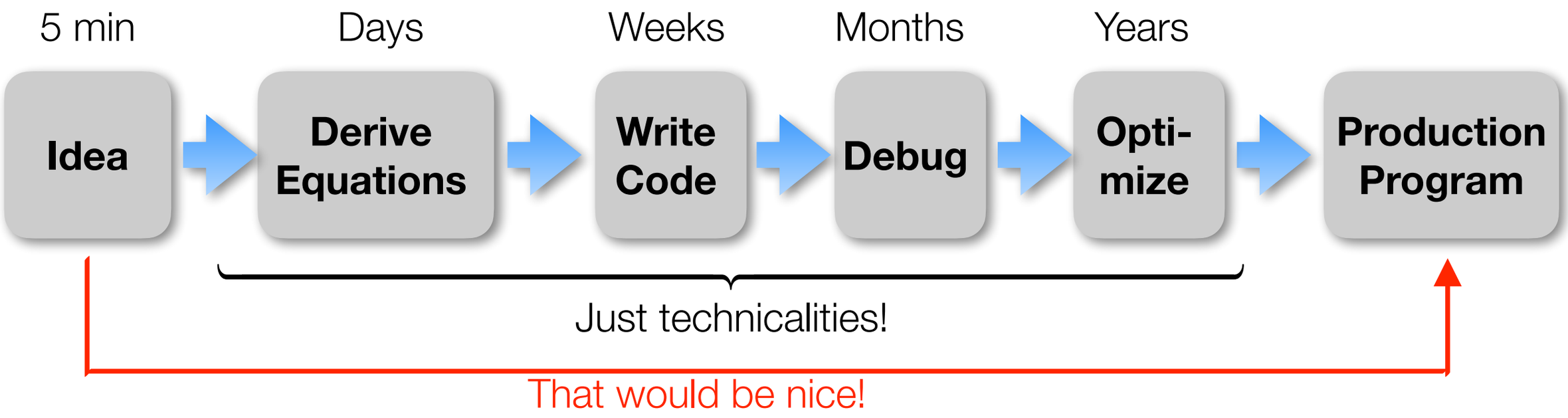
```
/* ----------------------------------------------
 *  FN 03/2021
 *
 *  For most integrals the right action is to just
 *  simply store it in a matrix.
 * ----------------------------------------------
 */
class TSymOneElectronIntegralStorer
                        : public TOneElectronConsumer{
  int64 KernelLength;
  TSharkBasis *BAS;
  TRMatrixSym *IOUT;
public:
  TSymOneElectronIntegralStorer(){
    KernelLength=0;
    BAS=0;
    IOUT=0;
  };
  void SetKernelLength(int64 x){
    KernelLength=x;
  }
  void SetBasis(TSharkBasis *x){
    BAS=x;
  }
  void SetOutput(TRMatrixSym *x){
    IOUT=x;
  }
  virtual void DigestIntegrals(int64 ish, int64 jsh,Tensor<3> &INTS);
};
```

```
void TSymOneElectronIntegralStorer::DigestIntegrals
                                    (int64 ish, int64 jsh,Tensor<3> &INTS)
{
  int64 li  = BAS->BG[ish].l;
  int64 lj  = BAS->BG[jsh].l;
  int64 ofsi= BAS->BG[ish].ofs;
  int64 ofsj= BAS->BG[jsh].ofs;
  int64 dimi= SHARK_LDIM(li);
  int64 dimj= SHARK_LDIM(lj);
  for (int64 i=0;i<dimi; i++){
    int64 addri= ofsi+i;
    int64 jend= dimj;
    if (Diagonal) jend=i+1;
    for (int64 j=0;j<jend; j++){
      int64 addrj= ofsj+j;
      for (int64 nk=0; nk<KernelLength; nk++){
        IOUT[nk](addri, addrj) = INTS(nk, i, j);
      }
    }
  }
}
```

PART 4

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

$$\left\langle \Psi_0 \mid E^n_m E^q_p \ldots E^s_r \mid \Psi_0 \right\rangle, \qquad E^q_p = a^+_{q\beta} a_{p\beta} + a^+_{q\alpha} a_{p\alpha}.$$

Or, in terms of elementary spin-orbital operators:

$$\left\langle \Psi_0 \mid a^n_m a^q_p \ldots a^s_r \mid \Psi_0 \right\rangle,$$

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:

$$E^p_i \left| \Psi_0 \right\rangle = 2\delta_{ip} \left| \Psi_0 \right\rangle, \qquad \left\langle \Psi_0 \right| E^i_p = 2\delta_{ip} \left\langle \Psi_0 \right|,$$

$$E^p_a \left| \Psi_0 \right\rangle = 0, \qquad \left\langle \Psi_0 \right| E^a_p = 0,$$

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

Awkward by hand, easy for a computer

➡ Creates 0's, Kronecker deltas and ‚pre-densities' (MR case)

$$\gamma^{uw\ldots y}_{tv\ldots x} = \left\langle \Psi_0 \mid E^u_t E^w_v \ldots E^y_x \mid \Psi_0 \right\rangle .$$

Issues:

- ✓ redundant terms are generated
- ✓ terms that cancel each other are generated
- ✓ Equivalent terms may have inequivalent labels
- ✓ …

Post-processing required

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‘

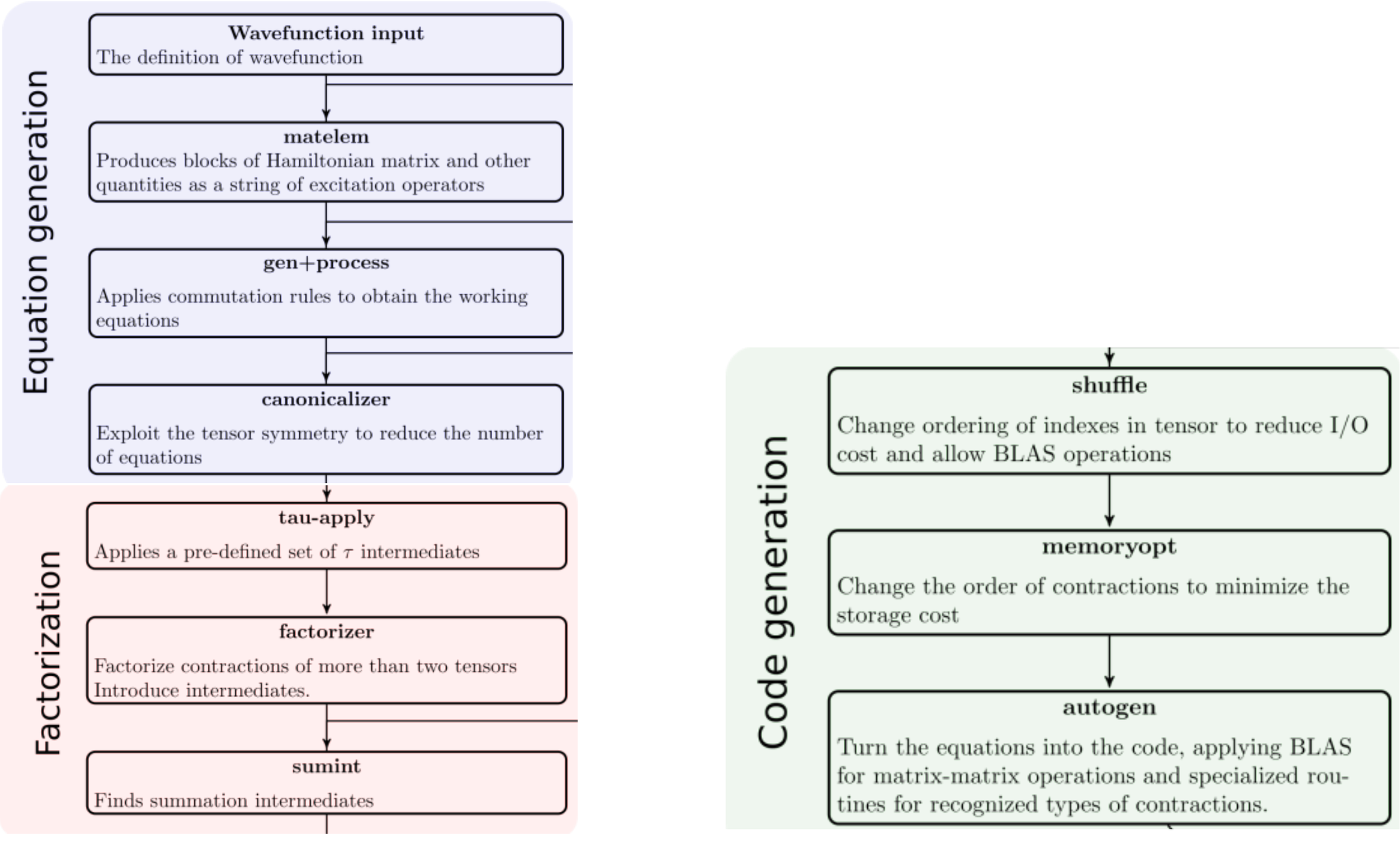
2. Factorizer

- ✓ 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

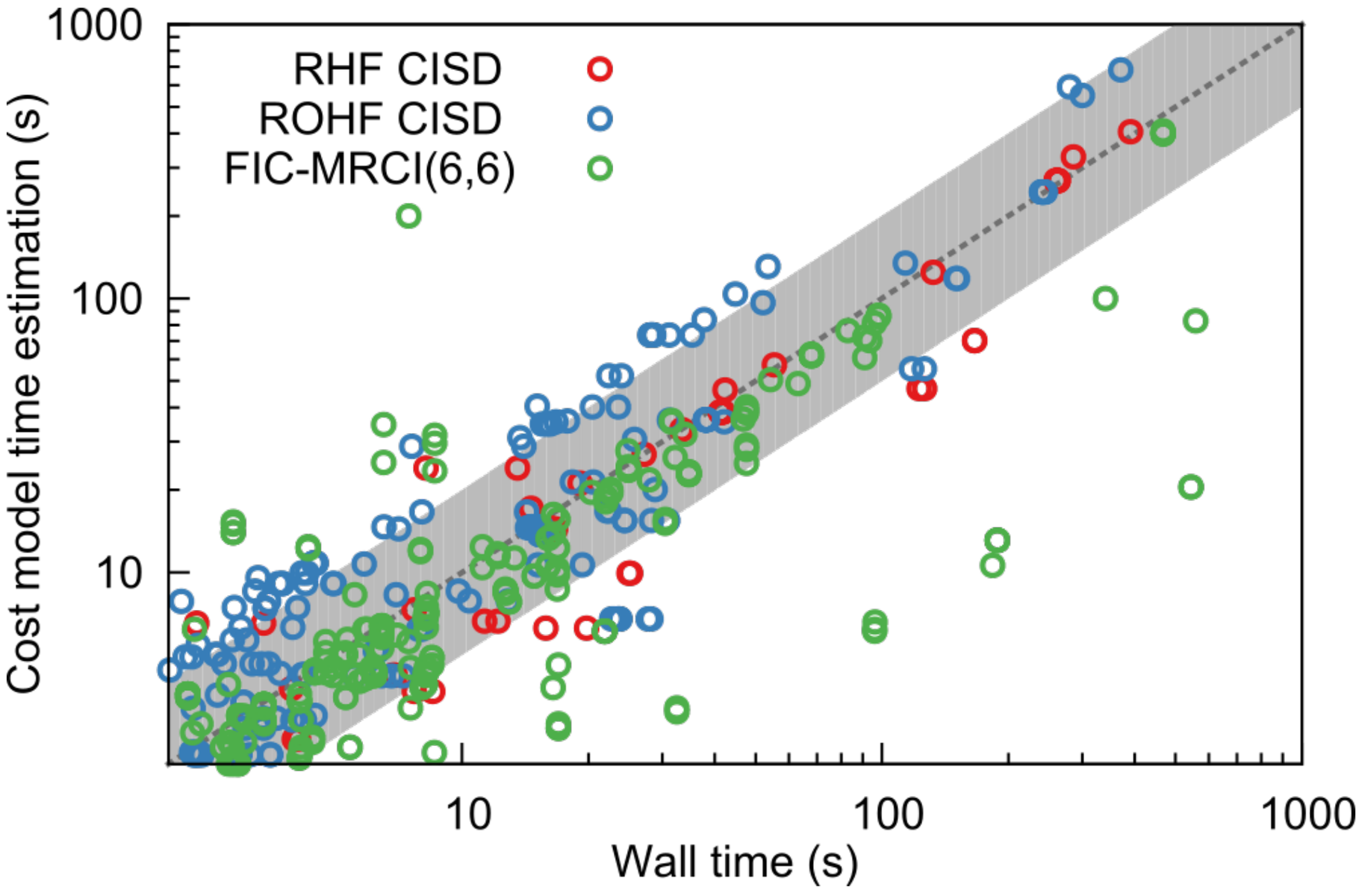
- ✓ Writes code for a specific electronic structure package
- ✓ Recognizes patterns/contractions for which highly optimized code exists
- ✓ Ensures that all terms have their correct formal scaling
- ✓ Ensures minimal I/O and maximal use of BLAS
- ✓ Generates parallel code, code for specific machines, ….

Realization of a Code generation chain (AGE)

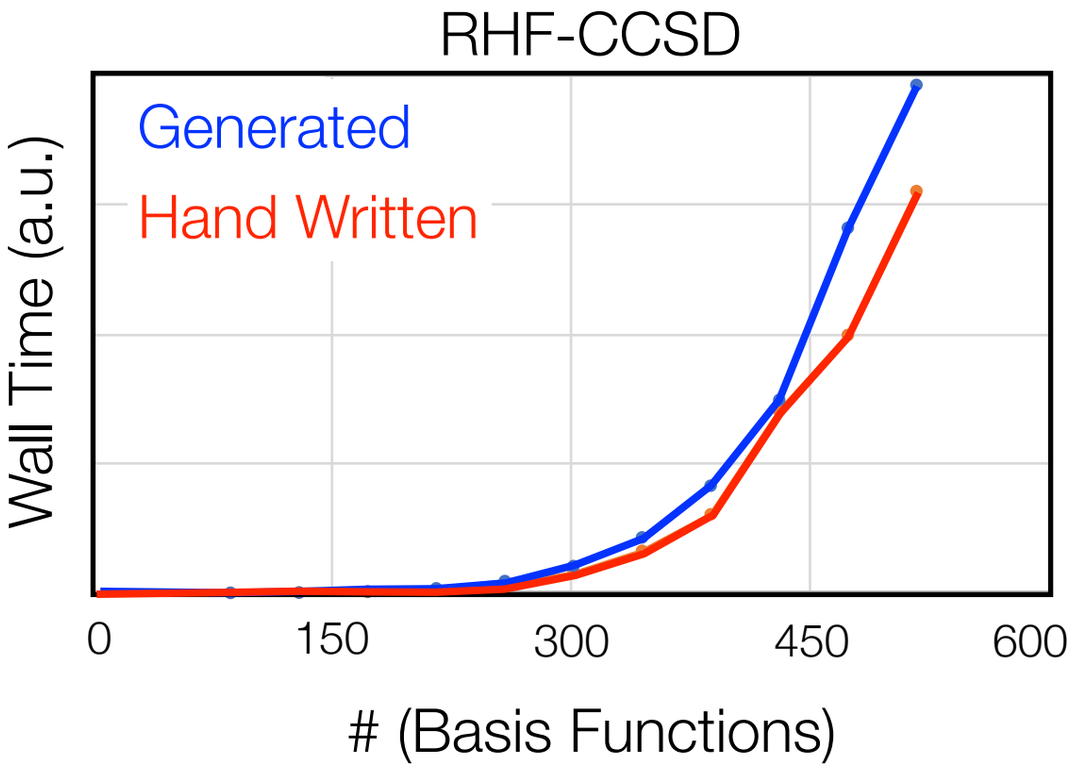


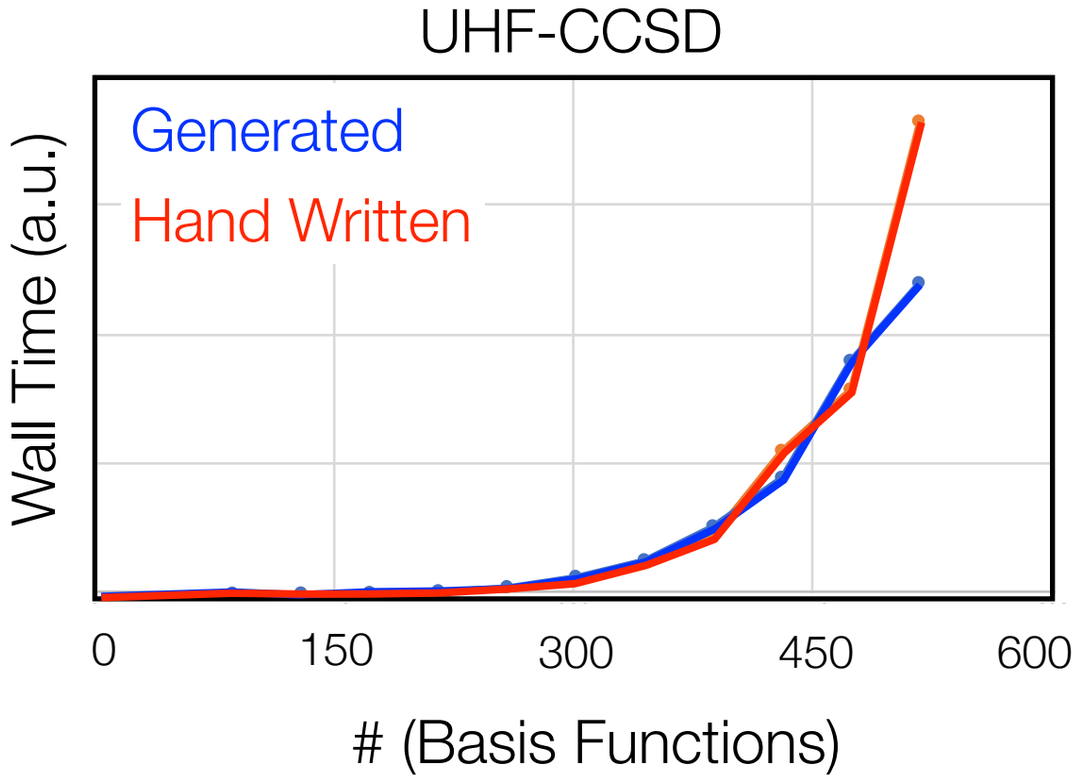
Cost model

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



Generated vs Hand Written Code





Where does the hand written Code win?

Hand code:

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

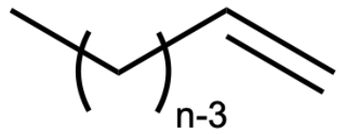
4 dgemm/k

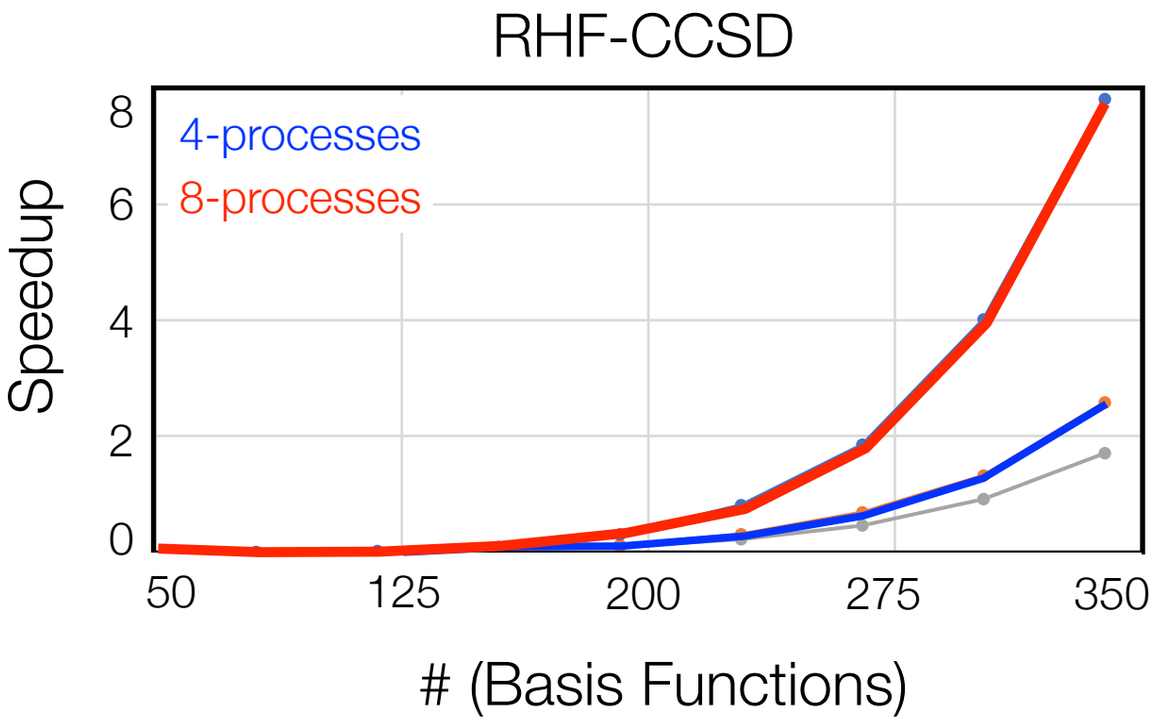
Generated code:

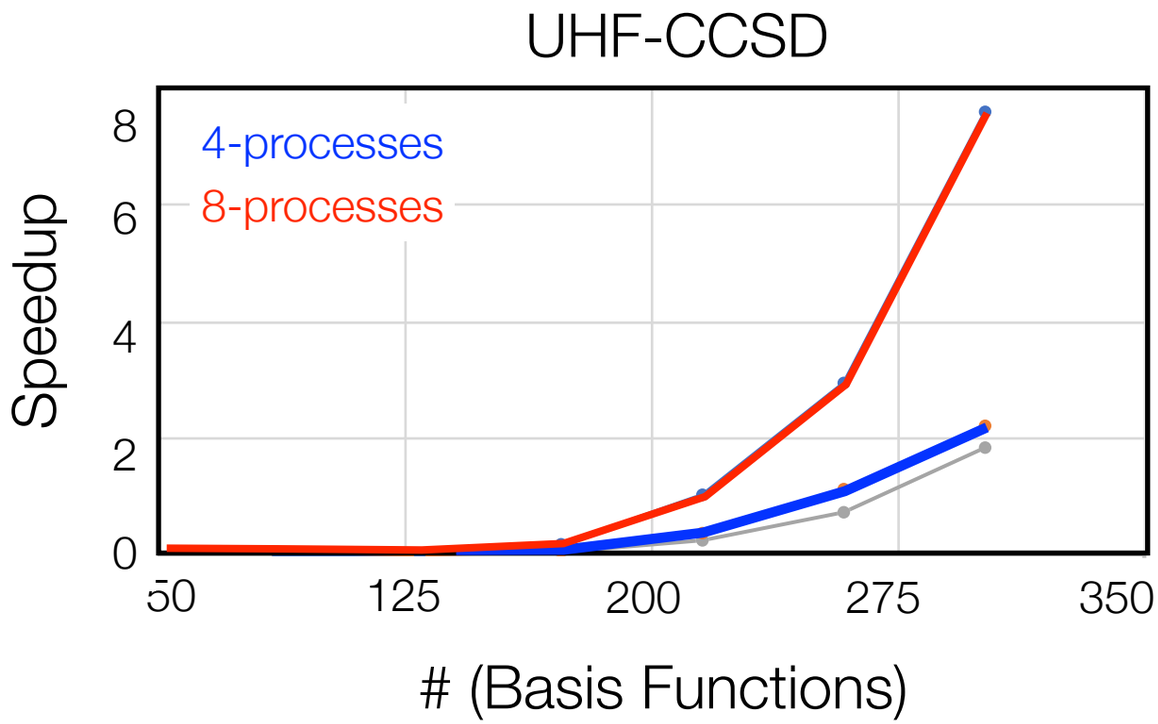
$$\sigma^{ij} \leftarrow \sum_k -\mathbf{J}^{ik}\mathbf{C}^{kj} - \mathbf{C}^{kj}\mathbf{J}^{ik} - \mathbf{J}^{kj}\mathbf{C}^{ik} - \mathbf{C}^{ik}\mathbf{J}^{kj} - \mathbf{C}^{ki}\mathbf{K}^{kj} - \mathbf{K}^{ik}\mathbf{C}^{jk} + 2\mathbf{K}^{ik}\mathbf{C}^{jk} + 2\mathbf{C}^{ik}\mathbf{K}^{kj}$$

8 dgemm/k

Coupled Cluster Gradients

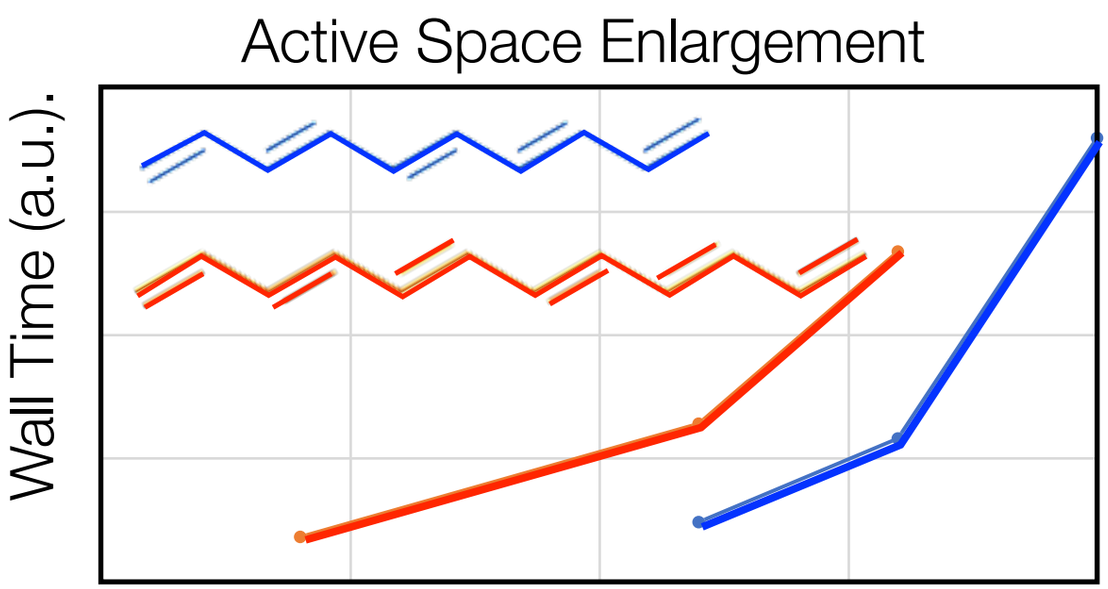


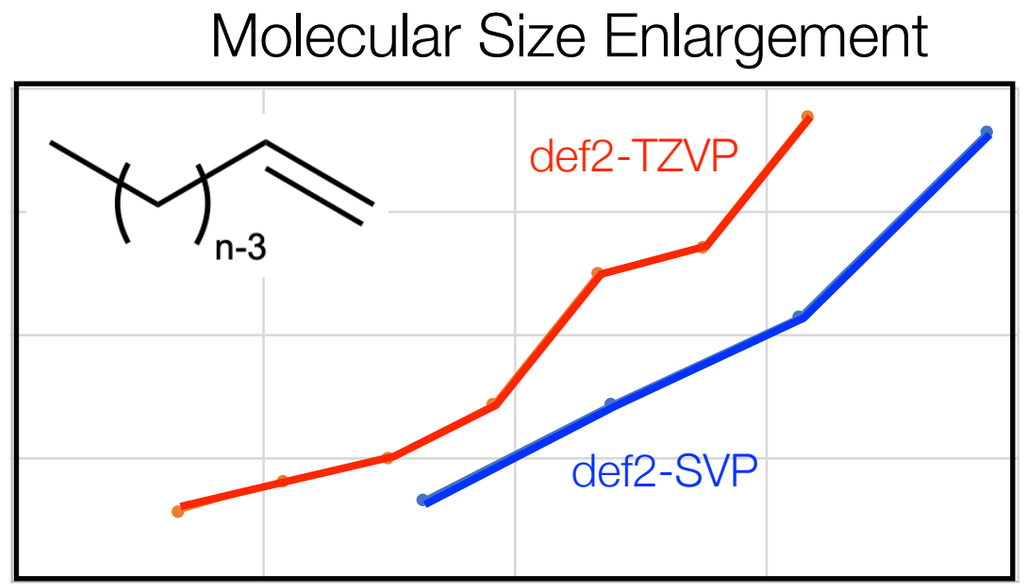




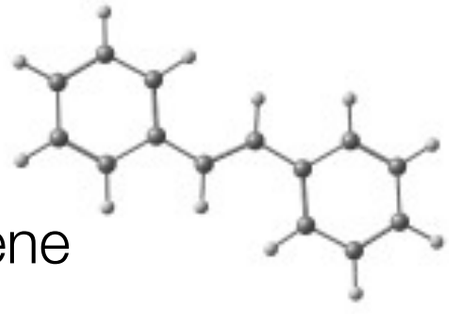
- Canonical Coupled Cluster gradients with perhaps >500 basis functions possible
- Parallel Scaling is good
- More than 10x faster than numerical gradients

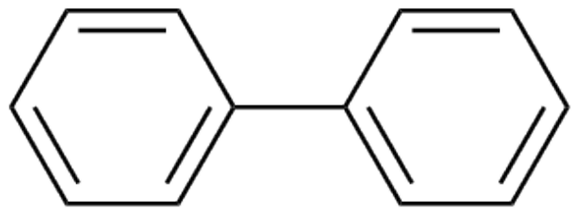
Reduced Scaling FIC-MRCC Implementation





- Accessible molecular size roughly the same as single reference CCSD
- Without reduced scaling limit about 8 active orbitals
- With reduced scaling limit about 12 active orbitals

Stilbene 

Biphenyl 

Complexity: Example

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

$$\left|\Phi_{ij}^{ta}\right\rangle = E_{ij}^{ta}\left|\Psi_0\right\rangle = \sum_I C_I^{(CASSCF)} E_{ij}^{ta}\left|\Phi_I^{(CAS)}\right\rangle$$

- ✓ 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)

MQM 2016

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

THE JOURNAL OF CHEMICAL PHYSICS **142**, 051103 (2015)

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.