

Algorithm Design

Frank Neese

Max Planck Institut für Chemische Energiekonversion
Stiftstr. 34-36
D-45470 Mülheim an der Ruhr
Frank.Neese@cec.mpg.de



Goals of Quantum Chemical Method Development

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

Scaling Laws

A quantum chemical algorithm can be characterized by its 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:

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

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

➔ Number of AOs is proportional to system size

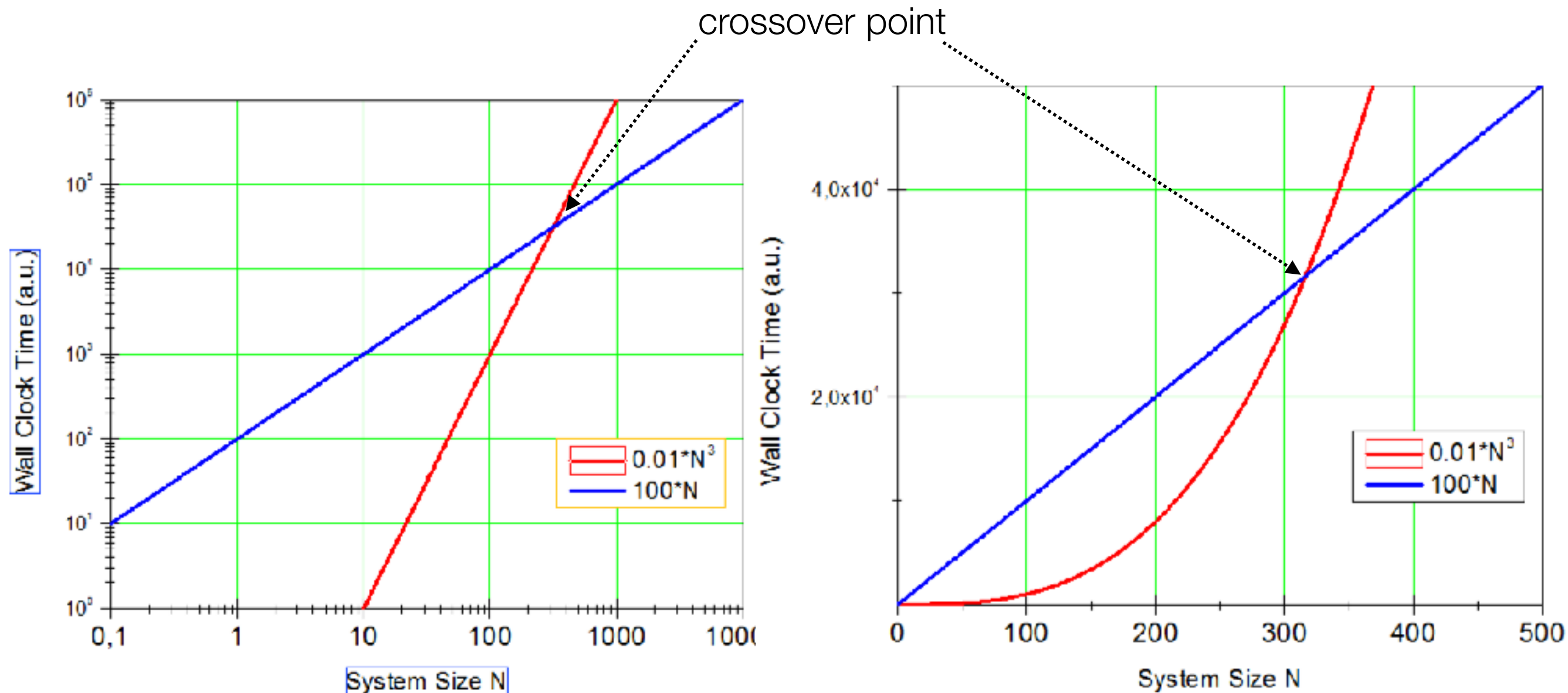
$(\mu\nu \mid \kappa\tau)$ Number of AOs integrals proportional to N^4 ($O(N^4)$)

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

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

$O(N^8)$

Prefactor vs Scaling



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

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

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%)
<O H D>	...	0.072 sec	(0.0% of sigma)
<O 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)

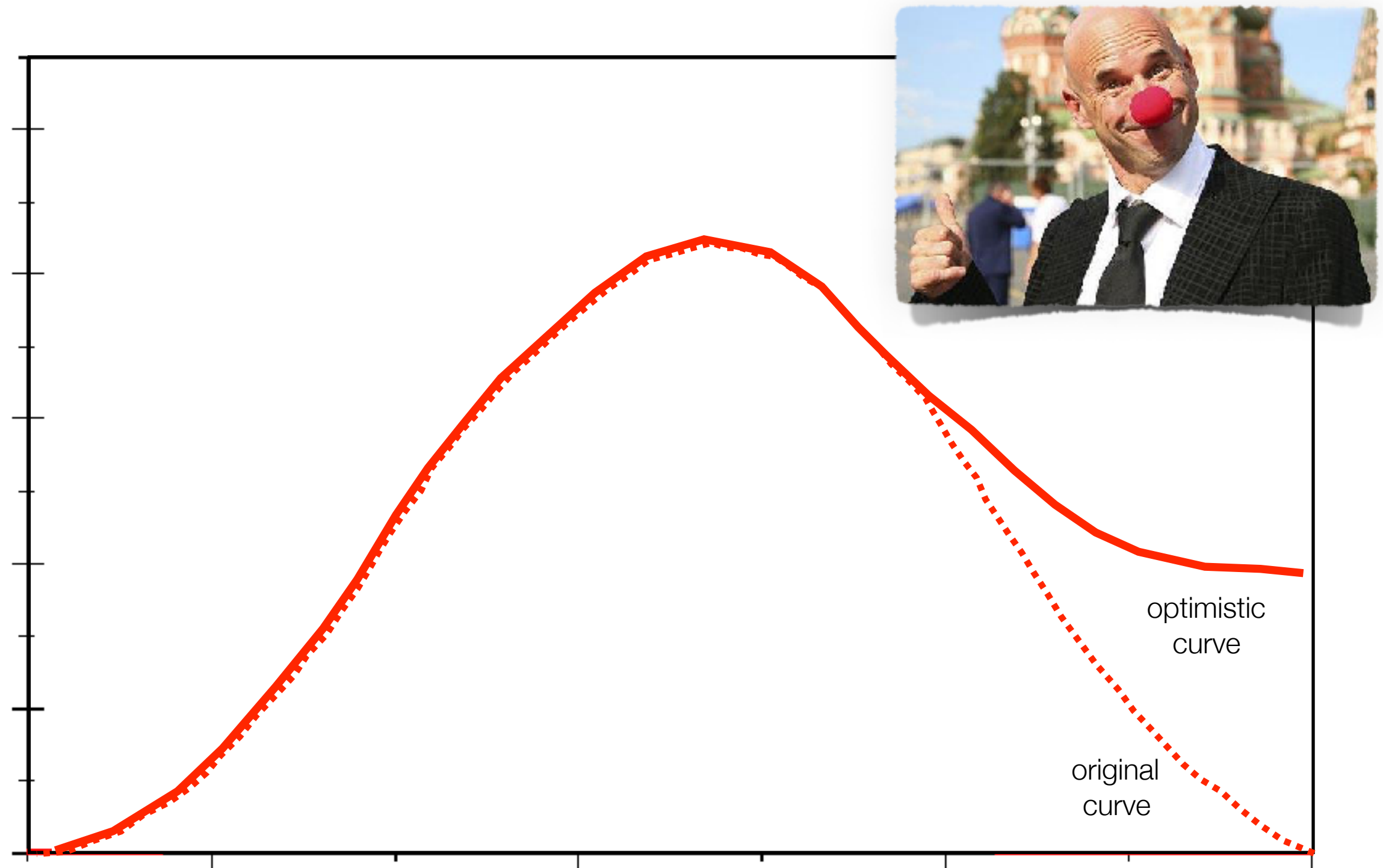
This is worth your while!

How much can you gain from optimizing these steps?

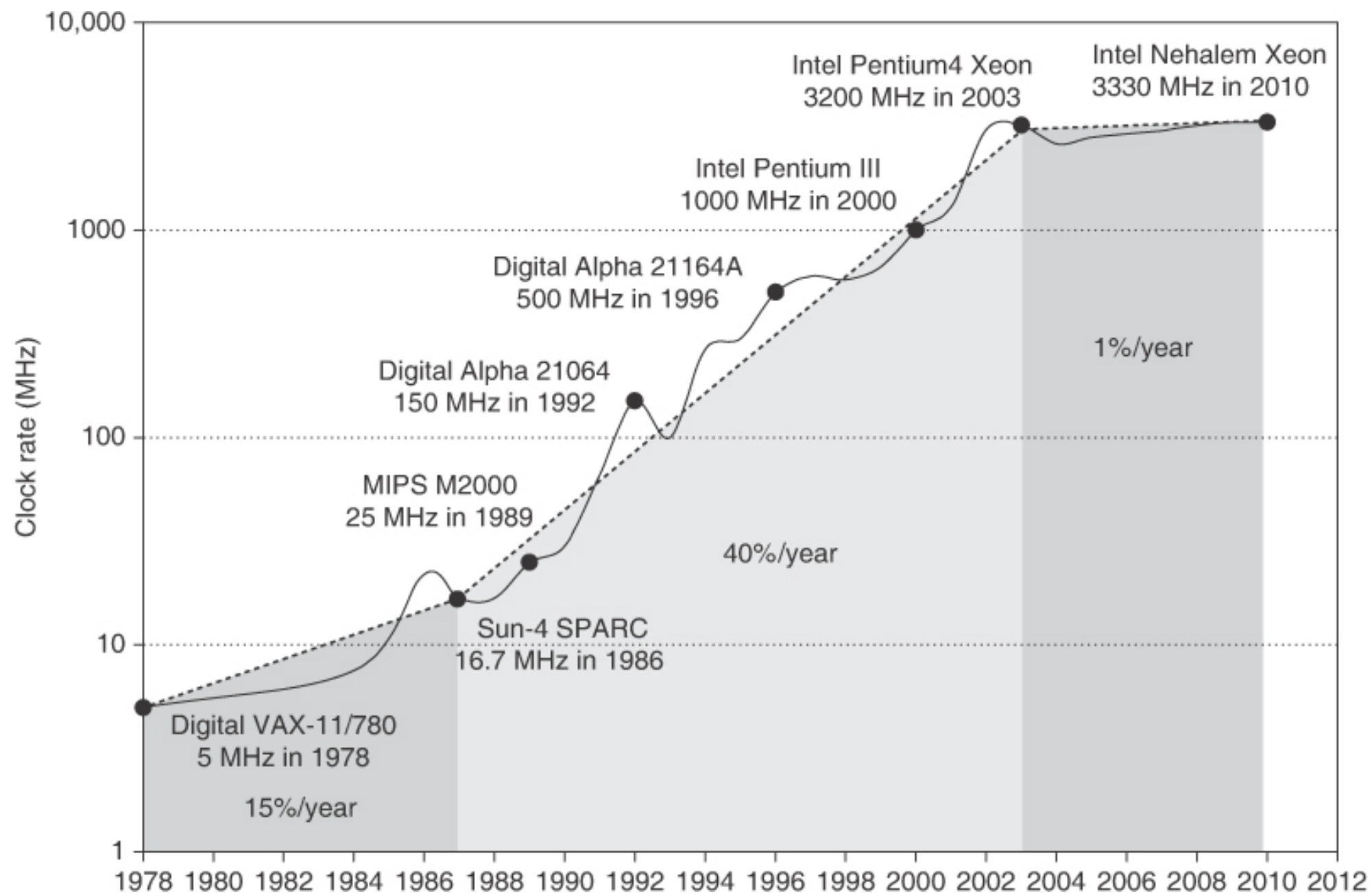
Prelude:

A little information on Computers

The ,Gauss-curve' of method development



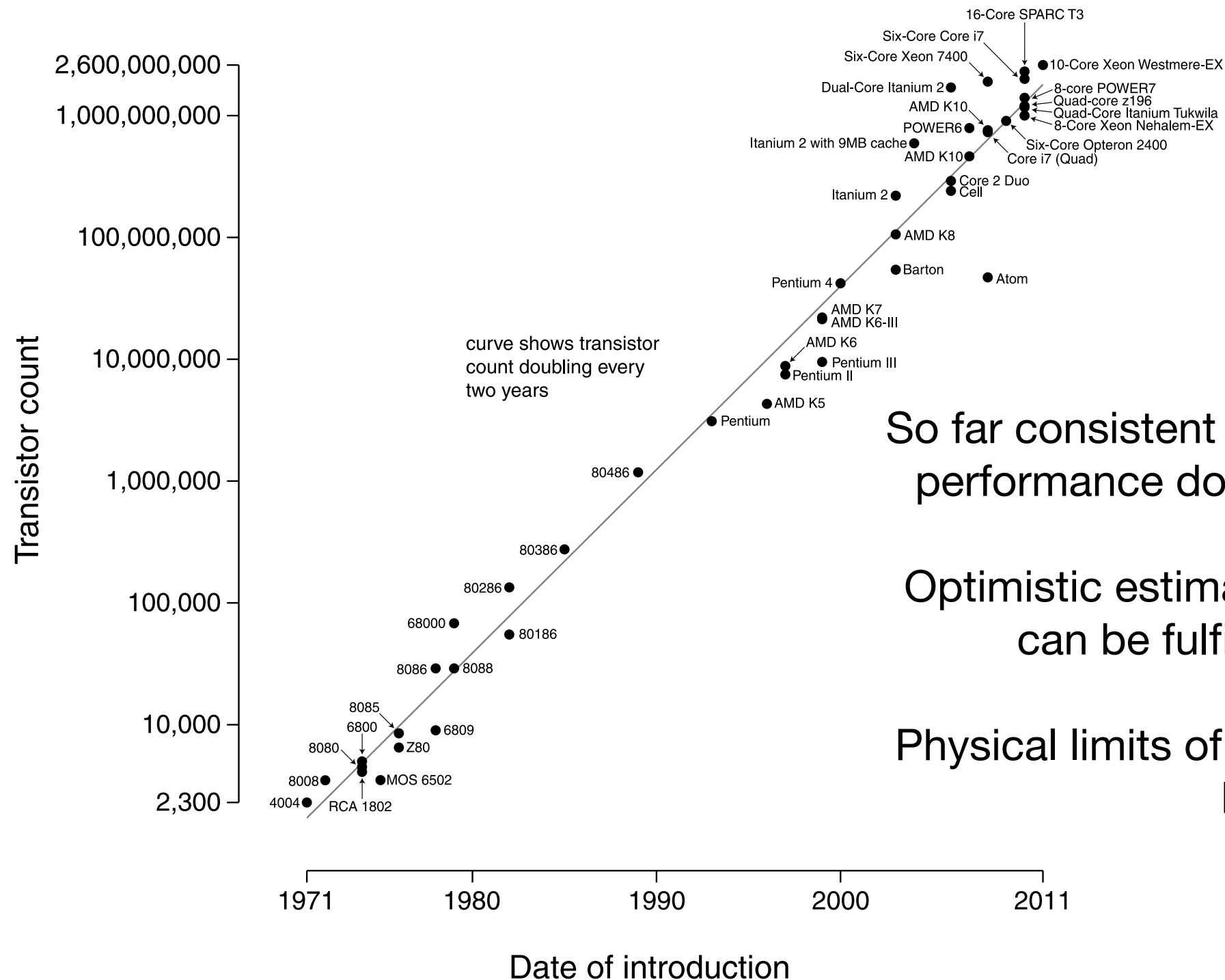
Single CPU Clockspeed



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: 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

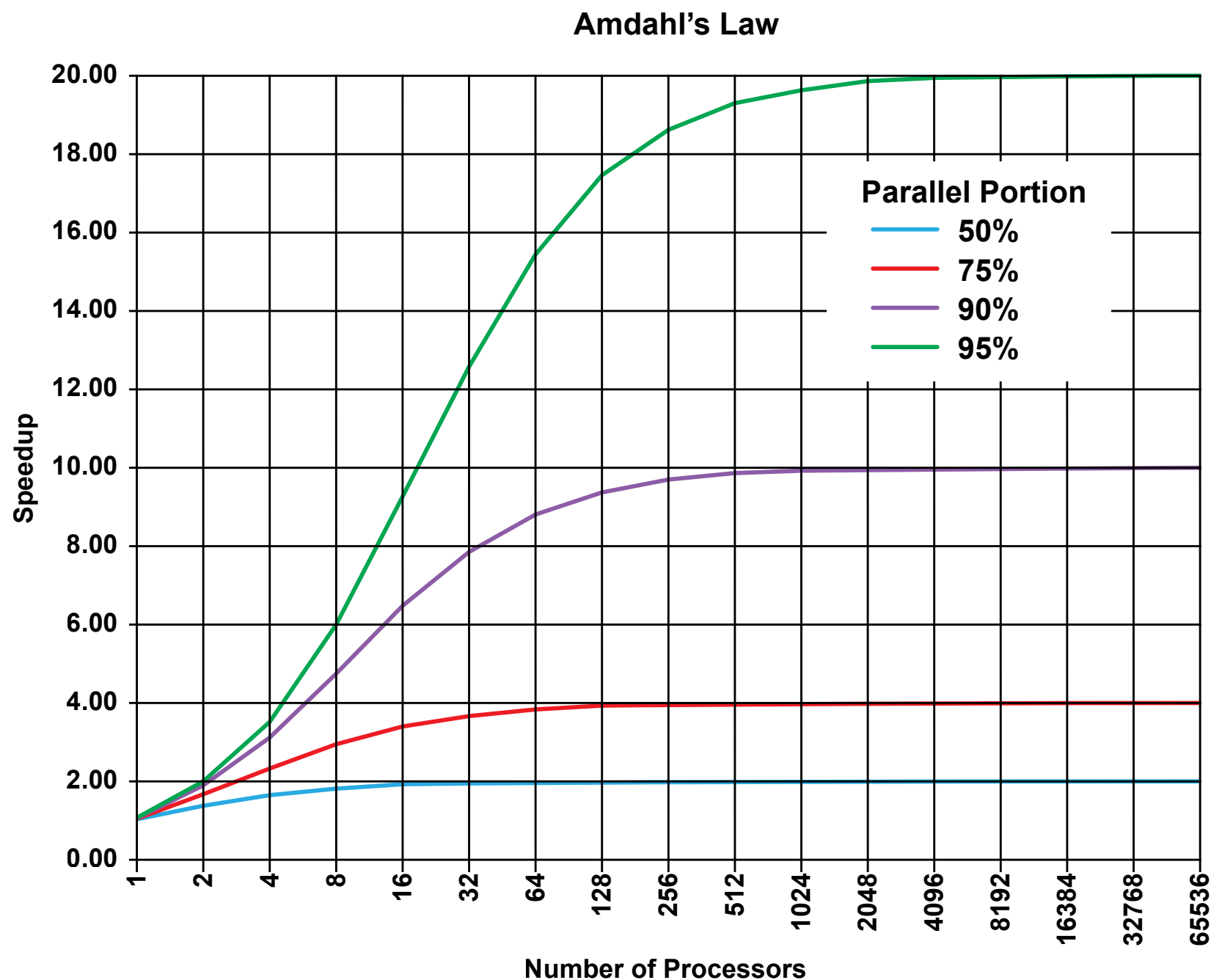
Paradigm Change:

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



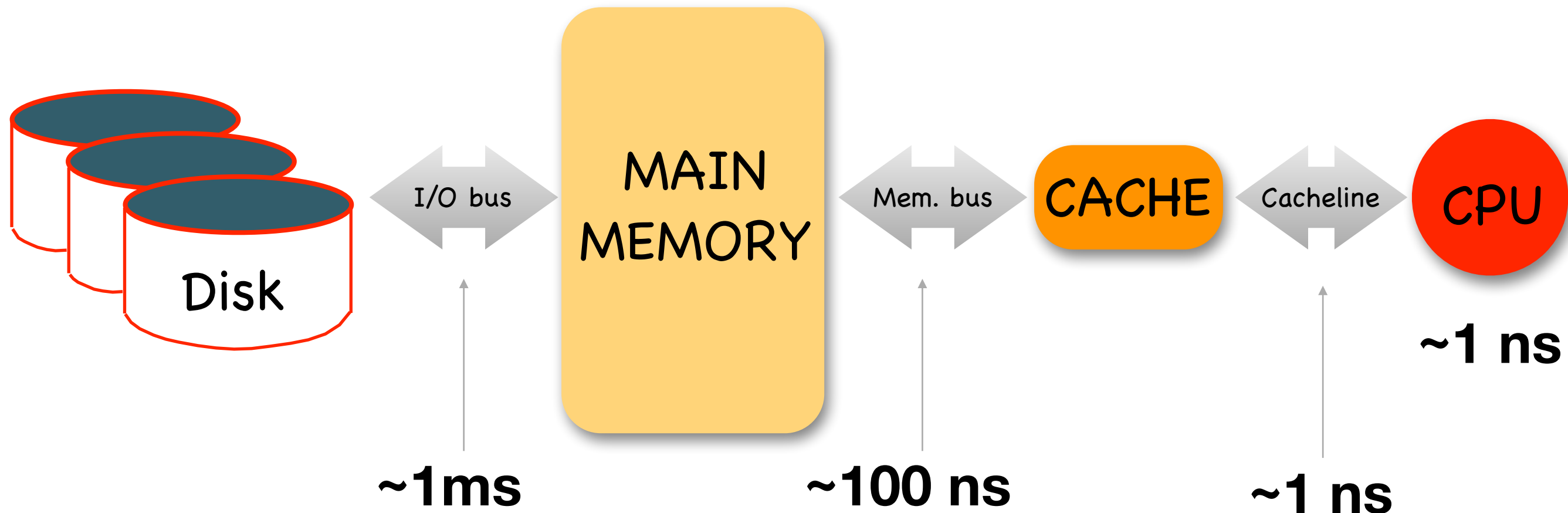
Speedup:

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

P: Parallel portion of code

N: Number of Processors

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**

Development Guidelines for Quantum Chemistry

„Getting Exactly the same number faster“



Take it with a grain of salt!

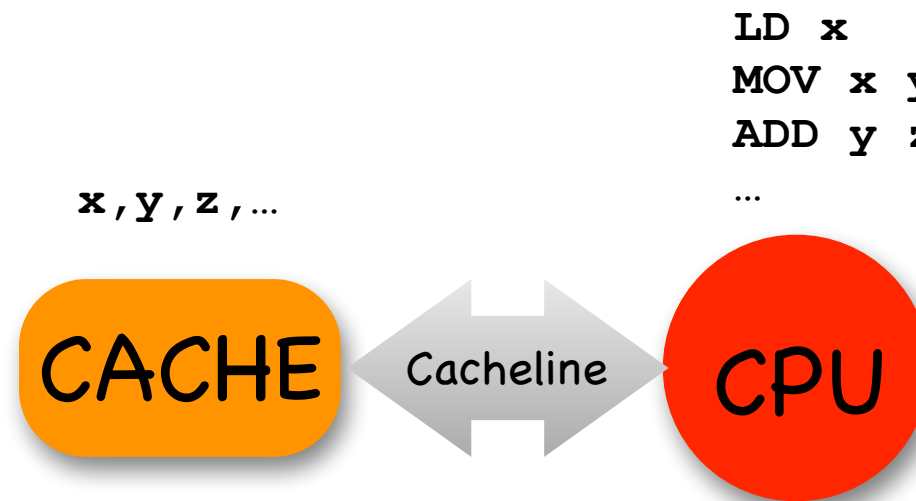
The Do's and Don't's of Programming: Overview

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

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

Instruction Pipelines and Logic

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



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

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

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

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

careful optimization
avoids logical
decisions in time
critical parts of the
program

Unit Stride Access

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

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

Good:

```
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.

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
```

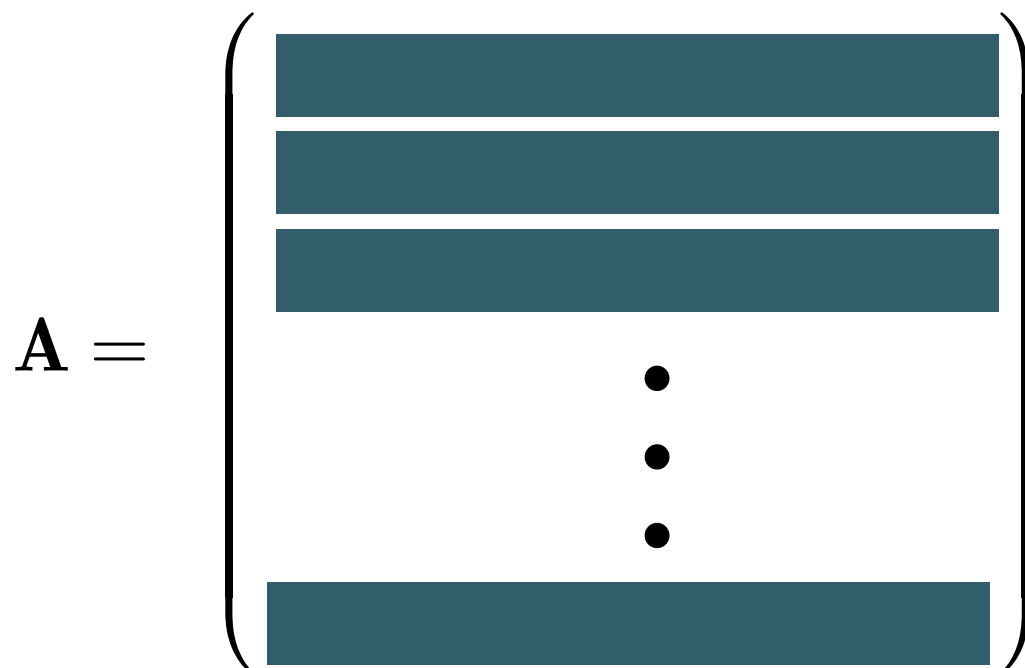
Example: The power of BLAS (II)

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

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

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

Why that?



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

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

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

Repulsion integral in MD:

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

const

*Expansion of
G^AG^B in
Hermite basis*

*Expansion of
G^CG^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 $\sim l_A + l_B$ (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

Transformation to Spherical Harmonics

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

$$(G_{\mu}^{l_a m_a} G_{\nu}^{l_b m_b} | G_{\kappa}^{l_c m_c} G_{\tau}^{l_d m_d}) = \sum_p \sum_q \sum_r \sum_s d_p^{l_a m_a} d_q^{l_b m_b} d_r^{l_c m_c} d_s^{l_d m_d} (G_{\mu}^{x_p y_p z_p} G_{\nu}^{x_q y_q z_q} | G_{\kappa}^{x_r y_r z_r} G_{\tau}^{x_s y_s z_s})$$

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

c++ -O3 -funroll-loops ...

Straightforward code

```
Cart2S1m(SRC,DST)
  loop i in xyz_b
    loop j in xyz_c
      loop k in xyz_d
        loop m in slm_a
          x=0
          loop l in xyz_a
            x=x+SRC[l,i,j,k]*d[m,l]
          end loop l
          TEMP(m,i,j,k)=x
        end loops i,j,k
      etc for the other three indices
      to fill target array DST
    end_subroutine
```

1000 x (gg|gg)=0.12 sec

Only factor ~2
➡ The compiler does a decent job here

Unrolled, optimized code

```
Cart2S1m_4_f(SRC,DST)
  loop i in xyz_a
    loop j in xyz_b
      loop k in xyz_c
        DST[ 0+ 7*(k+dim3*(j+dim2*i))]=
          +0.258198889747161153*SRC[ 2+10*(k+dim3*(j+dim2*i))]
          -0.387298334620741647*SRC[ 4+10*(k+dim3*(j+dim2*i))]
          -0.387298334620741647*SRC[ 6+10*(k+dim3*(j+dim2*i))];
        DST[ 1+ 7*(k+dim3*(j+dim2*i))]=
          -0.158113883008418971*SRC[ 0+10*(k+dim3*(j+dim2*i))]
          -0.158113883008418943*SRC[ 5+10*(k+dim3*(j+dim2*i))]
          +0.632455532033675771*SRC[ 7+10*(k+dim3*(j+dim2*i))];
        ...
      end loops
    end_subroutine
```

1000 x (gg|gg)=0.06 sec

Design of an algorithm: FLOP count

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

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

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

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

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

$O(N^8)$ scaling

Must be possible to do better than that

FLOP Count: Partial Integral transformation

Algorithm A: occupied indices first

$$\begin{aligned}
 (i\nu | \kappa\tau) &= \sum_{\mu} c_{\mu i}(\mu\nu | \kappa\tau) & (B^4O) & \quad \mathbf{3125} \\
 (i\nu | j\tau) &= \sum_{\kappa} c_{\kappa j}(i\nu | \kappa\tau) & (O^2B^3) & \quad \mathbf{312} \\
 (ia | j\tau) &= \sum_{\nu} c_{\nu a}(i\nu | j\tau) & (O^2VB^2) & \quad \mathbf{281} \\
 (ia | jb) &= \sum_{\tau} c_{\tau b}(ia | j\tau) & (O^2V^2B) & \quad \mathbf{253}
 \end{aligned}$$

Algorithm B: virtual indices first

$$\begin{aligned}
 (\mu a | \kappa\tau) &= \sum_{\nu} c_{\nu a}(\mu\nu | \kappa\tau) & (B^4V) & \quad \mathbf{28215} \\
 (\mu a | \nu b) &= \sum_{\tau} c_{\tau b}(\mu a | \kappa\tau) & (V^2B^3) & \quad \mathbf{25312} \\
 (ia | \nu b) &= \sum_{\mu} c_{\mu i}(\mu a | \nu b) & (OV^2B^2) & \quad \mathbf{2531} \\
 (ia | jb) &= \sum_{\nu} c_{\nu j}(ia | \nu b) & (O^2V^2B) & \quad \mathbf{253}
 \end{aligned}$$

Four $O(N^5)$ steps

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

0.07

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

FLOP count versus Performance

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

Example: Integral direct partial integral transformation for MP2

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

Key step: integral transformation

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

FLOP count optimized algorithm

```
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
```

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

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  $K[p,r](i,j) = (\mathbf{C}_{occ}^T * \mathbf{K}[p,r] * \mathbf{C}_{occ})_{ij}$ 
      Write matrix  $K[p,r]$  to disk
    end loops p,r
    Resort Integrals  $K[p,r](i,j)$  to give  $K[i,j](p,r)$  ( $i \leq j$ )
    Loop i= 1..Nocc
      loop j=1..i
        Read integrals  $K[i,j](p,r)$ 
        Perform transformation  $K[i,j](a,b) = (\mathbf{C}_{virt}^T * \mathbf{K}[i,j] * \mathbf{C}_{virt})_{ab}$ 
        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**

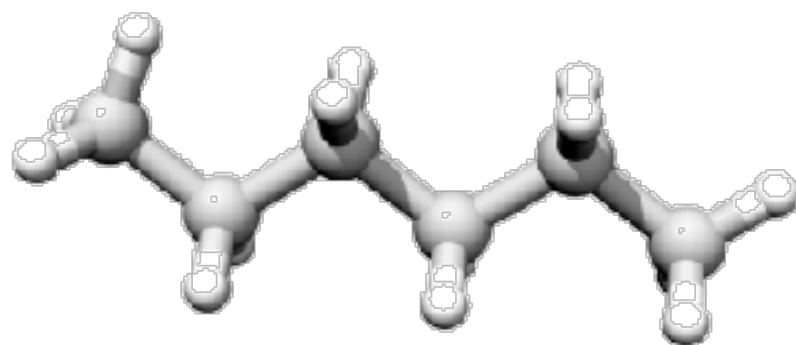
Two BLAS level 3 multiplications in the rate determining step

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 (I)



Hexane

def2-TZVP (258 basis functions)

4 GB main memory used

FLOP optimized algorithm

(1 batches necessary)

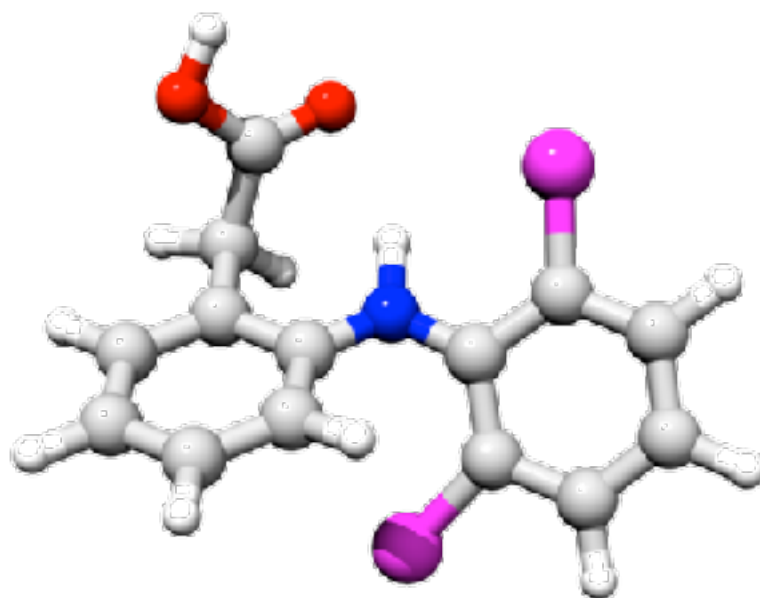
108 sec

BLAS optimized algorithm

79.8 sec

TOTAL TIME for half transformation:	79.4 sec
AO-integral generation	: 71.5 sec
Half transformation	: 5.7 sec
K-integral sorting	: 0.4 sec

Performance Test (II)



Diclophenac

def2-TZVP (667 basis functions)

4 GB main memory used

FLOP optimized algorithm

(25 batches necessary)

(too) many hours

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

Example: Factorization in Coupled Cluster

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

Look at one nonlinear term in the CCSD equations:

$$\sigma_{ab}^{ij} \leftarrow \sum_{kl} \sum_{cd} \langle kl || cd \rangle t_{cd}^{ij} t_{ab}^{kl} \quad \mathbf{O(N^8) \text{ scaling}}$$

- ➔ 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:

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

Example: Factorization in Coupled Cluster

$$\sigma_{ab}^{ij} \leftarrow \sum_{kl} t_{ab}^{kl} \underbrace{\sum_{cd} \langle kl || cd \rangle t_{cd}^{ij}}_{X_{kl}^{ij}} : X_{kl}^{ij} = \sum_{cd} \langle kl || cd \rangle t_{cd}^{ij} \quad \begin{array}{l} \mathbf{N_{occ}^4} \text{ Storage} \\ \mathbf{N_{occ}^4 N_{virt}^2} \text{ FLOPS} \end{array}$$

$$\sigma_{ab}^{ij} \leftarrow \sum_{kl} t_{ab}^{kl} X_{kl}^{ij} \quad \mathbf{N_{occ}^4 N_{virt}^2} \text{ FLOPS}$$

$O(N^6)$ scaling

$2 \times \mathbf{N_{occ}^4 N_{virt}^2}$ FLOPS
 $\mathbf{N_{occ}^4}$ Storage

$$\sigma_{ab}^{ij} \leftarrow \sum_{cd} t_{cd}^{ij} \underbrace{\sum_{kl} t_{ab}^{kl} \langle kl || cd \rangle}_{Y_{cd}^{ab}} : Y_{cd}^{ab} = \sum_{kl} t_{ab}^{kl} \langle kl || cd \rangle \quad \begin{array}{l} \mathbf{N_{virt}^4} \text{ Storage} \\ \mathbf{N_{occ}^2 N_{virt}^4} \text{ FLOPS} \end{array}$$

$$\sigma_{ab}^{ij} \leftarrow \sum_{cd} t_{cd}^{ij} Y_{cd}^{ab} \quad \mathbf{N_{occ}^2 N_{virt}^4} \text{ FLOPS}$$

$O(N^6)$ scaling

$2 \times \mathbf{N_{occ}^2 N_{virt}^4}$ FLOPS
 $\mathbf{N_{virt}^4}$ Storage

Algorithm 1 $\mathbf{N_{occ}^2}$ $\leftarrow \mathbf{MUCH}$ better and \mathbf{MUCH} less Storage!
 $\text{-----} = \text{----} \mathbf{FLOPS} \ll 1$
Algorithm 2 $\mathbf{N_{virt}^2}$

Move Work out of the Inner Loops: Split-J

Substantial performance gains can be realized by choosing intermediates wisely such that redundant work is move out of the inner loops

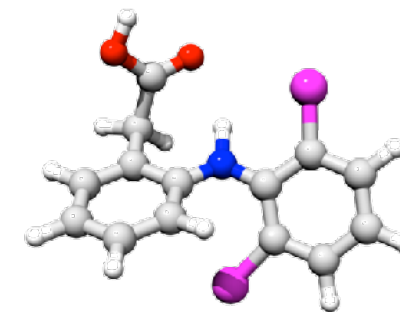
Example: Integrate integral evaluation as early as possible into the target quantities.
For the Coulomb matrix, Ahmadi & Almlöf suggested:

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

$$\begin{aligned}
 J_{\mu\nu} &= \sum_{\kappa\tau} P_{\kappa\tau}(\mu\nu | \kappa\tau) \\
 &= \sum_{\kappa\tau} P_{\kappa\tau} \underbrace{\sum_{tuv} E_{tuv}^{\mu\nu}}_{\text{independent of } \kappa\tau} \sum_{t'u'v'} (-1)^{t'+u'+v'} E_{t'u'v'}^{\kappa\tau} R_{t+t', u+u', v+v'} \\
 &= \sum_{tuv} E_{tuv}^{\mu\nu} \sum_{t'u'v'} R_{t+t', u+u', v+v'} \underbrace{\sum_{\kappa\tau} (-1)^{t'+u'+v'} P_{\kappa\tau} E_{t'u'v'}^{\kappa\tau}}_{\equiv P_{t'u'v'} \text{ independent of } \mu\nu, tuv} \\
 &= \sum_{tuv} E_{tuv}^{\mu\nu} \sum_{t'u'v'} P_{t'u'v'} R_{t+t', u+u', v+v'}
 \end{aligned}$$

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

Performance example



def2-TZVP=667 BFs

**Coulomb term (sec)
(20-builds)**

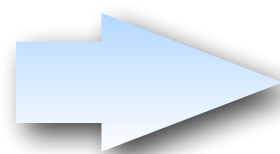
Traditional treatment

5796 sec

Split-J algorithm

2834 sec

=Ahmadi-Almlöf
=Head-Gordon J-engine



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

Example: I/O Heavy Algorithms

The I/O system is the slowest part of your computer!

- Use it as little as possible
- Move its usage as far outside in the loop structure as reasonably possible
- Avoid reading small chunks of data

Example: Integral symmetrization in EOM-CCSD

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  $K^{ia}(b,c)$  from IABC
    loop b=1..Nvir
      loop c=1..Nvir
         $K^{ib}(a,c) += K^{ia}(b,c);$ 
      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!!

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:

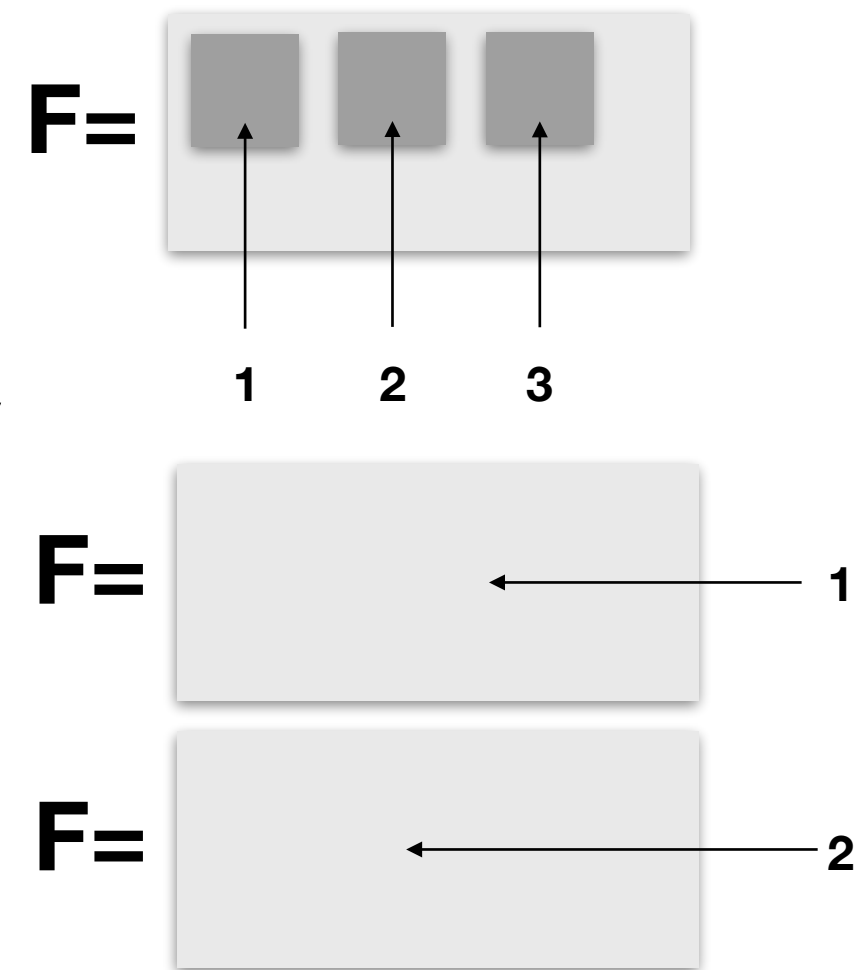
- Open MP, POSIX threads
- efficient use of resources, no memory replication
- difficult to debug large programs
- Can only be used on one machine with common memory

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 *outermost* 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.

Parallelization Examples

- Load balancing** e.g. integral calculation. The time required to calculate a given integral batch is a complicated function of angular momenta, contraction depth and orbital exponents ➤ Load balancing difficult to guarantee
- One possible solution: random distribution of batches among processors.
 - Unevenness will average out in the limit of many batches

`loop i=1..N,i+=1` \longrightarrow `loop i=1..N,i+=NProcs`

- Communication** Multiple and mixed communication of small amounts of data, interspersed by memory allocation can lead to random deadlocks
- Separate memory allocation and communication
 - Vectorize data (copy all data to a large storage vector communicate and then unwrap)

```
loop IP: 0 . . . number of parallel processes
  loop ipair: 0 . . . NPairs
    IP: broadcast N
    if (myID!=IP) ALLOCATE MEMory
    IP: broadcast Matrix of size N
  end loop over pairs
end loop over parallel processes
```

Automatic Code Generation

Problems with Method Development



Conclusions:

- ▶ The technical
- ▶ 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, Shamasundar, 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

$$\left\langle \Psi_0 \left| E_m^n E_p^q \dots E_r^s \right| \Psi_0 \right\rangle, \quad E_p^q = a_{q\beta}^+ a_{p\beta} + a_{q\alpha}^+ a_{p\alpha}.$$

Or, in terms of elementary spin-orbital operators:

$$\left\langle \Psi_0 \left| a_m^n a_p^q \dots a_r^s \right| \Psi_0 \right\rangle,$$

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:

$$\left[E_p^q, E_r^s \right] = E_p^s \delta_{qr} - E_r^q \delta_{ps},$$

Thus:

$$E_i^p \left| \Psi_0 \right\rangle = 2\delta_{ip} \left| \Psi_0 \right\rangle, \quad \left\langle \Psi_0 \left| E_p^i = 2\delta_{ip} \left\langle \Psi_0 \right|,$$

$$E_a^p \left| \Psi_0 \right\rangle = 0, \quad \left\langle \Psi_0 \left| E_p^a = 0,$$

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)

$$\gamma_{tv\dots x}^{uw\dots y} = \left\langle \Psi_0 \mid E_t^u E_v^w \dots E_x^y \mid \Psi_0 \right\rangle.$$

- 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‘

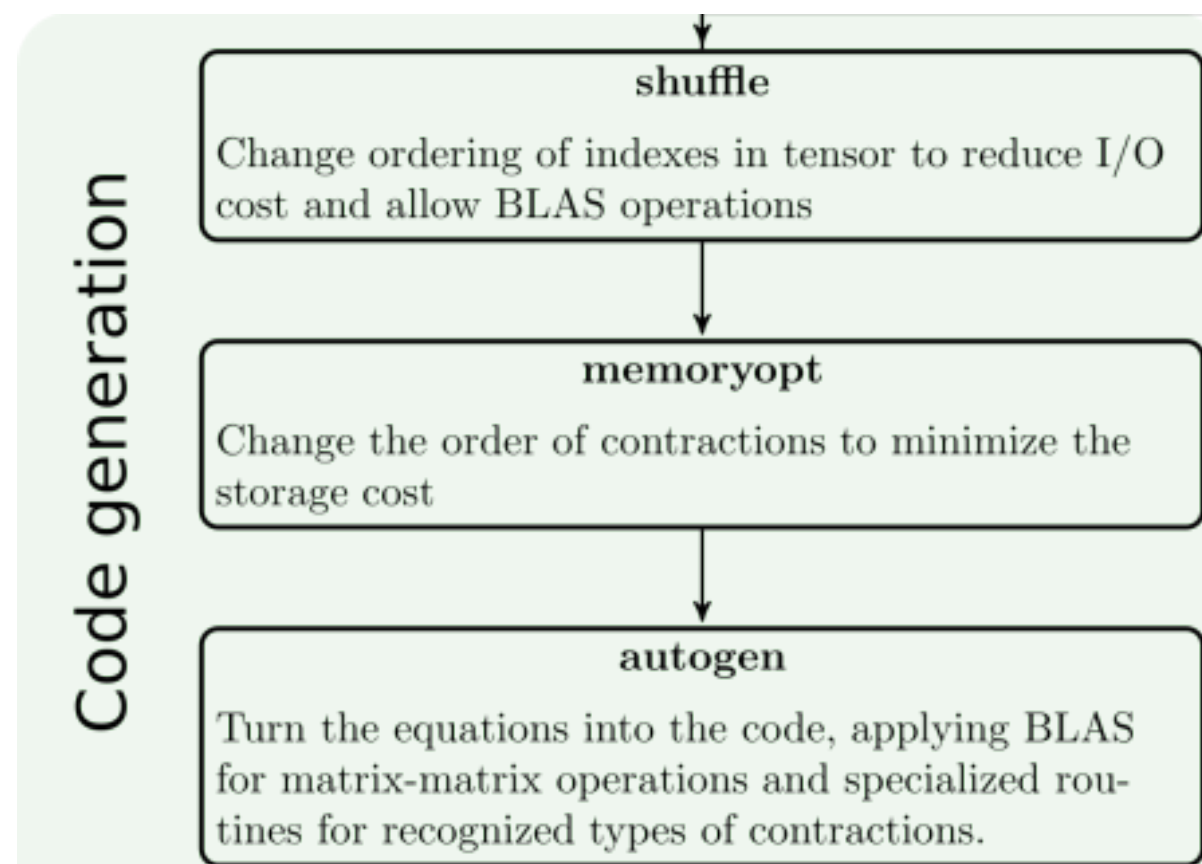
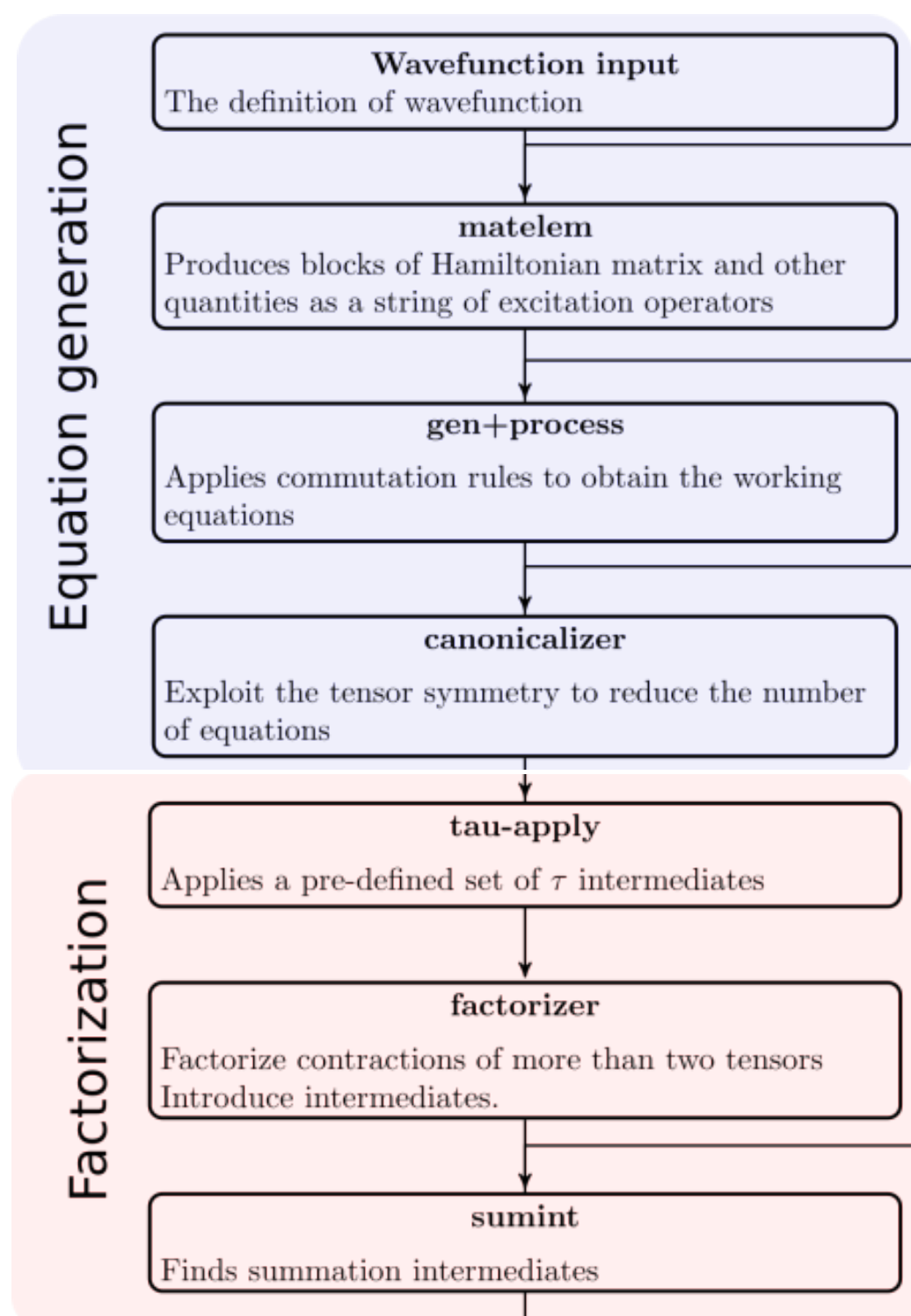
2. Factorizer

- ✓ 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

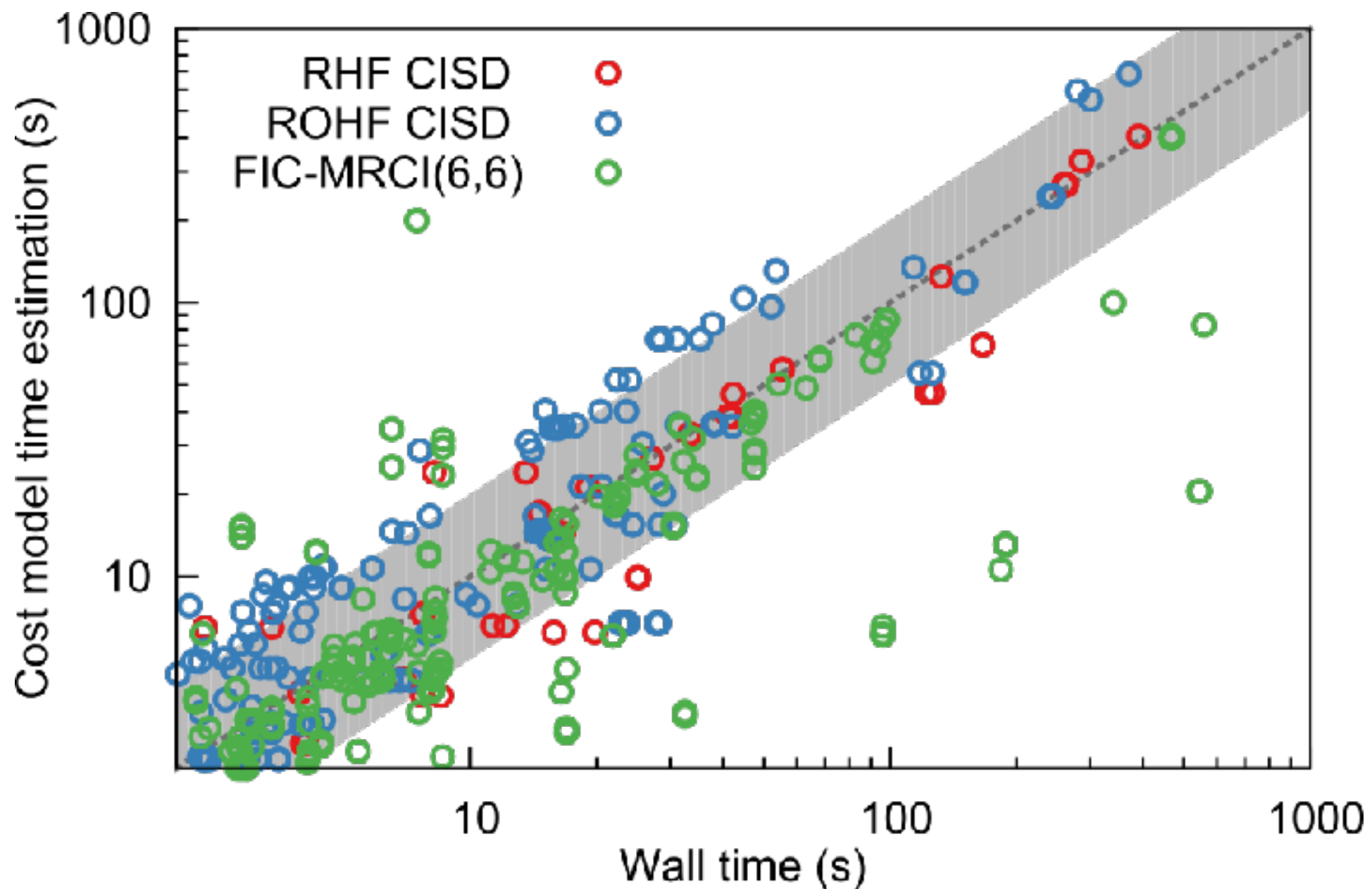
- ✓ Writes code for a specific electronic structure package
- ✓ Recognizes patterns/contractions for which highly optimized code exists
- ✓ Ensures that all terms have their correct formal scaling
- ✓ Ensures minimal I/O and maximal use of BLAS
- ✓ Generates parallel code, code for specific machines,

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.



Efficiency: Example

Naphtalene, CISD, no symmetry used

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

Hand code:

$$\sigma^{ij} \leftarrow \sum_k \left[\left(2\mathbf{C}^{ik} - (\mathbf{C}^{ik})^\dagger \right) \left(\mathbf{K}^{kj} - \frac{1}{2} \mathbf{J}^{kj} \right) - \frac{1}{2} \left((\mathbf{C}^{ik})^\dagger \mathbf{J}^{kj} \right) - \left((\mathbf{C}^{ik})^\dagger \mathbf{J}^{kj} \right)^\dagger + \right. \\ \left. \left(\mathbf{K}^{ik} - \frac{1}{2} \mathbf{J}^{ik} \right) \left(2\mathbf{C}^{kj} - (\mathbf{C}^{kj})^\dagger \right) - \frac{1}{2} \left(\mathbf{J}^{ik} (\mathbf{C}^{kj})^\dagger \right) - \left(\mathbf{J}^{ik} (\mathbf{C}^{kj})^\dagger \right)^\dagger \right]. \quad \text{4 dgemm/k}$$

Generated code:

$$\sigma^{ij} \leftarrow \sum_k -\mathbf{J}^{ik} \mathbf{C}^{kj} - \mathbf{C}^{kj} \mathbf{J}^{ik} - \mathbf{J}^{kj} \mathbf{C}^{ik} - \mathbf{C}^{ik} \mathbf{J}^{kj} - \mathbf{C}^{ki} \mathbf{K}^{kj} - \mathbf{K}^{ik} \mathbf{C}^{jk} + 2\mathbf{K}^{ik} \mathbf{C}^{jk} + 2\mathbf{C}^{ik} \mathbf{K}^{kj} \quad \text{8 dgemm/k}$$

Complexity: Example

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

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

- ✓ 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)

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.