Lazy matrices for apply-based algorithms

Michael F. Herbst* James Avery

*https://michael-herbst.com
michael.herbst@iwr.uni-heidelberg.de

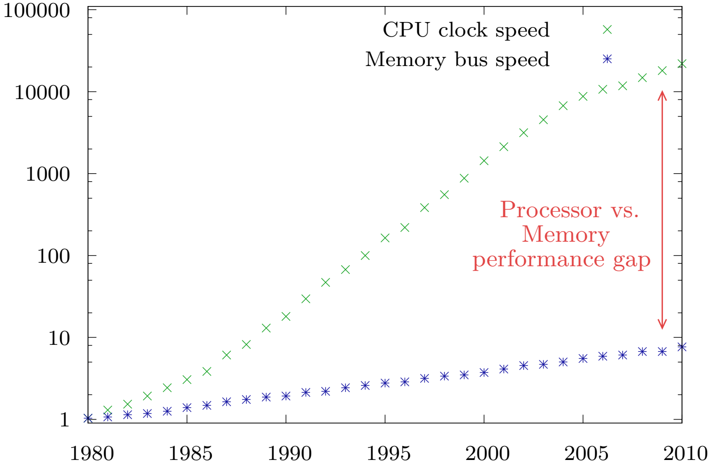
Interdisziplinäres Zentrum für wissenschaftliches Rechnen
Ruprecht-Karls-Universität Heidelberg

19th May 2017

Contents

Contents

Processor vs. memory performance improvement



Data from `https://dave.cheney.net/2014/06/07/five-things-that-make-go-fast`

Hartree-Fock equations

- Electronic structure theory
- Hartree-Fock equations

$$\left(-\frac{1}{2}\Delta + \hat{\mathcal{V}}_{\text{Nuc}} + \hat{\mathcal{V}}_{\text{2e}}\left[\,\{\psi_i\}_{i\in I}\,\right]\right)\psi_i = \varepsilon_i \psi_i$$

with

$-\frac{1}{2}\Delta$	Kinetic energy of electrons
$\hat{\mathcal{V}}_{\text{Nuc}}$	Electron-nuclear interaction
$\hat{\mathcal{V}}_{\text{2e}}\left[\,\{\psi_i\}_{i\in I}\,\right]$	Electron-electron interaction

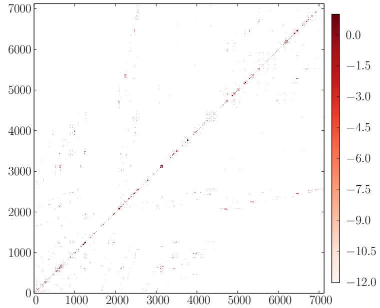
- Non-linear system of partial differential equations

Finite-element discretisation

- Finite elements: Piecewise polynomials with support only on a few neighbouring *cells*

$\Rightarrow$ Need many finite elements ($> 10^6$)

- Typically sparse matrix structures:

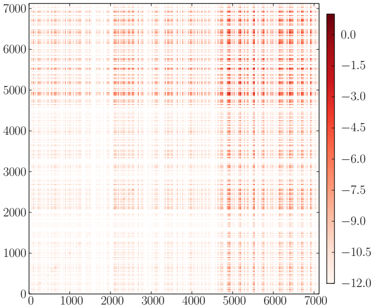


Typical discretisation of $-\frac{1}{2}\Delta + \hat{\mathcal{V}}_{\text{Nuc}}$

Finite-element discretisation

Caveat

- But … $\hat{\mathcal{V}}_{2e}$ is so-called *non-local*:



$\mathbf{V}_{2e}$: Discretisation of $\hat{\mathcal{V}}_{2e}$

- Building $\mathbf{V}_{2e}$ takes $\Omega(N^2)$ time and $\mathcal{O}(N^2)$ storage
- Typically $10^6 \cdot 10^6$ elements $\approx 8\,\text{TB}$ storage

Finite-element discretisation

Apply-based scheme

- Iterative solvers only need matrix-vector products
- Matrix-vector product of $\mathbf{V}_{2\mathrm{e}}$: Theoretically $\mathcal{O}(N)$

$\Rightarrow$ Apply-based or matrix-free algorithm:

- Never build $\mathbf{V}_{2\mathrm{e}}$ in storage
- Use expression for $\mathbf{V}_{2\mathrm{e}}$ to directly `apply` matrix to vectors

Characteristics of apply-based algorithms

Advantages

- Theoretical scaling (storage and time) reduced to $\mathcal{O}(N)$
- Parallelisation easier
  - $\Rightarrow$ Less data management
- Hardware trends are in favour

Disadvantages

- Matrices more intuitive than `apply`-functions
- More computations
  - $\Rightarrow$ Need efficient contraction schemes for the `apply`
  - Algorithms more complicated

Characteristics of apply-based algorithms

Advantages

- Theoretical scaling (storage and time) reduced to $\mathcal{O}(N)$
- Parallelisation easier
  - $\Rightarrow$ Less data management
- Hardware trends are in favour

Disadvantages

- Matrices more intuitive than `apply`-functions
- More computations
  - $\Rightarrow$ Need efficient contraction schemes for the `apply`
  - Algorithms more complicated

Characteristics of apply-based algorithms

Advantages

- Theoretical scaling (storage and time) reduced to $\mathcal{O}(N)$
- Parallelisation easier
  - $\Rightarrow$ Less data management
- Hardware trends are in favour

Disadvantages

- Matrices more intuitive than `apply`-functions
- More computations
  - $\Rightarrow$ Need efficient contraction schemes for the `apply`
  - Algorithms more complicated

Contents

Lazy matrices

- Stored matrix: All elements reside in memory
- Lazy matrix:
  - Generalisation of matrices
    - State
    - Non-linear
    - Elements may be expressions

  $\Rightarrow$ Obtaining elements expensive
  - Evaluation of internal expression: Delayed until `apply`
  - For convenience: Offer matrix-like interface

Using lazy matrices

- Program as usual

$$\mathbf{D} = \mathbf{A} + \mathbf{B}$$

- Build expression tree internally

$$\boxed{\mathbf{D}} = \boxed{\mathbf{A}} + \boxed{\mathbf{B}}$$

- On application:

$$\mathbf{D}\underline{x} = (\mathbf{A}\underline{x}) + (\mathbf{B}\underline{x})$$

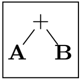
Using lazy matrices

- Program as usual

$$\mathbf{D} = \mathbf{A} + \mathbf{B}$$

- Build expression tree internally

$$\boxed{\mathbf{D}} = \boxed{\mathbf{A}} + \boxed{\mathbf{B}} =$$



- On application:

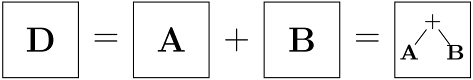
$$\mathbf{D}\underline{x} = (\mathbf{A}\underline{x}) + (\mathbf{B}\underline{x})$$

Using lazy matrices

- Program as usual

$$\mathbf{D} = \mathbf{A} + \mathbf{B}$$

- Build expression tree internally



- On application:

$$\mathbf{D}\underline{\boldsymbol{x}} = (\mathbf{A}\underline{\boldsymbol{x}}) + (\mathbf{B}\underline{\boldsymbol{x}})$$

Interface and example code

- `linalgwrap`[1]: Prototype `C++` implementation

```
typedef SmallVector<double> vector_type;
typedef SmallMatrix<double> matrix_type;
auto v = random<vector_type>(100);
DiagonalMatrix<matrix_type> diag(v);
auto mat = random<matrix_type>(100,100);

// No computation: Just build expression tree
auto sum = diag + mat;
auto prod = trans(sum) * diag * sum;
auto tree = mat + prod;

// Evaluate tree on application:
SmallVector<double> res = tree * v;
```

[1] https://linalgwrap.org

Notes and observations

- `linalgwrap`: Bookkeeping for `apply`-functions
- Programmer still sees matrices

⇒ Language for writing `apply`-based algorithms

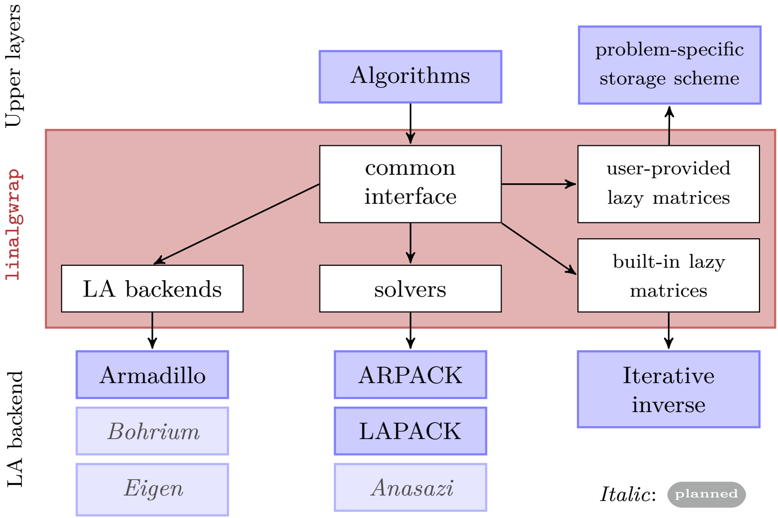
- Lazy matrices allow layered responsibility for computation, e.g. $(\mathbf{A}+\mathbf{B})\underline{\boldsymbol{x}}$
  - $\mathbf{A}\underline{\boldsymbol{x}}$ and $\mathbf{B}\underline{\boldsymbol{x}}$ decided by implementation of $\mathbf{A}$ and $\mathbf{B}$
  - $(\mathbf{A}\underline{\boldsymbol{x}})+(\mathbf{B}\underline{\boldsymbol{x}})$ done in linear algebra backend

⇒ Proper modularisation between

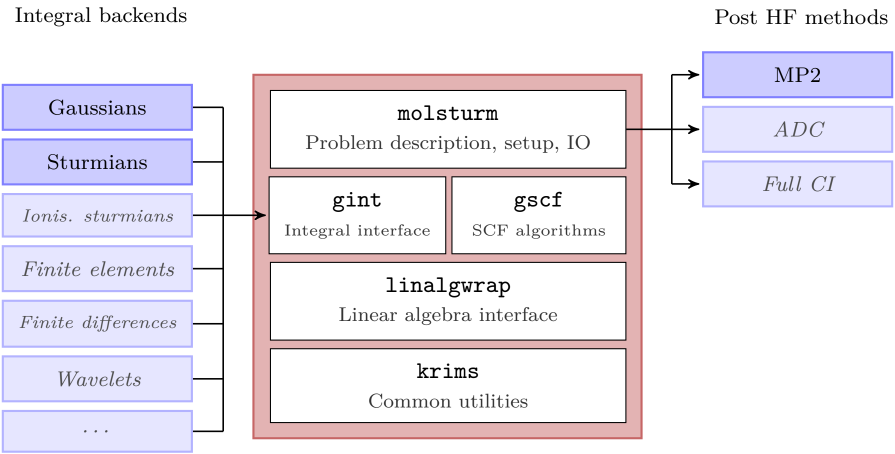
  - Higher-level algorithms
  - Lazy matrix implementations
  - LA backends

linalgwrap

Lazy linear algebra wrapper library



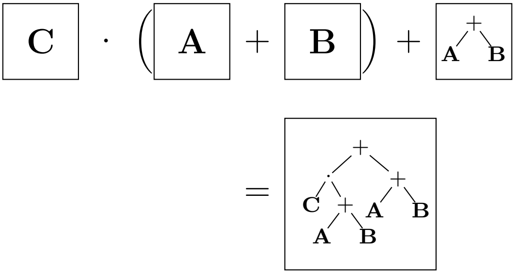
molsturm structure



- *Italic*: planned

Contents

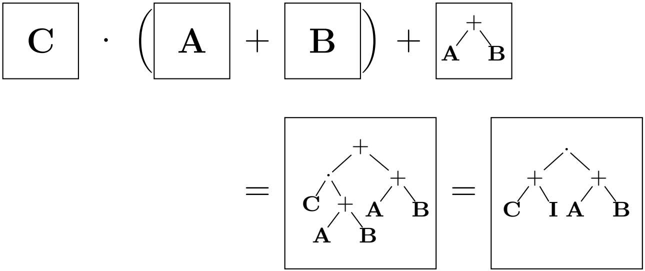
Lazy matrix expression optimisation



- Matrix expression tree $\equiv$ abstract syntax tree

$\Rightarrow$ May be optimised by standard methods

Lazy matrix expression optimisation



- Matrix expression tree $\equiv$ abstract syntax tree

$\Rightarrow$ May be optimised by standard methods

Extension to lazy tensors

- `apply` is a special tensor contraction

⇒ Lazy tensors:

- Delay all tensor contractions as long as possible

$$\text{e.g.} \qquad \tilde{k}_{bf} = \sum_{acdo\mu\nu} \mathcal{C}_{ao} \;\; c^{\mu}_{ab} I_{\mu\nu} c^{\nu}_{cd} \;\; \mathcal{C}_{co} \mathcal{C}_{df}$$

- Compare possible contraction schemes by complexity
- Execute cheapest evaluation scheme

⇒ Determine optimal contraction sequence automatically

Acknowledgements

- Dr. James Avery
- Prof. Andreas Dreuw and the Dreuw group

- Prof. Guido Kanschat
- HGS Mathcomp

Questions?

- EMail: `michael.herbst@iwr.uni-heidelberg.de`
- Website/blog: `https://michael-herbst.com`
- Projects: `https://linalgwrap.org` and `https://molsturm.org`