Efficient Algorithms for Tensor Contractions in Coupled Cluster

Edgar Solomonik

Department of Computer Science, ETH Zürich, Switzerland

13.2.2014

Max Planck Institute for Chemical Energy Conversion
Mülheim, Germany

Outline

1. Cyclops Tensor Framework
   - Aim
   - Interface
   - Internal Mechanism
   - Performance
   - Ongoing and future work
2. Symmetry preserving algorithm
   - Instances in matrix computations
   - General symmetric contractions
   - Application to coupled-cluster
3. Conclusion

Motivation and goals

Cyclops (**cycl**ic **op**eration**s**) Tensor Framework (CTF)

- aims to support distributed-memory tensor contractions

Motivation and goals

Cyclops (**cycl**ic **op**eration**s**) Tensor Framework (CTF)

- aims to support distributed-memory tensor contractions
- takes advantage of two-level parallelism via threading

Motivation and goals

Cyclops (**cycl**ic **op**eration**s**) Tensor Framework (CTF)

- aims to support distributed-memory tensor contractions
- takes advantage of two-level parallelism via threading
- leverages distributed and local matrix multiplication algorithms

Motivation and goals

Cyclops (**cycl**ic **op**eration**s**) Tensor Framework (CTF)

- aims to support distributed-memory tensor contractions
- takes advantage of two-level parallelism via threading
- leverages distributed and local matrix multiplication algorithms
- is packaged as a library and uses only MPI, BLAS, and OpenMP

Motivation and goals

Cyclops (**cycl**ic **op**eration**s**) Tensor Framework (CTF)

- aims to support distributed-memory tensor contractions
- takes advantage of two-level parallelism via threading
- leverages distributed and local matrix multiplication algorithms
- is packaged as a library and uses only MPI, BLAS, and OpenMP
- selects best mapping for tensors and contractions via performance models

Motivation and goals

Cyclops (**cycl**ic **op**eration**s**) Tensor Framework (CTF)

- aims to support distributed-memory tensor contractions
- takes advantage of two-level parallelism via threading
- leverages distributed and local matrix multiplication algorithms
- is packaged as a library and uses only MPI, BLAS, and OpenMP
- selects best mapping for tensors and contractions via performance models
- decomposes and redistributes tensor data dynamically

Distributed-memory context

CTF relies on MPI (Message Passing Interface) for bulk synchronous multiprocessor parallelism

```
CTF_World dw(comm)
```

- a set of processors in MPI corresponds to a communicator (MPI_Comm comm)

Distributed-memory context

CTF relies on MPI (Message Passing Interface) for bulk synchronous multiprocessor parallelism

```
CTF_World dw(comm)
```

- a set of processors in MPI corresponds to a communicator (MPI_Comm comm)
- MPI_COMM_WORLD is the default communicator containing all processes

Distributed-memory context

CTF relies on MPI (Message Passing Interface) for bulk synchronous multiprocessor parallelism

```
CTF_World dw(comm)
```

- a set of processors in MPI corresponds to a communicator (MPI_Comm comm)
- MPI_COMM_WORLD is the default communicator containing all processes
- data movement possible between a world and a ‘subworld’ (defined on a subcommunicator)

Tensor definition

A CTF tensor is a multidimensional distributed array, e.g.

$$T_{ij}^{ab}$$

where **T** is $m \times m \times n \times n$ antisymmetric in ab and in ij

```
CTF_Tensor T(4,\{m,m,n,n\},\{AS,NS,AS,NS\},dw)
```

- an 'AS' dimension is antisymmetric with the next

Tensor definition

A CTF tensor is a multidimensional distributed array, e.g.

$$T_{ij}^{ab}$$

where **T** is $m \times m \times n \times n$ antisymmetric in ab and in ij

```
CTF_Tensor T(4,\{m,m,n,n\},\{AS,NS,AS,NS\},dw)
```

- an ‘AS’ dimension is antisymmetric with the next
- symmetric ‘SY’ and symmetric-hollow ‘SH’ are also possible

Tensor definition

A CTF tensor is a multidimensional distributed array, e.g.

$$T_{ij}^{ab}$$

where **T** is $m \times m \times n \times n$ antisymmetric in *ab* and in *ij*

```
CTF_Tensor T(4,\{m,m,n,n\},\{AS,NS,AS,NS\},dw)
```

- an 'AS' dimension is antisymmetric with the next
- symmetric 'SY' and symmetric-hollow 'SH' are also possible
- tensors are allocated in packed form and set to zero when defined

Tensor definition

A CTF tensor is a multidimensional distributed array, e.g.

$$T_{ij}^{ab}$$

where **T** is $m \times m \times n \times n$ antisymmetric in *ab* and in *ij*

```
CTF_Tensor T(4,\{m,m,n,n\},\{AS,NS,AS,NS\},dw)
```

- an 'AS' dimension is antisymmetric with the next
- symmetric 'SY' and symmetric-hollow 'SH' are also possible
- tensors are allocated in packed form and set to zero when defined
- the first dimension of the tensor is mapped linearly onto memory

Tensor definition

A CTF tensor is a multidimensional distributed array, e.g.

$$T_{ij}^{ab}$$

where **T** is $m \times m \times n \times n$ antisymmetric in ab and in ij

```
CTF_Tensor T(4,\{m,m,n,n\},\{AS,NS,AS,NS\},dw)
```

- an 'AS' dimension is antisymmetric with the next
- symmetric 'SY' and symmetric-hollow 'SH' are also possible
- tensors are allocated in packed form and set to zero when defined
- the first dimension of the tensor is mapped linearly onto memory
- there are also obvious derived types for CTF_Tensor: CTF_Matrix, CTF_Vector, CTF_Scalar

Contract tensors

CTF can express a tensor contraction like

$$Z_{ij}^{ab} = Z_{ij}^{ab} + 2 \cdot P(a,b) \sum_k F_k^a \cdot T_{ij}^{kb}$$

where $P(a,b)$ implies antisymmetrization of index pair ab, as

```
Z["abij"] += 2.0*F["ak"]*T["kbij"]
```

- **for** loops and summations implicit in syntax

Contract tensors

CTF can express a tensor contraction like

$$Z_{ij}^{ab} = Z_{ij}^{ab} + 2 \cdot P(a, b) \sum_{k} F_{k}^{a} \cdot T_{ij}^{kb}$$

where $P(a, b)$ implies antisymmetrization of index pair ab, as

```
Z["abij"] += 2.0*F["ak"]*T["kbij"]
```

- **for** loops and summations implicit in syntax
- $P(a, b)$ is applied implicitly if **Z** is antisymmetric in ab

Contract tensors

CTF can express a tensor contraction like

$$Z_{ij}^{ab} = Z_{ij}^{ab} + 2 \cdot P(a,b) \sum_k F_k^a \cdot T_{ij}^{kb}$$

where $P(a,b)$ implies antisymmetrization of index pair ab, as

```
Z["abij"] += 2.0*F["ak"]*T["kbij"]
```

- **for** loops and summations implicit in syntax
- $P(a,b)$ is applied implicitly if **Z** is antisymmetric in ab
- **Z**, **F**, **T** should all be defined on the same world and all processes in the world must call the contraction bulk synchronously

CCSD

Extracted from Aquarius (Devin Matthews' code)

```
FMI["mi"] += 0.5*WMNEF["mnef"]*T(2)["efin"];
WMNIJ["mnij"] += 0.5*WMNEF["mnef"]*T(2)["efij"];
FAE["ae"] -= 0.5*WMNEF["mnef"]*T(2)["afmn"];
WAMEI["amei"] -= 0.5*WMNEF["mnef"]*T(2)["afin"];

Z(2)["abij"] = WMNEF["ijab"];
Z(2)["abij"] += FAE["af"]*T(2)["fbij"];
Z(2)["abij"] -= FMI["ni"]*T(2)["abnj"];
Z(2)["abij"] += 0.5*WABEF["abef"]*T(2)["efij"];
Z(2)["abij"] += 0.5*WMNIJ["mnij"]*T(2)["abmn"];
Z(2)["abij"] -= WAMEI["amei"]*T(2)["ebmj"];
```

CCSDT

Extracted from Aquarius (Devin Matthews' code)

```
Z(1)["ai"] += 0.25*WMNEF["mnef"]*T(3)["aefimn"];

Z(2)["abij"] += 0.5*WAMEF["bmef"]*T(3)["aefijm"];
Z(2)["abij"] -= 0.5*WMNEJ["mnej"]*T(3)["abeinm"];
Z(2)["abij"] += FME["me"]*T(3)["abeijm"];

Z(3)["abcijk"]  = WABEJ["bcek"]*T(2)["aeij"];
Z(3)["abcijk"] -= WAMIJ["bmjk"]*T(2)["acim"];
Z(3)["abcijk"] += FAE["ce"]*T(3)["abeijk"];
Z(3)["abcijk"] -= FMI["mk"]*T(3)["abcijm"];
Z(3)["abcijk"] += 0.5*WABEF["abef"]*T(3)["efcijk"];
Z(3)["abcijk"] += 0.5*WMNIJ["mnij"]*T(3)["abcmnk"];
Z(3)["abcijk"] -= WAMEI["amei"]*T(3)["ebcmjk"];
```

Access and write tensor data

CTF takes away the data pointer

- Access arbitrary sparse subsets of the tensor by global index (coordinate format)

Access and write tensor data

CTF takes away the data pointer

- Access arbitrary sparse subsets of the tensor by global index (coordinate format)
    - `T.write(int * indices, double * data)` (can also accumulate)

Access and write tensor data

CTF takes away the data pointer

- Access arbitrary sparse subsets of the tensor by global index (coordinate format)
  - T.write(int * indices, double * data) (can also accumulate)
  - T.read(int * indices, double * data) (can also accumulate)

Access and write tensor data

CTF takes away the data pointer

- Access arbitrary sparse subsets of the tensor by global index (coordinate format)
  - T.write(int * indices, double * data) (can also accumulate)
  - T.read(int * indices, double * data) (can also accumulate)
- Matlab submatrix notation: $A[j:k, l:m]$ (useful for CCSD(T) and CCSDT(Q))

Access and write tensor data

CTF takes away the data pointer

- Access arbitrary sparse subsets of the tensor by global index (coordinate format)
  - `T.write(int * indices, double * data)` (can also accumulate)
  - `T.read(int * indices, double * data)` (can also accumulate)
- Matlab submatrix notation: $A[j:k,l:m]$ (useful for CCSD(T) and CCSDT(Q))
  - `T.slice(int * offsets, int * ends)` returns the subtensor

Access and write tensor data

CTF takes away the data pointer

- Access arbitrary sparse subsets of the tensor by global index (coordinate format)
  - `T.write(int * indices, double * data)` (can also accumulate)
  - `T.read(int * indices, double * data)` (can also accumulate)
- Matlab submatrix notation: $A[j:k,l:m]$ (useful for CCSD(T) and CCSDT(Q))
  - `T.slice(int * offsets, int * ends)` returns the subtensor
  - `T.slice(int corner_off, int corner_end)` does the same

Access and write tensor data

CTF takes away the data pointer

- Access arbitrary sparse subsets of the tensor by global index (coordinate format)
  - `T.write(int * indices, double * data)` (can also accumulate)
  - `T.read(int * indices, double * data)` (can also accumulate)
- Matlab submatrix notation: $A[j:k, l:m]$ (useful for CCSD(T) and CCSDT(Q))
  - `T.slice(int * offsets, int * ends)` returns the subtensor
  - `T.slice(int corner_off, int corner_end)` does the same
  - can also sum subtensors

Access and write tensor data

CTF takes away the data pointer

- Access arbitrary sparse subsets of the tensor by global index (coordinate format)
  - T.write(int * indices, double * data) (can also accumulate)
  - T.read(int * indices, double * data) (can also accumulate)
- Matlab submatrix notation: $A[j : k, l : m]$ (useful for CCSD(T) and CCSDT(Q))
  - T.slice(int * offsets, int * ends) returns the subtensor
  - T.slice(int corner_off, int corner_end) does the same
  - can also sum subtensors
  - different subworlds can read different subtensors simultaneously

Access and write tensor data

CTF takes away the data pointer

- Access arbitrary sparse subsets of the tensor by global index (coordinate format)
  - `T.write(int * indices, double * data)` (can also accumulate)
  - `T.read(int * indices, double * data)` (can also accumulate)
- Matlab submatrix notation: $A[j : k, l : m]$ (useful for CCSD(T) and CCSDT(Q))
  - `T.slice(int * offsets, int * ends)` returns the subtensor
  - `T.slice(int corner_off, int corner_end)` does the same
  - can also sum subtensors
  - different subworlds can read different subtensors simultaneously
- Extract a subtensor of any permutation of the tensor

Access and write tensor data

CTF takes away the data pointer

- Access arbitrary sparse subsets of the tensor by global index (coordinate format)
  - T.write(int * indices, double * data) (can also accumulate)
  - T.read(int * indices, double * data) (can also accumulate)
- Matlab submatrix notation: $A[j : k, l : m]$ (useful for CCSD(T) and CCSDT(Q))
  - T.slice(int * offsets, int * ends) returns the subtensor
  - T.slice(int corner_off, int corner_end) does the same
  - can also sum subtensors
  - different subworlds can read different subtensors simultaneously
- Extract a subtensor of any permutation of the tensor
  - given mappings P, Q, does $B[i, j] = A[P[i], Q[j]]$ via permute()

Access and write tensor data

CTF takes away the data pointer

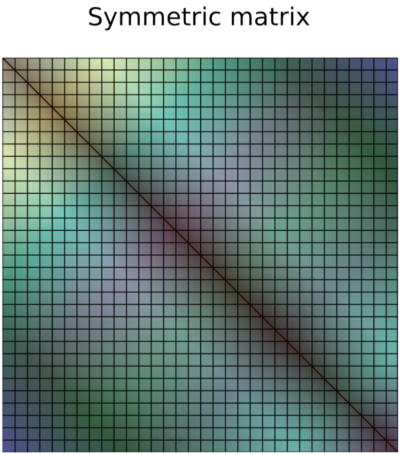
- Access arbitrary sparse subsets of the tensor by global index (coordinate format)
  - `T.write(int * indices, double * data)` (can also accumulate)
  - `T.read(int * indices, double * data)` (can also accumulate)
- Matlab submatrix notation: $A[j:k, l:m]$ (useful for CCSD(T) and CCSDT(Q))
  - `T.slice(int * offsets, int * ends)` returns the subtensor
  - `T.slice(int corner_off, int corner_end)` does the same
  - can also sum subtensors
  - different subworlds can read different subtensors simultaneously
- Extract a subtensor of any permutation of the tensor
  - given mappings P, Q, does $B[i,j] = A[P[i], Q[j]]$ via permute()
  - P and Q may access only subsets of $\mathbf{A}$ (if $\mathbf{B}$ is smaller)

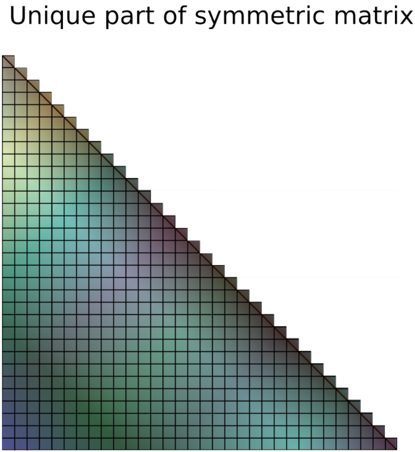
Access and write tensor data

CTF takes away the data pointer

- Access arbitrary sparse subsets of the tensor by global index (coordinate format)
  - T.write(int * indices, double * data) (can also accumulate)
  - T.read(int * indices, double * data) (can also accumulate)
- Matlab submatrix notation: $A[j:k, l:m]$ (useful for CCSD(T) and CCSDT(Q))
  - T.slice(int * offsets, int * ends) returns the subtensor
  - T.slice(int corner_off, int corner_end) does the same
  - can also sum subtensors
  - different subworlds can read different subtensors simultaneously
- Extract a subtensor of any permutation of the tensor
  - given mappings P, Q, does $B[i,j] = A[P[i], Q[j]]$ via permute()
  - P and Q may access only subsets of **A** (if **B** is smaller)
  - **B** may be defined on subworlds on the world on which **A** is defined and each subworld may specify different P and Q

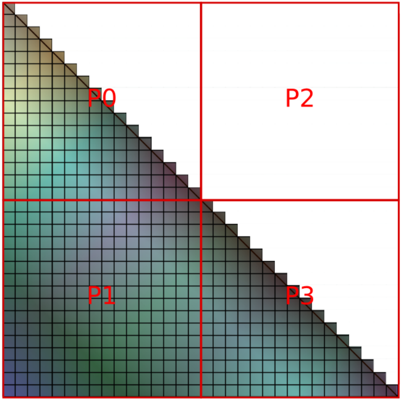
Symmetric matrix representation



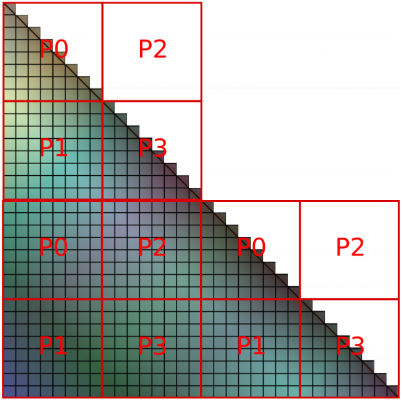


Blocked distributions of a symmetric matrix

Naive blocked layout

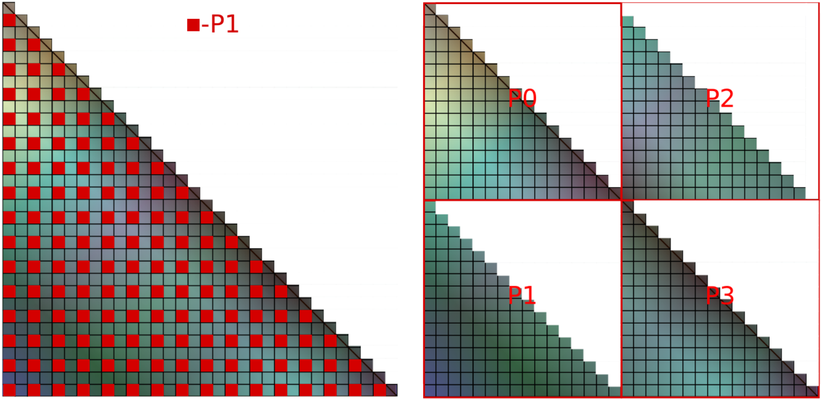


Block-cyclic layout



Cyclic distribution of a symmetric matrix

Cyclic layout ∼ Improved blocked layout



Tensor decomposition and mapping

CTF tensor decomposition

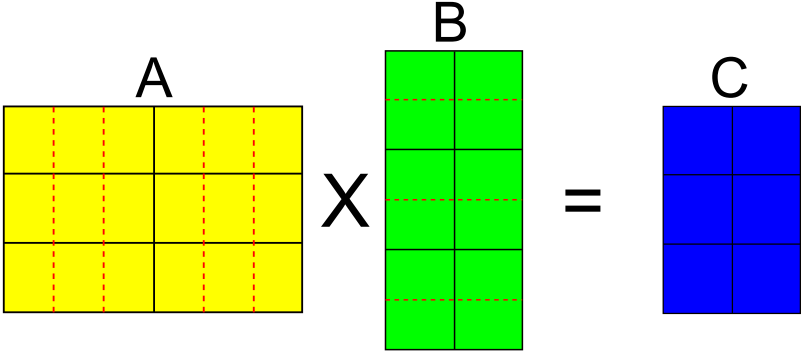
- cyclic layout used to preserve packed symmetric structure (hence Cyclops – cyclic ops)
- overdecomposition (virtualization) employed to decouple the decomposition from the physical processor grid

CTF mapping logic

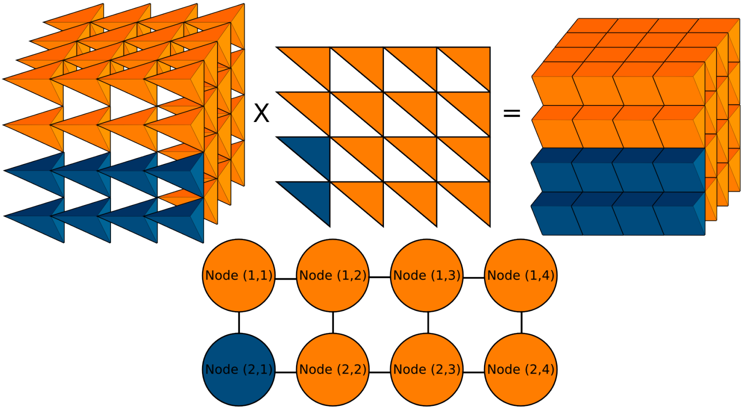
- arrange physical topology into all possible processor grids
- dynamically (in parallel) autotune over all topologies and over mapping strategies
- select best mapping based on model-based performance prediction

Virtualization (local blocking)

Matrix multiply on 2x3 processor grid. Red lines represent virtualized part of processor grid. Elements assigned to blocks by cyclic phase.



3D tensor mapping



Algorithms for tensor redistribution

The following three redistribution kernels are provided by CTF

- Sparse (key-value) redistribution

Algorithms for tensor redistribution

The following three redistribution kernels are provided by CTF

- Sparse (key-value) redistribution
  - requires all-to-all-v communication and expensive local binning work

Algorithms for tensor redistribution

The following three redistribution kernels are provided by CTF

- Sparse (key-value) redistribution
  - requires all-to-all-v communication and expensive local binning work
  - well-fit for user-level data entry and generalizable to sparse tensors

Algorithms for tensor redistribution

The following three redistribution kernels are provided by CTF

- Sparse (key-value) redistribution
  - requires all-to-all-v communication and expensive local binning work
  - well-fit for user-level data entry and generalizable to sparse tensors
- Dense mapping-to-mapping (no-explicit-key) redistribution

Algorithms for tensor redistribution

The following three redistribution kernels are provided by CTF

- Sparse (key-value) redistribution
  - requires all-to-all-v communication and expensive local binning work
  - well-fit for user-level data entry and generalizable to sparse tensors
- Dense mapping-to-mapping (no-explicit-key) redistribution
  - iterates over local data in global order, packs into send buffers, performs all-to-all-v

Algorithms for tensor redistribution

The following three redistribution kernels are provided by CTF

- Sparse (key-value) redistribution
  - requires all-to-all-v communication and expensive local binning work
  - well-fit for user-level data entry and generalizable to sparse tensors
- Dense mapping-to-mapping (no-explicit-key) redistribution
  - iterates over local data in global order, packs into send buffers, performs all-to-all-v
  - aggressively threaded and employs look-up arrays

Algorithms for tensor redistribution

The following three redistribution kernels are provided by CTF

- Sparse (key-value) redistribution
  - requires all-to-all-v communication and expensive local binning work
  - well-fit for user-level data entry and generalizable to sparse tensors
- Dense mapping-to-mapping (no-explicit-key) redistribution
  - iterates over local data in global order, packs into send buffers, performs all-to-all-v
  - aggressively threaded and employs look-up arrays
  - well-fit for redistribution between two arbitrary mappings

Algorithms for tensor redistribution

The following three redistribution kernels are provided by CTF

- Sparse (key-value) redistribution
  - requires all-to-all-v communication and expensive local binning work
  - well-fit for user-level data entry and generalizable to sparse tensors
- Dense mapping-to-mapping (no-explicit-key) redistribution
  - iterates over local data in global order, packs into send buffers, performs all-to-all-v
  - aggressively threaded and employs look-up arrays
  - well-fit for redistribution between two arbitrary mappings
- Block-to-block redistribution

Algorithms for tensor redistribution

The following three redistribution kernels are provided by CTF

- Sparse (key-value) redistribution
  - requires all-to-all-v communication and expensive local binning work
  - well-fit for user-level data entry and generalizable to sparse tensors
- Dense mapping-to-mapping (no-explicit-key) redistribution
  - iterates over local data in global order, packs into send buffers, performs all-to-all-v
  - aggressively threaded and employs look-up arrays
  - well-fit for redistribution between two arbitrary mappings
- Block-to-block redistribution
  - possible to use when the block decomposition does not change but only the processor grid does

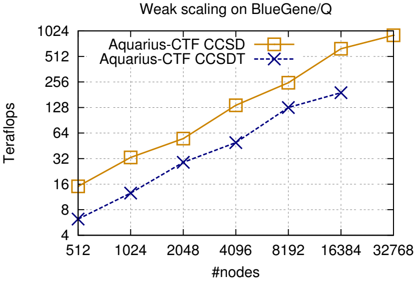
Algorithms for tensor redistribution

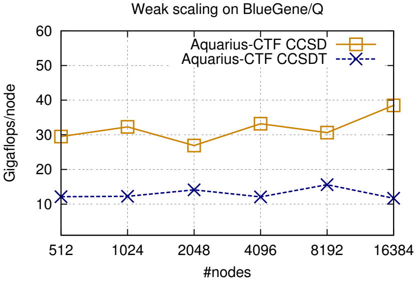
The following three redistribution kernels are provided by CTF

- Sparse (key-value) redistribution
  - requires all-to-all-v communication and expensive local binning work
  - well-fit for user-level data entry and generalizable to sparse tensors
- Dense mapping-to-mapping (no-explicit-key) redistribution
  - iterates over local data in global order, packs into send buffers, performs all-to-all-v
  - aggressively threaded and employs look-up arrays
  - well-fit for redistribution between two arbitrary mappings
- Block-to-block redistribution
  - possible to use when the block decomposition does not change but only the processor grid does
  - processors send blocks via point-to-point messages

Coupled-cluster code on BlueGene/Q (Mira)

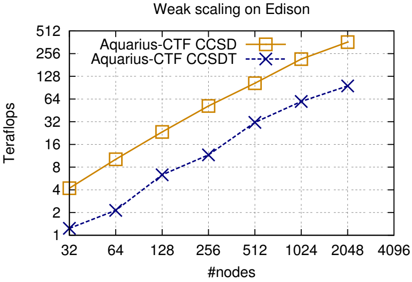
CCSD up to 55 water molecules with cc-pVDZ
CCSDT up to 10 water molecules with cc-pVDZ

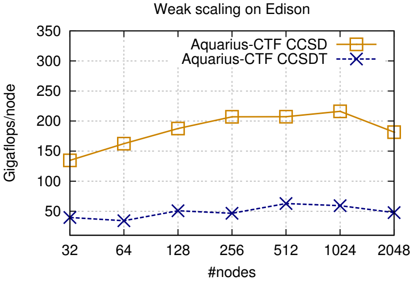




Coupled-cluster code on Cray XC30 (Edison)

CCSD up to 50 water molecules with cc-pVDZ
CCSDT up to 10 water molecules with cc-pVDZ

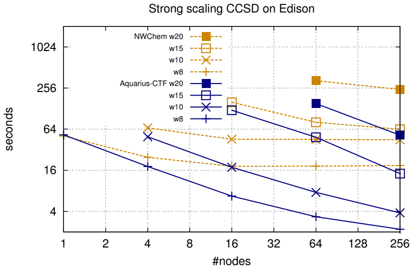


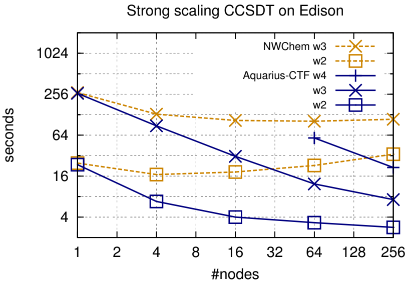


Comparison with NWChem

NWChem is a commonly-used distributed-memory quantum chemistry method suite

- provides CCSD and CCSDT
- uses Global Arrays a Partitioned Global Address Space (PGAS) for tensor data partitioning
- derives equations via Tensor Contraction Engine (TCE)





Ongoing work: arbitrary typed tensors and functions

- CTF v1.x is fully templated and instantiated to double and complex<double>

Ongoing work: arbitrary typed tensors and functions

- CTF v1.x is fully templated and instantiated to double and complex<double>
- CTF v2.x will have a light-weight templated layer but be type-oblivious internally

Ongoing work: arbitrary typed tensors and functions

- CTF v1.x is fully templated and instantiated to double and complex<double>
- CTF v2.x will have a light-weight templated layer but be type-oblivious internally
- A tensor contains elements from any set/monoid/group/semiring/ring

Ongoing work: arbitrary typed tensors and functions

- CTF v1.x is fully templated and instantiated to double and complex<double>
- CTF v2.x will have a light-weight templated layer but be type-oblivious internally
- A tensor contains elements from any set/monoid/group/semiring/ring
- Tensor functions with parameters/output of different type will now be possible

Ongoing work: arbitrary typed tensors and functions

- CTF v1.x is fully templated and instantiated to double and complex<double>
- CTF v2.x will have a light-weight templated layer but be type-oblivious internally
- A tensor contains elements from any set/monoid/group/semiring/ring
- Tensor functions with parameters/output of different type will now be possible
    - makes mixed-precision operations possible

Ongoing work: arbitrary typed tensors and functions

- CTF v1.x is fully templated and instantiated to double and complex<double>
- CTF v2.x will have a light-weight templated layer but be type-oblivious internally
- A tensor contains elements from any set/monoid/group/semiring/ring
- Tensor functions with parameters/output of different type will now be possible
  - makes mixed-precision operations possible
  - enables graph algorithms on the (min,+) semiring

Ongoing work: arbitrary typed tensors and functions

- CTF v1.x is fully templated and instantiated to double and complex<double>
- CTF v2.x will have a light-weight templated layer but be type-oblivious internally
- A tensor contains elements from any set/monoid/group/semiring/ring
- Tensor functions with parameters/output of different type will now be possible
    - makes mixed-precision operations possible
    - enables graph algorithms on the (min,+) semiring
    - more exotic use-cases possible such as tensors of particles

Future work for CTF

- (Aquarius) CCSD(T), CCSDT(Q), CCSDTQ

Future work for CTF

- (Aquarius) CCSD(T), CCSDT(Q), CCSDTQ
- time-accurate performance models

Future work for CTF

- (Aquarius) CCSD(T), CCSDT(Q), CCSDTQ
- time-accurate performance models
- simultaneous multi-contraction scheduling

Future work for CTF

- (Aquarius) CCSD(T), CCSDT(Q), CCSDTQ
- time-accurate performance models
- simultaneous multi-contraction scheduling
- sparse tensors and contractions

Future work for CTF

- (Aquarius) CCSD(T), CCSDT(Q), CCSDTQ
- time-accurate performance models
- simultaneous multi-contraction scheduling
- sparse tensors and contractions
- faster algorithms for symmetric contractions (theory in next part of this talk)

Symmetric-matrix–vector multiplication

- Consider symmetric $n \times n$ matrix $\mathbf{A}$ and vectors $\mathbf{b}, \mathbf{c}$

Symmetric-matrix–vector multiplication

- Consider symmetric $n \times n$ matrix $\mathbf{A}$ and vectors $\mathbf{b}, \mathbf{c}$
- $\mathbf{c} = \mathbf{A} \cdot \mathbf{b}$ is usually computed by forming a *nonsymmetric* intermediate matrix $\mathbf{W}$,

$$W_{ij} = A_{ij} \cdot b_j \qquad\qquad c_i = \sum_{j=1}^{n} W_{ij}$$

which requires n^2 multiplications and n^2 additions

Symmetric-matrix–vector multiplication

- Consider symmetric $n \times n$ matrix $\mathbf{A}$ and vectors $\mathbf{b}, \mathbf{c}$
- $\mathbf{c} = \mathbf{A} \cdot \mathbf{b}$ is usually computed by forming a *nonsymmetric* intermediate matrix $\mathbf{W}$,

$$W_{ij} = A_{ij} \cdot b_j \qquad\qquad c_i = \sum_{j=1}^{n} W_{ij}$$

which requires n^2 multiplications and n^2 additions

- The *symmetry preserving algorithm* employs a *symmetric* intermediate matrix $\mathbf{Z}$,

$$Z_{ij} = A_{ij} \cdot (b_i + b_j) \qquad\qquad c_i = \sum_{j=1}^{n} Z_{ij} - \left(\sum_{j=1}^{n} A_{ij}\right) \cdot b_i$$

which requires $\frac{n^2}{2}$ multiplications and $\frac{5n^2}{2}$ additions

Symmetrized rank-two outer product

- Consider vectors $\mathbf{a}, \mathbf{b}$ of dimension n

Symmetrized rank-two outer product

- Consider vectors $\mathbf{a}, \mathbf{b}$ of dimension n
- Symmetric matrix $\mathbf{C} = \mathbf{a} \cdot \mathbf{b}^\mathsf{T} + \mathbf{b} \cdot \mathbf{a}^\mathsf{T}$ is usually computed by forming a *nonsymmetric* intermediate matrix $\mathbf{W}$,

$$W_{ij} = a_i \cdot b_j \qquad\qquad C_{ij} = W_{ij} + W_{ji}$$

which requires n^2 multiplications and $n^2/2$ additions

Symmetrized rank-two outer product

- Consider vectors $\mathbf{a}, \mathbf{b}$ of dimension n
- Symmetric matrix $\mathbf{C} = \mathbf{a} \cdot \mathbf{b}^\mathsf{T} + \mathbf{b} \cdot \mathbf{a}^\mathsf{T}$ is usually computed by forming a *nonsymmetric* intermediate matrix $\mathbf{W}$,

$$W_{ij} = a_i \cdot b_j \qquad\qquad C_{ij} = W_{ij} + W_{ji}$$

  which requires n^2 multiplications and $n^2/2$ additions
- The *symmetry preserving algorithm* employs a *symmetric* intermediate matrix $\mathbf{Z}$,

$$Z_{ij} = (a_i + a_j) \cdot (b_i + b_j) \qquad C_{ij} = Z_{ij} - a_i \cdot b_i - a_j \cdot b_j$$

  which requires $\frac{n^2}{2}$ multiplications and $2n^2$ additions

Symmetrized matrix multiplication

- Consider symmetric $n \times n$ matrices **A**, **B**, and **C**

Symmetrized matrix multiplication

- Consider symmetric $n \times n$ matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$
- $\mathbf{C} = \mathbf{A} \cdot \mathbf{B} + \mathbf{B} \cdot \mathbf{A}$ is usually computed via a nonsymmetric intermediate order 3 tensor $\mathbf{W}$,

$$W_{ijk} = A_{ik} \cdot B_{kj} \qquad \bar{W}_{ij} = \sum_k W_{ijk} \qquad C_{ij} = W_{ij} + W_{ji}.$$

which requires n^3 multiplications and n^3 additions.

Symmetrized matrix multiplication

- Consider symmetric $n \times n$ matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$
- $\mathbf{C} = \mathbf{A} \cdot \mathbf{B} + \mathbf{B} \cdot \mathbf{A}$ is usually computed via a nonsymmetric intermediate order 3 tensor $\mathbf{W}$,

$$W_{ijk} = A_{ik} \cdot B_{kj} \qquad \bar{W}_{ij} = \sum_k W_{ijk} \qquad C_{ij} = W_{ij} + W_{ji}.$$

  which requires n^3 multiplications and n^3 additions.
- The *symmetry preserving algorithm* employs a *symmetric* intermediate tensor $\mathbf{Z}$ using $n^3/6$ multiplications and $7n^3/6$ additions,

$$Z_{ijk} = (A_{ij} + A_{ik} + A_{jk}) \cdot (B_{ij} + B_{ik} + B_{jk}) \qquad v_i = \sum_{k=1}^{n} A_{ik} \cdot B_{ik}$$

$$C_{ij} = \sum_{k=1}^{n} Z_{ijk} - n \cdot A_{ij} \cdot B_{ij} - v_i - v_j - \left(\sum_{k=1}^{n} A_{ik}\right) \cdot B_{ij} - A_{ij} \cdot \left(\sum_{k=1}^{n} B_{ik}\right)$$

Symmetry preserving algorithm generalization

- Any fully symmetrized contraction of two fully symmetric tensors with a total of ω indices can be done with $n^\omega/\omega! + O(n^{\omega-1})$ multiplications

Symmetry preserving algorithm generalization

- Any fully symmetrized contraction of two fully symmetric tensors with a total of ω indices can be done with $n^\omega/\omega! + O(n^{\omega-1})$ multiplications
- Extensions to antisymmetric tensors and antisymmetrized contractions possible, but not for all cases

Symmetry preserving algorithm generalization

- Any fully symmetrized contraction of two fully symmetric tensors with a total of ω indices can be done with $n^\omega/\omega! + O(n^{\omega-1})$ multiplications
- Extensions to antisymmetric tensors and antisymmetrized contractions possible, but not for all cases
- Extends to all complex/Hermitian cases

Symmetry preserving algorithm generalization

- Any fully symmetrized contraction of two fully symmetric tensors with a total of ω indices can be done with $n^\omega/\omega! + O(n^{\omega-1})$ multiplications
- Extensions to antisymmetric tensors and antisymmetrized contractions possible, but not for all cases
- Extends to all complex/Hermitian cases
- Also applicable to contractions of a tensor with itself, in particular $\mathbf{A}^2$ for symmetric or antisymmetric matrix $\mathbf{A}$ requires $n^3/6$ multiplications

Symmetry preserving algorithm generalization

- Any fully symmetrized contraction of two fully symmetric tensors with a total of ω indices can be done with $n^\omega/\omega! + O(n^{\omega-1})$ multiplications
- Extensions to antisymmetric tensors and antisymmetrized contractions possible, but not for all cases
- Extends to all complex/Hermitian cases
- Also applicable to contractions of a tensor with itself, in particular $\mathbf{A}^2$ for symmetric or antisymmetric matrix $\mathbf{A}$ requires $n^3/6$ multiplications
- Nonsymmetric $\mathbf{A}^2$ (or more generally $\mathbf{A} \cdot \mathbf{B} + \mathbf{B} \cdot \mathbf{A}$ for nonsymmetric matrices $\mathbf{A}$, $\mathbf{B}$) can be done in $2n^3/3$ operations

Symmetry preserving algorithm generalization

- Any fully symmetrized contraction of two fully symmetric tensors with a total of ω indices can be done with $n^\omega/\omega! + O(n^{\omega-1})$ multiplications
- Extensions to antisymmetric tensors and antisymmetrized contractions possible, but not for all cases
- Extends to all complex/Hermitian cases
- Also applicable to contractions of a tensor with itself, in particular $\mathbf{A}^2$ for symmetric or antisymmetric matrix $\mathbf{A}$ requires $n^3/6$ multiplications
- Nonsymmetric $\mathbf{A}^2$ (or more generally $\mathbf{A}\cdot\mathbf{B} + \mathbf{B}\cdot\mathbf{A}$ for nonsymmetric matrices $\mathbf{A}$, $\mathbf{B}$) can be done in $2n^3/3$ operations
- Numerical stability confirmed via proof and experiments

Application to CCSD

The CCSD contraction

$$Z^{a\bar{k}}_{i\bar{c}} = \sum_{b}\sum_{j} T^{ab}_{ij} \cdot V^{j\bar{k}}_{b\bar{c}}$$

usually requires $2n^6$ total operations.
The symmetry-preserving algorithm can be applied over the indices

$$\mathbf{Z^a} = \sum_{b} \mathbf{T^{ab}} \cdot \mathbf{V_b}$$

with each multiplication being a contraction over the other four indices $i,j,\bar{c},\bar{k}$, which is more expensive than the addition operations, yielding n^6 operations to leading order.

Application to CCSD(T) and CCSDT(Q)

The CCSD(T) contraction

$$T^{ab\bar{c}}_{ij\bar{k}} = P(a,b)P(i,j)\sum_{\bar{l}=1}^{n} T^{a\bar{c}}_{i\bar{l}} \cdot W^{\bar{l}b}_{j\bar{k}}$$

usually requires $2n^7$ total operations.
The symmetry-preserving algorithm can be applied over the indices

$$\mathbf{T^{ab}} = P(a,b)\mathbf{T^{a}} \cdot \mathbf{W^{b}} \qquad \text{and} \qquad \mathbf{T_{ij}} = P(i,j)\mathbf{T_{i}} \cdot \mathbf{T_{j}}$$

with each multiplication in the latter being a contraction over the remaining three indices $\bar{c}$,$\bar{k}$, and $\bar{l}$, for a total of $n^7/2$ leading order operations.
For a similar CCSDT(Q) contraction, which usually requires $n^9/2$ operations, the symmetry preserving algorithm achieves $n^9/36$.

Conclusion

Future work on symmetry-preserving algorithms

- Full cost derivations for CC methods
- High performance implementation and integration into CTF

References

- CTF (latest): E.S., D. Matthews, J.R. Hammond, J.F. Stanton, J. Demmel, "A massively parallel tensor contraction framework for coupled-cluster", JPDC, 2015. computations
- symmetry preserving algorithms: E.S., J. Demmel, "Contracting symmetric tensors using fewer multiplications", ETH Report, 2015.
- communication cost of symmetry preserving algorithms: E.S., J. Demmel, T. Hoefler, "Communication lower bounds for tensor contraction algorithms", ETH Report, 2015.

Backup slides