Efficiency and Accuracy of Parallel Accumulator-based $\mathcal{H}$-Arithmetic

Steffen Börm
University of Kiel

Ronald Kriemann
Max Planck Inst. for Math. i.t.S.

SIAM ALA18

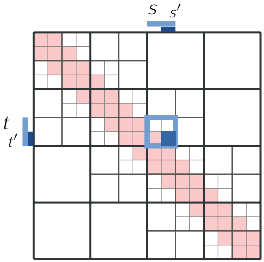
$\mathcal{H}$-Arithmetic with Accumulators

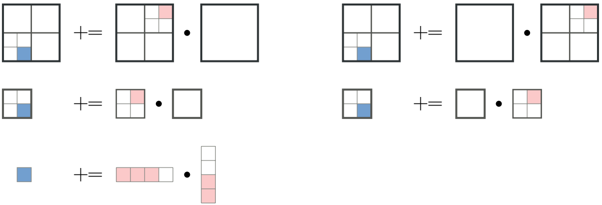
Motivating Example

Let A, B and C be $\mathcal{H}$-matrices with the shown structure.

For the multiplication $C := A \cdot B$ several updates from different levels of the $\mathcal{H}$-hierarchy are applied to a single block.

As an example, the updates for $C_{t',s'}$ are:

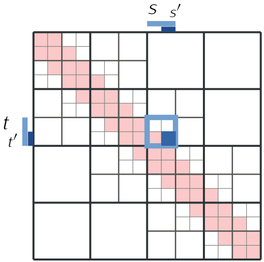




Similar updates are computed for all other sub blocks of the parent block $C_{t,s}$.

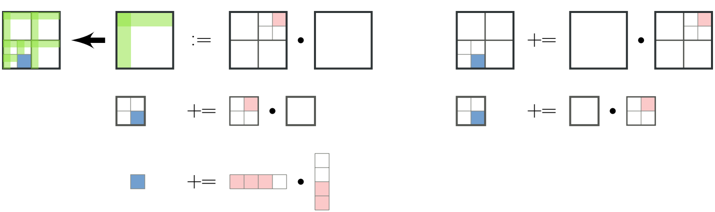
Motivating Example

Let A, B and C be $\mathcal{H}$-matrices with the shown structure.



For the multiplication $C := A \cdot B$ several updates from different levels of the $\mathcal{H}$-hierarchy are applied to a single block.

As an example, the updates for $C_{t',s'}$ are:

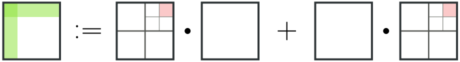


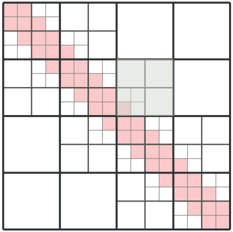
Similar updates are computed for all other sub blocks of the parent block $C_{t,s}$.

In a classical implementation, all sub multiplications sum up to 24 truncations for the 3 low-rank blocks in $C_{t,s}$.

Motivating Example

Instead, updates are first collected for each destination block and afterwards shifted down following the hierarchy.[1]

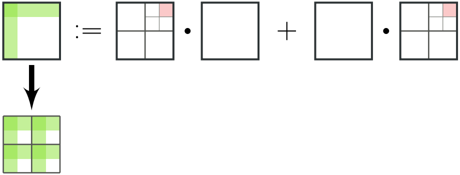


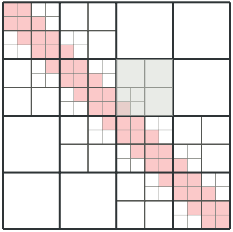


[1] S. Börm, *"Hierarchical matrix arithmetic with accumulated updates"*, submitted to Computing and Visualization in Science, 2017.

Motivating Example

Instead, updates are first collected for each destination block and afterwards shifted down following the hierarchy.[1]

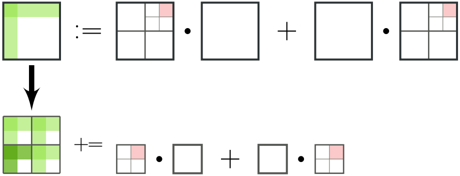


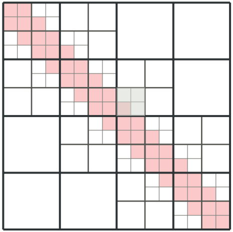


[1] S. Börm, *"Hierarchical matrix arithmetic with accumulated updates"*, submitted to Computing and Visualization in Science, 2017.

Motivating Example

Instead, updates are first collected for each destination block and afterwards shifted down following the hierarchy.[1]

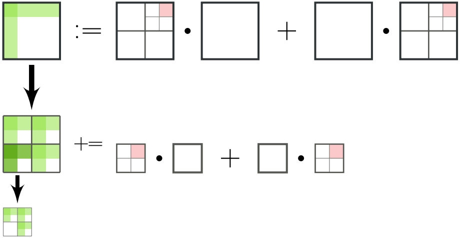


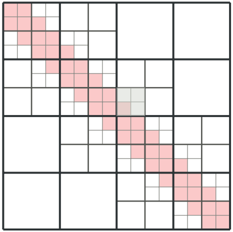


[1] S. Börm, *"Hierarchical matrix arithmetic with accumulated updates"*, submitted to Computing and Visualization in Science, 2017.

Motivating Example

Instead, updates are first collected for each destination block and afterwards shifted down following the hierarchy.[1]

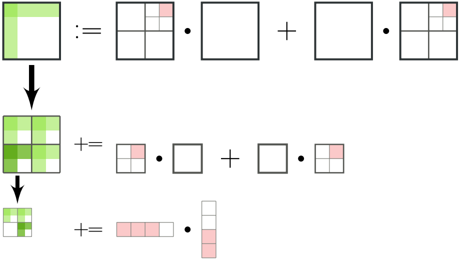


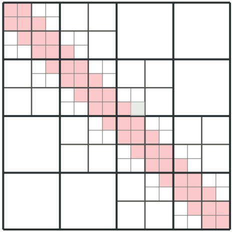


[1] S. Börm, *"Hierarchical matrix arithmetic with accumulated updates"*, submitted to Computing and Visualization in Science, 2017.

Motivating Example

Instead, updates are first collected for each destination block and afterwards shifted down following the hierarchy.[1]

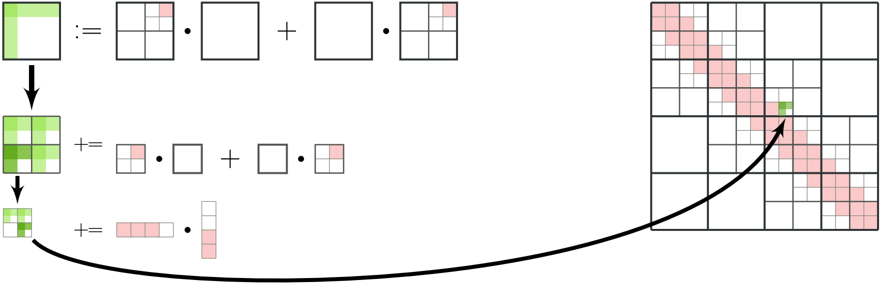




[1] S. Börm, *"Hierarchical matrix arithmetic with accumulated updates"*, submitted to Computing and Visualization in Science, 2017.

Motivating Example

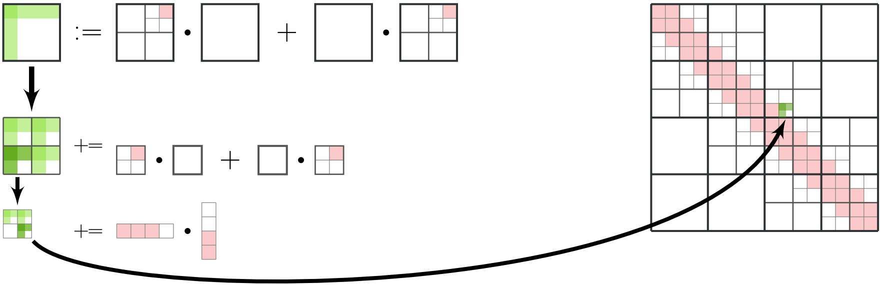
Instead, updates are first collected for each destination block and afterwards shifted down following the hierarchy.[1]



[1] S. Börm, *"Hierarchical matrix arithmetic with accumulated updates"*, submitted to Computing and Visualization in Science, 2017.

Motivating Example

Instead, updates are first collected for each destination block and afterwards shifted down following the hierarchy.[1]

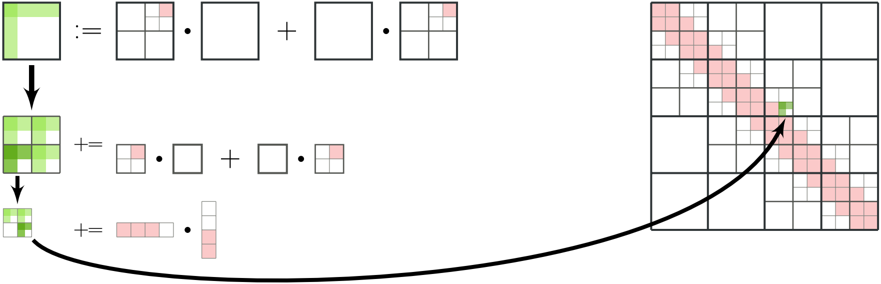


We now have 1 truncation on level 2, 2 truncations for level 3 and 4 truncations per subblock on level 4, summing up to 15 truncations for all low-rank blocks in $C_{t,s}$.

[1] S. Börm, *"Hierarchical matrix arithmetic with accumulated updates"*, submitted to Computing and Visualization in Science, 2017.

Motivating Example

Instead, updates are first collected for each destination block and afterwards shifted down following the hierarchy.[1]



We now have 1 truncation on level 2, 2 truncations for level 3 and 4 truncations per subblock on level 4, summing up to 15 truncations for all low-rank blocks in $C_{t,s}$.

Performing this for the full $\mathcal{H}$-multiplication $C := C + A \cdot B$ the number of truncations is reduced from 646 to 500.

[1] S. Börm, *"Hierarchical matrix arithmetic with accumulated updates"*, submitted to Computing and Visualization in Science, 2017.

Arithmetic

Let I be an index set, $T(I)$ a cluster tree over I and $T = T(I \times I)$ a block cluster tree over $T(I)$. For $t \in T(I)$ let $\mathcal{S}_t$ denote the set of sons of t. Furthermore, let A, B, C be $\mathcal{H}$–matrices over T.

For each matrix block $C_{t,s}$ we define an *accumulator* $U_{t,s} \in \mathbb{C}^{t \times s}$ and a set $\mathcal{P}_{t,s}$ of *pending* updates. Both are initialised to zero at the start of any $\mathcal{H}$–arithmetic, e.g., $U_{t,s} = 0$ and $\mathcal{P}_{t,s} = \emptyset$ for all $(t, s) \in T$.

Arithmetic

Let I be an index set, $T(I)$ a cluster tree over I and $T = T(I \times I)$ a block cluster tree over $T(I)$. For $t \in T(I)$ let $\mathcal{S}_t$ denote the set of sons of t. Furthermore, let A, B, C be $\mathcal{H}$-matrices over T.

For each matrix block $C_{t,s}$ we define an *accumulator* $U_{t,s} \in \mathbb{C}^{t \times s}$ and a set $\mathcal{P}_{t,s}$ of *pending* updates. Both are initialised to zero at the start of any $\mathcal{H}$-arithmetic, e.g., $U_{t,s} = 0$ and $\mathcal{P}_{t,s} = \emptyset$ for all $(t, s) \in T$.

$\mathcal{H}$-multiplication is split into two functions, which collect the updates and shift them down to sub blocks:

procedure ADDPRODUCT($A_{t,r}, B_{r,s}, C_{t,s}$)
    if $A_{t,r}, B_{r,s}, C_{t,s}$ are block matrices **then**
        $\mathcal{P}_{t,s} := \mathcal{P}_{t,s} \cup \{(A_{t,r}, B_{r,s})\}$;
    else
        $U_{t,s} := U_{t,s} + A_{t,r} \cdot B_{r,s}$;

procedure APPLYUPDATES($C_{t,s}$)
    if $C_{t,s}$ is a block matrix **then**
        for $t' \in \mathcal{S}_t, s' \in \mathcal{S}_s$ **do**
            $U_{t',s'} := U_{t',s'} + U_{t,s}|_{t',s'}$;
            for $(A_{t,r}, B_{r,s}) \in \mathcal{P}_{t,s}, r' \in \mathcal{S}_r$ **do**
                ADDPRODUCT($A_{t',r'}, B_{r',s'}, C_{t',s'}$);
            APPLYUPDATES($C_{t',s'}$);
    else
        $C_{t,s} := C_{t,s} + U_{t,s}$;

Arithmetic

Let I be an index set, $T(I)$ a cluster tree over I and $T = T(I \times I)$ a block cluster tree over $T(I)$. For $t \in T(I)$ let $\mathcal{S}_t$ denote the set of sons of t. Furthermore, let A, B, C be $\mathcal{H}$-matrices over T.

For each matrix block $C_{t,s}$ we define an *accumulator* $U_{t,s} \in \mathbb{C}^{t \times s}$ and a set $\mathcal{P}_{t,s}$ of *pending* updates. Both are initialised to zero at the start of any $\mathcal{H}$-arithmetic, e.g., $U_{t,s} = 0$ and $\mathcal{P}_{t,s} = \emptyset$ for all $(t,s) \in T$.

$\mathcal{H}$-multiplication is split into two functions, which collect the updates and shift them down to sub blocks:

```
procedure AddProduct(A_{t,r}, B_{r,s}, C_{t,s})
    if A_{t,r}, B_{r,s}, C_{t,s} are block matrices then
        P_{t,s} := P_{t,s} ∪ {(A_{t,r}, B_{r,s})};
    else
        U_{t,s} := U_{t,s} + A_{t,r} · B_{r,s};
```

```
procedure ApplyUpdates(C_{t,s}, type)
    if C_{t,s} is a block matrix then
        for t' ∈ S_t, s' ∈ S_s do
            U_{t',s'} := U_{t',s'} + U_{t,s}|_{t',s'};
            for (A_{t,r}, B_{r,s}) ∈ P_{t,s}, r' ∈ S_r do
                AddProduct( A_{t',r'}, B_{r',s'}, C_{t',s'} );
            if type = recursive then
                ApplyUpdates( C_{t',s'} );
    else
        C_{t,s} := C_{t,s} + U_{t,s};
```

Numerical Results

$\mathcal{H}$-matrix multiplication experiments are computed with $\mathcal{H}$-matrix based on Laplace SLP operator, on a unit sphere with block-wise accuracy of 10^{-4}.

n	t_{std}	t_{accu}	Speedup	#Trunc.
2.048	3.7	1.6	2.34x	42%
8.192	25.7	14.7	1.75x	50%
32.786	141.7	78.5	1.81x	44%
131.072	809.8	404.7	2.00x	36%
524.288	4313.3	2090.5	2.06x	31%
2.097.152	22944.3	10478.5	2.19x	25%

(time in seconds on Xeon E7-8857)

$\mathcal{H}$-LU factorization

The classical, recursive formulation of $\mathcal{H}$-LU factorization consists almost entirely off $\mathcal{H}$-matrix multiplications:

```
procedure LU(A_{t,t}, L_{t,t}, U_{t,t})
  if A_{t,t} is a block matrix then

    for 0 ≤ i < #S_t do
      LU( A_{t_i,t_i}, L_{t_i,t_i}, U_{t_i,t_i} );
      for i+1 ≤ j < #S_t do
        SolveLL( A_{t_i,t_j}, L_{t_i,t_i}, U_{t_i,t_j} );
        SolveUR( A_{t_j,t_i}, L_{t_j,t_i}, U_{t_i,t_i} );
      for i+1 ≤ j,ℓ < #S_t do
        Multiply( −1, L_{t_j,t_i}, U_{t_i,t_ℓ}, A_{t_j,t_ℓ} );
  else

    A_{t,t} = L_{t,t} U_{t,t};
```

```
procedure SolveLL(A_{t,s}, L_{t,t}, B_{t,s})
  if A_{t,s}, L_{t,t}, B_{t,s} are block matrices then

    for 0 ≤ i < #S_t do
      for 0 ≤ j < #S_s do
        SolveLL( A_{t_i,s_j}, L_{t_i,t_i}, B_{t_i,s_j} );
      for i+1 ≤ ℓ < #S_t do
        for 0 ≤ j < #S_s do
          Multiply( −1, L_{t_ℓ,t_i}, B_{t_i,s_j}, A_{t_ℓ,s_j} );
  else

    L_{t,t} B_{t,s} = A_{t,s};
```

A direct replacement of the $\mathcal{H}$-multiplication is not optimal, since it does not handle multiple updates during $\mathcal{H}$-LU.

$\mathcal{H}$-LU factorization

The classical, recursive formulation of $\mathcal{H}$-LU factorization consists almost entirely off $\mathcal{H}$-matrix multiplications:

```
procedure LU(A_{t,t}, L_{t,t}, U_{t,t})
  if A_{t,t} is a block matrix then
    ApplyUpdates( A_{t,t}, nonrecursive );
    for 0 ≤ i < #S_t do
      LU( A_{t_i,t_i}, L_{t_i,t_i}, U_{t_i,t_i} );
      for i+1 ≤ j < #S_t do
        SolveLL( A_{t_i,t_j}, L_{t_i,t_i}, U_{t_i,t_j} );
        SolveUR( A_{t_j,t_i}, L_{t_j,t_i}, U_{t_i,t_i} );
      for i+1 ≤ j,ℓ < #S_t do
        AddProduct(-1, L_{t_j,t_i}, U_{t_i,t_ℓ}, A_{t_j,t_ℓ});
  else
    ApplyUpdates( A_{t,t}, recursive );
    A_{t,t} = L_{t,t} U_{t,t};
```

```
procedure SolveLL(A_{t,s}, L_{t,t}, B_{t,s})
  if A_{t,s}, L_{t,t}, B_{t,s} are block matrices then
    ApplyUpdates( A_{t,s}, nonrecursive );
    for 0 ≤ i < #S_t do
      for 0 ≤ j < #S_s do
        SolveLL( A_{t_i,s_j}, L_{t_i,t_i}, B_{t_i,s_j} );
      for i+1 ≤ ℓ < #S_t do
        for 0 ≤ j < #S_s do
          AddProduct(-1, L_{t_ℓ,t_i}, B_{t_i,s_j}, A_{t_ℓ,s_j});
  else
    ApplyUpdates( A_{t,s}, recursive );
    L_{t,t} B_{t,s} = A_{t,s};
```

A direct replacement of the $\mathcal{H}$-multiplication is not optimal, since it does not handle multiple updates during $\mathcal{H}$-LU.

Instead, collecting and applying updates is separated and accumulators are shifted down level by level in the hierarchy.

$\mathcal{H}$-LU factorization

Results for Laplace SLP operator:

n	t_{std}	t_{accu}	Speedup	#Trunc.
2.048	1.0	0.7	1.46x	61%
8.192	7.2	4.8	1.48x	51%
32.786	42.3	23.7	1.79x	38%
131.072	259.1	122.9	2.11x	27%
524.288	1469.2	654.4	2.25x	21%
2.097.152	7908.0	3484.9	2.27x	17%

(time in seconds on Xeon E7-8857)

$\mathcal{H}$-LU factorization

Results for Laplace SLP operator:

n	t_{std}	t_{accu}	Speedup	#Trunc.
2.048	1.0	0.7	1.46x	61%
8.192	7.2	4.8	1.48x	51%
32.786	42.3	23.7	1.79x	38%
131.072	259.1	122.9	2.11x	27%
524.288	1469.2	654.4	2.25x	21%
2.097.152	7908.0	3484.9	2.27x	17%

(time in seconds on Xeon E7-8857)

Results for Helmholtz SLP operator with wavenumber $\kappa = 2$:

n	t_{std}	t_{accu}	Speedup	#Trunc.
2.048	1.9	1.3	1.50x	54%
8.192	14.5	10.6	1.37x	53%
32.786	86.1	52.8	1.63x	38%
131.072	537.5	284.5	1.89x	27%
524.288	3101.2	1548.0	2.00x	21%

(time in seconds on Xeon E7-8857)

Complexity and Accuracy

Complexity and Accuracy

The theoretical complexity of $\mathcal{H}$-multiplication in the standard and the accumulator based form is $\mathcal{O}\left(k^2 n \log^2 n\right)$.

As indicated by the numerical result, the complexity of the accumulator version seems reduced compared to the standard version.

Complexity and Accuracy

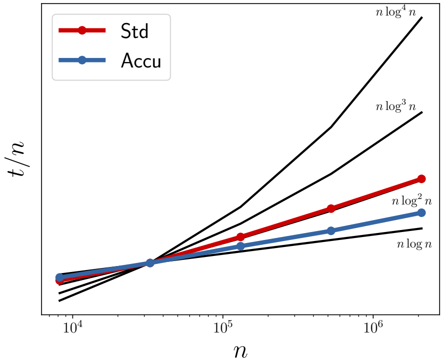
The theoretical complexity of $\mathcal{H}$-multiplication in the standard and the accumulator based form is $\mathcal{O}\left(k^2 n \log^2 n\right)$.

As indicated by the numerical result, the complexity of the accumulator version seems reduced compared to the standard version.

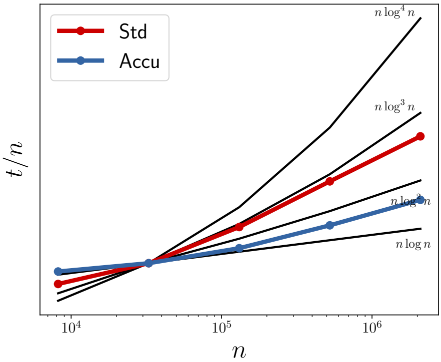
Complexity in Practice

$$\varepsilon = 10^{-2}$$

Multiplication



$\mathcal{H}$-LU



Complexity and Accuracy

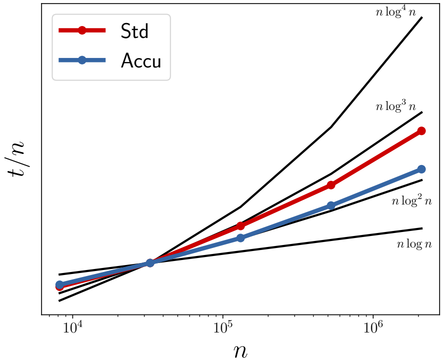
The theoretical complexity of $\mathcal{H}$-multiplication in the standard and the accumulator based form is $\mathcal{O}\left(k^2 n \log^2 n\right)$.

As indicated by the numerical result, the complexity of the accumulator version seems reduced compared to the standard version.

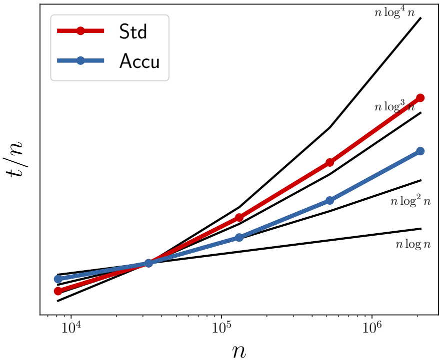
Complexity in Practice

$\varepsilon = 10^{-4}$

Multiplication



$\mathcal{H}$-LU



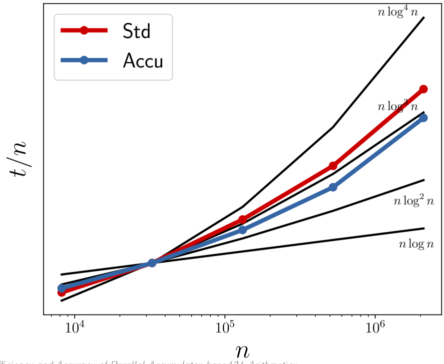
The theoretical complexity of $\mathcal{H}$–multiplication in the standard and the accumulator based form is $\mathcal{O}\left(k^2 n \log^2 n\right)$.

As indicated by the numerical result, the complexity of the accumulator version seems reduced compared to the standard version.

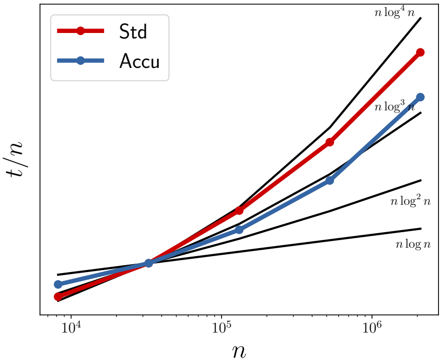
Complexity in Practice

$\varepsilon = 10^{-6}$

Multiplication



$\mathcal{H}$–LU



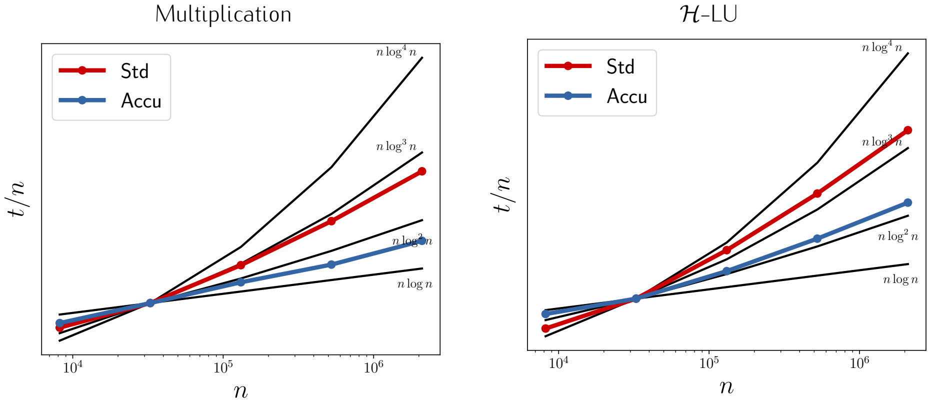
Complexity and Accuracy

The theoretical complexity of $\mathcal{H}$-multiplication in the standard and the accumulator based form is $\mathcal{O}\left(k^2 n \log^2 n\right)$.

As indicated by the numerical result, the complexity of the accumulator version seems reduced compared to the standard version.

Complexity in Practice

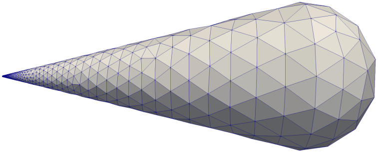
$k = 7$



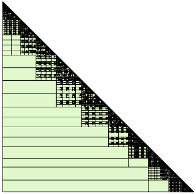
Complexity and Accuracy

The behaviour remains with a grid resulting in a degenerate $\mathcal{H}$-structure:

Grid



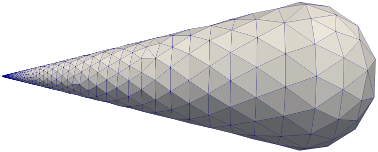
$\mathcal{H}$-matrix



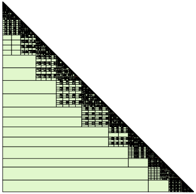
Complexity and Accuracy

The behaviour remains with a grid resulting in a degenerate $\mathcal{H}$-structure:

Grid

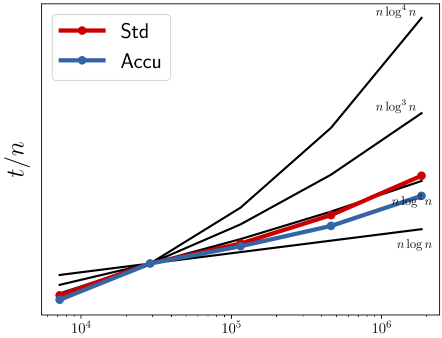


$\mathcal{H}$-matrix

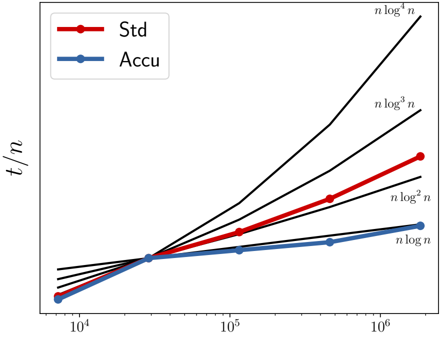


$\varepsilon = 10^{-2}$

Multiplication



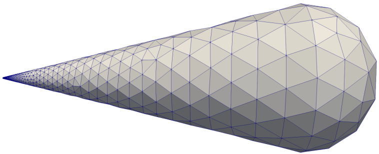
$\mathcal{H}$-LU



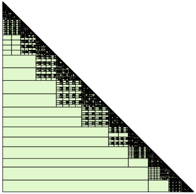
Complexity and Accuracy

The behaviour remains with a grid resulting in a degenerate $\mathcal{H}$-structure:

Grid

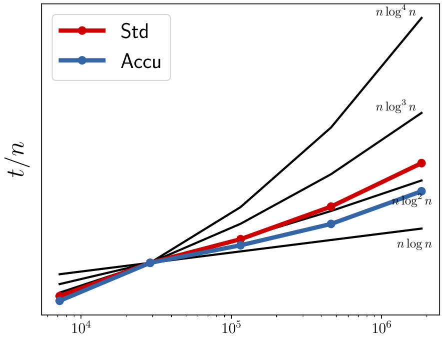


$\mathcal{H}$-matrix

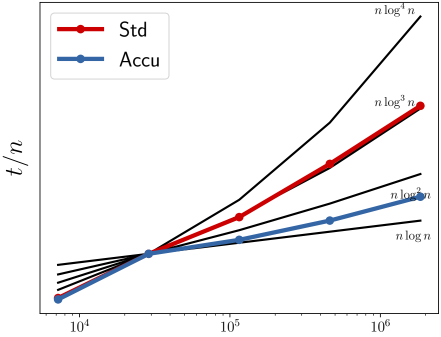


$\varepsilon = 10^{-4}$

Multiplication

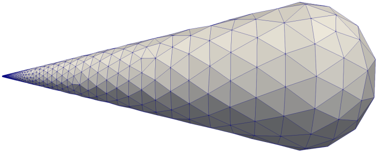


$\mathcal{H}$-LU

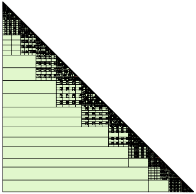


The behaviour remains with a grid resulting in a degenerate $\mathcal{H}$-structure:

Grid

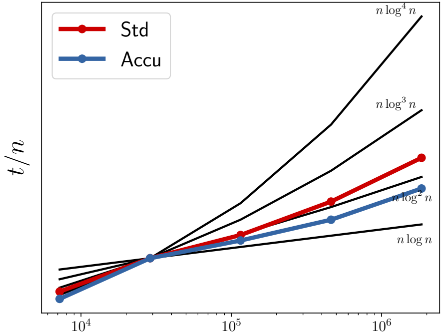


$\mathcal{H}$-matrix

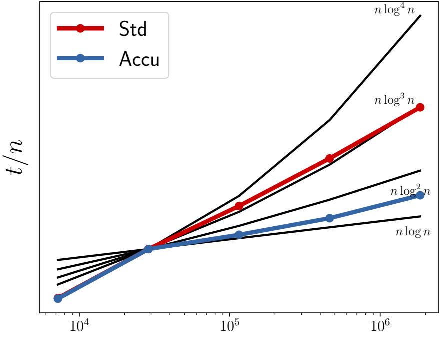


$\varepsilon = 10^{-6}$

Multiplication

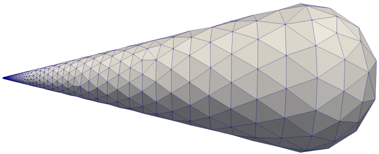


$\mathcal{H}$-LU

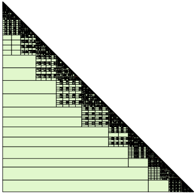


The behaviour remains with a grid resulting in a degenerate $\mathcal{H}$-structure:

Grid

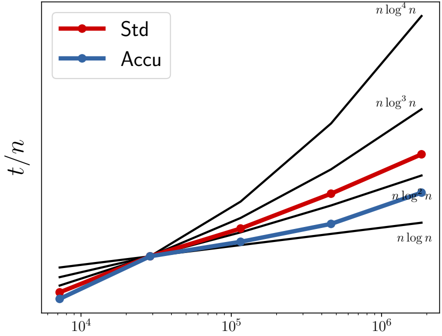


$\mathcal{H}$-matrix

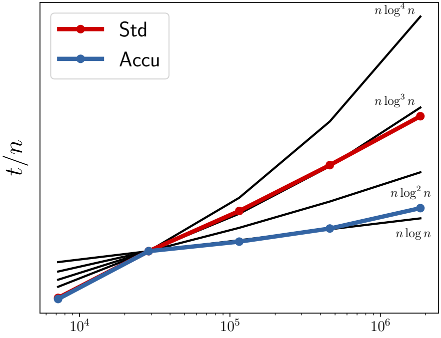


$k = 7$

Multiplication



$\mathcal{H}$-LU



Complexity and Accuracy

Due to the different summation order of low-rank blocks, accumulator based $\mathcal{H}$-arithmetic shows higher ranks compared to standard $\mathcal{H}$-arithmetic.

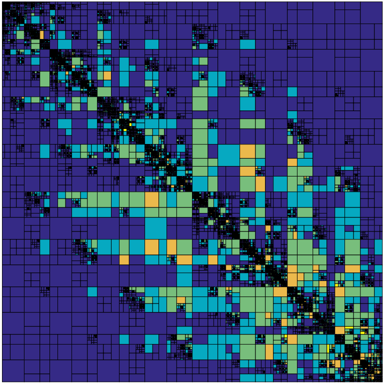
Also the accuracy is slightly worse compared to standard $\mathcal{H}$-arithmetic.

n	Mem_{std}	Mem_{accu}	Increase
32.786	385	422	9.6 %
131.072	1760	1970	11.9 %
524.288	8160	9210	12.9 %
2.097.152	36900	41960	13.7 %

(memory in MB)

$\varepsilon = 10^{-4}$	$\text{Error}_{\text{std}}$	$\text{Error}_{\text{accu}}$
32.786	$1.5_{10}{-3}$	$4.3_{10}{-3}$
131.072	$2.4_{10}{-3}$	$6.2_{10}{-3}$
524.288	$3.2_{10}{-3}$	$8.8_{10}{-3}$
2.097.152	$4.6_{10}{-3}$	$1.3_{10}{-2}$

(error is $\|I - (LU)^{-1}A\|_2$)



Rank difference between standard and accumulator $\mathcal{H}$-LU.

Complexity and Accuracy

Due to the different summation order of low-rank blocks, accumulator based $\mathcal{H}$-arithmetic shows higher ranks compared to standard $\mathcal{H}$-arithmetic.

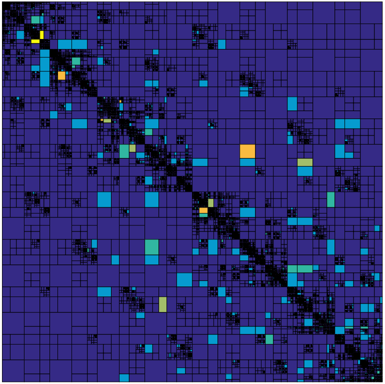
Also the accuracy is slightly worse compared to standard $\mathcal{H}$-arithmetic.

n	Mem_{std}	Mem_{accu}	Increase
32.786	613	622	1.5 %
131.072	3000	3050	1.7 %
524.288	14490	14730	1.7 %
2.097.152	67650	68780	1.7 %

(memory in MB)

$\varepsilon = 10^{-6}$	$\text{Error}_{\text{std}}$	$\text{Error}_{\text{accu}}$
32.786	$4.6_{10}{-5}$	$6.2_{10}{-5}$
131.072	$6.7_{10}{-5}$	$8.1_{10}{-5}$
524.288	$9.6_{10}{-5}$	$1.2_{10}{-4}$
2.097.152	$1.3_{10}{-4}$	$1.8_{10}{-4}$

(error is $\|I - (LU)^{-1}A\|_2$)



Rank difference between standard and accumulator $\mathcal{H}$-LU.

However, this effect is dependent on the predefined accuracy of the $\mathcal{H}$-arithmetic. The better the approximation, the less the difference.

Task-Parallel $\mathcal{H}$-LU

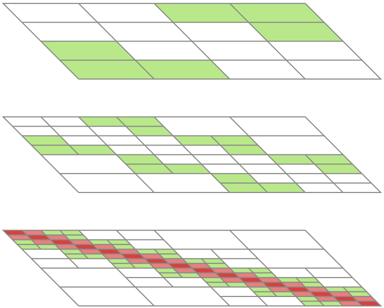
$\mathcal{H}$-LU with Tasks

The standard, task-based $\mathcal{H}$-LU factorisation defines individual tasks for block factorisation, solving and updates based on the recursive $\mathcal{H}$-LU algorithm modified to have *global* scope.

procedure DAGLU($A_{t,t}, L_{t,t}, U_{t,t}$)
  task(LU($A_{t,t}, L_{t,t}, U_{t,t}$));
  if $A_{t,t}$ is a block matrix **then**
    for $0 \leq i < \#\mathcal{S}_t$ **do**
      DAGLU($A_{t_i,t_i}, L_{t_i,t_i}, U_{t_i,t_i}$);

  for $s \in T^{\ell(t)}, s >_I t$ **do**
    task(SolveLL($A_{t,s}, L_{t,t_i}, U_{t,s}$));
    task(SolveUR($A_{s,t}, L_{s,t}, U_{t,t}$));

  for $s, r \in T^{\ell(t)}, s, r >_I t$ **do**
    task(Multiply($-1, L_{s,t}, U_{t,r}, A_{s,r}$));



With the level set $T^{\ell(t)} := \{s \in T : \text{level}(s) = \text{level}(t)\}$ and the index set relation $s >_I t :\Leftrightarrow \forall i \in s, j \in t : i > j$.

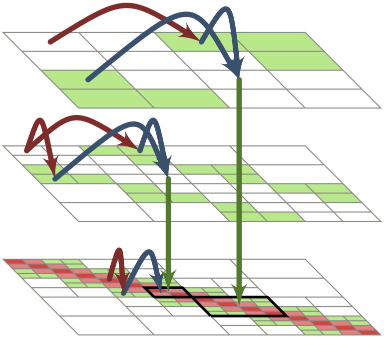
$\mathcal{H}$-LU with Tasks

The standard, task-based $\mathcal{H}$-LU factorisation defines individual tasks for block factorisation, solving and updates based on the recursive $\mathcal{H}$-LU algorithm modified to have *global* scope.

procedure DAGLU($A_{t,t}, L_{t,t}, U_{t,t}$)
  task(LU($A_{t,t}, L_{t,t}, U_{t,t}$));
  if $A_{t,t}$ is a block matrix **then**
    for $0 \leq i < \#\mathcal{S}_t$ **do**
      DAGLU($A_{t_i,t_i}, L_{t_i,t_i}, U_{t_i,t_i}$);

  for $s \in T^{\ell(t)}, s >_I t$ **do**
    task(SolveLL($A_{t,s}, L_{t,t_i}, U_{t,s}$));
    task(SolveUR($A_{s,t}, L_{s,t}, U_{t,t}$));

  for $s, r \in T^{\ell(t)}, s, r >_I t$ **do**
    task(Multiply($-1, L_{s,t}, U_{t,r}, A_{s,r}$));



With the level set $T^{\ell(t)} := \{s \in T : \text{level}(s) = \text{level}(t)\}$ and the index set relation $s >_I t :\Leftrightarrow \forall i \in s, j \in t : i > j$.

Dependencies exist between factorisation and solve tasks on the same level or due to updates tasks on different levels.

Accumulator $\mathcal{H}$-LU with Tasks

The accumulator based $\mathcal{H}$-LU with tasks follows the same modifications as in the recursive case: multiplication is replaced by collecting updates and accumulated updates are applied following the hierarchy.

procedure DAGLU($A_{t,t}, L_{t,t}, U_{t,t}$)
  task(LU($A_{t,t}, L_{t,t}, U_{t,t}$));
  if $A_{t,t}$ is a block matrix **then**
    for $0 \leq i < \#\mathcal{S}_t$ **do**
      DAGLU($A_{t_i,t_i}, L_{t_i,t_i}, U_{t_i,t_i}$);

  for $s \in T^{\ell(t)}, s >_I t$ **do**
    task(SolveLL($A_{t,s}, L_{t,t_i}, U_{t,s}$));
    task(SolveUR($A_{s,t}, L_{s,t}, U_{t,t}$));

  for $s, r \in T^{\ell(t)}, s, r >_I t$ **do**
    task(AddProduct($-1, L_{s,t_i}, U_{t_i,r}, A_{s,r}$));

Let $\mathcal{U}_{t,s}$ be the set of all AddProduct tasks for $A_{t,s}$.

Accumulator $\mathcal{H}$-LU with Tasks

The accumulator based $\mathcal{H}$-LU with tasks follows the same modifications as in the recursive case: multiplication is replaced by collecting updates and accumulated updates are applied following the hierarchy.

```
procedure DAGLU(A_{t,t}, L_{t,t}, U_{t,t})
   task(LU( A_{t,t}, L_{t,t}, U_{t,t} ));
   if  A_{t,t} is a block matrix  then
      for  0 ≤ i < #S_t  do
         DAGLU( A_{t_i,t_i}, L_{t_i,t_i}, U_{t_i,t_i});

   for  s ∈ T^{ℓ(t)}, s >_l t  do
      task(SolveLL( A_{t,s}, L_{t,t_i}, U_{t,s} ));
      task(SolveUR( A_{s,t}, L_{s,t}, U_{t,t} ));

   for  s, r ∈ T^{ℓ(t)}, s, r >_l t  do
      task(AddProduct(−1, L_{s,t_i}, U_{t_i,r}, A_{s,r}));
```

Let $\mathcal{U}_{t,s}$ be the set of all AddProduct tasks for $A_{t,s}$.

```
procedure BuildApplyTasks(A_{t,s})
   if 𝒰_{t,s} ≠ ∅  then
      task( ApplyUpdates(A_{t,s}) );
      for  U ∈ 𝒰_{t,s}  do
         U  ──→  task( ApplyUpdates(A_{t,s}) );
```

Dependency rules:

If updates exist, an ApplyUpdates task is required and depends on them.

Accumulator $\mathcal{H}$-LU with Tasks

The accumulator based $\mathcal{H}$-LU with tasks follows the same modifications as in the recursive case: multiplication is replaced by collecting updates and accumulated updates are applied following the hierarchy.

```
procedure DAGLU(A_{t,t}, L_{t,t}, U_{t,t})
   task(LU( A_{t,t}, L_{t,t}, U_{t,t} ));
   if  A_{t,t} is a block matrix  then
      for  0 ≤ i < #S_t  do
         DAGLU( A_{t_i,t_i}, L_{t_i,t_i}, U_{t_i,t_i});

   for  s ∈ T^{ℓ(t)}, s >_l t  do
      task(SolveLL( A_{t,s}, L_{t,t_i}, U_{t,s} ));
      task(SolveUR( A_{s,t}, L_{s,t}, U_{t,t} ));

   for  s, r ∈ T^{ℓ(t)}, s, r >_l t  do
      task(AddProduct(−1, L_{s,t_i}, U_{t_i,r}, A_{s,r}));
```

```
procedure BuildApplyTasks(A_{t,s})
   if U_{t,s} ≠ ∅ or task(parent) exists then
      task( ApplyUpdates(A_{t,s}) );
      for  U ∈ U_{t,s}  do
         U  ──►  task( ApplyUpdates(A_{t,s}) );
```

Let $\mathcal{U}_{t,s}$ be the set of all AddProduct tasks for $A_{t,s}$.

Dependency rules:

If a block has an ApplyUpdates task, so have all subblocks.

Accumulator $\mathcal{H}$-LU with Tasks

The accumulator based $\mathcal{H}$-LU with tasks follows the same modifications as in the recursive case: multiplication is replaced by collecting updates and accumulated updates are applied following the hierarchy.

```
procedure DAGLU(A_{t,t}, L_{t,t}, U_{t,t})
   task(LU( A_{t,t}, L_{t,t}, U_{t,t} ));
   if A_{t,t} is a block matrix then
      for 0 ≤ i < #S_t do
         DAGLU( A_{t_i,t_i}, L_{t_i,t_i}, U_{t_i,t_i});

   for s ∈ T^{ℓ(t)}, s >_l t do
      task(SolveLL( A_{t,s}, L_{t,t_i}, U_{t,s} ));
      task(SolveUR( A_{s,t}, L_{s,t}, U_{t,t} ));

   for s, r ∈ T^{ℓ(t)}, s, r >_l t do
      task(AddProduct(−1, L_{s,t_i}, U_{t_i,r}, A_{s,r}));
```

Let $\mathcal{U}_{t,s}$ be the set of all AddProduct tasks for $A_{t,s}$.

```
procedure BuildApplyTasks(A_{t,s})
   if U_{t,s} ≠ ∅ or task(parent) exists then
      task( ApplyUpdates(A_{t,s}) );
      for U ∈ U_{t,s} do
         U ⟶ task( ApplyUpdates(A_{t,s}) );

   if task(parent) exists then
      task(parent) ⟶ task(ApplyUpdates(A_{t,s}));
```

Dependency rules:

Parent tasks need to be executed before son tasks.

Accumulator $\mathcal{H}$-LU with Tasks

The accumulator based $\mathcal{H}$-LU with tasks follows the same modifications as in the recursive case: multiplication is replaced by collecting updates and accumulated updates are applied following the hierarchy.

```
procedure DAGLU(A_{t,t}, L_{t,t}, U_{t,t})
  task(LU( A_{t,t}, L_{t,t}, U_{t,t} ));
  if A_{t,t} is a block matrix then
    for 0 ≤ i < #S_t do
      DAGLU( A_{t_i,t_i}, L_{t_i,t_i}, U_{t_i,t_i});

  for s ∈ T^{ℓ(t)}, s >_I t do
    task(SolveLL( A_{t,s}, L_{t,t_i}, U_{t,s} ));
    task(SolveUR( A_{s,t}, L_{s,t}, U_{t,t} ));

  for s, r ∈ T^{ℓ(t)}, s, r >_I t do
    task(AddProduct(−1, L_{s,t_i}, U_{t_i,r}, A_{s,r}));
```

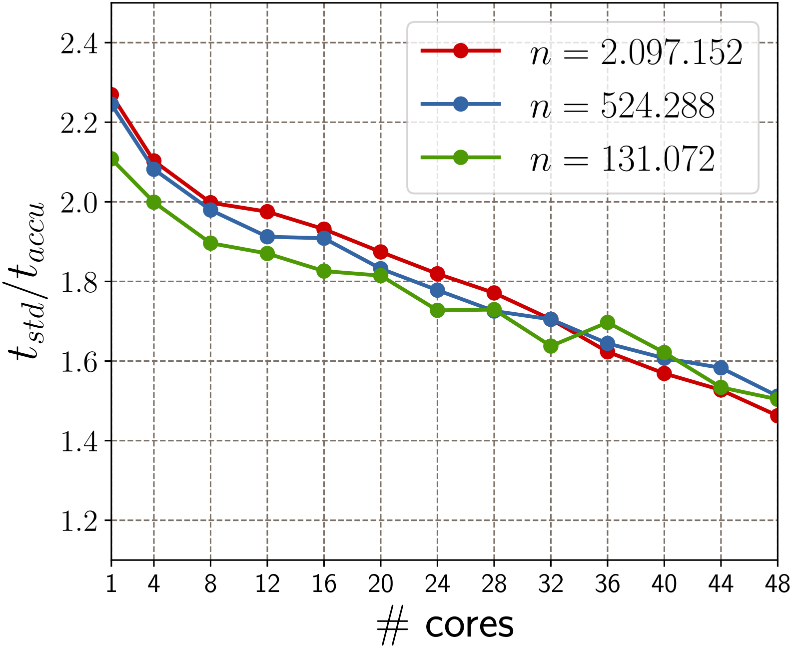
Let $\mathcal{U}_{t,s}$ be the set of all AddProduct tasks for $A_{t,s}$.

```
procedure BuildApplyTasks(A_{t,s})
  if U_{t,s} ≠ ∅ or task(parent) exists then
    task( ApplyUpdates(A_{t,s}) );
    for U ∈ U_{t,s} do
      U ⟶ task( ApplyUpdates(A_{t,s}) );

  if task(parent) exists then
    task(parent) ⟶ task(ApplyUpdates(A_{t,s}));

  if task(LU(A_{t,s})) or task(Solve(A_{t,s})) exists then
    task( ApplyUpdates(A_{t,s}) ) ⟶
          task(LU(A_{t,s})) / task(Solve(A_{t,s}, ·, ·))
  else
    for (t', s') ∈ S_{t,s} do
      BuildApplyTasks(A_{t',s'});
```

Dependency rules:

If LU/solve task exists, it depends on the ApplyUpdates task.

Numerical Results



Conclusion

Accumulator based $\mathcal{H}$-arithmetic significantly reduces the number of truncations during $\mathcal{H}$-arithmetic with a reduction in practical complexity.

Modification of existing implementations is simple and straight forward.

Parallel speedup is reduced compared to standard $\mathcal{H}$-arithmetic but still significant overall speedup.

Conclusion

Accumulator based $\mathcal{H}$-arithmetic significantly reduces the number of truncations during $\mathcal{H}$-arithmetic with a reduction in practical complexity.

Modification of existing implementations is simple and straight forward.

Parallel speedup is reduced compared to standard $\mathcal{H}$-arithmetic but still significant overall speedup.