Accumulator based Task-parallel $\mathcal{H}$-Factorization

Steffen Börm
University of Kiel

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

SIAM PP18

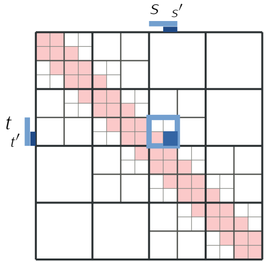
$\mathcal{H}$-Arithmetic with Accumulators

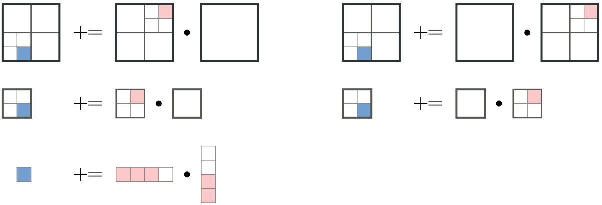
Motivating Example

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:

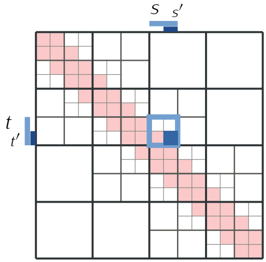




Similar updates are computed for all other sub blocks of the parent block $C_{t,s}$.

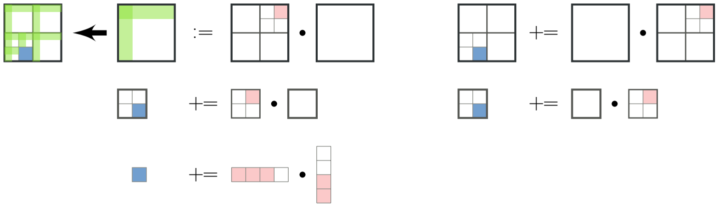
Motivating Example

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:

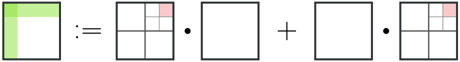


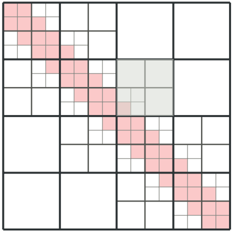
Similar updates are computed for all other sub blocks of the parent block $C_{t,s}$.

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]

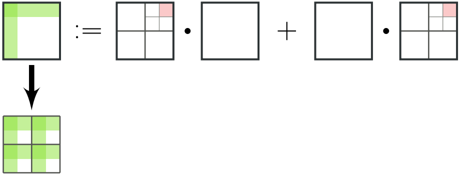


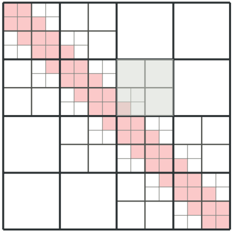


[1] S. Börm, *"Hierarchical matrix arithmetic with accumulated updates"*, submitted to Computing and Visualization in Science, 2017.

Motivating Example

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

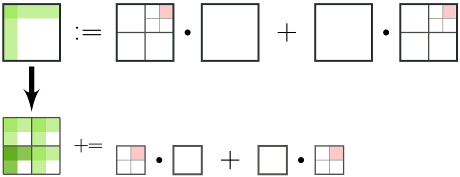


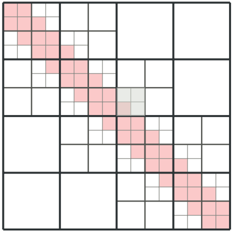


[1] S. Börm, *"Hierarchical matrix arithmetic with accumulated updates"*, submitted to Computing and Visualization in Science, 2017.

Motivating Example

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

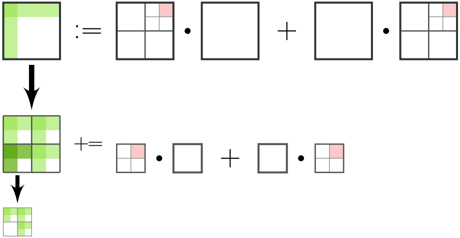


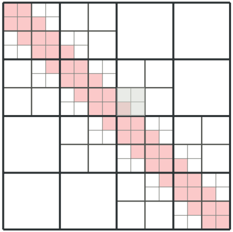


[1] S. Börm, *"Hierarchical matrix arithmetic with accumulated updates"*, submitted to Computing and Visualization in Science, 2017.

Motivating Example

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

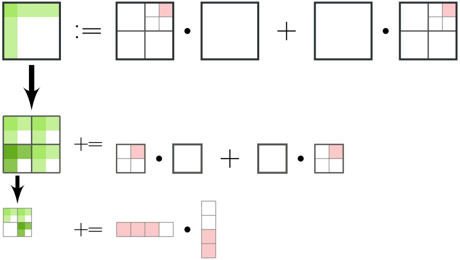


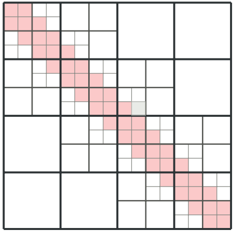


[1] S. Börm, *"Hierarchical matrix arithmetic with accumulated updates"*, submitted to Computing and Visualization in Science, 2017.

Motivating Example

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

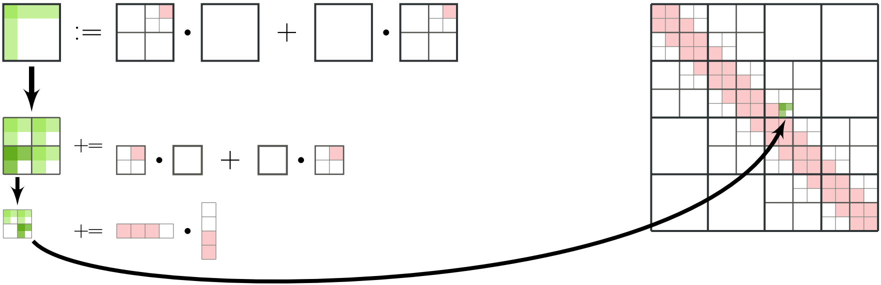




[1] S. Börm, *"Hierarchical matrix arithmetic with accumulated updates"*, submitted to Computing and Visualization in Science, 2017.

Motivating Example

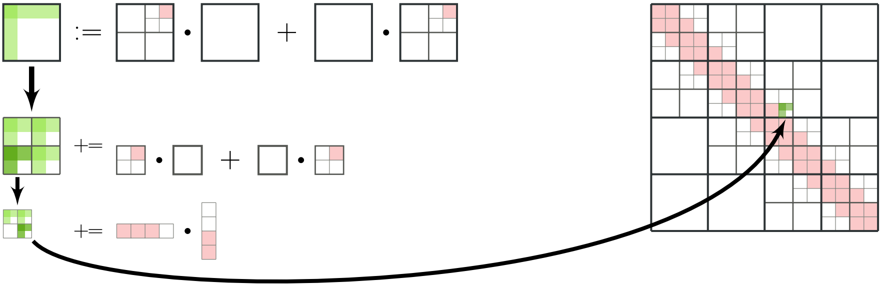
Instead, updates are first collected for each destination block and afterwards shifted down following the hierarchy.[1]



[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]

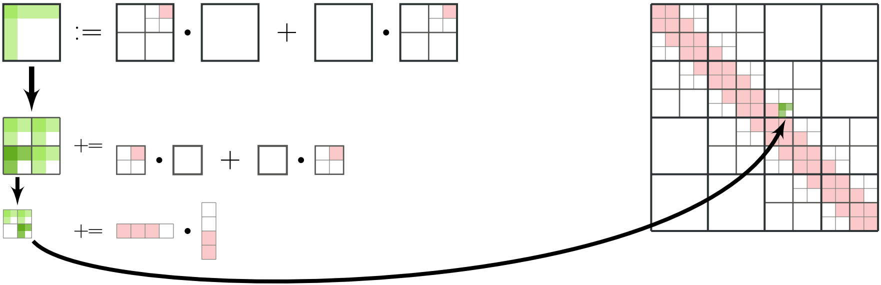


We now have 1 truncation on level 2, 2 truncations for level 3 and 4 truncations per subblock on level 4, summing up to 15 truncations for all low-rank blocks in $C_{t,s}$.

[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**
    $\mathcal{P}_{t,s} := \mathcal{P}_{t,s} \cup \{(A_{t,r}, B_{r,s})\}$;
  else
    $U_{t,s} := U_{t,s} + A_{t,r} \cdot B_{r,s}$;

procedure APPLYUPDATES($C_{t,s}$, type)
  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($A_{t',r'}, B_{r',s'}, C_{t',s'}$);
      if type = recursive **then**
        APPLYUPDATES($C_{t',s'}$);
  else
    $C_{t,s} := C_{t,s} + U_{t,s}$;

Numerical Experiments

$\mathcal{H}$-matrix multiplication experiments are computed with $\mathcal{H}$-matrix based on Helmholtz SLP operator, $\kappa = 2$ on a unit sphere with block-wise accuracy of 10^{-4}.

Xeon E7-8857 (**Ivybridge**)

n	t_{std}	t_{accu}	Speedup	#Trunc.
2.048	7.6	3.4	2.23x	39%
8.192	55.7	35.9	1.55x	52%
32.786	311.6	199.1	1.56x	46%
131.072	1801.9	1024.4	1.76x	37%
524.288	9836.4	5322.0	1.85x	30%

Xeon Platinum 8176 (**Skylake**)

n	t_{std}	t_{accu}	Speedup
2.048	5.1	2.1	2.46x
8.192	38.5	23.6	1.63x
32.786	222.6	136.8	1.63x
131.072	1274.8	713.9	1.79x

(time in seconds)

$\mathcal{H}$-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};
```

$\mathcal{H}$-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 $0 \leq i < \#\mathcal{S}_t$ **do**
      LU(A_{t_i,t_i}, L_{t_i,t_i}, U_{t_i,t_i});
      for $i+1 \leq j < \#\mathcal{S}_t$ **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 $i+1 \leq j, \ell < \#\mathcal{S}_t$ **do**
        AddProduct(−1, L_{t_j,t_i}, U_{t_i,t_ℓ}, A_{t_j,t_ℓ});
        ApplyUpdates(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 $0 \leq i < \#\mathcal{S}_t$ **do**
      for $0 \leq j < \#\mathcal{S}_s$ **do**
        SolveLL(A_{t_i,s_j}, L_{t_i,t_i}, B_{t_i,s_j});
      for $i+1 \leq \ell < \#\mathcal{S}_t$ **do**
        for $0 \leq j < \#\mathcal{S}_s$ **do**
          AddProduct(−1, L_{t_ℓ,t_i}, B_{t_i,s_j}, A_{t_ℓ,s_j});
          ApplyUpdates(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 $0 \leq i < \#\mathcal{S}_t$ **do**
      LU($A_{t_i,t_i}, L_{t_i,t_i}, U_{t_i,t_i}$);
      for $i+1 \leq j < \#\mathcal{S}_t$ **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 $i+1 \leq j, \ell < \#\mathcal{S}_t$ **do**
        AddProduct($-1, L_{t_j,t_i}, U_{t_i,t_\ell}, A_{t_j,t_\ell}$);
  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 $0 \leq i < \#\mathcal{S}_t$ **do**
      for $0 \leq j < \#\mathcal{S}_s$ **do**
        SolveLL($A_{t_i,s_j}, L_{t_i,t_i}, B_{t_i,s_j}$);
      for $i+1 \leq \ell < \#\mathcal{S}_t$ **do**
        for $0 \leq j < \#\mathcal{S}_s$ **do**
          AddProduct($-1, L_{t_\ell,t_i}, B_{t_i,s_j}, A_{t_\ell,s_j}$);
  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

Xeon E7-8857 (**Ivybridge**)

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%

Xeon Platinum 8176 (**Skylake**)

n	t_{std}	t_{accu}	Speedup
2.048	1.4	0.8	1.64x
8.192	10.0	7.1	1.41x
32.786	62.2	38.6	1.61x
131.072	387.1	205.0	1.89x

(time in seconds)

Rank Growth 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.

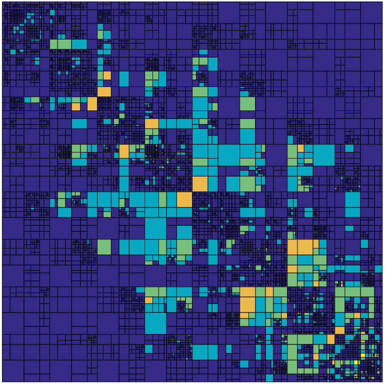
Also the accuracy is slightly worse compared to standard $\mathcal{H}$-arithmetic.

n	Mem_{std}	Mem_{accu}	Increase
8.192	175	185	5.7 %
32.786	837	907	8.3 %
131.072	3820	4210	10.2 %
524.288	17580	19590	11.4 %

(memory in MB)

$\varepsilon = 10^{-4}$	$\text{Error}_{\text{std}}$	$\text{Error}_{\text{accu}}$
8.192	$1.3_{10}{-3}$	$2.7_{10}{-3}$
32.786	$1.7_{10}{-3}$	$4.1_{10}{-3}$
131.072	$2.4_{10}{-3}$	$6.0_{10}{-3}$
524.288	$3.6_{10}{-3}$	$8.8_{10}{-3}$

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



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

Rank Growth 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.

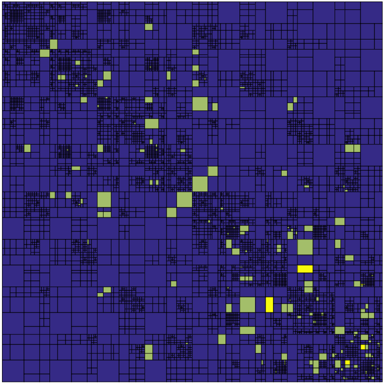
Also the accuracy is slightly worse compared to standard $\mathcal{H}$-arithmetic.

n	Mem_{std}	Mem_{accu}	Increase
8.192	249	253	1.6 %
32.786	1280	1310	2.3 %
131.072	6420	6560	2.2 %
524.288	30840	31510	2.2 %

(memory in MB)

$\varepsilon = 10^{-6}$	$\text{Error}_{\text{std}}$	$\text{Error}_{\text{accu}}$
8.192	$4.4_{10}{-5}$	$3.5_{10}{-5}$
32.786	$6.5_{10}{-5}$	$6.7_{10}{-5}$
131.072	$9.8_{10}{-5}$	$1.0_{10}{-4}$
524.288	$1.4_{10}{-4}$	$1.5_{10}{-4}$

(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.

Adding Tasks

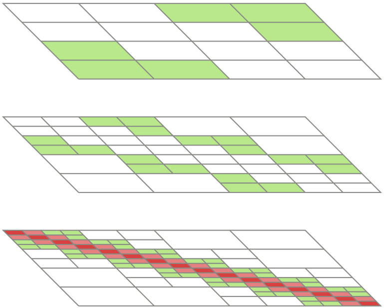
$\mathcal{H}$-LU with Tasks

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($A_{t,t}, L_{t,t}, U_{t,t}$));

  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 $s >_I t :\Leftrightarrow \forall i \in s, j \in t : i > j$.

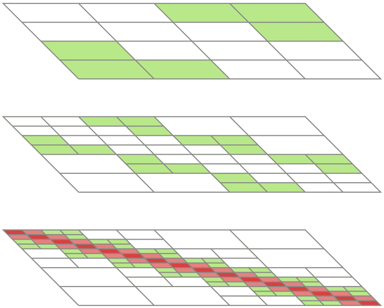
$\mathcal{H}$-LU with Tasks

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 $s >_I t :\Leftrightarrow \forall i \in s, j \in t : i > j$.

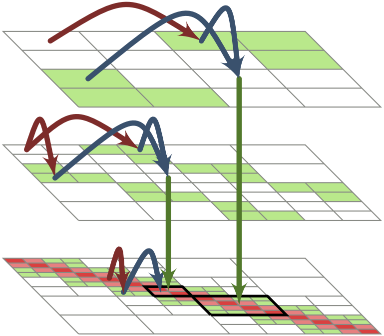
$\mathcal{H}$-LU with Tasks

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 $s \in T^{\ell(t)}, s >_l t$ **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 $s, r \in T^{\ell(t)}, s, r >_l t$ **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 $A_{t,s}$.

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 $0 \leq i < \#\mathcal{S}_t$ **do**
      DAGLU(A_{t_i,t_i}, L_{t_i,t_i}, U_{t_i,t_i});

  for $s \in T^{\ell(t)}, s >_l t$ **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 $s, r \in T^{\ell(t)}, s, r >_l t$ **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 $\mathcal{U}_{t,s} \neq \emptyset$ or **task**(parent) exists **then**
    task(ApplyUpdates($A_{t,s}$));
    for $U \in \mathcal{U}_{t,s}$ **do**
      $U \longrightarrow$ **task**(ApplyUpdates($A_{t,s}$));

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 $0 \leq i < \#\mathcal{S}_t$ **do**
            DAGLU($A_{t_i,t_i}, L_{t_i,t_i}, U_{t_i,t_i}$);

    for $s \in T^{\ell(t)}, s >_l t$ **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 $s, r \in T^{\ell(t)}, s, r >_l t$ **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 $\mathcal{U}_{t,s} \neq \emptyset$ or **task**(parent) exists **then**
        task(ApplyUpdates($A_{t,s}$));
        for $U \in \mathcal{U}_{t,s}$ **do**
            $U \longrightarrow$ **task**(ApplyUpdates($A_{t,s}$));

    if **task**(parent) exists **then**
        task(parent) $\longrightarrow$ **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 Experiments

Experiments are computed for the Helmholtz example with $n = 524.288$ (Ivybridge) and $n = 131.072$ (Skylake) .

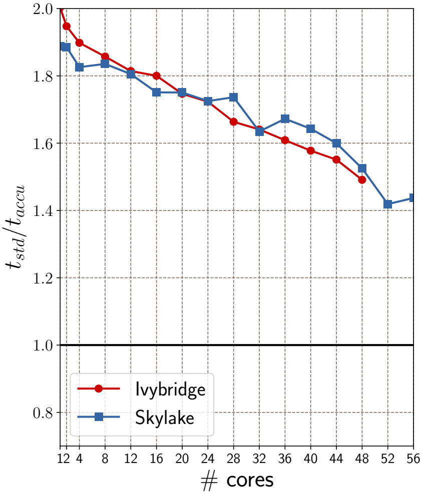
Xeon E7–8857 (**Ivybridge**)

# cores	t_{std}	t_{accu}	Speedup
1	3101.2	1548.0	2.00x
12	285.3	157.3	1.81x
24	156.4	91.2	1.72x
48	99.1	66.9	1.48x

Xeon Platinum 8176 (**Skylake**), no HT

# cores	t_{std}	t_{accu}	Speedup
1	387.1	205.0	1.89x
28	21.2	12.2	1.74x
56	14.3	9.9	1.44x

(time in seconds)



Further Modifications

Further Modifications

Up to now, all direct updates are evaluated and applied immediately to the accumulator. Instead, this may be postponed until all updates are available and then applied together.

For this, an additional set $\mathcal{P}^{\text{direct}}_{t,s}$ of *pending direct* updates is introduced:

procedure $\textsc{AddProductLazy}(A_{t,r}, B_{r,s}, C_{t,s})$
  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
    $\mathcal{P}^{\text{direct}}_{t,s} := \mathcal{P}^{\text{direct}}_{t,s} \cup \{(A_{t,r}, B_{r,s})\}$;

procedure $\textsc{ApplyUpdatesLazy}(C_{t,s})$
  if $C_{t,s}$ is a block matrix **then**
    for $(A, B) \in \mathcal{P}^{\text{direct}}_{t,s}$ **do**
      $U_{t,s} := U_{t,s} + A \cdot B$;
    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**
        $\textsc{AddProduct}(A_{t',r'}, B_{r',s'}, C_{t',s'})$;
      $\textsc{ApplyUpdates}(\ C_{t',s'}\ )$;
  else
    $C_{t,s} := C_{t,s} + U_{t,s}$;

Lazy evaluation applies to the accumulators per level and not the destination block.

Further Modifications

Up to now, all direct updates are evaluated and applied immediately to the accumulator. Instead, this may be postponed until all updates are available and then applied together.

For this, an additional set $\mathcal{P}_{t,s}^{\text{direct}}$ of *pending direct* updates is introduced:

```
procedure AddProductLazy(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
    P^direct_{t,s} := P^direct_{t,s} ∪ {(A_{t,r}, B_{r,s})};
```

```
procedure ApplyUpdatesLazy(C_{t,s})
  if C_{t,s} is a block matrix then
    for U ∈ sort({A · B : (A, B) ∈ P^direct_{t,s}}) do
      U_{t,s} := U_{t,s} + U;
    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'});
      ApplyUpdates( C_{t',s'} );
  else
    C_{t,s} := C_{t,s} + U_{t,s};
```

Lazy evaluation applies to the accumulators per level and not the destination block.

Since all updates are available, it also permits update sorting.

Further Modifications

Up to now, all direct updates are evaluated and applied immediately to the accumulator. Instead, this may be postponed until all updates are available and then applied together.

For this, an additional set $\mathcal{P}^{\text{direct}}_{t,s}$ of *pending direct* updates is introduced:

procedure AddProductLazy($A_{t,r}, B_{r,s}, C_{t,s}$)
  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
    $\mathcal{P}^{\text{direct}}_{t,s} := \mathcal{P}^{\text{direct}}_{t,s} \cup \{(A_{t,r}, B_{r,s})\}$;

procedure ApplyUpdatesLazy($C_{t,s}$)
  if $C_{t,s}$ is a block matrix **then**
    $\mathcal{U} := \text{batch}(\{A \cdot B : (A, B) \in \mathcal{P}^{\text{direct}}_{t,s}\})$;
    $U_{t,s} := U_{t,s} + \text{reduce}(\mathcal{U})$;
    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
    $C_{t,s} := C_{t,s} + U_{t,s}$;

Lazy evaluation applies to the accumulators per level and not the destination block.

Since all updates are available, it also permits update sorting or batch execution.

Numerical Tests

$\mathcal{H}$-matrix multiplication

n	t_{eager}	t_{lazy}
2.048	3.4	3.4
8.192	35.9	35.9
32.786	199.1	195.0
131.072	1024.4	1023.1

$\mathcal{H}$-LU factorization

	eager		**lazy**	
n	Time	Error	Time	Error
2.048	1.3	$1.0_{10}\text{-}3$	1.4	$1.2_{10}\text{-}3$
8.192	10.6	$2.7_{10}\text{-}3$	10.8	$2.3_{10}\text{-}3$
32.786	52.8	$4.2_{10}\text{-}3$	54.2	$3.6_{10}\text{-}3$
131.072	284.5	$6.0_{10}\text{-}3$	291.8	$5.4_{10}\text{-}3$

All computed without update sorting.

Conclusion

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

Modification of existing implementations is simple and straight forward.

Parallel speedup is slightly 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 possible reduction in complexity.

Modification of existing implementations is simple and straight forward.

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