Task-Based $\mathcal{H}$-Matrix Arithmetics

Part I: Algorithm Design

Ronald Kriemann
MPI MIS

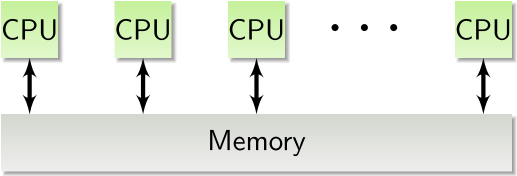
Winterschool on $\mathcal{H}$-Matrices

2014

Introduction

Parallel Systems

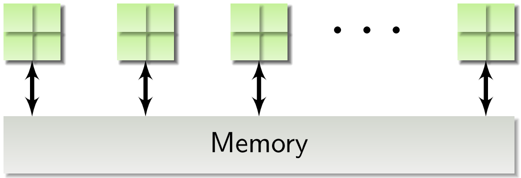
We consider *shared-memory* systems, i.e. computers with several CPUs all accessing the same memory.



Having a single address space for all processors, simplifies parallel programming because inter-process *communication is free*.

Parallel Systems

We consider *shared-memory* systems, i.e. computers with several CPUs all accessing the same memory.

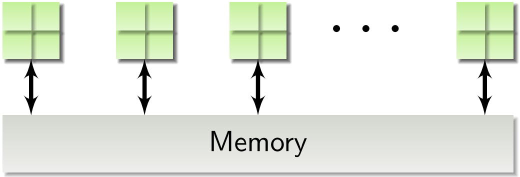


Having a single address space for all processors, simplifies parallel programming because inter-process *communication is free*.

Nowadays, each CPU consists of several compute *cores*. *Multi-core* CPUs have 8-16 cores, e.g. Intel Xeon or AMD Opteron CPUs, whereas *many-core* CPUs have 64 or more cores, e.g. Intel XeonPhi.

Parallel Systems

We consider *shared-memory* systems, i.e. computers with several CPUs all accessing the same memory.



Having a single address space for all processors, simplifies parallel programming because inter-process *communication is free*.

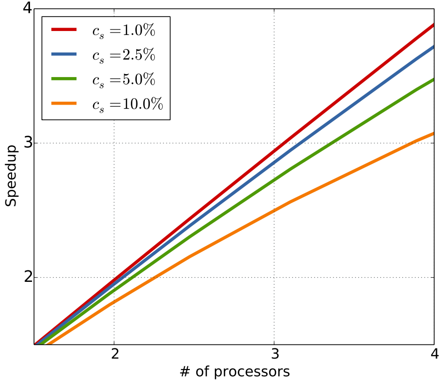
Nowadays, each CPU consists of several compute *cores*. *Multi-core* CPUs have 8-16 cores, e.g. Intel Xeon or AMD Opteron CPUs, whereas *many-core* CPUs have 64 or more cores, e.g. Intel XeonPhi.

The main problem on such systems is to *keep all cores busy*.

Parallel Systems

If cores are idle during the execution of an algorithm, the *parallel speedup* will deteriorate very rapidly.

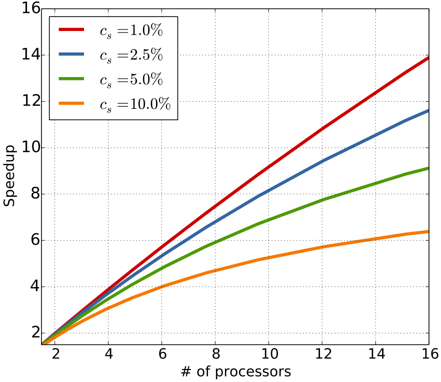
The reason is *Amdahl's Law*: the influence of the sequential part on the speedup:



Parallel Systems

If cores are idle during the execution of an algorithm, the *parallel speedup* will deteriorate very rapidly.

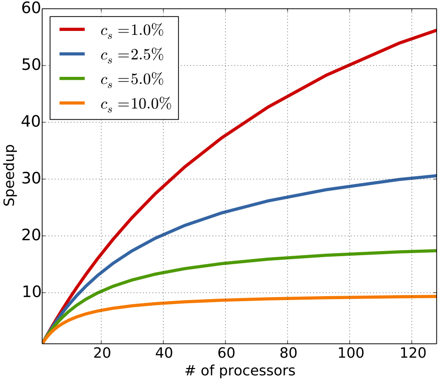
The reason is *Amdahl's Law*: the influence of the sequential part on the speedup:



Parallel Systems

If cores are idle during the execution of an algorithm, the *parallel speedup* will deteriorate very rapidly.

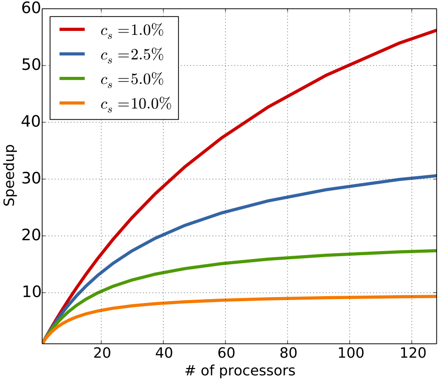
The reason is *Amdahl's Law*: the influence of the sequential part on the speedup:



Parallel Systems

If cores are idle during the execution of an algorithm, the *parallel speedup* will deteriorate very rapidly.

The reason is *Amdahl's Law*: the influence of the sequential part on the speedup:



Sources of idleness: sequential code, overhead, inefficiencies.

Tasks

An algorithm typically consists of many individual operations, e.g. each update of y_i in the dense matrix-vector multiplication

$$
\begin{array}{l}
\textbf{for} \;\; i = 0, \ldots, n-1 \;\; \textbf{do} \\
\quad \textbf{for} \;\; j = 0, \ldots, n-1 \;\; \textbf{do} \\
\quad\quad y_i = y_i + A_{ij} x_j;
\end{array}
$$

Tasks

An algorithm typically consists of many individual operations, e.g. each update of y_i in the dense matrix-vector multiplication

for $i = 0, \ldots, n-1$ **do**
  for $j = 0, \ldots, n-1$ **do**
    $y_i = y_i + A_{ij} x_j;$

In a parallel algorithm, an *atomic* set of operations, which is executed by a *single* processor is called a *task*.

Designing algorithms by concentrating on tasks can help to reduce idle times on many-core systems.

Tasks

An algorithm can be implemented with a different task *granularity*:

```
for i = 0, ..., n-1 do
    for j = 0, ..., n-1 do
        task                                  // One task per matrix entry
            y_i = y_i + A_ij x_j;
```

```
for i = 0, ..., n-1 do
    task                                      // One task per row
        for j = 0, ..., n-1 do
            y_i = y_i + A_ij x_j;
```

If a task is too small, too much overhead due to task management may occur.

If task granularity is too large, too few tasks may result, leaving processors idle.

Tasks

Often, between different tasks *dependencies* exists, e.g. the result of one task is the input of another task:

procedure $\textsc{DotProduct}(x, y, i, j)$
  if $i = j$ **then**
    task
      return $x_i \cdot y_i$;
  else
    task
      $d_0 := \textsc{DotProduct}(x, y, i, (i+j)/2 - 1)$;
      $d_1 := \textsc{DotProduct}(x, y, (i+j)/2, j)$;
      return $d_0 + d_1$;

Here, the computation of the sub intervals has to finish *before* computing the final result.

Tasks

To achieve an optimal parallel speedup, the task granularity and the *execution order* of all tasks need to be optimal for a specific computer system.

Various factors are to consider for an optimal granularity and execution order: costs of tasks, number of processors, processor layout, memory hierarchy, etc..

Often, some of these factors are *unknown* or very *specific* to a computer system.

Tasks

To achieve an optimal parallel speedup, the task granularity and the *execution order* of all tasks need to be optimal for a specific computer system.

Various factors are to consider for an optimal granularity and execution order: costs of tasks, number of processors, processor layout, memory hierarchy, etc..

Often, some of these factors are *unknown* or very *specific* to a computer system.

Fortunately, there is software available, which may be used to

- simplify task definition and
- optimise execution order.

However, for this, algorithm design has to be changed to be

task-based

$\mathcal{H}$-Matrix Construction

$\mathcal{H}$-Matrix Construction

Let I be an index set, $T(I)$ a (binary) $\mathcal{H}$-tree over I and $T = T(I \times I)$ a $\mathcal{H}_\times$-tree over $T(I)$ with $\mathcal{L}(T)$ being the set of leaves of T.

The basic algorithm for $\mathcal{H}$-matrix construction is

procedure $\textsc{MatrixConstruct}(T)$
    for all $b \in \mathcal{L}(T)$ **do**

        if b is admissible **then**
            build low-rank matrix;
        else
            build dense matrix;

$\mathcal{H}$-Matrix Construction

Let I be an index set, $T(I)$ a (binary) $\mathcal{H}$-tree over I and $T = T(I \times I)$ a $\mathcal{H}_\times$-tree over $T(I)$ with $\mathcal{L}(T)$ being the set of leaves of T.

The basic algorithm for $\mathcal{H}$-matrix construction is

procedure $\textsc{MatrixConstruct}(T)$
  for all $b \in \mathcal{L}(T)$ **do**
    task
      if b is admissible **then**
        build low-rank matrix;
      else
        build dense matrix;

The construction of *each* leaf in T defines a new task.

$\mathcal{H}$-Matrix Construction

The main properties of

procedure $\textsc{MatrixConstruct}(T)$

  for all $b \in \mathcal{L}(T)$ **do**

    task

      build dense/low-rank matrix for b;

are

- much more tasks $(\#\mathcal{L}(T))$ than processors and
- construction of a block does *not depend* on other blocks.

$\mathcal{H}$-Matrix Construction

The main properties of

procedure $\text{MatrixConstruct}(T)$
    #pragma omp parallel for   *// loop-parallelisation in OpenMP*
    for all $b \in \mathcal{L}(T)$ **do**   *// each b on a different p*

        build dense/low-rank matrix for b;

are

- much more tasks $(\#\mathcal{L}(T))$ than processors and
- construction of a block does *not depend* on other blocks.

With this, a simple *loop-parallelisation* will result in an optimal parallel speedup.

$\mathcal{H}$-Matrix Construction with Coarsening

When *coarsening* is added to matrix construction, the algorithm is implemented via recursion, creating *dependencies* between tasks:

procedure $\textsc{MatrixConstruct}(b \in T)$
  if $b \in \mathcal{L}(T)$ **then**
    build dense/low-rank matrix for b;
  else
    for all $b' \in \mathcal{S}(b)$ **do**
      $\textsc{MatrixConstruct}(b')$;
    coarsen matrix for b;

The coarsening may be performed only after all sub blocks have been created!

$\mathcal{H}$-Matrix Construction with Coarsening

When *coarsening* is added to matrix construction, the algorithm is implemented via recursion, creating *dependencies* between tasks:

```
procedure MatrixConstruct(b ∈ T)
    if b ∈ L(T) then
        build dense/low-rank matrix for b;
    else
        for all b' ∈ S(b) do
            MatrixConstruct(b');
        coarsen matrix for b;
```

The coarsening may be performed only after all sub blocks have been created!

Properties:

- still much more tasks ($\#T$) than processors and
- tasks are *not* independent (dependency follows hierarchy).

$\mathcal{H}$-Matrix Construction with Coarsening

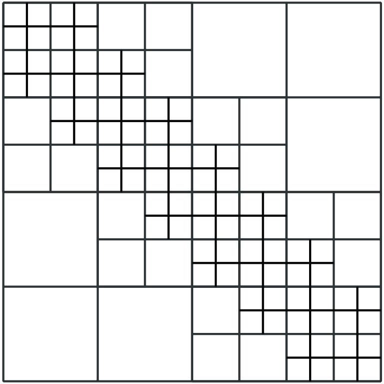
A parallel version of MatrixConstruct with simple loop-parallelisation:

```
procedure MatrixConstruct(b ∈ T)
    if b ∈ L(T) then
        build dense/low-rank matrix for b;
    else
        #pragma omp parallel for
        for all b' ∈ S(b) do
            MatrixConstruct(b');
        coarsen matrix for b;
```

Here, blocks of the $\mathcal{H}_\times$-tree are mapped to processors in a *top-down* way.

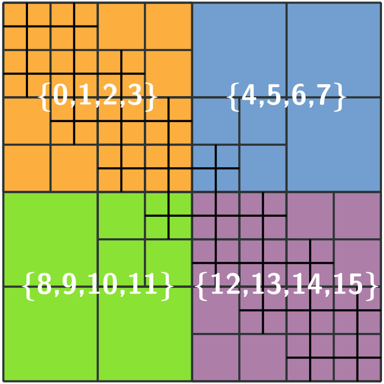
$\mathcal{H}$-Matrix Construction with Coarsening

Top-down mapping during matrix construction for $\mathcal{P} = \{0, \ldots, 15\}$:



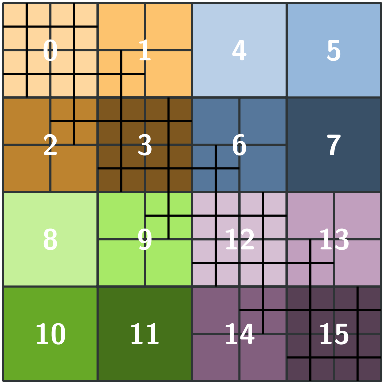
$\mathcal{H}$-Matrix Construction with Coarsening

Top-down mapping during matrix construction for $\mathcal{P} = \{0, \ldots, 15\}$:



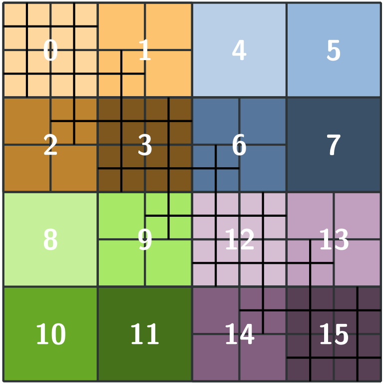
$\mathcal{H}$-Matrix Construction with Coarsening

Top-down mapping during matrix construction for $\mathcal{P} = \{0, \ldots, 15\}$:



$\mathcal{H}$-Matrix Construction with Coarsening

Top-down mapping during matrix construction for $\mathcal{P} = \{0, \ldots, 15\}$:



Problem: costs for matrix construction *differ* depending on position in matrix leading to load imbalance and hence, idle processors.

$\mathcal{H}$-Matrix Construction with Coarsening

As an alternative, only the tasks and their dependencies are defined, without processor mapping (bottom-up approach):

```
procedure MatrixConstruct(b ∈ T)
    if b ∈ L(T) then
        task
            build leaf matrix;
    else
        task
            for all b' ∈ S(b) do              // define task dependencies
                sub task: MatrixConstruct(b');
            coarsen matrix for b;
```

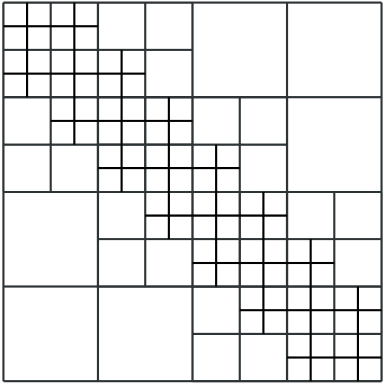
This creates a task dependency tree equal to the $\mathcal{H}_\times$-tree.

Since tasks appear early in the hierarchy, hierarchy traversal is distributed to all processors.

As long as there are ready tasks, no processor idles.

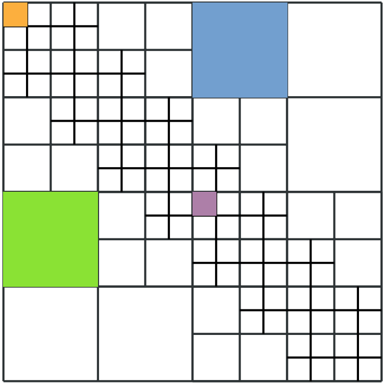
$\mathcal{H}$-Matrix Construction with Coarsening

Mapping of matrix blocks to processors when using tasks:



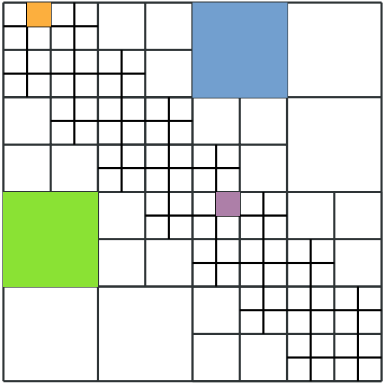
$\mathcal{H}$-Matrix Construction with Coarsening

Mapping of matrix blocks to processors when using tasks:



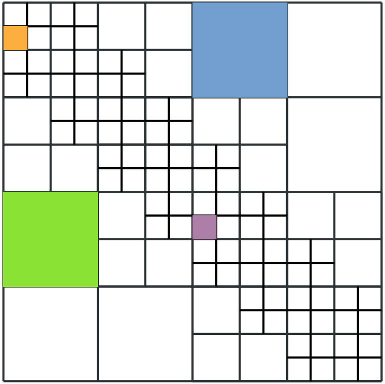
$\mathcal{H}$-Matrix Construction with Coarsening

Mapping of matrix blocks to processors when using tasks:



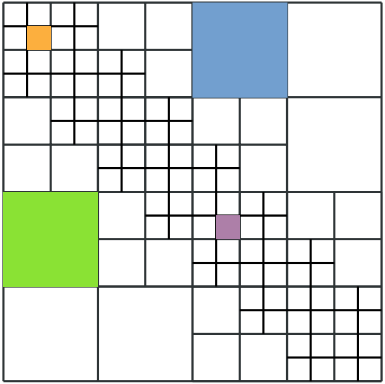
$\mathcal{H}$-Matrix Construction with Coarsening

Mapping of matrix blocks to processors when using tasks:



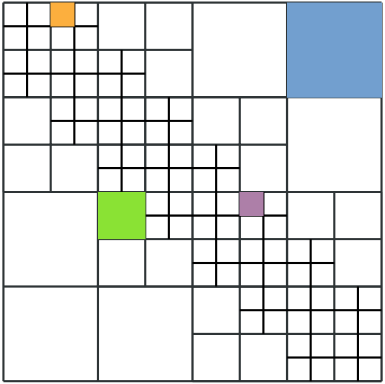
$\mathcal{H}$-Matrix Construction with Coarsening

Mapping of matrix blocks to processors when using tasks:



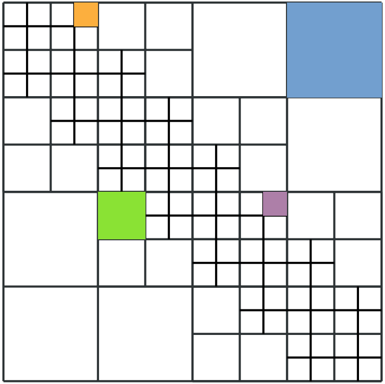
$\mathcal{H}$-Matrix Construction with Coarsening

Mapping of matrix blocks to processors when using tasks:



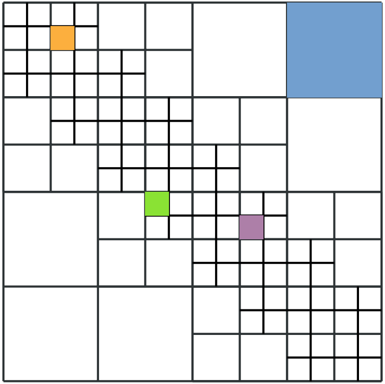
$\mathcal{H}$-Matrix Construction with Coarsening

Mapping of matrix blocks to processors when using tasks:



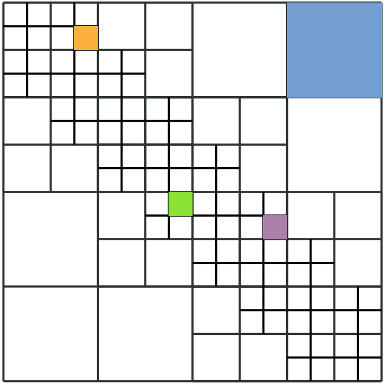
$\mathcal{H}$-Matrix Construction with Coarsening

Mapping of matrix blocks to processors when using tasks:



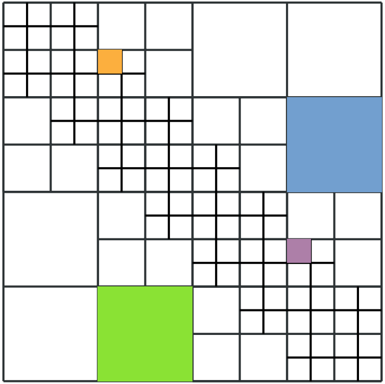
$\mathcal{H}$-Matrix Construction with Coarsening

Mapping of matrix blocks to processors when using tasks:



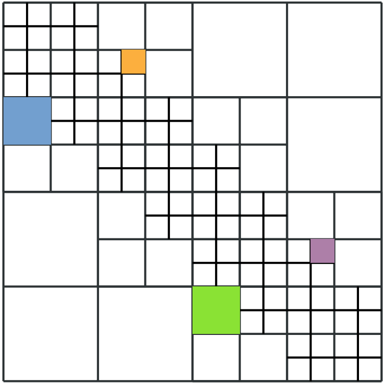
$\mathcal{H}$-Matrix Construction with Coarsening

Mapping of matrix blocks to processors when using tasks:



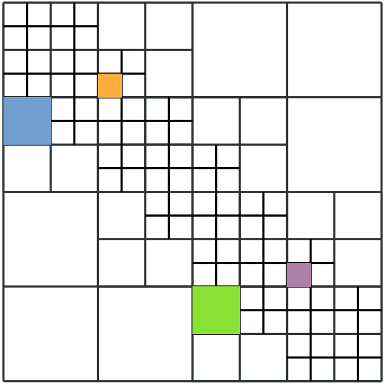
$\mathcal{H}$-Matrix Construction with Coarsening

Mapping of matrix blocks to processors when using tasks:



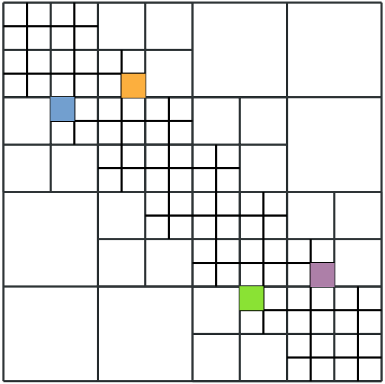
$\mathcal{H}$-Matrix Construction with Coarsening

Mapping of matrix blocks to processors when using tasks:



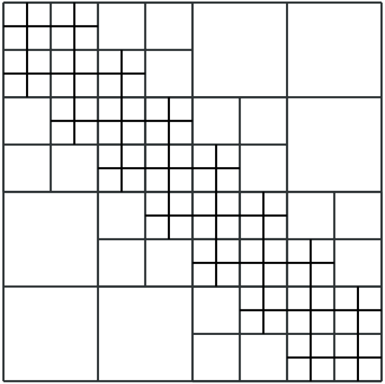
$\mathcal{H}$-Matrix Construction with Coarsening

Mapping of matrix blocks to processors when using tasks:



$\mathcal{H}$-Matrix Construction with Coarsening

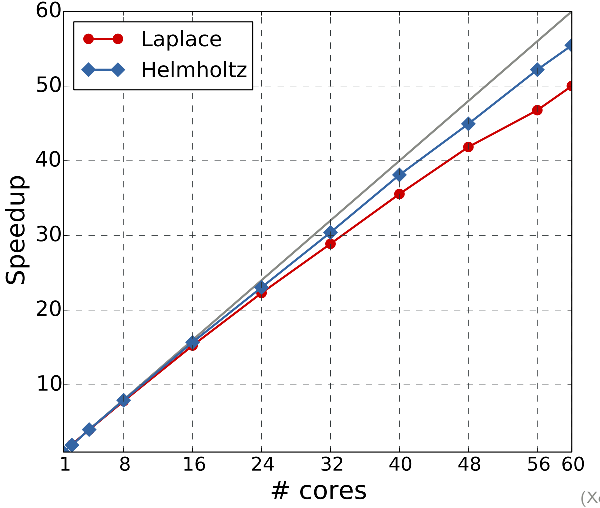
Mapping of matrix blocks to processors when using tasks:



Idling may still happen, e.g. if a very costly task is scheduled at the end of the computation (but very unlikely in a typical $\mathcal{H}$-matrix).

Numerical Results

$\mathcal{H}$-matrix construction for Laplace-/Helmholtz-SLP on unit sphere:



(XeonPhi 5110P)

$\mathcal{H}$-Matrix Multiplication

$\mathcal{H}$-Matrix Multiplication

We consider the general update form $A := \alpha B \cdot C + A$, which results in the following recursion:

procedure $\textsc{mul}(\alpha, A, B, C)$
    if A, B, C are block matrices **then**
        for $i \in 0, 1$ **do**
            for $j \in 0, 1$ **do**
                for $\ell \in 0, 1$ **do**
                    mul($\alpha, A_{ij}, B_{i\ell}, C_{\ell j}$);
    else

            $A := A + \alpha BC$;

$\mathcal{H}$-Matrix Multiplication

We consider the general update form $A := \alpha B \cdot C + A$, which results in the following recursion:

```
procedure MUL(α, A, B, C)
    if A, B, C are block matrices then
        for i ∈ 0,1 do
            for j ∈ 0,1 do
                for ℓ ∈ 0,1 do
                    mul( α, A_ij, B_iℓ, C_ℓj );
    else
        task
            A := A + αBC;
```

The work is performed if one of the matrices is a leaf matrix. Hence, this forms a task.

Critical Sections

During matrix multiplication, different tasks update the same matrix block, which therefore forms a *critical section*, i.e. at most one processor may write to the same matrix block at a time.

procedure $\textsc{mul}(\alpha, A, B, C)$
  if A, B, C are block matrices **then**
    for $i, j, l \in 0, 1$ **do**
      mul($\alpha, A_{ij}, B_{i\ell}, C_{\ell j}$);
  else
    task

      Critical: $A := A + \alpha BC$;

Critical Sections

During matrix multiplication, different tasks update the same matrix block, which therefore forms a *critical section*, i.e. at most one processor may write to the same matrix block at a time.

```
procedure MUL(α, A, B, C)
    if  A, B, C are block matrices  then
        for  i, j, l ∈ 0, 1  do
            mul( α, A_ij, B_iℓ, C_ℓj );
    else
        task
            lock A
                A := A + αBC;
            unlock A
```

A *mutex* ensures, that only one processor may enter a critical section while all other processors will wait for the mutex to be unlocked.

Critical Sections

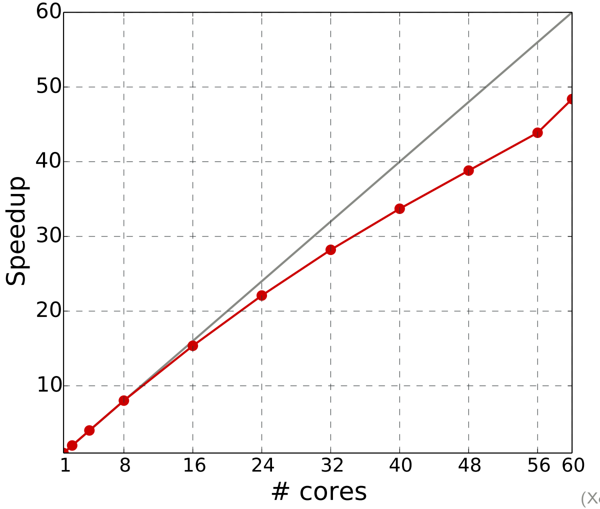
To avoid processor idling while waiting for a locked mutex, the update may be split into computing the update matrix and applying the update:

```
procedure MUL(α, A, B, C)
    if A, B, C are block matrices then
        for i, j, l ∈ 0, 1 do
            mul( α, A_ij, B_iℓ, C_ℓj );
    else
        task                                    // compute update
            T := αBC;
        task                                    // apply update
            lock A
                A := A + T;
            unlock A
```

Computing T is *independent* from all other tasks.

Numerical Results

$\mathcal{H}$-matrix multiplication for (unsymmetric) Laplace-SLP matrix on unit sphere:



(XeonPhi 5110P)

$\mathcal{H}$-LU Factorisation

$\mathcal{H}$-LU Factorisation

For an $\mathcal{H}$-Matrix A over T, the LU factorisation $A = LU$ is defined by the block structure of A, L and U

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \cdot \begin{pmatrix} U_{00} & U_{01} \\ & U_{11} \end{pmatrix},$$

which leads to the following equations:

$$A_{00} = L_{00}U_{00} \qquad \text{(Recursion)}$$
$$A_{01} = L_{00}U_{01} \qquad \text{(Matrix Solve)}$$
$$A_{10} = L_{10}U_{00} \qquad \text{(Matrix Solve)}$$
$$A_{11} = L_{10}U_{01} + L_{11}U_{11} \qquad \text{(Update and Recursion)}$$

Classical $\mathcal{H}$-LU Algorithm

The above equations directly translate into an algorithm for the $\mathcal{H}$-LU factorisation:

procedure $\mathrm{LU}(A, L, U)$
- $\mathrm{LU}(\ A_{00}, L_{00}, U_{00}\ );$
- $\textsc{SolveLower}(\ A_{01}, L_{00}, U_{01}\ );$
- $\textsc{SolveUpper}(\ A_{10}, L_{10}, U_{00}\ );$
- $\textsc{Multiply}(\ -1, L_{10}, U_{01}, A_{11}\ );$
- $\mathrm{LU}(\ A_{11}, L_{11}, U_{11}\ );$

Classical $\mathcal{H}$-LU Algorithm

The above equations directly translate into an algorithm for the $\mathcal{H}$-LU factorisation and matrix solves:

procedure $\mathrm{LU}(A, L, U)$
  $\mathrm{LU}(\ A_{00}, L_{00}, U_{00}\ )$;
  $\textsc{SolveLower}(\ A_{01}, L_{00}, U_{01}\ )$;
  $\textsc{SolveUpper}(\ A_{10}, L_{10}, U_{00}\ )$;
  $\textsc{Multiply}(\ -1, L_{10}, U_{01}, A_{11}\ )$;
  $\mathrm{LU}(\ A_{11}, L_{11}, U_{11}\ )$;

procedure $\textsc{SolveLower}(A, L, B)$
  $\textsc{SolveLower}(\ A_{00}, L_{00}, B_{00}\ )$;
  $\textsc{SolveLower}(\ A_{01}, L_{00}, B_{01}\ )$;
  $\textsc{Multiply}(\ -1, L_{10}, B_{00}, A_{11}\ )$;
  $\textsc{Multiply}(\ -1, L_{10}, B_{01}, A_{11}\ )$;
  $\textsc{SolveLower}(\ A_{10}, L_{11}, B_{10}\ )$;
  $\textsc{SolveLower}(\ A_{11}, L_{11}, B_{11}\ )$;

Classical $\mathcal{H}$-LU Algorithm

The above equations directly translate into an algorithm for the $\mathcal{H}$-LU factorisation and matrix solves:

procedure $\text{LU}(A, L, U)$
  $\text{LU}(\ A_{00}, L_{00}, U_{00}\ )$;
  $\text{SolveLower}(\ A_{01}, L_{00}, U_{01}\ )$;
  $\text{SolveUpper}(\ A_{10}, L_{10}, U_{00}\ )$;
  $\text{Multiply}(\ -1, L_{10}, U_{01}, A_{11}\ )$;
  $\text{LU}(\ A_{11}, L_{11}, U_{11}\ )$;

procedure $\text{SolveLower}(A, L, B)$
  $\text{SolveLower}(\ A_{00}, L_{00}, B_{00}\ )$;
  $\text{SolveLower}(\ A_{01}, L_{00}, B_{01}\ )$;
  $\text{Multiply}(\ -1, L_{10}, B_{00}, A_{11}\ )$;
  $\text{Multiply}(\ -1, L_{10}, B_{01}, A_{11}\ )$;
  $\text{SolveLower}(\ A_{10}, L_{11}, B_{10}\ )$;
  $\text{SolveLower}(\ A_{11}, L_{11}, B_{11}\ )$;

Both procedures only consist of *recursion* and *matrix multiplication*.

Only at the level of leaves, specialised algorithms are needed, e.g. factorise dense matrix or solve low-rank matrix.

Parallelisation

The algorithm is by itself inherently *sequential*.

Only the matrix solves may be performed in parallel:

procedure LU(A, L, U)
  LU(A_{00}, L_{00}, U_{00});
  { SolveLower(A_{01}, L_{00}, U_{01}) | solveUpper(A_{10}, L_{10}, U_{00}); }
  Multiply($-1, L_{10}, U_{01}, A_{11}$);
  LU(A_{11}, L_{11}, U_{11});

Parallelisation

The algorithm is by itself inherently *sequential*.

Only the matrix solves may be performed in parallel:

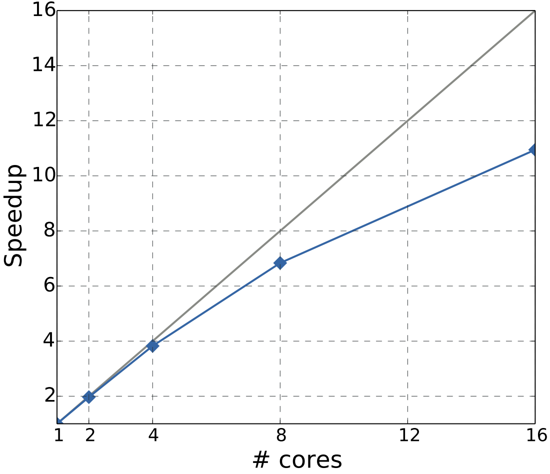
procedure $\mathrm{LU}(A, L, U)$
  $\mathrm{LU}(\ A_{00}, L_{00}, U_{00}\ )$;
  { $\textsc{SolveLower}(\ A_{01}, L_{00}, U_{01}\ )$ | $\textsc{solveUpper}(\ A_{10}, L_{10}, U_{00}\ )$; }
  $\textsc{Multiply}(\ -1, L_{10}, U_{01}, A_{11}\ )$;
  $\mathrm{LU}(\ A_{11}, L_{11}, U_{11}\ )$;

Matrix solve algorithm can be parallelised only slightly better:

procedure $\textsc{SolveLower}(A, L, B)$
  { $\textsc{SolveLower}(\ A_{00}, L_{00}, B_{00}\ )$; | $\textsc{SolveLower}(\ A_{01}, L_{00}, B_{01}\ )$; }
  { $\textsc{Multiply}(\ -1, L_{10}, B_{00}, A_{10}\ )$; | $\textsc{Multiply}(\ -1, L_{10}, B_{01}, A_{11}\ )$; }
  { $\textsc{SolveLower}(\ A_{10}, L_{11}, B_{10}\ )$; | $\textsc{SolveLower}(\ A_{11}, L_{11}, B_{11}\ )$; }

Numerical Results

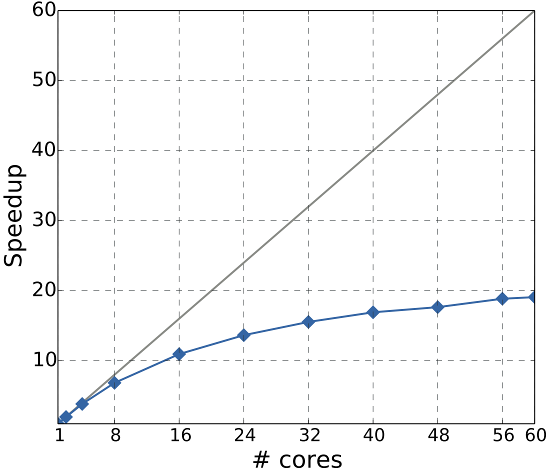
Parallel speedup for the $\mathcal{H}$-LU factorisation of the $\mathcal{H}$-matrix defined by the Laplace SLP on the unit sphere:



(XeonPhi 5110P)

Numerical Results

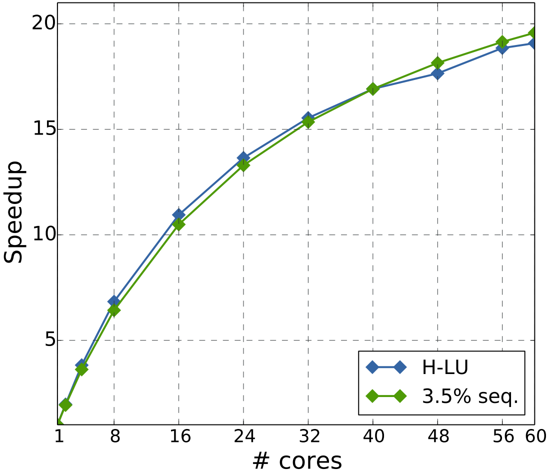
Parallel speedup for the $\mathcal{H}$-LU factorisation of the $\mathcal{H}$-matrix defined by the Laplace SLP on the unit sphere:



(XeonPhi 5110P)

Numerical Results

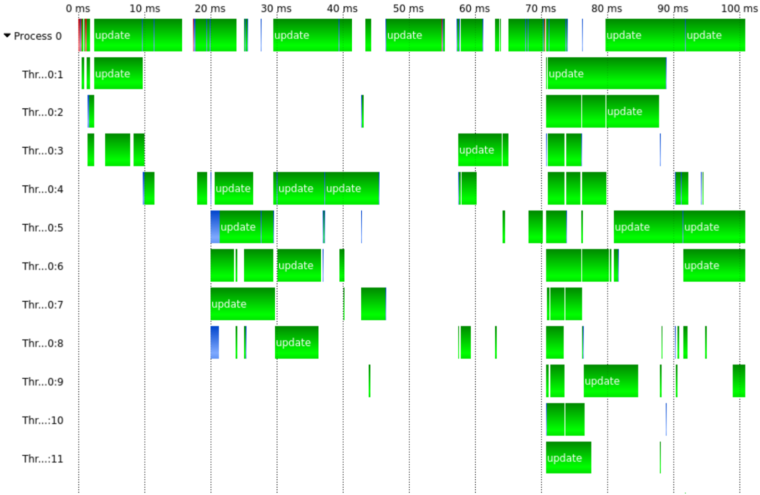
Parallel speedup for the $\mathcal{H}$-LU factorisation of the $\mathcal{H}$-matrix defined by the Laplace SLP on the unit sphere:



(XeonPhi 5110P)

Numerical Results

Function trace of $\mathcal{H}$-LU factorisation:



(Xeon E5-2640)

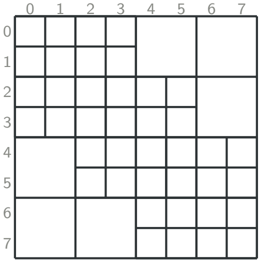
Task-based $\mathcal{H}$-LU Factorisation

$\mathcal{H}$-LU Factorisation Tasks

The equations

$$A_{00} = L_{00}U_{00} \qquad A_{11} = L_{10}U_{01} + L_{11}U_{11}$$
$$A_{01} = L_{00}U_{01} \qquad A_{10} = L_{10}U_{00}$$

define the computations on a per-block level. After recursion, this defines all tasks of the computation:



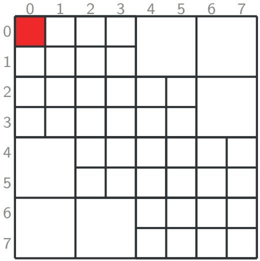
$$\begin{aligned}
A_{\{0\}\times\{0\}} &= L_{\{0\}\times\{0\}}U_{\{0\}\times\{0\}} \\
A_{\{1\}\times\{0\}} &= L_{\{1\}\times\{0\}}U_{\{0\}\times\{0\}} \quad \cdots \\
A_{\{0\}\times\{1\}} &= L_{\{0\}\times\{0\}}U_{\{0\}\times\{1\}} \quad \cdots \\
A_{\{4,5\}\times\{0,1\}} &= L_{\{4,5\}\times\{0,1\}}U_{\{0,1\}\times\{0,1\}} \quad \cdots \\
&\vdots \\
A_{\{2,3\}\times\{6,7\}} &= L_{\{2,3\}\times\{2,3\}}U_{\{2,3\}\times\{6,7\}} \\
&\vdots \\
A_{\{7\}\times\{7\}} &= L_{\{7\}\times\{7\}}U_{\{7\}\times\{7\}}
\end{aligned}$$

$\mathcal{H}$-LU Factorisation Tasks

The equations

$$A_{00} = L_{00}U_{00} \qquad A_{11} = L_{10}U_{01} + L_{11}U_{11}$$
$$A_{01} = L_{00}U_{01} \qquad A_{10} = L_{10}U_{00}$$

define the computations on a per-block level. After recursion, this defines all tasks of the computation:



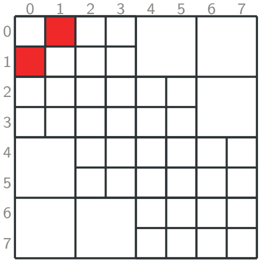
$$\begin{aligned}
A_{\{0\}\times\{0\}} &= L_{\{0\}\times\{0\}}U_{\{0\}\times\{0\}} \\
A_{\{1\}\times\{0\}} &= L_{\{1\}\times\{0\}}U_{\{0\}\times\{0\}} \quad \cdots \\
A_{\{0\}\times\{1\}} &= L_{\{0\}\times\{0\}}U_{\{0\}\times\{1\}} \quad \cdots \\
A_{\{4,5\}\times\{0,1\}} &= L_{\{4,5\}\times\{0,1\}}U_{\{0,1\}\times\{0,1\}} \quad \cdots \\
&\vdots \\
A_{\{2,3\}\times\{6,7\}} &= L_{\{2,3\}\times\{2,3\}}U_{\{2,3\}\times\{6,7\}} \\
&\vdots \\
A_{\{7\}\times\{7\}} &= L_{\{7\}\times\{7\}}U_{\{7\}\times\{7\}}
\end{aligned}$$

$\mathcal{H}$-LU Factorisation Tasks

The equations

$$A_{00} = L_{00}U_{00} \qquad A_{11} = L_{10}U_{01} + L_{11}U_{11}$$
$$A_{01} = L_{00}U_{01} \qquad A_{10} = L_{10}U_{00}$$

define the computations on a per-block level. After recursion, this defines all tasks of the computation:



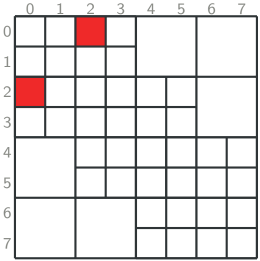
$$\begin{array}{rcll}
A_{\{0\}\times\{0\}} & = & L_{\{0\}\times\{0\}}U_{\{0\}\times\{0\}} & \\
A_{\{1\}\times\{0\}} & = & L_{\{1\}\times\{0\}}U_{\{0\}\times\{0\}} & \cdots \\
A_{\{0\}\times\{1\}} & = & L_{\{0\}\times\{0\}}U_{\{0\}\times\{1\}} & \cdots \\
A_{\{4,5\}\times\{0,1\}} & = & L_{\{4,5\}\times\{0,1\}}U_{\{0,1\}\times\{0,1\}} & \cdots \\
& \vdots & & \\
A_{\{2,3\}\times\{6,7\}} & = & L_{\{2,3\}\times\{2,3\}}U_{\{2,3\}\times\{6,7\}} & \\
& \vdots & & \\
A_{\{7\}\times\{7\}} & = & L_{\{7\}\times\{7\}}U_{\{7\}\times\{7\}} &
\end{array}$$

$\mathcal{H}$-LU Factorisation Tasks

The equations

$$A_{00} = L_{00}U_{00} \qquad A_{11} = L_{10}U_{01} + L_{11}U_{11}$$
$$A_{01} = L_{00}U_{01} \qquad A_{10} = L_{10}U_{00}$$

define the computations on a per-block level. After recursion, this defines all tasks of the computation:



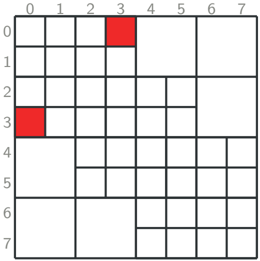
$$\begin{aligned}
A_{\{0\}\times\{0\}} &= L_{\{0\}\times\{0\}}U_{\{0\}\times\{0\}} \\
\color{red}{A_{\{1\}\times\{0\}}} &= \color{red}{L_{\{1\}\times\{0\}}U_{\{0\}\times\{0\}}} \quad \cdots \\
\color{red}{A_{\{0\}\times\{1\}}} &= \color{red}{L_{\{0\}\times\{0\}}U_{\{0\}\times\{1\}}} \quad \cdots \\
A_{\{4,5\}\times\{0,1\}} &= L_{\{4,5\}\times\{0,1\}}U_{\{0,1\}\times\{0,1\}} \quad \cdots \\
&\vdots \\
A_{\{2,3\}\times\{6,7\}} &= L_{\{2,3\}\times\{2,3\}}U_{\{2,3\}\times\{6,7\}} \\
&\vdots \\
A_{\{7\}\times\{7\}} &= L_{\{7\}\times\{7\}}U_{\{7\}\times\{7\}}
\end{aligned}$$

$\mathcal{H}$-LU Factorisation Tasks

The equations

$$A_{00} = L_{00}U_{00} \qquad A_{11} = L_{10}U_{01} + L_{11}U_{11}$$
$$A_{01} = L_{00}U_{01} \qquad A_{10} = L_{10}U_{00}$$

define the computations on a per-block level. After recursion, this defines all tasks of the computation:



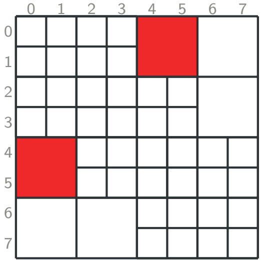
$$\begin{aligned}
A_{\{0\}\times\{0\}} &= L_{\{0\}\times\{0\}}U_{\{0\}\times\{0\}} \\
A_{\{1\}\times\{0\}} &= L_{\{1\}\times\{0\}}U_{\{0\}\times\{0\}} \quad \cdots \\
A_{\{0\}\times\{1\}} &= L_{\{0\}\times\{0\}}U_{\{0\}\times\{1\}} \quad \cdots \\
A_{\{4,5\}\times\{0,1\}} &= L_{\{4,5\}\times\{0,1\}}U_{\{0,1\}\times\{0,1\}} \quad \cdots \\
&\vdots \\
A_{\{2,3\}\times\{6,7\}} &= L_{\{2,3\}\times\{2,3\}}U_{\{2,3\}\times\{6,7\}} \\
&\vdots \\
A_{\{7\}\times\{7\}} &= L_{\{7\}\times\{7\}}U_{\{7\}\times\{7\}}
\end{aligned}$$

$\mathcal{H}$-LU Factorisation Tasks

The equations

$$A_{00} = L_{00}U_{00} \qquad A_{11} = L_{10}U_{01} + L_{11}U_{11}$$
$$A_{01} = L_{00}U_{01} \qquad A_{10} = L_{10}U_{00}$$

define the computations on a per-block level. After recursion, this defines all tasks of the computation:



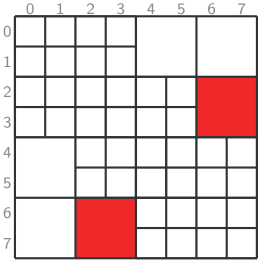
$$
\begin{aligned}
A_{\{0\}\times\{0\}} &= L_{\{0\}\times\{0\}}U_{\{0\}\times\{0\}} \\
A_{\{1\}\times\{0\}} &= L_{\{1\}\times\{0\}}U_{\{0\}\times\{0\}} \quad \cdots \\
A_{\{0\}\times\{1\}} &= L_{\{0\}\times\{0\}}U_{\{0\}\times\{1\}} \quad \cdots \\
A_{\{4,5\}\times\{0,1\}} &= L_{\{4,5\}\times\{0,1\}}U_{\{0,1\}\times\{0,1\}} \quad \cdots \\
&\vdots \\
A_{\{2,3\}\times\{6,7\}} &= L_{\{2,3\}\times\{2,3\}}U_{\{2,3\}\times\{6,7\}} \\
&\vdots \\
A_{\{7\}\times\{7\}} &= L_{\{7\}\times\{7\}}U_{\{7\}\times\{7\}}
\end{aligned}
$$

$\mathcal{H}$-LU Factorisation Tasks

The equations

$$A_{00} = L_{00}U_{00} \qquad A_{11} = L_{10}U_{01} + L_{11}U_{11}$$
$$A_{01} = L_{00}U_{01} \qquad A_{10} = L_{10}U_{00}$$

define the computations on a per-block level. After recursion, this defines all tasks of the computation:



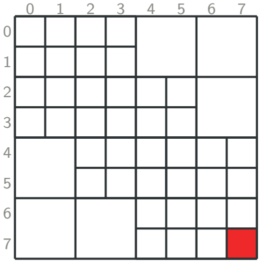
$$\begin{aligned}
A_{\{0\}\times\{0\}} &= L_{\{0\}\times\{0\}}U_{\{0\}\times\{0\}} \\
A_{\{1\}\times\{0\}} &= L_{\{1\}\times\{0\}}U_{\{0\}\times\{0\}} \quad \cdots \\
A_{\{0\}\times\{1\}} &= L_{\{0\}\times\{0\}}U_{\{0\}\times\{1\}} \quad \cdots \\
A_{\{4,5\}\times\{0,1\}} &= L_{\{4,5\}\times\{0,1\}}U_{\{0,1\}\times\{0,1\}} \quad \cdots \\
&\vdots \\
A_{\{2,3\}\times\{6,7\}} &= L_{\{2,3\}\times\{2,3\}}U_{\{2,3\}\times\{6,7\}} \\
&\vdots \\
A_{\{7\}\times\{7\}} &= L_{\{7\}\times\{7\}}U_{\{7\}\times\{7\}}
\end{aligned}$$

$\mathcal{H}$-LU Factorisation Tasks

The equations

$$A_{00} = L_{00}U_{00} \qquad A_{11} = L_{10}U_{01} + L_{11}U_{11}$$
$$A_{01} = L_{00}U_{01} \qquad A_{10} = L_{10}U_{00}$$

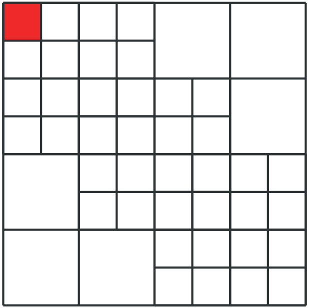
define the computations on a per-block level. After recursion, this defines all tasks of the computation:



$$
\begin{aligned}
A_{\{0\}\times\{0\}} &= L_{\{0\}\times\{0\}}U_{\{0\}\times\{0\}} \\
A_{\{1\}\times\{0\}} &= L_{\{1\}\times\{0\}}U_{\{0\}\times\{0\}} \quad \cdots \\
A_{\{0\}\times\{1\}} &= L_{\{0\}\times\{0\}}U_{\{0\}\times\{1\}} \quad \cdots \\
A_{\{4,5\}\times\{0,1\}} &= L_{\{4,5\}\times\{0,1\}}U_{\{0,1\}\times\{0,1\}} \quad \cdots \\
&\vdots \\
A_{\{2,3\}\times\{6,7\}} &= L_{\{2,3\}\times\{2,3\}}U_{\{2,3\}\times\{6,7\}} \\
&\vdots \\
A_{\{7\}\times\{7\}} &= L_{\{7\}\times\{7\}}U_{\{7\}\times\{7\}}
\end{aligned}
$$

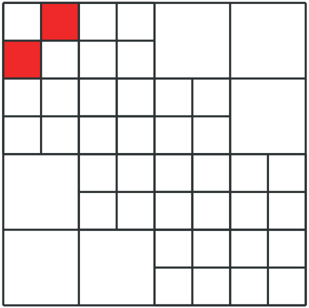
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



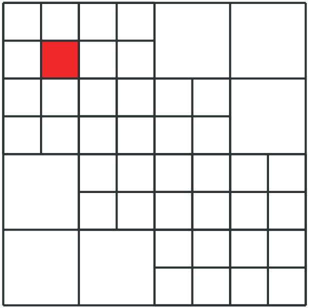
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



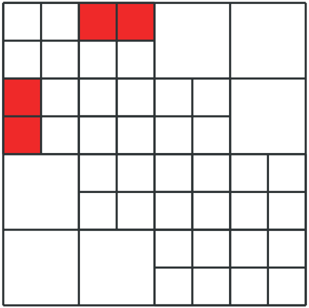
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



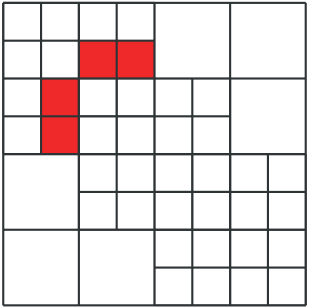
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



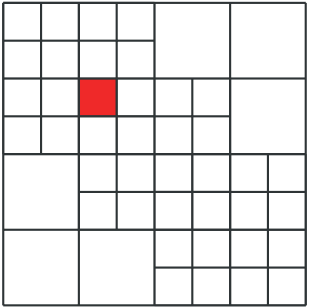
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



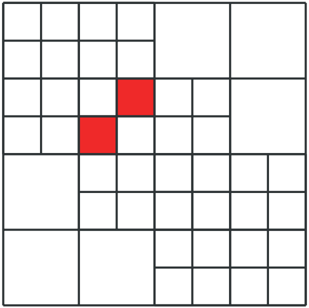
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



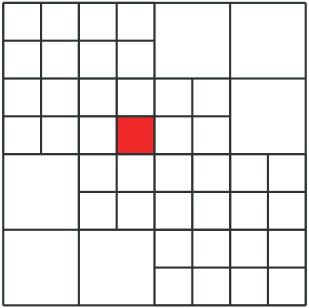
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



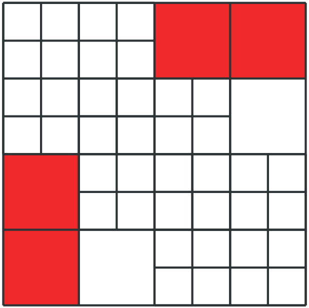
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



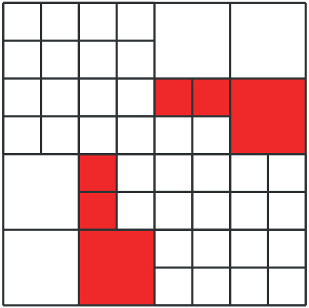
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



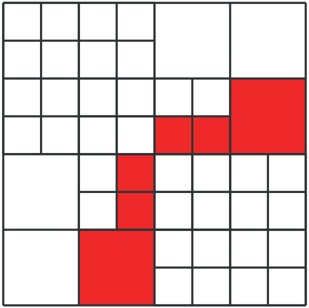
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



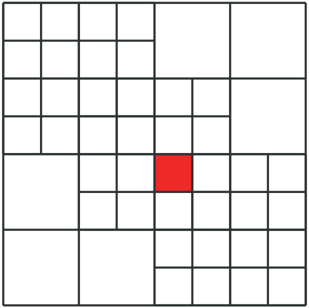
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



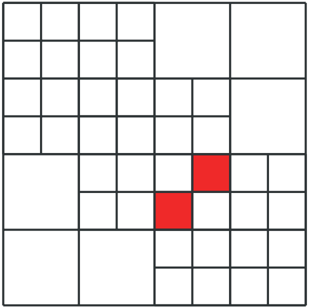
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



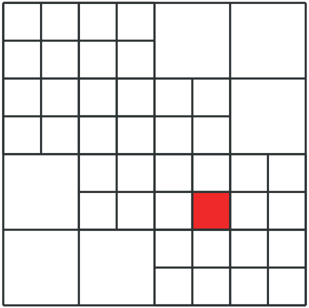
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



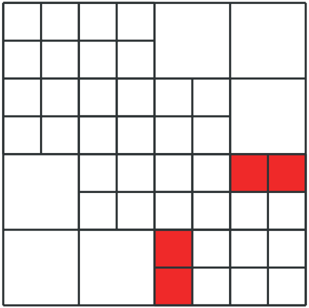
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



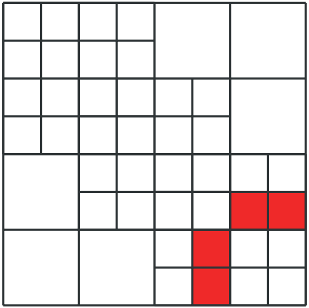
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



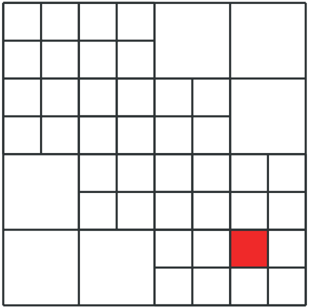
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



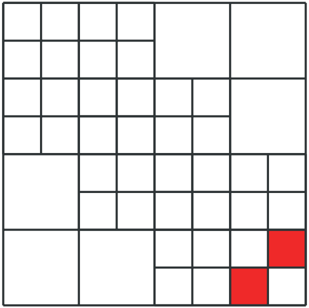
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



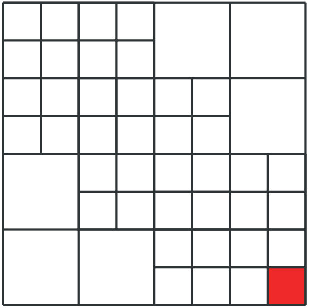
Task Execution Order

Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



Task Execution Order

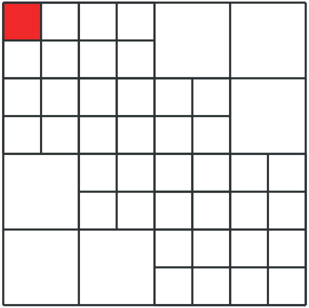
Using the classical recursive $\mathcal{H}$-LU algorithm, those tasks are processed in a *localised* execution order with *single* task execution on the diagonal:



To handle all tasks, *19 steps* are needed.

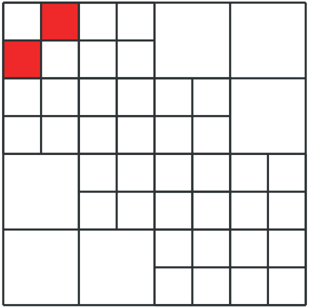
Task Execution Order

An optimal execution order only needs *15* steps and diagonal tasks can be executed *simultaneously* with off-diagonal tasks:



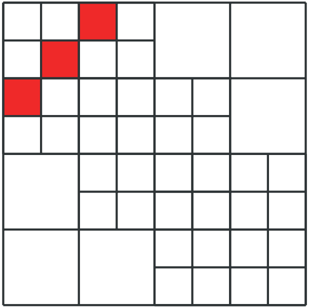
Task Execution Order

An optimal execution order only needs *15* steps and diagonal tasks can be executed *simultaneously* with off-diagonal tasks:



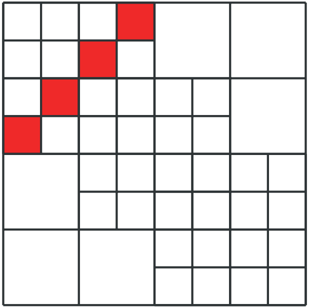
Task Execution Order

An optimal execution order only needs *15* steps and diagonal tasks can be executed *simultaneously* with off-diagonal tasks:



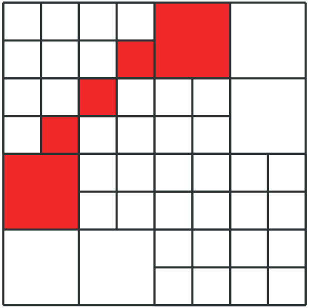
Task Execution Order

An optimal execution order only needs *15* steps and diagonal tasks can be executed *simultaneously* with off-diagonal tasks:



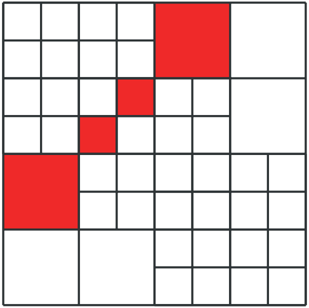
Task Execution Order

An optimal execution order only needs *15* steps and diagonal tasks can be executed *simultaneously* with off-diagonal tasks:



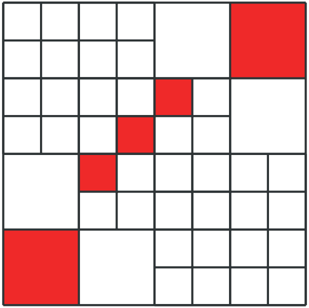
Task Execution Order

An optimal execution order only needs *15* steps and diagonal tasks can be executed *simultaneously* with off-diagonal tasks:



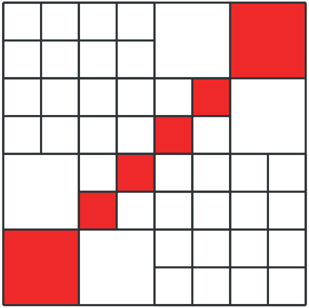
Task Execution Order

An optimal execution order only needs *15* steps and diagonal tasks can be executed *simultaneously* with off-diagonal tasks:



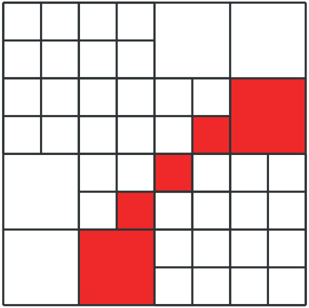
Task Execution Order

An optimal execution order only needs *15* steps and diagonal tasks can be executed *simultaneously* with off-diagonal tasks:



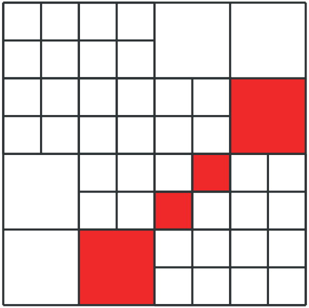
Task Execution Order

An optimal execution order only needs *15* steps and diagonal tasks can be executed *simultaneously* with off-diagonal tasks:



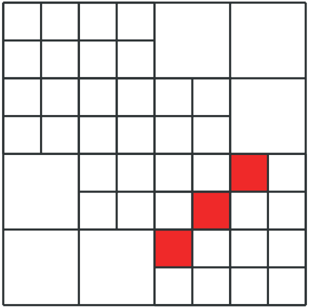
Task Execution Order

An optimal execution order only needs *15* steps and diagonal tasks can be executed *simultaneously* with off-diagonal tasks:



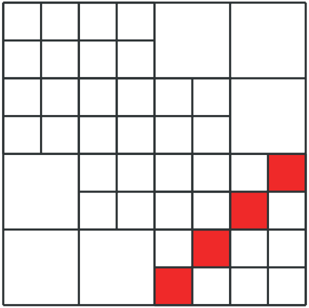
Task Execution Order

An optimal execution order only needs *15* steps and diagonal tasks can be executed *simultaneously* with off-diagonal tasks:



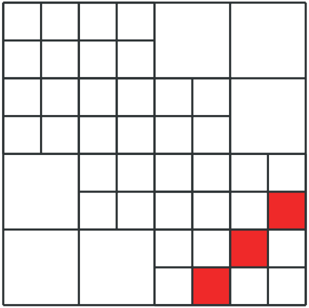
Task Execution Order

An optimal execution order only needs *15* steps and diagonal tasks can be executed *simultaneously* with off-diagonal tasks:



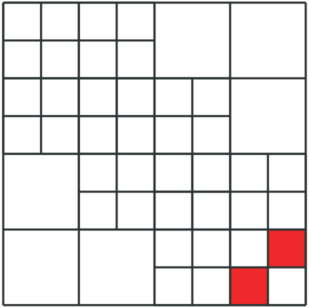
Task Execution Order

An optimal execution order only needs *15* steps and diagonal tasks can be executed *simultaneously* with off-diagonal tasks:



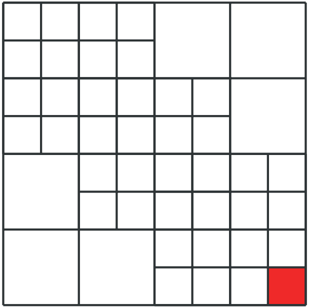
Task Execution Order

An optimal execution order only needs *15* steps and diagonal tasks can be executed *simultaneously* with off-diagonal tasks:



Task Execution Order

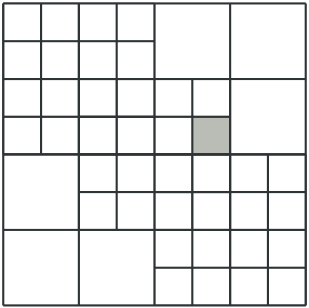
An optimal execution order only needs *15* steps and diagonal tasks can be executed *simultaneously* with off-diagonal tasks:



For the 46 tasks in the example, the parallel speedup is increased from $\frac{46}{19} \approx 2.42$ to $\frac{46}{15} \approx 3.07$ (not counting update tasks).

Task Dependencies

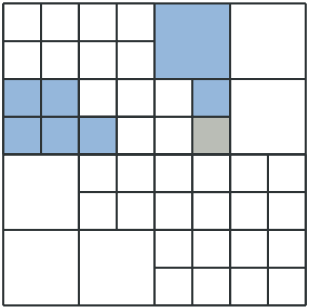
The equations of the $\mathcal{H}$-LU factorisation also define *data dependencies* between matrix blocks



Task Dependencies

The equations of the $\mathcal{H}$-LU factorisation also define *data dependencies* between matrix blocks, e.g.

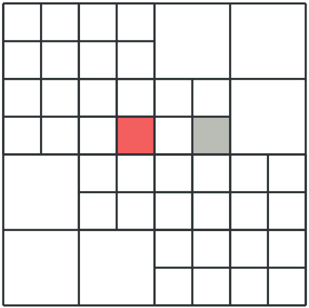
- factorise or solve matrix blocks after applying all updates,



Task Dependencies

The equations of the $\mathcal{H}$-LU factorisation also define *data dependencies* between matrix blocks, e.g.

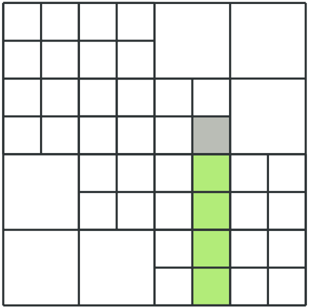
- factorise or solve matrix blocks after applying all updates,
- solve off-diagonal blocks after diagonal factorisation, and



Task Dependencies

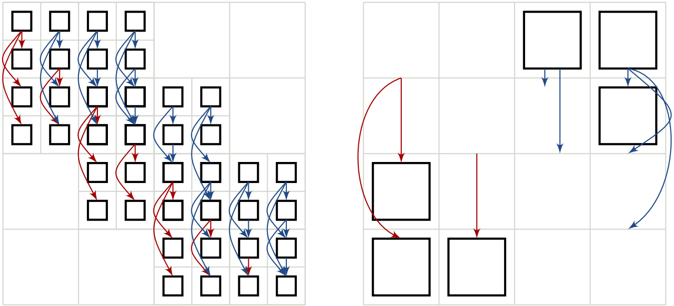
The equations of the $\mathcal{H}$-LU factorisation also define *data dependencies* between matrix blocks, e.g.

- factorise or solve matrix blocks after applying all updates,
- solve off-diagonal blocks after diagonal factorisation, and
- perform matrix updates after matrix solves.



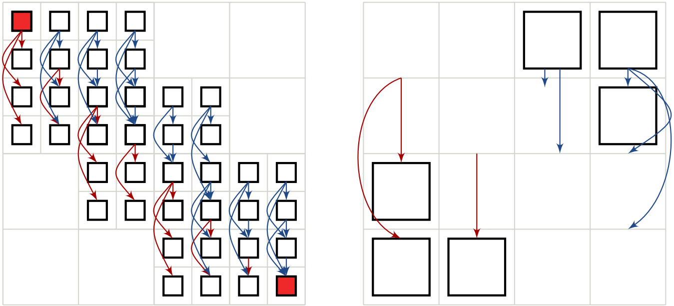
Task Dependencies

The tasks and their dependencies can also be represented in the form of a *directed acyclic graph* (*DAG*) with tasks as nodes and dependencies as edges:



Task Dependencies

The tasks and their dependencies can also be represented in the form of a *directed acyclic graph* (*DAG*) with tasks as nodes and dependencies as edges:

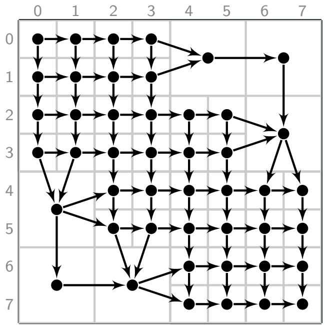


The *start node* of this DAG is the upper left matrix block, while the *end node* is the lower left matrix block.

DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

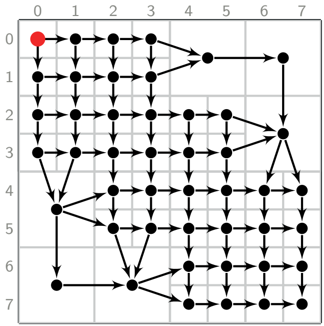
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

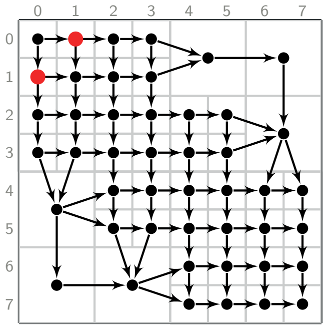
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

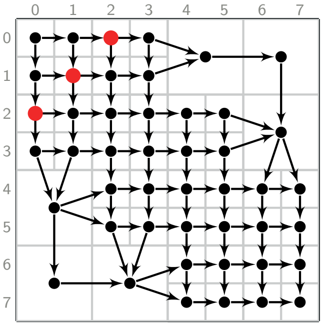
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

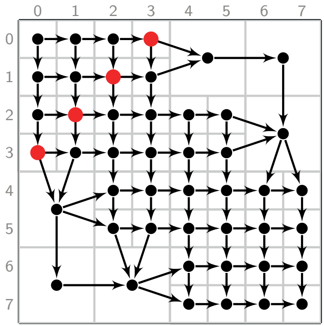
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

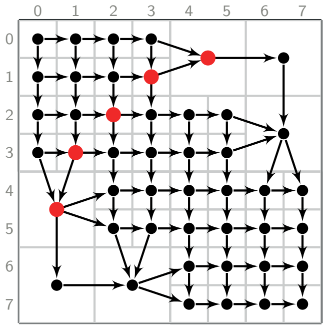
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

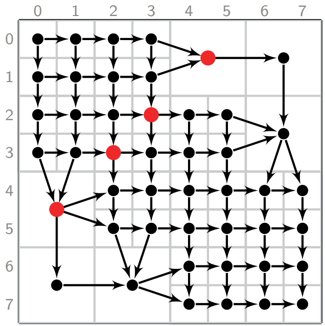
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

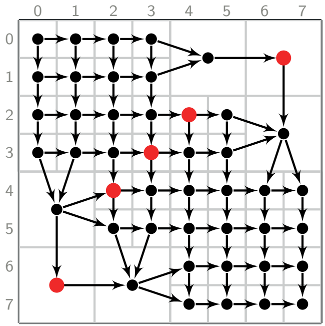
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

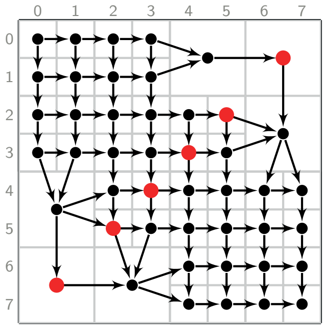
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

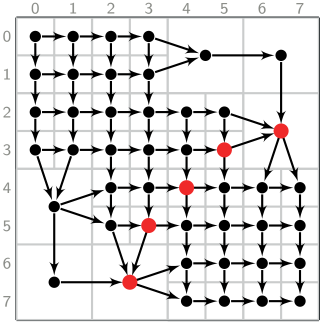
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

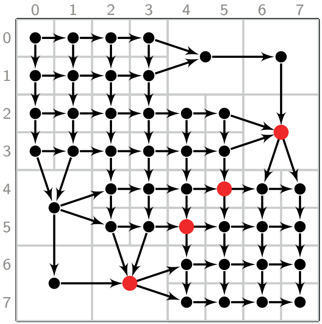
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

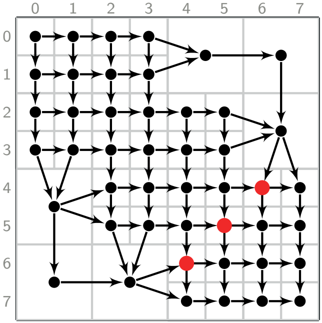
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

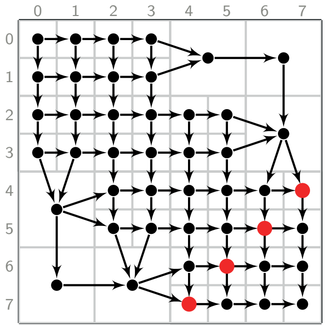
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

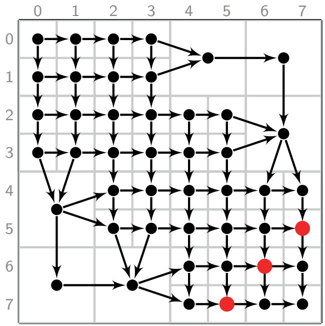
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

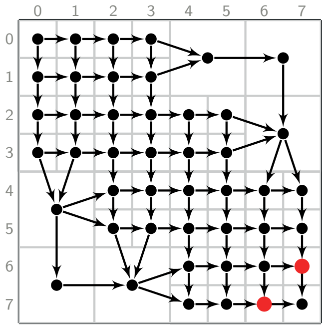
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

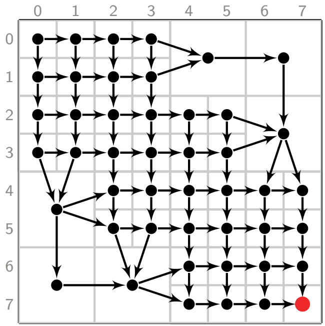
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

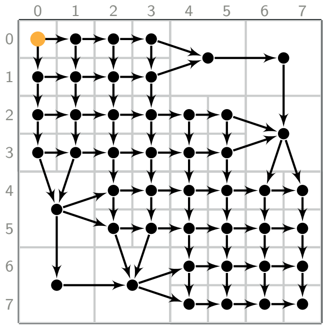
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

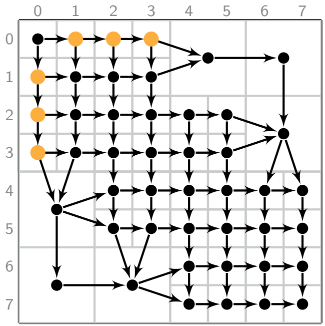
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

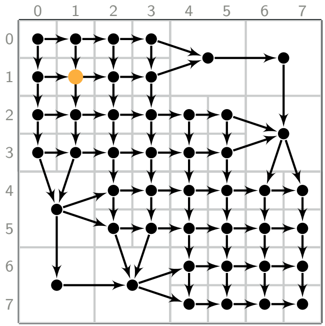
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

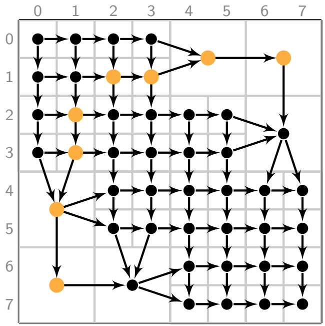
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

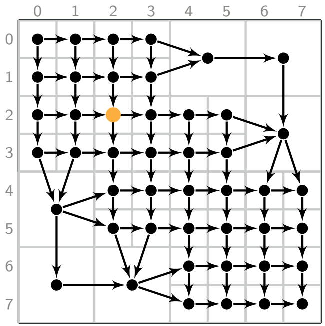
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

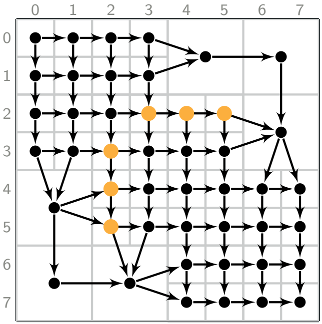
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

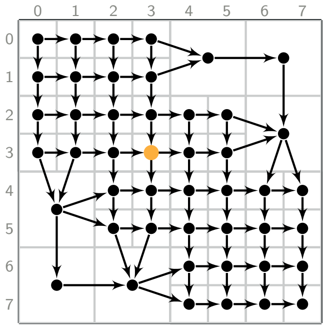
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

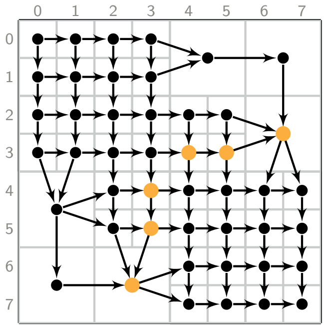
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

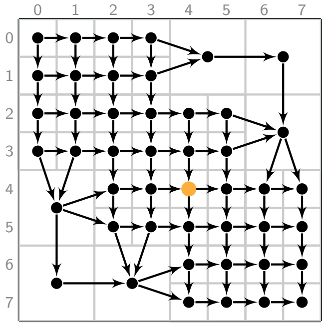
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

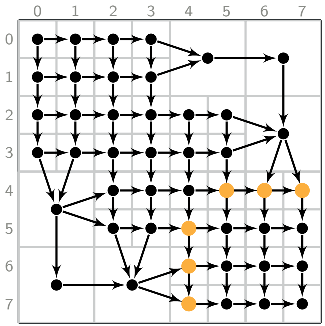
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

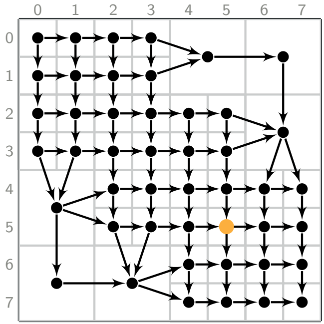
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

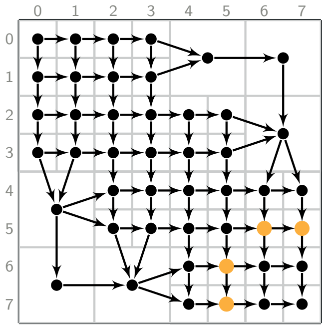
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

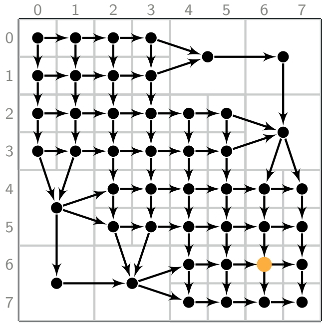
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

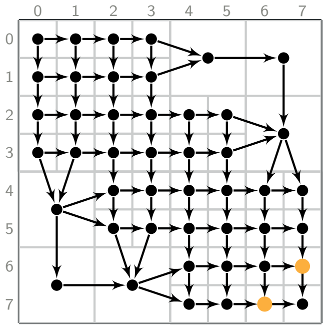
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

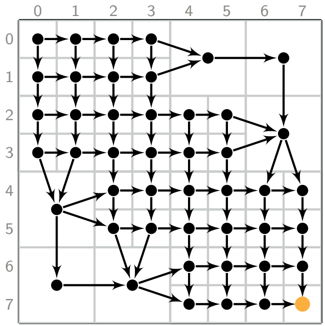
Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



DAG Execution

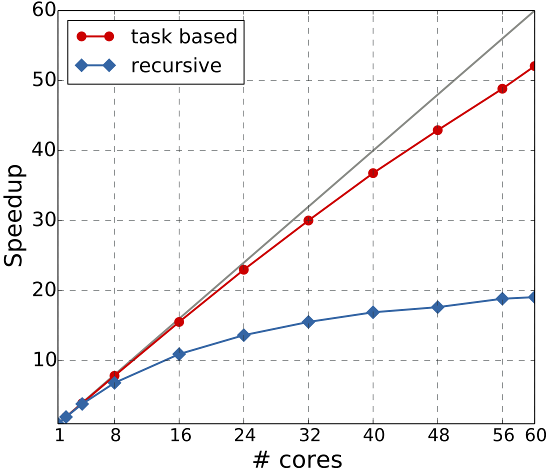
As with implicit task dependencies, nodes in a DAG are scheduled for execution, when all dependencies are met, i.e. predecessor tasks have finished.

Equivalently: all nodes with the same *maximal* distance from the start node may be executed in parallel.



Numerical Results

Again, the $\mathcal{H}$-LU factorisation of the Laplace SLP operator is computed. The speedup of the task based algorithm is:



(XeonPhi 5110P)

Numerical Results

Function trace of $\mathcal{H}$-LU factorisation:

(Xeon E5-2640)

Numerical Results

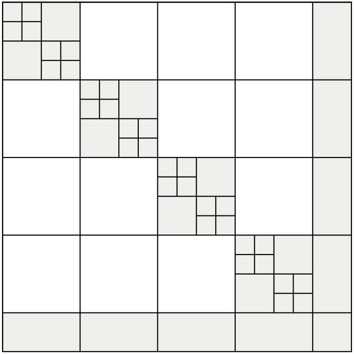
Function trace of $\mathcal{H}$-LU factorisation:

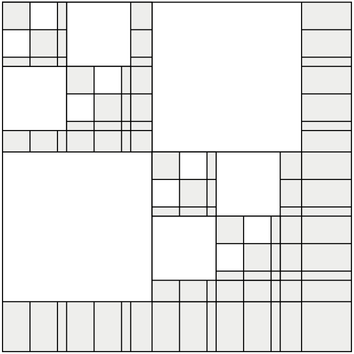
(Xeon E5-2640)

Domain-Decomposition

Domain-Decomposition

If domain decomposition or nested dissection is applied, $\mathcal{H}$-matrices have large, zero, off-diagonal blocks:

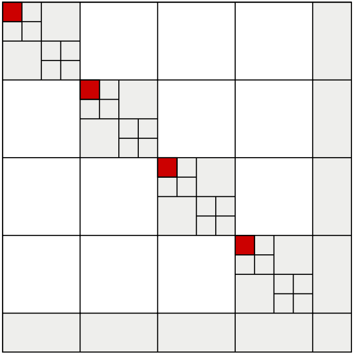


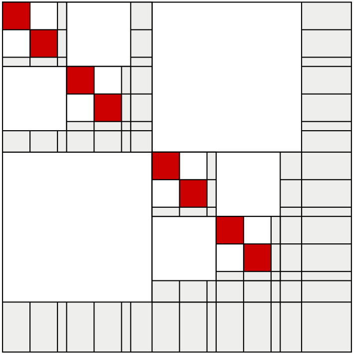


During LU factorisation, these blocks will remain zero, resulting in a higher level of parallelism.

Domain-Decomposition

The task-based $\mathcal{H}$-LU factorisation algorithm *automatically* exploits this parallelism by using *several* start nodes in the DAG:

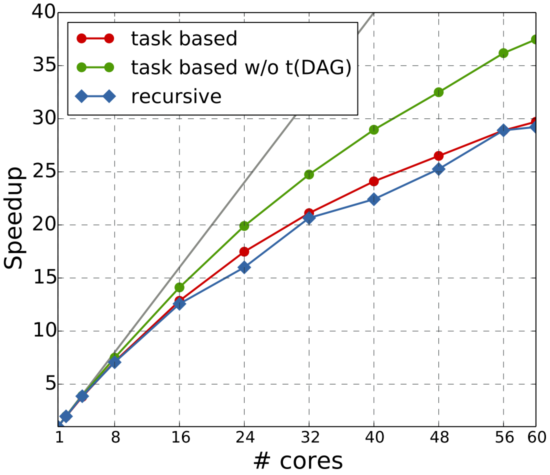




The parallel speedup of the recursive $\mathcal{H}$-LU algorithm is limited by the size of the interface.

Numerical Results

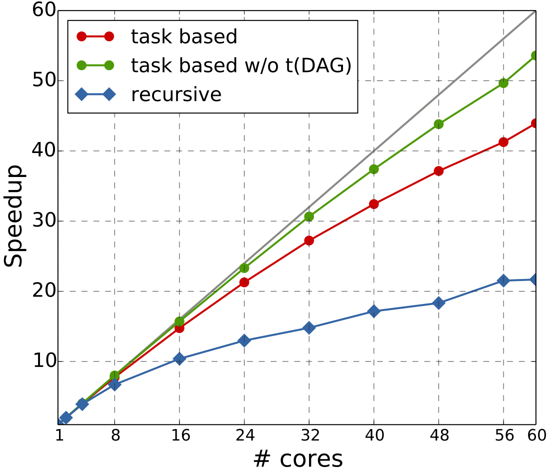
$\mathcal{H}$-LU factorisation for convection-diffusion equation in $\mathbb{R}^2$:



(XeonPhi 5110P)

Numerical Results

$\mathcal{H}$-LU factorisation for convection-diffusion equation in $\mathbb{R}^3$:



(XeonPhi 5110P)

Literature

Literature

R. Kriemann,
$\mathcal{H}$-LU Factorization on Many-Core Systems,
MIS Preprint, 5/2014.

R. Kriemann,
Parallel $\mathcal{H}$-Matrix Arithmetics on Shared Memory Systems,
Computing, 74:273–297, 2005.