$\mathcal{H}$-Matrix Parallelisation

Ronald Kriemann

Max Planck Institute for Mathematics
in the
Sciences Leipzig

Winterschool on $\mathcal{H}$-Matrices
2009

Overview

Introduction

Introduction

Problem

- Matrix construction is very expensive for BEM problems,
- solving equation systems often needs preconditioning, but $\mathcal{H}$-LU factorisation is also expensive.

Introduction

Problem

- Matrix construction is very expensive for BEM problems,
- solving equation systems often needs preconditioning, but $\mathcal{H}$-LU factorisation is also expensive.

Goal

Exploit parallel capabilities of PCs or workstations to accelerate $\mathcal{H}$-arithmetics.

Introduction

Problem

- Matrix construction is very expensive for BEM problems,
- solving equation systems often needs preconditioning, but $\mathcal{H}$-LU factorisation is also expensive.

Goal

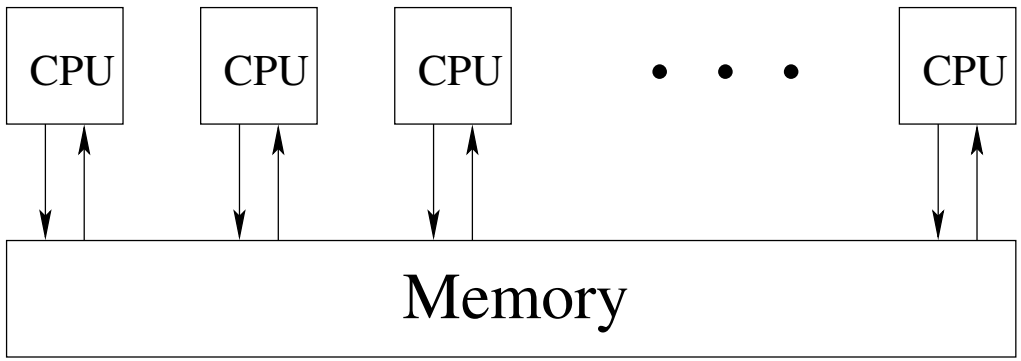
Exploit parallel capabilities of PCs or workstations to accelerate $\mathcal{H}$-arithmetics.

Conditions

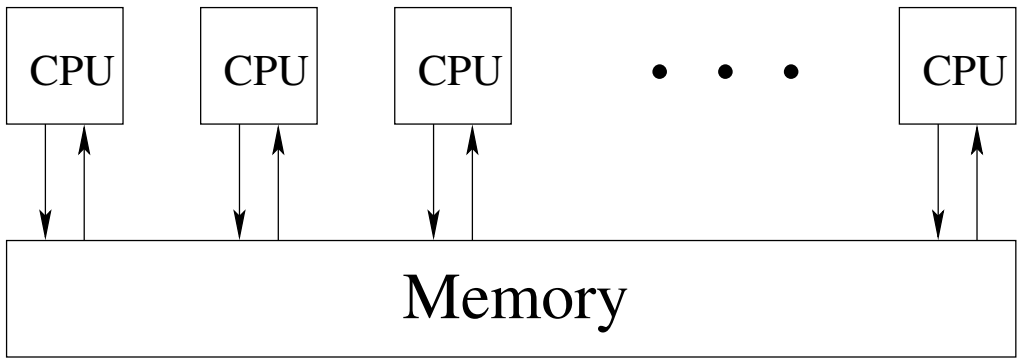
- parallelise only for few processors,
- recycle existing algorithm,
- achieve high parallel speedup

Technical Prerequisites

Workstations and small compute servers are usually systems with a shared memory, e.g. all p processors have direct access to the same memory:



Workstations and small compute servers are usually systems with a shared memory, e.g. all p processors have direct access to the same memory:



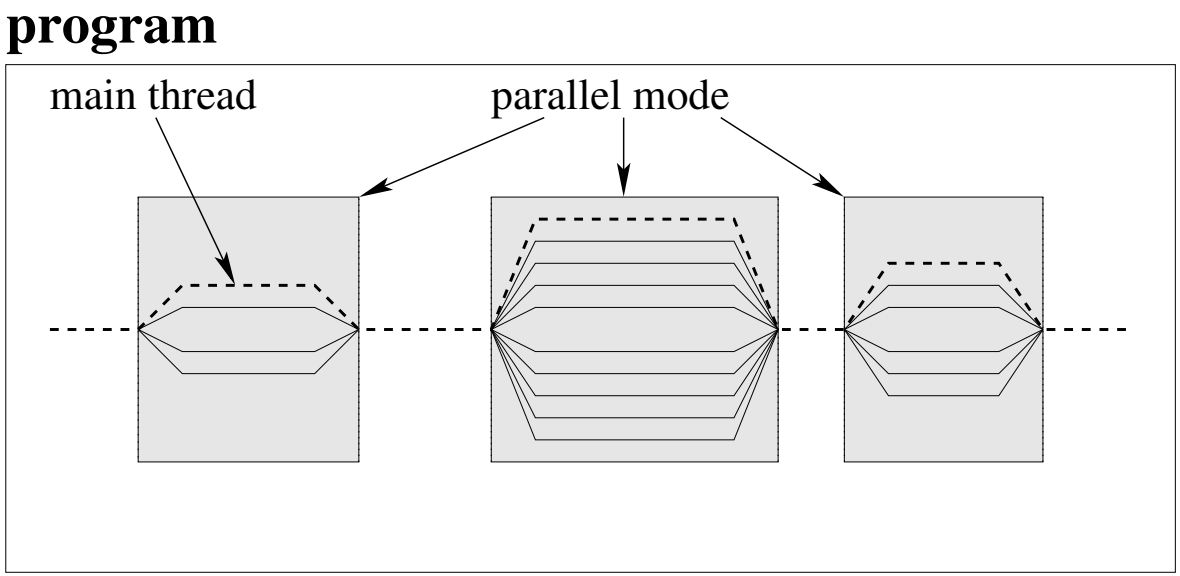
Consequences

- simplified programming because no communications involved,
- but protection of critical resources necessary, e.g. simultaneous access to the same memory position

The standard parallelisation tool on shared memory systems are threads, i.e. parallel computation paths in a program. All threads can read and write all memory used by the program.

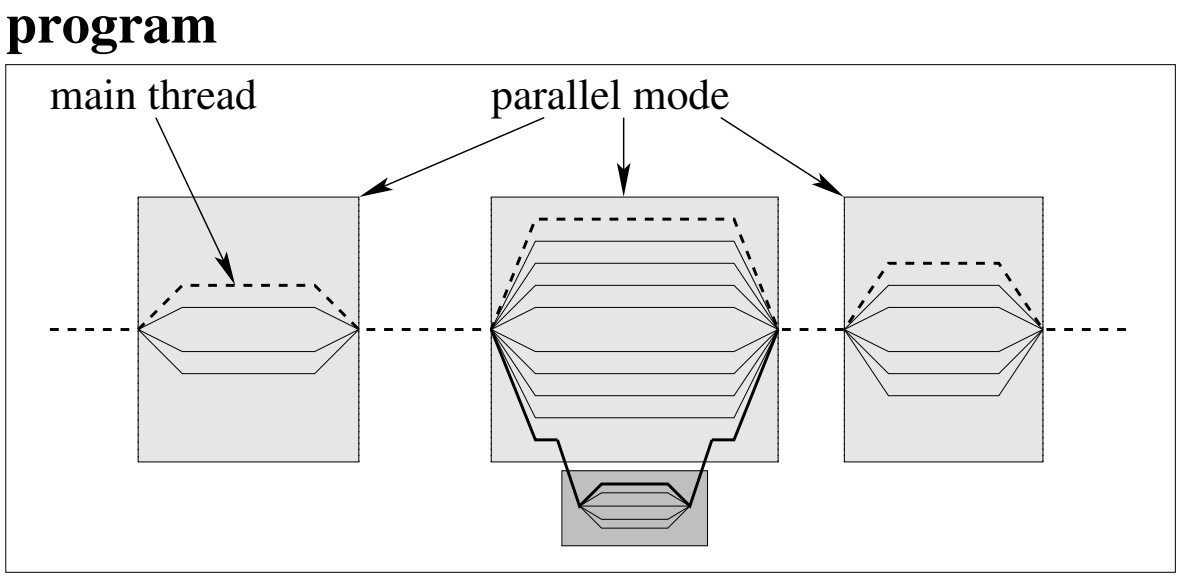
The standard parallelisation tool on shared memory systems are threads, i.e. parallel computation paths in a program. All threads can read and write all memory used by the program.

Each program has a main thread, e.g. the `main` function in a C program. Afterwards, new threads can be started, e.g:



The standard parallelisation tool on shared memory systems are threads, i.e. parallel computation paths in a program. All threads can read and write all memory used by the program.

Each program has a main thread, e.g. the `main` function in a C program. Afterwards, new threads can be started, e.g:



Also possible is nested parallelism: starting new threads from other threads.

A mutex is a tool for mutual exclusion of critical sections in a program. It is either `LOCKED` or `UNLOCKED`.

Locking an already locked mutex blocks the thread until the mutex is unlocked by another thread.

A mutex is a tool for mutual exclusion of critical sections in a program. It is either `LOCKED` or `UNLOCKED`.

Locking an already locked mutex blocks the thread until the mutex is unlocked by another thread.

Example

Compute $A := A + \sum_{i=1}^{4} A_i$, with matrices A and $A_i, i = 1, \ldots, 4$:

On thread 1:

$T_1 := A_1 + A_2;$

$A := A + T_1;$

On thread 2:

$T_2 := A_3 + A_4;$

$A := A + T_2;$

A mutex is a tool for mutual exclusion of critical sections in a program. It is either `LOCKED` or `UNLOCKED`.

Locking an already locked mutex blocks the thread until the mutex is unlocked by another thread.

Example

Compute $A := A + \sum_{i=1}^{4} A_i$, with matrices A and $A_i, i = 1, \ldots, 4$:

On thread 1:

$T_1 := A_1 + A_2$;
lock mutex m;
$A := A + T_1$;
unlock mutex m;

On thread 2:

$T_2 := A_3 + A_4$;
lock mutex m;
$A := A + T_2$;
unlock mutex m;

with shared mutex m.

How to access threads in a program, e.g. how to start new threads and synchronise them via mutices?

How to access threads in a program, e.g. how to start new threads and synchronise them via mutices?

Widely used implementations:

- POSIX threads:
  - \+ powerful, almost everywhere available
  - − complicate interface

How to access threads in a program, e.g. how to start new threads and synchronise them via mutices?

Widely used implementations:

- POSIX threads:
  - \+ powerful, almost everywhere available
  - − complicate interface
- OpenMP:
  - \+ simple interface
  - − mainly developed for loop parallelisation, needs compiler support

Language enhancements for C/C++ and FORTRAN for thread creation, loop parallelisation and synchronisation.

Language enhancements for C/C++ and FORTRAN for thread creation, loop parallelisation and synchronisation.

Example for linear combination of vectors: $y := y + \alpha x$:

```
#pragma omp parallel for
    for ( i = 0; i < n; i++ )
        y[i] := y[i] + alpha * x[i];
```

At the **pragma** directive p threads are started. The loop is automatically parallelised and after finishing, all threads are synchronised. If no OpenMP support is available, the directive is ignored by the compiler.

Language enhancements for C/C++ and FORTRAN for thread creation, loop parallelisation and synchronisation.

Example for linear combination of vectors: $y := y + \alpha x$:

```
#pragma omp parallel for
    for ( i = 0; i < n; i++ )
        y[i] := y[i] + alpha * x[i];
```

At the **pragma** directive p threads are started. The loop is automatically parallelised and after finishing, all threads are synchronised. If no OpenMP support is available, the directive is ignored by the compiler.

OpenMP is supported by all major compiler vendors: GNU, Intel, Sun, Microsoft but with varying degree, e.g. nested parallelism.

Other OpenMP Directives

#pragma omp parallel

Starts p threads executing the following code block.

#pragma omp critical

Provides mutual exclusion, e.g. mutices, for the following code block.

Other OpenMP Directives

#pragma omp parallel

Starts p threads executing the following code block.

#pragma omp critical

Provides mutual exclusion, e.g. mutices, for the following code block.

OpenMP Functions and Types

For explicit mutices, the type `omp_lock_t` is defined by OpenMP with the functions

```
// lock given mutex
void  omp_set_lock    ( omp_lock_t * mutex );
// unlock given mutex
void  omp_unset_lock  ( omp_lock_t * mutex );
```

Matrix Construction

Let $\mathcal{I}$ be an index set with $\#\mathcal{I} = n$, $T(\mathcal{I})$ a cluster tree over $\mathcal{I}$ and $T(\mathcal{I} \times \mathcal{I})$ a block cluster tree over $\mathcal{I} \times \mathcal{I}$.

Build matrix blocks only for leaves in block cluster tree:

for $(t, s) \in \mathcal{L}(T(\mathcal{I} \times \mathcal{I}))$ **do**
    if (t, s) is admissible **then** build low rank matrix;
    else build dense matrix;
endfor;

Let $\mathcal{I}$ be an index set with $\#\mathcal{I} = n$, $T(\mathcal{I})$ a cluster tree over $\mathcal{I}$ and $T(\mathcal{I} \times \mathcal{I})$ a block cluster tree over $\mathcal{I} \times \mathcal{I}$.

Build matrix blocks only for leaves in block cluster tree:

for $(t, s) \in \mathcal{L}(T(\mathcal{I} \times \mathcal{I}))$ **do**
  if (t, s) is admissible **then** build low rank matrix;
  else build dense matrix;
endfor;

Properties:

- all matrix blocks can be built independently,
- $\#\mathcal{L}(T(\mathcal{I} \times \mathcal{I})) \gg p$, e.g. enough to do for each processor and no explicit scheduling necessary.

Apply OpenMP parallelisation directly to loop:

```
leaves = ... // list of leaves
#pragma omp parallel {
    while ( ! is_empty( leaves ) ) {
        // guard change of leaves set by mutex
        #pragma omp critical {
            cluster = head( leaves );
            leaves  = tail( leaves );
        }

        if ( is_adm( cluster ) )
            M = build_lowrank( cluster );
        else
            M = build_dense( cluster );
}   }
```

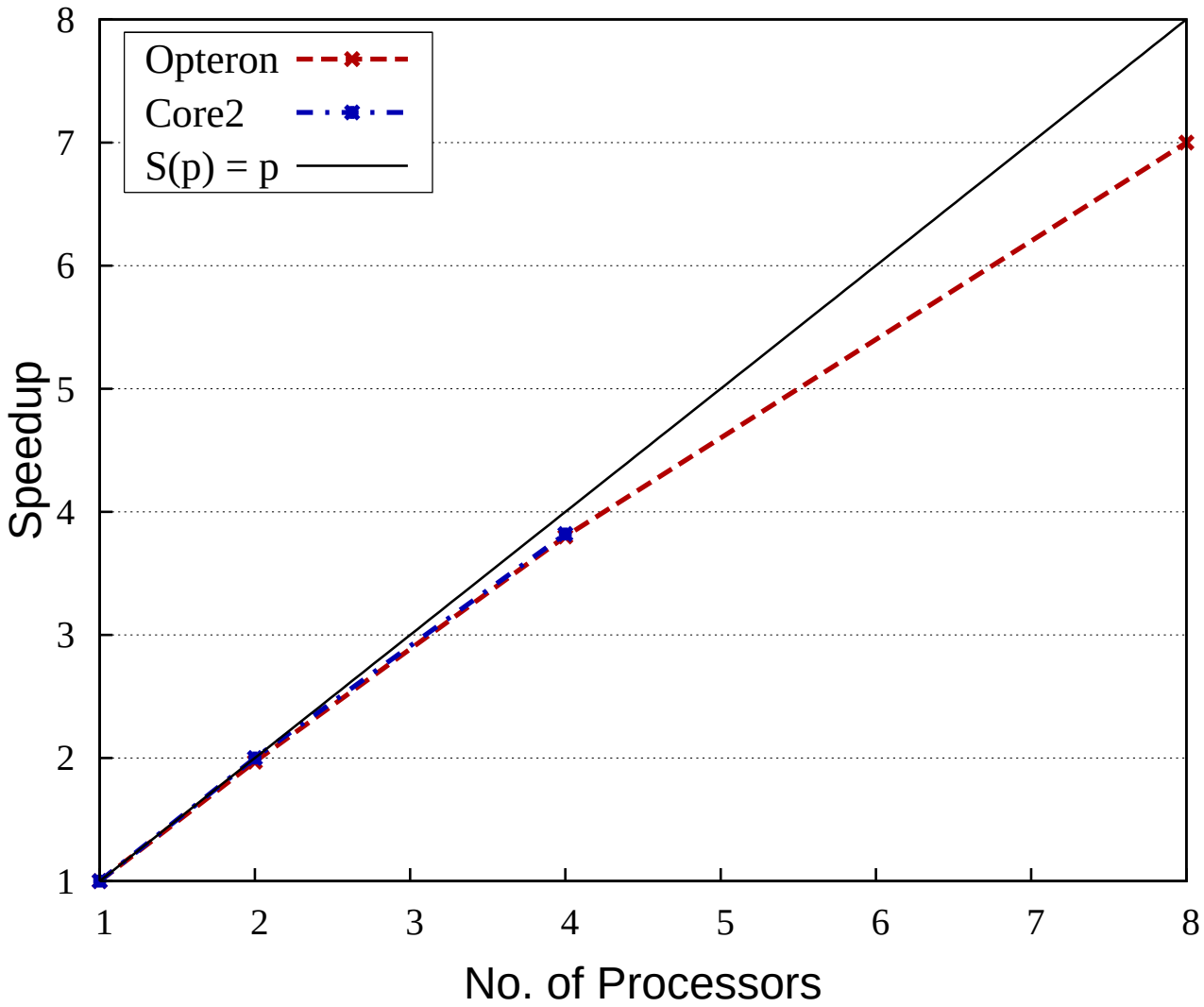
Apply OpenMP parallelisation directly to loop:

```
leaves = ... // list of leaves
#pragma omp parallel {
    while ( ! is_empty( leaves ) ) {
        // guard change of leaves set by mutex
        #pragma omp critical {
            cluster = head( leaves );
            leaves  = tail( leaves );
        }

        if ( is_adm( cluster ) )
            M = build_lowrank( cluster );
        else
            M = build_dense( cluster );
}   }
```

Building block matrices is cheap and can be done sequentially.

For Helmholtz Double Layer Potential in $\mathbb{R}^3$:



Matrix Construction with Hierarchy

1. first build all leaves in parallel
2. build block matrices corresponding to inner nodes afterwards sequentially

Matrix Construction with Hierarchy

1. first build all leaves in parallel
2. build block matrices corresponding to inner nodes afterwards sequentially

Matrix construction with Coarsening

1. insert all leaves of block cluster tree into work set
2. iterate until work set is empty
   1. obtain block cluster and handle corresponding matrix
      - if leaf: construct matrix
      - if inner node: try to coarsen
   2. check all sons of father block cluster,
   3. if all son matrices have been constructed, insert block cluster into work set

Preconditioning

Preconditioning

We are looking for a good preconditioner for the linear equation system

$$Ax = y$$

where A is an $\mathcal{H}$-Matrix.

Good guess for a preconditioner: $P = A^{-1}$. Therefore use $\mathcal{H}$-matrix inverse $\tilde{P} = A^{\sim 1}$.

For iterative schemes one only needs matrix-vector multiplication, therefore $\mathcal{H}$-LU factorisation is sufficient for evaluating $A^{\sim 1}$ and cheaper to compute.

Assume A has 2×2 block structure:

$$\begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix} \cdot \begin{pmatrix} U_{11} & U_{12} \\ & U_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

Solving the above system, one obtains:

$$\begin{aligned} L_{11}U_{11} &= A_{11}, & L_{11}U_{12} &= A_{12}, \\ L_{21}U_{11} &= A_{21}, & L_{22}U_{22} &= A'_{22} \end{aligned}$$

with

$$A'_{22} = A_{22} - L_{21}U_{12}.$$

This involves two recursive calls, two matrix solves and one multiplication (with addition).

Again, assume 2×2 block structure and consider $BU = C$ with known C and upper triangular U, e.g. $L_{21}U_{11} = A_{21}$ from above.

$$\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \cdot \begin{pmatrix} U_{11} & U_{12} \\ & U_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

This leads to

$$\begin{aligned} B_{11}U_{11} &= C_{11}, & B_{12}U_{22} &= C'_{12}, \\ B_{21}U_{11} &= C_{21}, & B_{22}U_{22} &= C'_{22} \end{aligned}$$

with

$$C'_{12} = C_{12} - B_{11}U_{12} \quad \text{and} \quad C'_{22} = C_{22} - B_{21}U_{12}$$

involving four recursions and two multiplications.

$\mathcal{H}$-LU factorisation and matrix solve only involves recursive calls and multiplications. Therefore, parallelising the multiplications, parallelises the $\mathcal{H}$-LU factorisation.

$\mathcal{H}$-LU factorisation and matrix solve only involves recursive calls and multiplications. Therefore, parallelising the multiplications, parallelises the $\mathcal{H}$-LU factorisation.

$$C := \alpha AB + \beta C$$

Sequential algorithm for a $m \times m$ block matrix:

```
void mul ( alpha, A, B, beta, C ) {
   if ( is_blocked( A ) && is_blocked( B ) &&
        is_blocked( C ) )
      for ( i = 0; i < m; i++ )
         for ( j = 0; j < m; j++ )
            for ( l = 0; l < m; l++ )
               mul( alpha, A_il, B_lj, beta, C_ij );
   else
      C := alpha * A * B + beta * C;
}
```

Collect all atomic multiplications into list and apply OpenMP parallelisation to list:

```
void mul_sim ( A, B, C, list ) {
   if ( is_blocked( A ) && is_blocked( B ) && is_blocked( C ) )
      for ( i, j, l = 0;   i, j, l < m;   i++, j++, l++ )
          mul_sim( A_il, B_lj, C_ij, list );
   else
      append( list, { C, A, B } ); }
```

Collect all atomic multiplications into list and apply OpenMP parallelisation to list:

```
void mul_sim ( A, B, C, list ) {
   if ( is_blocked( A ) && is_blocked( B ) && is_blocked( C ) )
      for ( i, j, l = 0;   i, j, l < m;   i++, j++, l++ )
          mul_sim( A_il, B_lj, C_ij, list );
   else
      append( list, { C, A, B } ); }
```

```
void mul ( alpha, A, B, beta, C ) {
    mul_sim( A, B, C, list );
    #pragma omp parallel {
        while ( ! is_empty( list ) ) {
            #pragma omp critical {
                product = head( list );
                list    = tail( list );
            }

            T = alpha * product.A * product.B;

            omp_set_lock( mutex( product.C ) ); // guard access to C
            product.C = beta * product.C + T;
            omp_unset_lock( mutex( product.C ) );
}   }   }
```

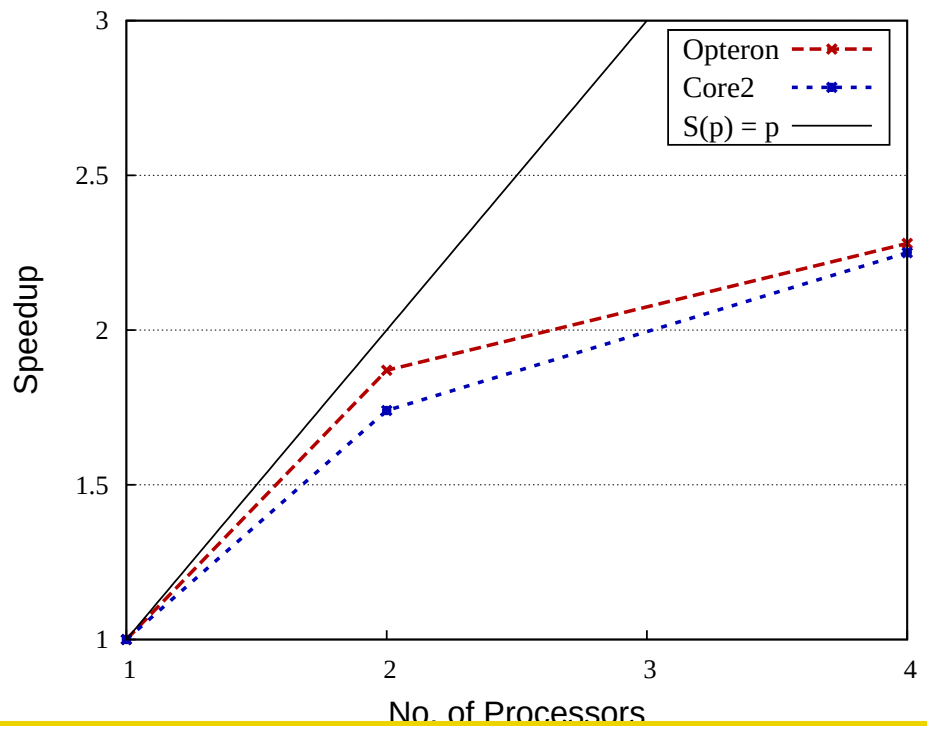
Collect products per matrix to reduce collisions:

```
void mul_sim2 ( A, B, C ) {
   if ( is_blocked( A ) && is_blocked( B ) && is_blocked( C ) )
      for ( i, j, l = 0;   i, j, l < m;   i++, j++, l++ )
          mul_sim2( A_il, B_lj, C_ij );
   else
      append( C.list, { A, B } ); }
```

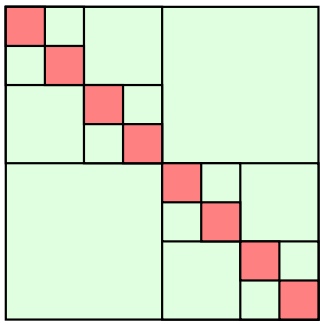
Collect products per matrix to reduce collisions:

```
void mul_sim2 ( A, B, C ) {
   if ( is_blocked( A ) && is_blocked( B ) && is_blocked( C ) )
      for ( i, j, l = 0;   i, j, l < m;   i++, j++, l++ )
          mul_sim2( A_il, B_lj, C_ij );
   else
      append( C.list, { A, B } ); }
```

Numerical Experiments (LU, Helmholtz DLP)

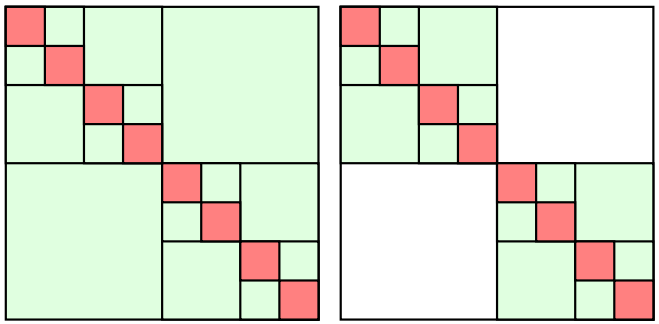


Successively remove off-diagonal blocks from $\mathcal{H}$-matrix with increasing number of processors:



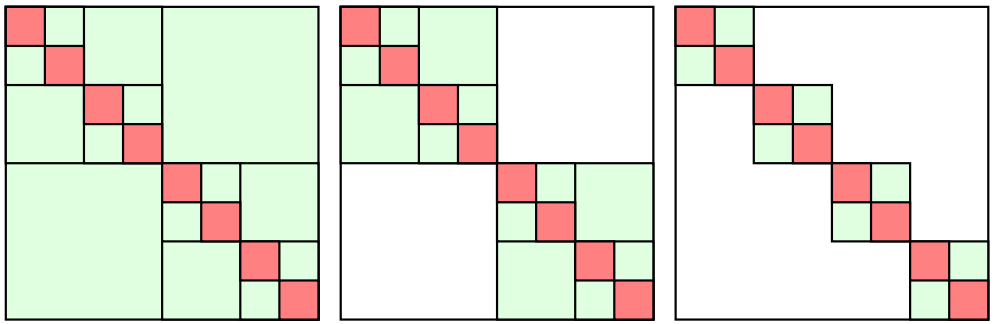
$p = 1$

Successively remove off-diagonal blocks from $\mathcal{H}$-matrix with increasing number of processors:



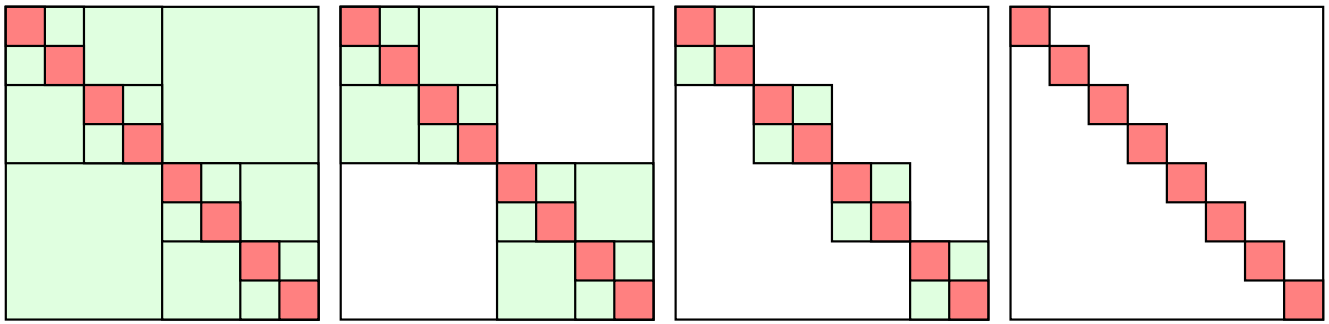
$p = 1$ $p = 2$

Successively remove off-diagonal blocks from $\mathcal{H}$-matrix with increasing number of processors:



$p = 1$ $p = 2$ $p = 4$

Successively remove off-diagonal blocks from $\mathcal{H}$-matrix with increasing number of processors:



$p = 1$ $p = 2$ $p = 4$ $p = 8$

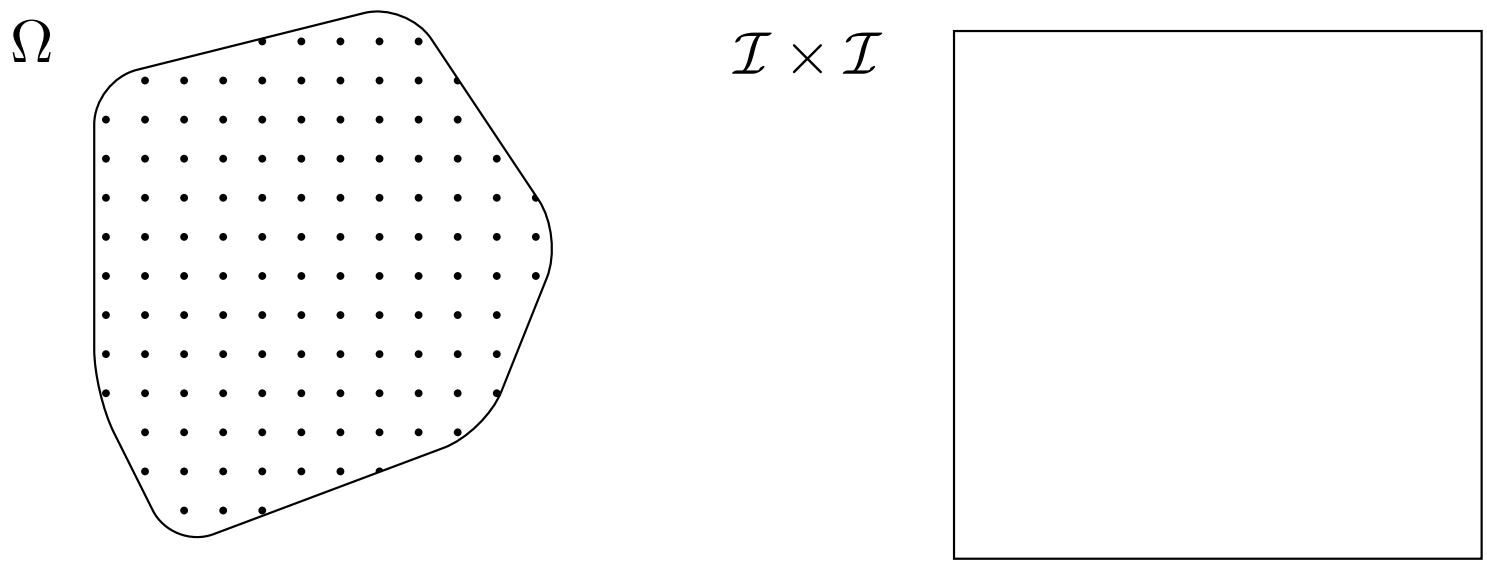
Assume 2×2 block structure of all block matrices in given $\mathcal{H}$-matrix. Then, the algorithm for the parallel LU factorisation for block diagonal matrices is:

```
void blockdiag_LU ( p, A ) {
    if ( p == 1 ) LU( A );
    else {
        #pragma omp parallel for num_threads(2)
            for ( i = 0; i < 2; i++ )
                blockdiag_LU( p/2, A_ii );
}   }   }
```

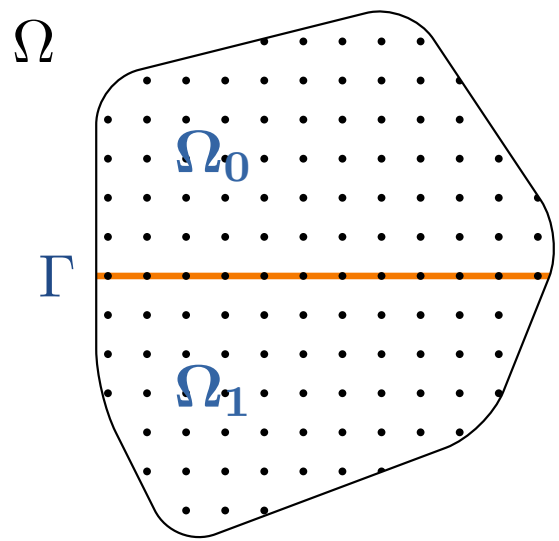
The OpenMP option `num_threads(2)` ensures, that only two threads are started.

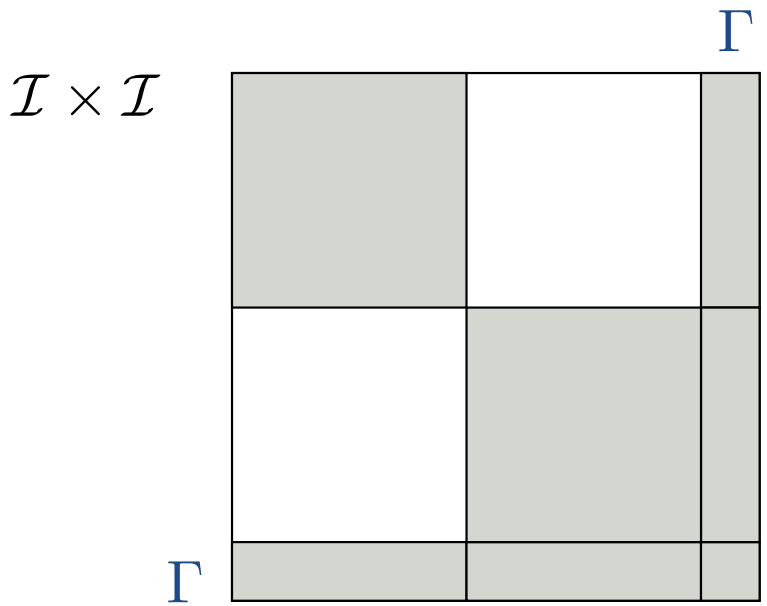
This approach requires support for nested parallelism in OpenMP implementation.

Assume a finite element discretisation for a partial differential equation, e.g. with piecewise linear ansatz functions. Since the support of the basis functions is local, one can find a subset Γ of $\mathcal{I}$, such that the remaining indices are decomposed into decoupled sets:

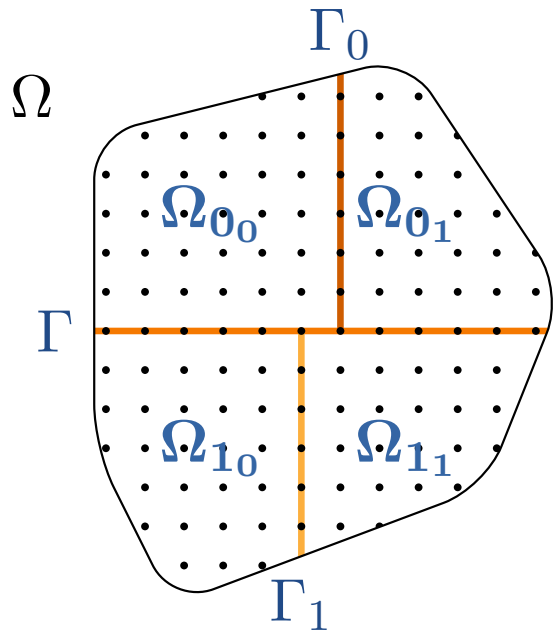


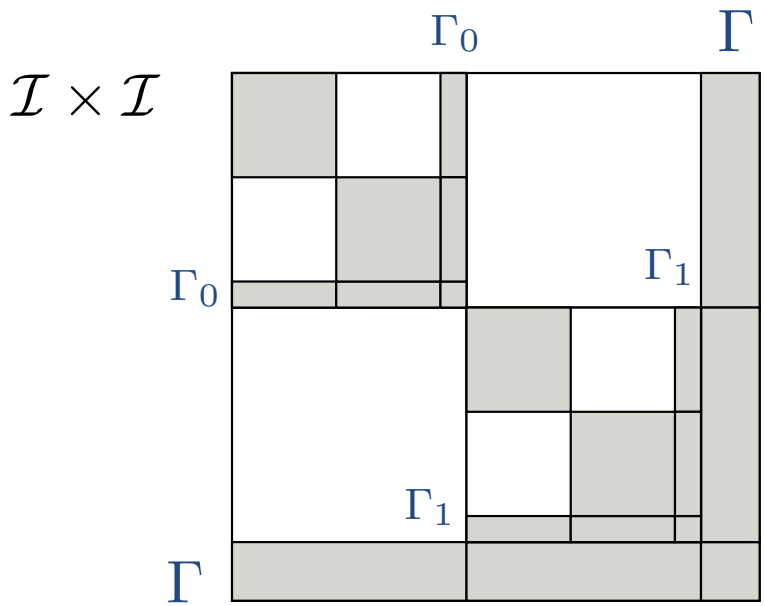
Assume a finite element discretisation for a partial differential equation, e.g. with piecewise linear ansatz functions. Since the support of the basis functions is local, one can find a subset Γ of $\mathcal{I}$, such that the remaining indices are decomposed into decoupled sets:





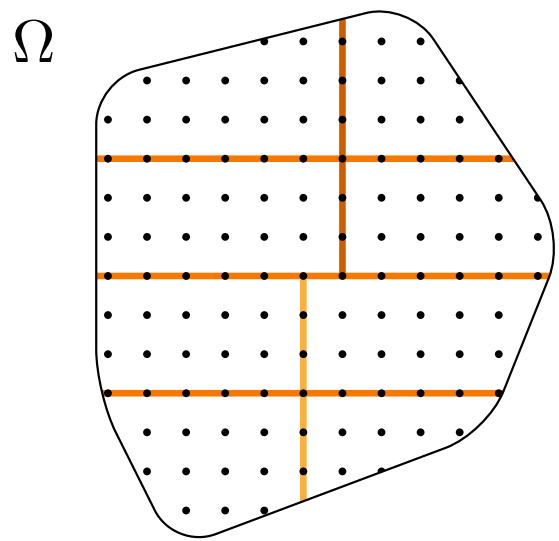
Assume a finite element discretisation for a partial differential equation, e.g. with piecewise linear ansatz functions. Since the support of the basis functions is local, one can find a subset Γ of $\mathcal{I}$, such that the remaining indices are decomposed into decoupled sets:

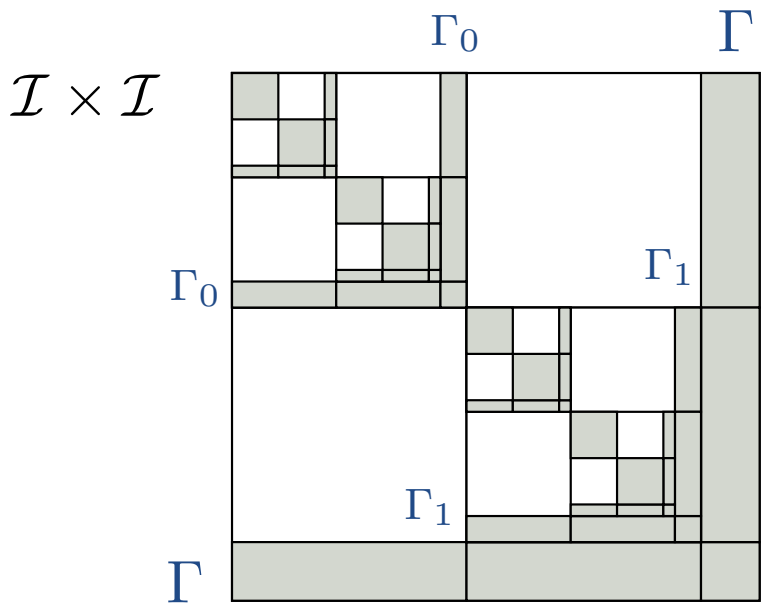




Recursively apply this procedure to the created sub index sets.

Assume a finite element discretisation for a partial differential equation, e.g. with piecewise linear ansatz functions. Since the support of the basis functions is local, one can find a subset Γ of $\mathcal{I}$, such that the remaining indices are decomposed into decoupled sets:





Recursively apply this procedure to the created sub index sets.

The L and U factors in a LU factorisation of A have the same structure as A:

$$\begin{pmatrix} L_{11} & & \\ & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \cdot \begin{pmatrix} U_{11} & & U_{13} \\ & U_{22} & U_{23} \\ & & U_{33} \end{pmatrix} = \begin{pmatrix} A_{11} & & A_{13} \\ & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$$

leading to

$$\begin{aligned} L_{11}U_{11} &= A_{11} & L_{22}U_{22} &= A_{22} \\ L_{31}U_{11} &= A_{31} & L_{32}U_{22} &= A_{32} \\ L_{11}U_{13} &= A_{13} & L_{22}U_{23} &= A_{23} \end{aligned}$$

which can be handled independently and

$$L_{33}U_{33} = A'_{33} \quad \text{with} \quad A'_{33} = A_{33} - L_{31}U_{13} - L_{32}U_{23}.$$

The L and U factors in a LU factorisation of A have the same structure as A:

$$\begin{pmatrix} L_{11} & & \\ & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \cdot \begin{pmatrix} U_{11} & & U_{13} \\ & U_{22} & U_{23} \\ & & U_{33} \end{pmatrix} = \begin{pmatrix} A_{11} & & A_{13} \\ & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$$

leading to

$$\begin{aligned} L_{11}U_{11} &= A_{11} & L_{22}U_{22} &= A_{22} \\ L_{31}U_{11} &= A_{31} & L_{32}U_{22} &= A_{32} \\ L_{11}U_{13} &= A_{13} & L_{22}U_{23} &= A_{23} \end{aligned}$$

which can be handled independently and

$$L_{33}U_{33} = A'_{33} \quad \text{with} \quad A'_{33} = A_{33} - L_{31}U_{13} - L_{32}U_{23}.$$

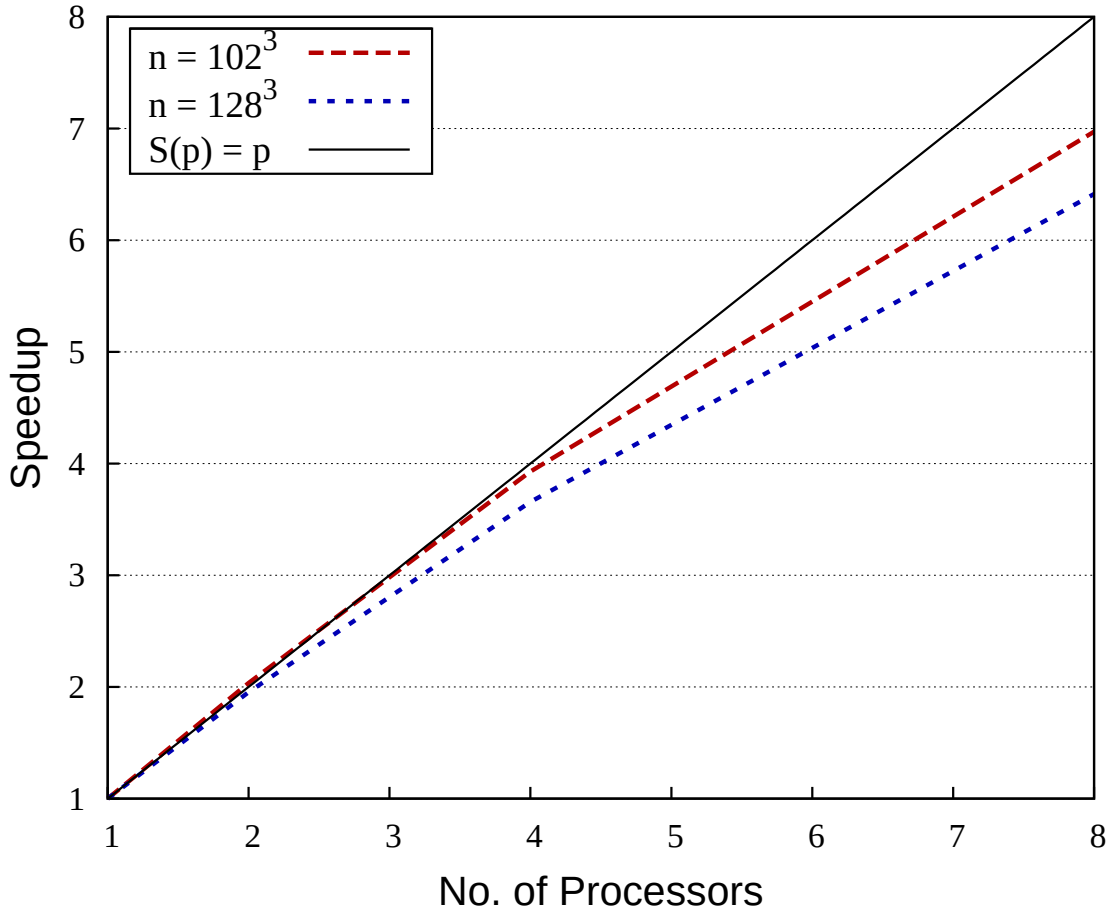
$\#\Gamma$ should be small to have small sequential part.

```
void nd_LU ( p, A ) {
    if ( p == 1 ) LU( A );
    else {
        #pragma omp parallel for num_threads(2)
            for ( i = 0; i < 2; i++ ) {
                nd_LU( p/2, A_ii );
                nd_solve_lower( p/2, A_3i, A_ii );
                nd_solve_upper( p/2, A_ii, A_i3 );
                T_i = nd_mul( p/2, A_3i, A_i3 );

                omp_set_lock( mutex( A_33 ) );
                A_33 = A_33 - T_i;
                omp_unset_lock( mutex( A_33 ) );
        }
        LU( A_33 );
}   }
```

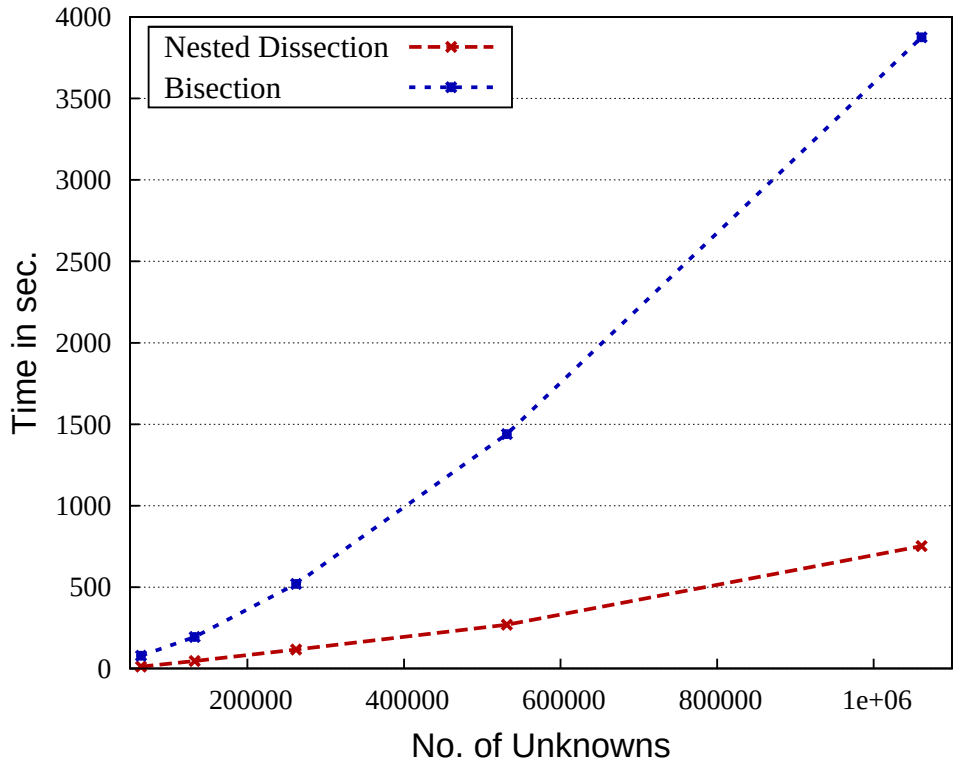
Numerical Experiments

Poisson problem in $\Omega = [0,1]^3$



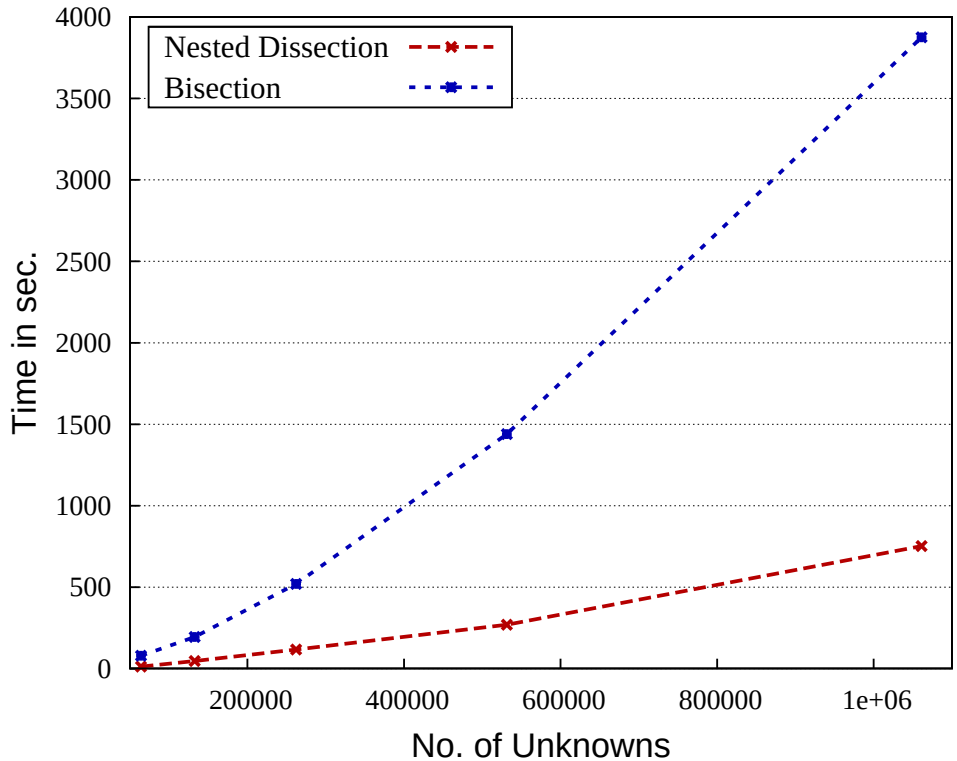
Nested Dissection: Remarks

- due to sparsity structure, nested dissection approach much faster than standard bisection even sequentially:



Nested Dissection: Remarks

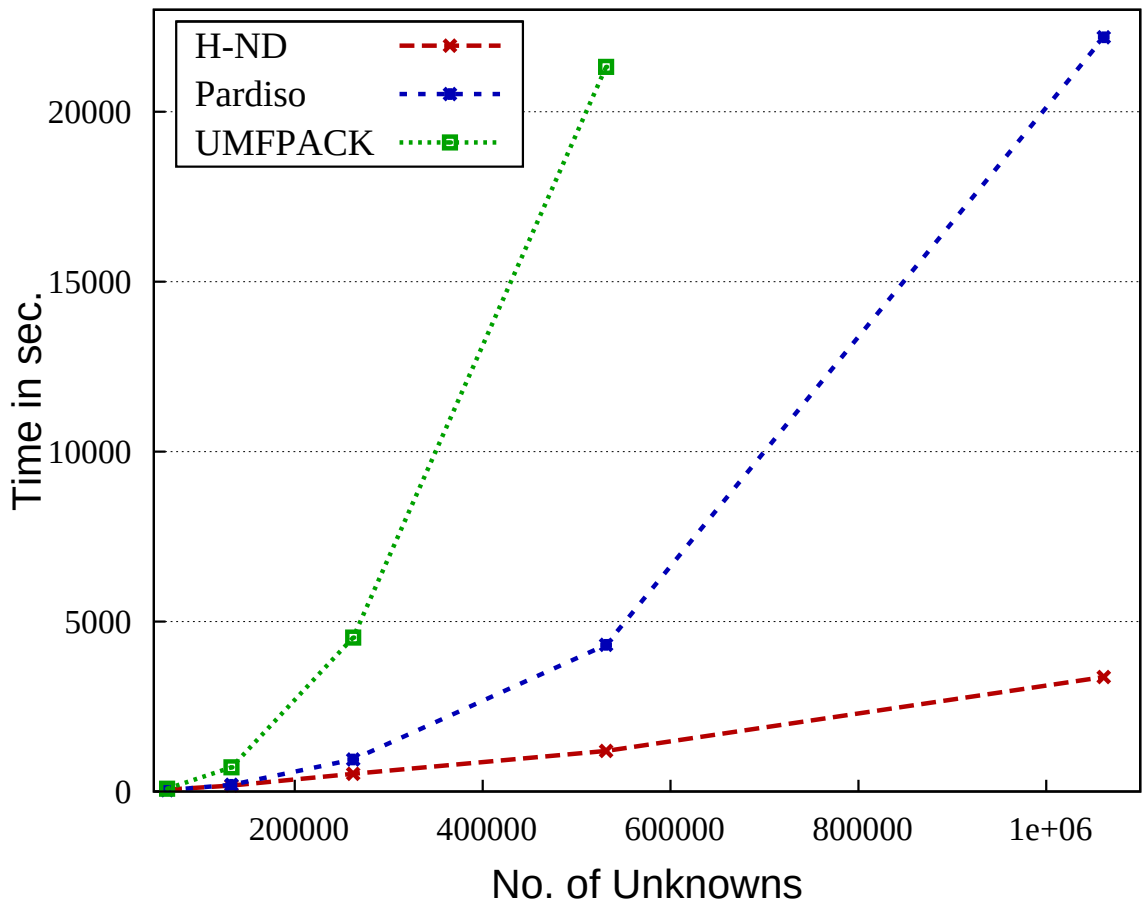
- due to sparsity structure, nested dissection approach much faster than standard bisection even sequentially:



- computation of Γ and clustering $\mathcal{I}$ can be done purely algebraically using graph partitioning for the sparse matrix.

Nested Dissection: Remarks

- $\mathcal{H}$-matrices with nested dissection faster than optimised direct solvers:



Conclusion and Outlook

Conclusion and Outlook

Presented algorithms are either leaf oriented (matrix construction) or use hierarchical parallelisation (LU with nested dissection).

Conclusion and Outlook

Presented algorithms are either leaf oriented (matrix construction) or use hierarchical parallelisation (LU with nested dissection).

Parallelisable with same techniques:

- Matrix Addition: only set of leaves involved

Conclusion and Outlook

Presented algorithms are either leaf oriented (matrix construction) or use hierarchical parallelisation (LU with nested dissection).

Parallelisable with same techniques:

- Matrix Addition: only set of leaves involved,
- Matrix Vector Multiplication:
  - again, with set of leaves,
  - private destination vector per thread; has to be summed up in parallel (axpy)

Conclusion and Outlook

Presented algorithms are either leaf oriented (matrix construction) or use hierarchical parallelisation (LU with nested dissection).

Parallelisable with same techniques:

- Matrix Addition: only set of leaves involved,
- Matrix Vector Multiplication:
  - again, with set of leaves,
  - private destination vector per thread; has to be summed up in parallel (axpy),
- Matrix Inversion:
  - like LU factorisation, uses parallel matrix multiplication,
  - much better speedup than LU factorisation.

Conclusion and Outlook

Presented algorithms are either leaf oriented (matrix construction) or use hierarchical parallelisation (LU with nested dissection).

Parallelisable with same techniques:

- Matrix Addition: only set of leaves involved,
- Matrix Vector Multiplication:
  - again, with set of leaves,
  - private destination vector per thread; has to be summed up in parallel (axpy),
- Matrix Inversion:
  - like LU factorisation, uses parallel matrix multiplication,
  - much better speedup than LU factorisation.

For all algorithms, the achievable speedup is high compared to the implementation costs.

Conclusion and Outlook

Presented algorithms are either leaf oriented (matrix construction) or use hierarchical parallelisation (LU with nested dissection).

Parallelisable with same techniques:

- Matrix Addition: only set of leaves involved,
- Matrix Vector Multiplication:
  - again, with set of leaves,
  - private destination vector per thread; has to be summed up in parallel (axpy),
- Matrix Inversion:
  - like LU factorisation, uses parallel matrix multiplication,
  - much better speedup than LU factorisation.

For all algorithms, the achievable speedup is high compared to the implementation costs.

Alternative: $\mathcal{H}$-Lib^{pro} already implements all presented algorithms and much more.

Literature

R. Kriemann,

Parallele Algorithmen für $\mathcal{H}$-Matrizen,

Ph.D. Thesis, University of Kiel, 2005.

R. Kriemann,

Parallel $\mathcal{H}$-Matrix Arithmetics on Shared Memory Systems,

Computing, 74:273–297, 2005.

M. Bebendorf and R. Kriemann,

Fast Parallel Solution of Boundary Integral Equations and Related Problems,

Computing and Visualization in Science, 8(3–4):121–135, 2005.

L. Grasedyck, R. Kriemann and S. Le Borne,

Domain Decomposition Based $\mathcal{H}$-LU Preconditioning,

to appear in "Numerische Mathematik".

L. Grasedyck, R. Kriemann and S. Le Borne,

Parallel Black Box $\mathcal{H}$-LU Preconditioning for Elliptic Boundary Value Problems,

"Computing and Visualization in Science", 11(4-6), pp. 273–291, 2008.