Parallel $\mathcal{H}$-matrix Arithmetic for Shared Memory Systems

Parallel $\mathcal{H}$-matrix Arithmetic for Shared Memory Systems

I) Matrix Building

Parallel $\mathcal{H}$-matrix Arithmetic for Shared Memory Systems

I) Matrix Building

II) Matrix-Vector Multiplication

Parallel $\mathcal{H}$-matrix Arithmetic for Shared Memory Systems

I) Matrix Building

II) Matrix-Vector Multiplication

III) Matrix Multiplication

Parallel $\mathcal{H}$-matrix Arithmetic for Shared Memory Systems

I) Matrix Building

II) Matrix-Vector Multiplication

III) Matrix Multiplication

IV) Matrix Inversion

Model Problem

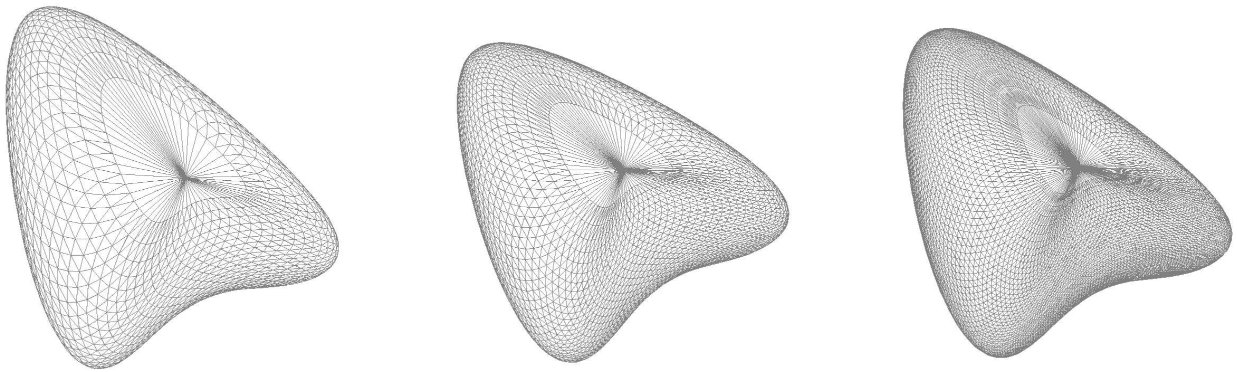
- Problem for all numerical examples:
  - single layer potential, piecewise constant ansatz
  - Galerkin discretisation

Model Problem

- Problem for all numerical examples:
  - single layer potential, piecewise constant ansatz
  - Galerkin discretisation
- representation by $\mathcal{H}$-matrices; rank-k blocks computed with *ACA*

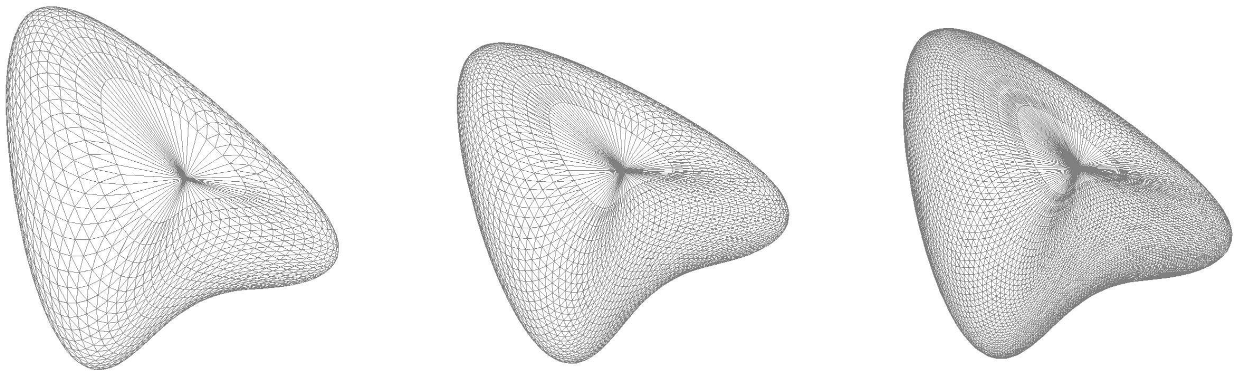
Model Problem

- Problem for all numerical examples:
  - single layer potential, piecewise constant ansatz
  - Galerkin discretisation
- representation by $\mathcal{H}$-matrices; rank-k blocks computed with *ACA*
- geometry:



Model Problem

- Problem for all numerical examples:
  - single layer potential, piecewise constant ansatz
  - Galerkin discretisation
- representation by $\mathcal{H}$-matrices; rank-k blocks computed with *ACA*
- geometry:



- computed on shared memory system with p processors (HP9000 Superdome, PA-RISC 875 MHz)

Notation

- Index set $I = \{0, \cdots, n-1\}$
- Cluster tree $T(I)$ constructed by *binary space partitioning*,
- $\text{depth}(T(I)) = \log_2 n$
- Block cluster tree $T(I \times I)$ with standard admissibility ($\eta = 1.0$)
- Leafs of block cluster tree: $\mathcal{L}(T(I \times I))$

Matrix Building

Matrix Building

Sequential algorithm:

for all $(\tau, \sigma) \in \mathcal{L}(T(I \times I))$ **do**
  if (τ, σ) is admissible **then**
    create rank-k matrix;
  else
    create dense matrix;
endfor;

Matrix Building

Sequential algorithm:

for all $(\tau, \sigma) \in \mathcal{L}(T(I \times I))$ **do**
  if (τ, σ) is admissible **then**
    create rank-k matrix;
  else
    create dense matrix;
endfor;

Straightforward parallelisation:

create each block on different processor

Load Balancing

Load Balancing

- use *online scheduling* algorithm (load balancing during computation)

Load Balancing

- use *online scheduling* algorithm (load balancing during computation)
- Advantage: no cost function needed

Load Balancing

- use *online scheduling* algorithm (load balancing during computation)
- Advantage: no cost function needed
- *List Scheduling*: first idle processor executes first not yet computed block

---

for all $(\tau, \sigma) \in \mathcal{L}(T(I \times I))$ **do**
    p := first idle processor;
    if (τ, σ) is admissible **then**
        create rank-k matrix on p;
    else
        create dense matrix on p;
endfor;

Load Balancing

- use *online scheduling* algorithm (load balancing during computation)
- Advantage: no cost function needed
- *List Scheduling*: first idle processor executes first not yet computed block

---

for all $(\tau, \sigma) \in \mathcal{L}(T(I \times I))$ **do**
  p := first idle processor;
  if (τ, σ) is admissible **then**
    create rank-k matrix on p;
  else
    create dense matrix on p;
endfor;

Parallel Speedup (Graham '69) and Complexity:

$$\frac{t(1)}{t(p)} \geq \frac{p}{\left(2 - \frac{1}{p}\right)} \quad , \quad \mathcal{W}_{\mathrm{MB}}(n,p) = \mathcal{O}\left(\frac{n \log n}{p}\right).$$

Programming Shared Memory Systems

Programming Shared Memory Systems

Threads:

- parallel execution paths in a single process

Programming Shared Memory Systems

Threads:

- parallel execution paths in a single process
- all threads share same address space: no communication

Programming Shared Memory Systems

Threads:

- parallel execution paths in a single process
- all threads share same address space: no communication
- *POSIX* threads (*Pthreads*) as common interface on many computer systems

Programming Shared Memory Systems

Threads:

- parallel execution paths in a single process
- all threads share same address space: no communication
- *POSIX* threads (*Pthreads*) as common interface on many computer systems

Implementation with *Thread Pool*:

Programming Shared Memory Systems

Threads:

- parallel execution paths in a single process
- all threads share same address space: no communication
- *POSIX* threads (*Pthreads*) as common interface on many computer systems

Implementation with *Thread Pool*:

- consists of p threads which execute given jobs

Programming Shared Memory Systems

Threads:

- parallel execution paths in a single process
- all threads share same address space: no communication
- *POSIX* threads (*Pthreads*) as common interface on many computer systems

Implementation with *Thread Pool*:

- consists of p threads which execute given jobs
- much simpler interface than Pthreads: simplifies programming

Programming Shared Memory Systems

Threads:

- parallel execution paths in a single process
- all threads share same address space: no communication
- *POSIX* threads (*Pthreads*) as common interface on many computer systems

Implementation with *Thread Pool*:

- consists of p threads which execute given jobs
- much simpler interface than Pthreads: simplifies programming
- more efficient: less startup time per job because no real thread is started

Matrix Building with Thread Pool

```
procedure build_matrix ( (τ, σ) )
    if (τ, σ) is admissible then
        build a rank-k matrix using ACA;
    else
        build a dense matrix;
end;

for all (τ, σ) ∈ ℒ(T(I × I)) do
    run ( build_matrix( (τ, σ) ) );
endfor;

sync_all ();
```

Numerical Results

Fixed rank: $k = 15$.

Time and Parallel Efficiency

$$E(p) = \frac{t(1)}{p \cdot t(p)}$$

n	$t(1)$	$E(4)$	$E(8)$	$E(12)$	$E(16)$
3 968	134.9 s	100 %	99.9 %	99.7 %	99.6 %
7 920	341.4 s	99.9 %	99.6 %	99.2 %	99.6 %
19 320	1040.8 s	99.9 %	99.8 %	99.7 %	99.6 %
43 680	2798.1 s	99.9 %	99.9 %	99.7 %	99.7 %
89 400	6587.7 s	100 %	100 %	100 %	100 %
184 040	15313.9 s	99.6 %	99.2 %	99.1 %	98.4 %

Matrix-Vector Multiplication

To compute:

$$y := \alpha A x + \beta y$$

Let y_i, x_i denote local part of y and x on proc. i, $|y_i| = |x_i| = n/p$.

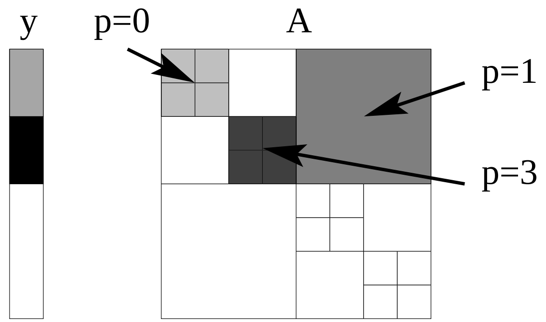
Matrix-Vector Multiplication

To compute:

$$y := \alpha A x + \beta y$$

Let y_i, x_i denote local part of y and x on proc. i, $|y_i| = |x_i| = n/p$.

Problem: two processors write to same part of y



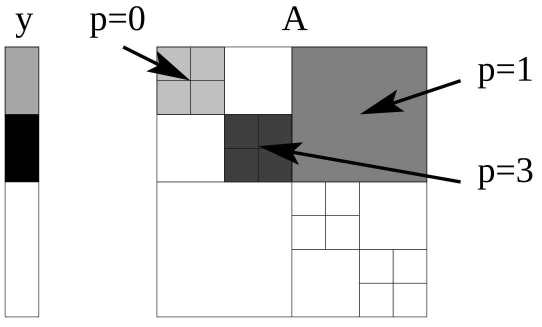
Matrix-Vector Multiplication

To compute:

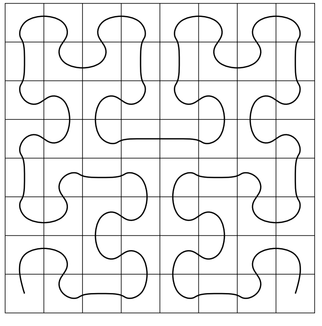
$$y := \alpha A x + \beta y$$

Let y_i, x_i denote local part of y and x on proc. i, $|y_i| = |x_i| = n/p$.

Problem: two processors write to same part of y



Solution: load balancing with *space-filling curves*



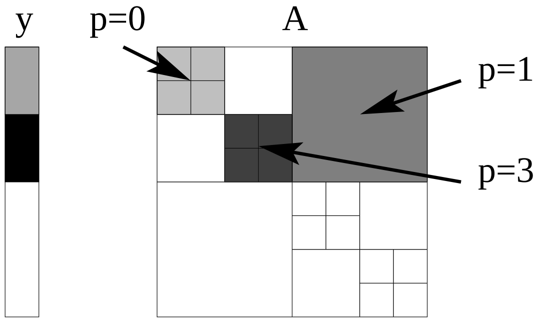
Matrix-Vector Multiplication

To compute:

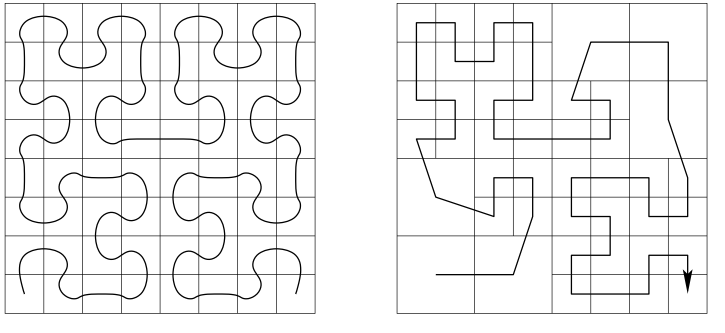
$$y := \alpha A x + \beta y$$

Let y_i, x_i denote local part of y and x on proc. i, $|y_i| = |x_i| = n/p$.

Problem: two processors write to same part of y



Solution: load balancing with *space-filling curves*



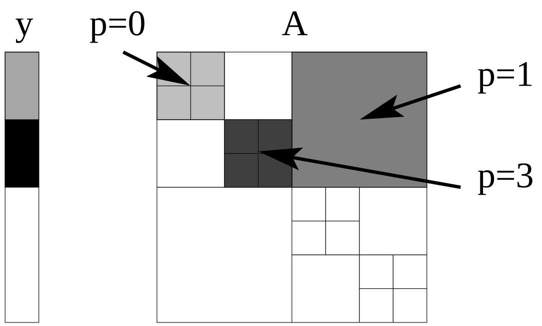
Matrix-Vector Multiplication

To compute:

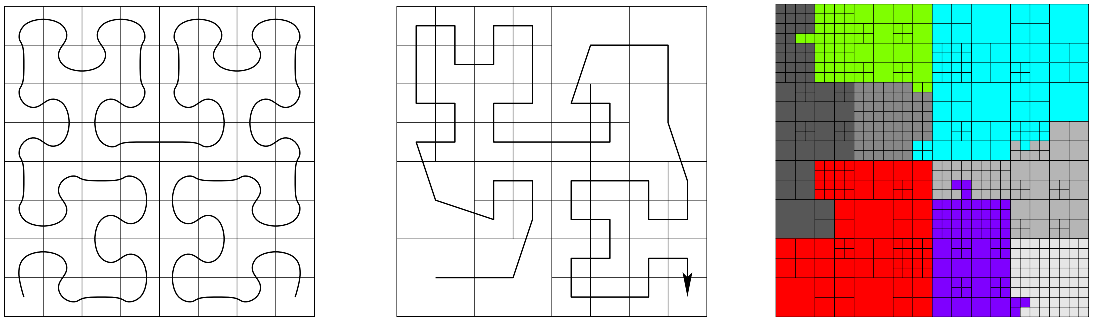
$$y := \alpha A x + \beta y$$

Let y_i, x_i denote local part of y and x on proc. i, $|y_i| = |x_i| = n/p$.

Problem: two processors write to same part of y



Solution: load balancing with *space-filling curves*



Load Balancing

- cost function: number of entries per block

Load Balancing

- cost function: number of entries per block
- use order defined by space-filling curve to form list of matrix blocks

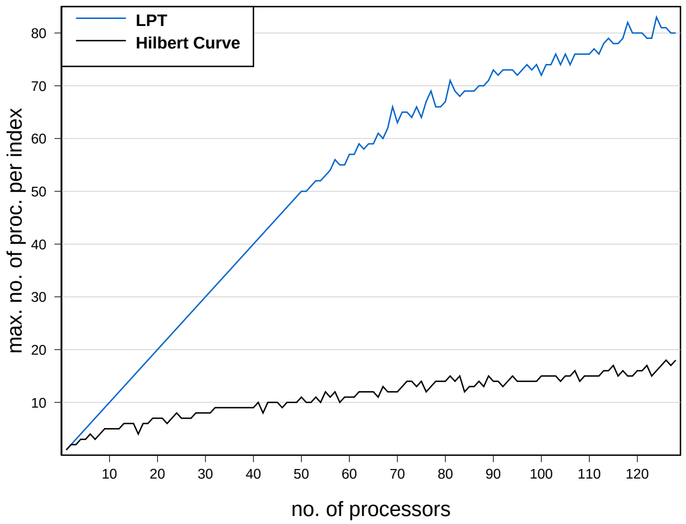
Load Balancing

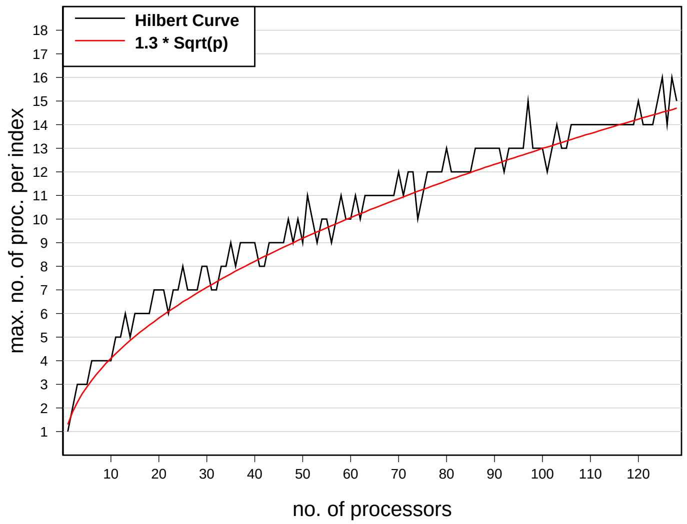
- cost function: number of entries per block
- use order defined by space-filling curve to form list of matrix blocks
- use *sequence partitioning* to schedule list (Olstad/Manne'95: solution in $\mathcal{O}\left(np\right)$)

Load Balancing

- cost function: number of entries per block
- use order defined by space-filling curve to form list of matrix blocks
- use *sequence partitioning* to schedule list (Olstad/Manne'95: solution in $\mathcal{O}(np)$)

Sharing Degree





Matrix-Vector Multiplication Algorithm

```
procedure step_1 ( i, β, y, A, x )
    y_i := β · y_i;
    y'_i := α A_i x;
end;

procedure step_2 ( i, y, y'_i )
    y_i := Σ y'_i;
end;

procedure mv_mul( i, α, A, x, β, y )
    for 0 ≤ i < p do
        run( step_1( i, β, y, A, x ) );
    sync_all();
    for 0 ≤ i < p do
        run( step_2( i, y, y'_i ) );
    sync_all();
end;
```

Complexity of parallel Matrix-Vector Multiplication

$$\mathcal{W}_{\mathrm{MV}}(n,p) = \mathcal{O}\left(\frac{n\log n}{p} + \frac{n}{\sqrt{p}}\right)$$

Complexity of parallel Matrix-Vector Multiplication

$$\mathcal{W}_{\mathrm{MV}}(n,p) = \mathcal{O}\left(\frac{n\log n}{p} + \frac{n}{\sqrt{p}}\right)$$

Numerical Results

n	$t(1)$	$E(4)$	$E(8)$	$E(12)$	$E(16)$
3 968	$1.47_{10}{-1}$ s	85.3 %	77.6 %	66.3 %	49.7 %
7 920	$3.99_{10}{-1}$ s	83.4 %	79.5 %	74.3 %	64.9 %
19 320	$1.27_{10}{-0}$ s	86.4 %	83.8 %	79.6 %	72.3 %
43 680	$3.40_{10}{-0}$ s	87.2 %	87.0 %	82.8 %	78.7 %
89 400	$7.84_{10}{-0}$ s	90.1 %	85.1 %	83.9 %	80.4 %
184 040	$1.79_{10}{+1}$ s	90.0 %	85.1 %	86.5 %	80.7 %

Matrix Multiplication

To compute:

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

Matrix Multiplication

To compute:

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

Sequential Algorithm for a $m \times m$ blockmatrix:

procedure mul(α, A, B, β, C)
    if A, B and C are blockmatrices **then**
        for $i := 0, \ldots, m-1$ **do**
            for $j := 0, \ldots, m-1$ **do**
                for $l := 0, \ldots, m-1$ **do**
                    mul($\alpha, A_{il}, B_{lj}, \beta, C_{ij}$);
    else
        $C := \alpha AB + \beta C$;
end;

Matrix Multiplication

To compute:

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

Sequential Algorithm for a $m \times m$ blockmatrix:

procedure mul(α, A, B, β, C)
  if A, B and C are blockmatrices **then**
    for $i := 0, \ldots, m-1$ **do**
      for $j := 0, \ldots, m-1$ **do**
        for $l := 0, \ldots, m-1$ **do**
          mul($\alpha, A_{il}, B_{lj}, \beta, C_{ij}$);
  else
8:    $C := \alpha AB + \beta C$;
end;

Parallelisation: execute line 8 on different processors (online scheduling)

Collisions

Consider

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

Parallel execution of

$$C_{00} = C_{00} + A_{00}B_{00} \quad \text{and} \quad C_{00} = C_{00} + A_{01}B_{10}.$$

leads to collision and blocking of one processor.

Collisions

Consider

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

Parallel execution of

$$C_{00} = C_{00} + A_{00}B_{00} \quad \text{and} \quad C_{00} = C_{00} + A_{01}B_{10}.$$

leads to collision and blocking of one processor.

Solution:

- simulate matrix multiplication to collect all products AB for a destination block C

Collisions

Consider

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

Parallel execution of

$$C_{00} = C_{00} + A_{00}B_{00} \quad \text{and} \quad C_{00} = C_{00} + A_{01}B_{10}.$$

leads to collision and blocking of one processor.

Solution:

- simulate matrix multiplication to collect all products AB for a destination block C
- execute list of products for each C on a different processor

Algorithm

procedure sim_mul(A, B, C)
    if A, B and C are blockmatrices **then**
        for $i := 0, \ldots, m-1$ **do**
            for $j := 0, \ldots, m-1$ **do**
                for $l := 0, \ldots, m-1$ **do**
                    sim_mul(A_{il}, B_{lj}, C_{ij});
    else
        $P_C := P_C \cup \{(A, B)\}$; $\mathcal{L}_{\mathrm{MM}} := \mathcal{L}_{\mathrm{MM}} \cup \{C\}$;
end;

Algorithm

procedure sim_mul(A, B, C)
  if A, B and C are blockmatrices **then**
    for $i := 0, \ldots, m-1$ **do**
      for $j := 0, \ldots, m-1$ **do**
        for $l := 0, \ldots, m-1$ **do**
          sim_mul(A_{il}, B_{lj}, C_{ij});
  else
    $P_C := P_C \cup \{(A, B)\}$; $\mathcal{L}_{\mathrm{MM}} := \mathcal{L}_{\mathrm{MM}} \cup \{C\}$;
end;

procedure mul_block(C)
  for all $(A, B) \in P_C$ **do** $C := C + \alpha AB$;

procedure par_mul($\alpha, \beta, \mathcal{L}_{\mathrm{MM}}$)
  for all $C \in \mathcal{L}_{\mathrm{MM}}$ **do**
    run(mul_block(C));

Complexity of parallel $\mathcal{H}$-Matrix Multiplication

Using List scheduling:

$$\mathcal{W}_{\mathrm{MM}}(n,p) = \mathcal{O}\left(\frac{n \log^2 n}{p}\right)$$

Complexity of parallel $\mathcal{H}$-Matrix Multiplication

Using List scheduling:

$$\mathcal{W}_{\mathrm{MM}}(n,p) = \mathcal{O}\left(\frac{n \log^2 n}{p}\right)$$

Numerical Results

n	$t(1)$	$E(4)$	$E(8)$	$E(12)$	$E(16)$
3 968	98.5 s	98.3 %	97.0 %	95.3 %	95.0 %
7 920	287.8 s	98.2 %	97.5 %	97.0 %	95.6 %
19 320	945.5 s	99.0 %	97.7 %	96.9 %	96.2 %
43 680	2817.2 s	99.1 %	98.2 %	97.1 %	96.1 %
89 400	7432.7 s	100 %	99.5 %	99.0 %	97.6 %
184 040	19292.2 s	99.8 %	98.8 %	98.0 %	96.4 %

Matrix Inversion

Sequential Schur-complement algorithm for a 2×2 blockmatrix:

procedure invert(A, C, T)

  if A is a blockmatrix **then**

    invert(A_{00}, C_{00}, T_{00});

    $T_{01} := C_{00} A_{01}$; $T_{10} := A_{10} C_{00}$;

    $A_{11} := A_{11} - A_{10} T_{01}$;

    invert(A_{11}, C_{11}, T_{11});

    $C_{01} := -T_{01} C_{11}$; $C_{10} := -C_{11} T_{10}$;

    $C_{00} := C_{00} - T_{01} C_{10}$;

  else

    $C := A^{-1}$;

  endif;

end;

Matrix Inversion

Sequential Schur-complement algorithm for a 2×2 blockmatrix:

procedure invert(A, C, T)
  if A is a blockmatrix **then**
    invert(A_{00}, C_{00}, T_{00});
    $T_{01} := C_{00} A_{01}$; $T_{10} := A_{10} C_{00}$;
    $A_{11} := A_{11} - A_{10} T_{01}$;
    invert(A_{11}, C_{11}, T_{11});
    $C_{01} := -T_{01} C_{11}$; $C_{10} := -C_{11} T_{10}$;
    $C_{00} := C_{00} - T_{01} C_{10}$;
  else
    $C := A^{-1}$;
  endif;
end;

Parallelisation: use parallel matrix multiplication for all 6 products

Complexity of Parallel $\mathcal{H}$-Matrix Inversion

$$\mathcal{W}_{\mathrm{MI}}(n,p) = \mathcal{O}\left(n + \frac{n\log^2 n}{p}\right)$$

Complexity of Parallel $\mathcal{H}$-Matrix Inversion

$$\mathcal{W}_{\mathrm{MI}}(n,p) = \mathcal{O}\left(n + \frac{n \log^2 n}{p}\right)$$

Numerical Results

n	$t(1)$	$E(4)$	$E(8)$	$E(12)$	$E(16)$
3 968	97.6 s	92.9 %	82.1 %	71.5 %	60.6 %
7 920	286.3 s	93.7 %	83.5 %	73.2 %	62.2 %
19 320	939.2 s	94.5 %	83.7 %	73.3 %	63.8 %
43 680	2796.7 s	94.2 %	83.3 %	72.7 %	62.9 %
89 400	10106.2 s	94.9 %	83.8 %	73.2 %	63.8 %
184 040	19191.0 s	94.8 %	83.8 %	73.1 %	63.7 %

Conclusion

Speedup of parallel $\mathcal{H}$-matrix arithmetic