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 $H$-matrix Arithmetic for Shared Memory Systems

I) Matrix Building

II) Matrix-Vector Multiplication

III) Matrix Multiplication
Parallel $H$-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:

  ![Geometry Diagram]

- 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: $\mathcal{L}(T(I \times I))$
Sequential algorithm:

for all \((\tau, \sigma) \in \mathcal{L}(T(I \times I))\) do

if \((\tau, \sigma)\) is admissible then

create rank-\(k\) matrix;

else

create dense matrix;

endfor;
Matrix Building

Sequential algorithm:

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

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 \((\tau, \sigma)\) 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

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

Parallel Speedup (Graham ’69) and Complexity:

\[
\frac{t(1)}{t(p)} \geq \frac{p}{2 - \frac{1}{p}}, \quad \mathcal{W}_{MB}(n, p) = \mathcal{O}\left(\frac{n\log n}{p}\right).
\]
Programming Shared Memory Systems
Programming Shared Memory Systems

Threads:

- parallel execution paths in a single process
Threads:

- parallel execution paths in a single process
- all threads share same address space: no communication
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
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:
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
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) ∈ $\mathcal{L}(T(I \times I))$ do
    run ( build_matrix( (τ, σ) ) );
endfor;

sync_all ();
Numerical Results

Fixed rank: \( k = 15 \).

Time and Parallel Efficiency

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

<table>
<thead>
<tr>
<th>( n )</th>
<th>( t(1) )</th>
<th>( E(4) )</th>
<th>( E(8) )</th>
<th>( E(12) )</th>
<th>( E(16) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3968</td>
<td>134.9 s</td>
<td>100 %</td>
<td>99.9 %</td>
<td>99.7 %</td>
<td>99.6 %</td>
</tr>
<tr>
<td>7920</td>
<td>341.4 s</td>
<td>99.9 %</td>
<td>99.6 %</td>
<td>99.2 %</td>
<td>99.6 %</td>
</tr>
<tr>
<td>19320</td>
<td>1040.8 s</td>
<td>99.9 %</td>
<td>99.8 %</td>
<td>99.7 %</td>
<td>99.6 %</td>
</tr>
<tr>
<td>43680</td>
<td>2798.1 s</td>
<td>99.9 %</td>
<td>99.9 %</td>
<td>99.7 %</td>
<td>99.7 %</td>
</tr>
<tr>
<td>89400</td>
<td>6587.7 s</td>
<td>100 %</td>
<td>100 %</td>
<td>100 %</td>
<td>100 %</td>
</tr>
<tr>
<td>184040</td>
<td>15313.9 s</td>
<td>99.6 %</td>
<td>99.2 %</td>
<td>99.1 %</td>
<td>98.4 %</td>
</tr>
</tbody>
</table>
Matrix-Vector Multiplication

To compute:

\[ y := \alpha Ax + \beta y \]

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:

\[ y := \alpha Ax + \beta y \]

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:

\[ y := \alpha Ax + \beta y \]

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 \textit{space-filling curves}
Matrix-Vector Multiplication

To compute:

\[ y := \alpha Ax + \beta y \]

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:

\[ y := \alpha Ax + \beta y \]

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 $\mathcal{O}(np)$)
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

![Graphs showing sharing degree for different numbers of processors](image-url)
Matrix-Vector Multiplication Algorithm

procedure step_1 (\(i, \beta, y, A, x\))
\[
\begin{align*}
y_i & := \beta \cdot y_i; \\
y'_i & := \alpha A_i x;
\end{align*}
\]
end;

procedure step_2 (\(i, y, y'_i\))
\[
y_i := \sum y'_i;
\]
end;

procedure mv_mul(\(i, \alpha, A, x, \beta, y\))
\[
\begin{align*}
\text{for } 0 \leq i < p & \text{ do} \\
& \text{run( step_1( } i, \beta, y, A, x ) ); \\
& \text{sync_all();} \\
& \text{for } 0 \leq i < p & \text{ do} \\
& \text{run( step_2( } i, y, y'_i ) ); \\
& \text{sync_all();}
\end{align*}
\]
end;
Complexity of parallel Matrix-Vector Multiplication

\[ \mathcal{W}_{MV}(n, p) = O \left( \frac{n \log n}{p} + \frac{n}{\sqrt{p}} \right) \]
Complexity of parallel Matrix-Vector Multiplication

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

**Numerical Results**

<table>
<thead>
<tr>
<th>(n)</th>
<th>(t(1))</th>
<th>(E(4))</th>
<th>(E(8))</th>
<th>(E(12))</th>
<th>(E(16))</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 968</td>
<td>1.47\times 10^{-1} s</td>
<td>85.3 %</td>
<td>77.6 %</td>
<td>66.3 %</td>
<td>49.7 %</td>
</tr>
<tr>
<td>7 920</td>
<td>3.99\times 10^{-1} s</td>
<td>83.4 %</td>
<td>79.5 %</td>
<td>74.3 %</td>
<td>64.9 %</td>
</tr>
<tr>
<td>19 320</td>
<td>1.27\times 10^{-0} s</td>
<td>86.4 %</td>
<td>83.8 %</td>
<td>79.6 %</td>
<td>72.3 %</td>
</tr>
<tr>
<td>43 680</td>
<td>3.40\times 10^{-0} s</td>
<td>87.2 %</td>
<td>87.0 %</td>
<td>82.8 %</td>
<td>78.7 %</td>
</tr>
<tr>
<td>89 400</td>
<td>7.84\times 10^{-0} s</td>
<td>90.1 %</td>
<td>85.1 %</td>
<td>83.9 %</td>
<td>80.4 %</td>
</tr>
<tr>
<td>184 040</td>
<td>1.79\times 10^{+1} s</td>
<td>90.0 %</td>
<td>85.1 %</td>
<td>86.5 %</td>
<td>80.7 %</td>
</tr>
</tbody>
</table>
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:

\begin{verbatim}
procedure mul(\alpha, A, B, \beta, 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;
\end{verbatim}
Matrix Multiplication

To compute:

\[ C := \alpha AB + \beta C \]

Sequential Algorithm for a \( m \times m \) blockmatrix:

```plaintext
procedure mul( \( \alpha, A, B, \beta, C \) )
  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( \( \alpha, A_{il}, B_{lj}, \beta, C_{ij} \) );
        end;
      end;
    end
  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, . . . , m − 1 do
            for j := 0, . . . , m − 1 do
                for l := 0, . . . , m − 1 do
                    sim_mul( A_{il}, B_{lj}, C_{ij} );
            end
        end
    else
        P_C := P_C ∪ {(A, B)}; \mathcal{L}_{\text{MM}} := \mathcal{L}_{\text{MM}} ∪ \{ C \};
    end;
Algorithm

procedure sim_mul( A, B, C )
    if A, B and C are blockmatrices then
        for i := 0, ..., m - 1 do
            for j := 0, ..., m - 1 do
                for l := 0, ..., m - 1 do
                    sim_mul( A_{il}, B_{lj}, C_{ij} );
                else
                    P_C := P_C ∪ {(A, B)}; L_{MM} := L_{MM} ∪ {C};
    end;

procedure mul_block( C )
    for all (A, B) ∈ P_C do C := C + αAB;

procedure par_mul( α, β, L_{MM} )
    for all C ∈ L_{MM} do
        run( mul_block( C ) );
Complexity of parallel $\mathcal{H}$-Matrix Multiplication

Using List scheduling:

$$\mathcal{W}_{MM}(n, p) = \mathcal{O}\left(\frac{n \log^2 n}{p}\right)$$
Complexity of parallel H-Matrix Multiplication

Using List scheduling:

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

Numerical Results

<table>
<thead>
<tr>
<th>n</th>
<th>t(1)</th>
<th>E(4)</th>
<th>E(8)</th>
<th>E(12)</th>
<th>E(16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3968</td>
<td>98.5 s</td>
<td>98.3 %</td>
<td>97.0 %</td>
<td>95.3 %</td>
<td>95.0 %</td>
</tr>
<tr>
<td>7920</td>
<td>287.8 s</td>
<td>98.2 %</td>
<td>97.5 %</td>
<td>97.0 %</td>
<td>95.6 %</td>
</tr>
<tr>
<td>19320</td>
<td>945.5 s</td>
<td>99.0 %</td>
<td>97.7 %</td>
<td>96.9 %</td>
<td>96.2 %</td>
</tr>
<tr>
<td>43680</td>
<td>2817.2 s</td>
<td>99.1 %</td>
<td>98.2 %</td>
<td>97.1 %</td>
<td>96.1 %</td>
</tr>
<tr>
<td>89400</td>
<td>7432.7 s</td>
<td>100 %</td>
<td>99.5 %</td>
<td>99.0 %</td>
<td>97.6 %</td>
</tr>
<tr>
<td>184040</td>
<td>19292.2 s</td>
<td>99.8 %</td>
<td>98.8 %</td>
<td>98.0 %</td>
<td>96.4 %</td>
</tr>
</tbody>
</table>
Sequential Schur-complement algorithm for a $2 \times 2$ blockmatrix:

```plaintext
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;
```
Sequential Schur-complement algorithm for a $2 \times 2$ blockmatrix:

\begin{verbatim}
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;
\end{verbatim}

Parallelisation: use parallel matrix multiplication for all 6 products
Complexity of Parallel $H$-Matrix Inversion

$$W_{MI}(n, p) = O\left( n + \frac{n \log^2 n}{p} \right)$$
Complexity of Parallel $\mathcal{H}$-Matrix Inversion

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

### Numerical Results

<table>
<thead>
<tr>
<th>$n$</th>
<th>$t(1)$</th>
<th>$E(4)$</th>
<th>$E(8)$</th>
<th>$E(12)$</th>
<th>$E(16)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3968</td>
<td>97.6 s</td>
<td>92.9 %</td>
<td>82.1 %</td>
<td>71.5 %</td>
<td>60.6 %</td>
</tr>
<tr>
<td>7920</td>
<td>286.3 s</td>
<td>93.7 %</td>
<td>83.5 %</td>
<td>73.2 %</td>
<td>62.2 %</td>
</tr>
<tr>
<td>19320</td>
<td>939.2 s</td>
<td>94.5 %</td>
<td>83.7 %</td>
<td>73.3 %</td>
<td>63.8 %</td>
</tr>
<tr>
<td>43680</td>
<td>2796.7 s</td>
<td>94.2 %</td>
<td>83.3 %</td>
<td>72.7 %</td>
<td>62.9 %</td>
</tr>
<tr>
<td>89400</td>
<td>10106.2 s</td>
<td>94.9 %</td>
<td>83.8 %</td>
<td>73.2 %</td>
<td>63.8 %</td>
</tr>
<tr>
<td>184040</td>
<td>19191.0 s</td>
<td>94.8 %</td>
<td>83.8 %</td>
<td>73.1 %</td>
<td>63.7 %</td>
</tr>
</tbody>
</table>
Conclusion

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