Parallel Algorithms for $\mathcal{H}$-matrices (Part I)

Parallel Algorithms for $\mathcal{H}$-matrices (Part I)

1. $\mathcal{H}$-matrices and Model Problem

Parallel Algorithms for $\mathcal{H}$-matrices (Part I)

1. $\mathcal{H}$-matrices and Model Problem
2. Matrix Building

Parallel Algorithms for $\mathcal{H}$-matrices (Part I)

1. $\mathcal{H}$-matrices and Model Problem
2. Matrix Building
3. Bulk Synchronous Parallel Machine

Parallel Algorithms for $\mathcal{H}$-matrices (Part I)

1. $\mathcal{H}$-matrices and Model Problem
2. Matrix Building
3. Bulk Synchronous Parallel Machine
4. Matrix-Vector Multiplication

$\mathcal{H}$-matrices

- Index set $I = \{0, \cdots, n-1\}$

$\mathcal{H}$-matrices

- Index set $I = \{0, \cdots, n-1\}$
- Cluster tree $T(I)$ constructed by *binary space partitioning*,

$\mathcal{H}$-matrices

- Index set $I = \{0, \cdots, n-1\}$
- Cluster tree $T(I)$ constructed by *binary space partitioning*,
- depth$(T(I)) = \log_2 n$

$\mathcal{H}$-matrices

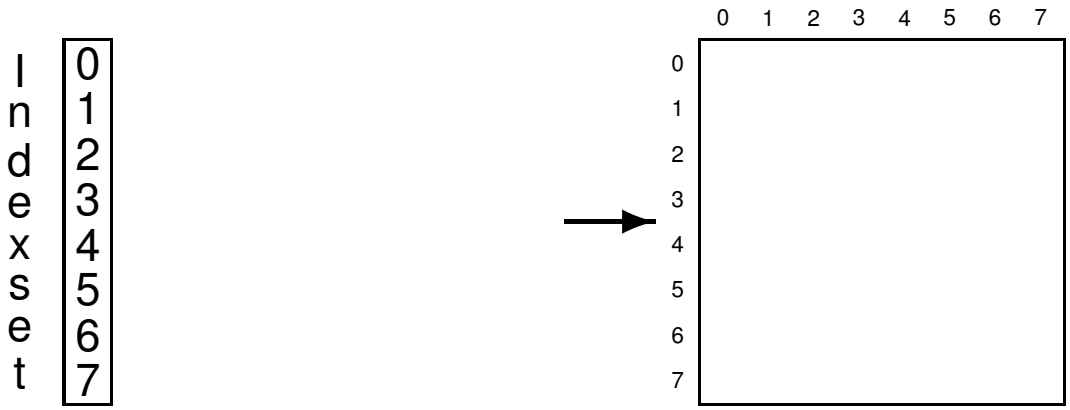
- Index set $I = \{0, \cdots, n-1\}$
- Cluster tree $T(I)$ constructed by *binary space partitioning*,
- depth$(T(I)) = \log_2 n$
- Block cluster tree $T(I \times I)$ with standard admissibility ($\eta = 0.5$)

$\mathcal{H}$-matrices

- Index set $I = \{0, \cdots, n-1\}$
- Cluster tree $T(I)$ constructed by *binary space partitioning*,
- depth$(T(I)) = \log_2 n$
- Block cluster tree $T(I \times I)$ with standard admissibility ($\eta = 0.5$)
- Leafs of block cluster tree: $\mathcal{L}(T(I \times I))$

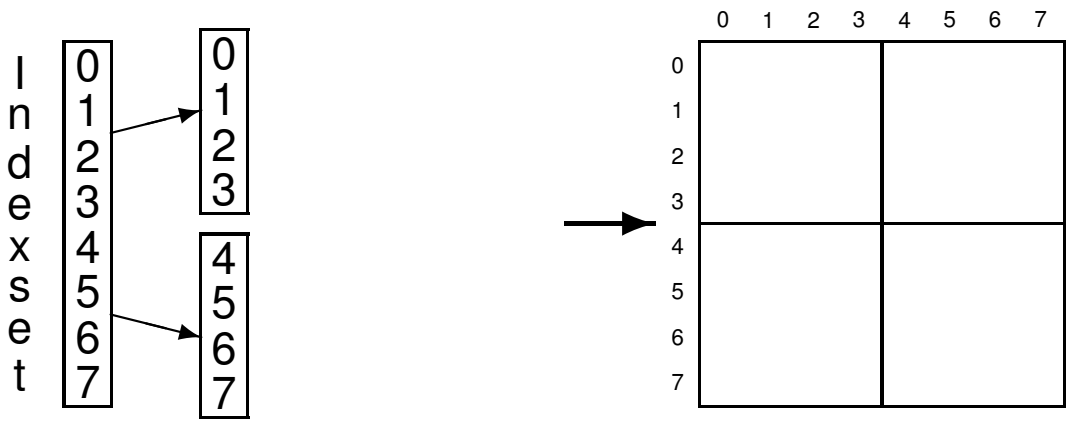
$\mathcal{H}$-matrices

- Index set $I = \{0, \cdots, n-1\}$
- Cluster tree $T(I)$ constructed by *binary space partitioning*,
- depth$(T(I)) = \log_2 n$
- Block cluster tree $T(I \times I)$ with standard admissibility ($\eta = 0.5$)
- Leafs of block cluster tree: $\mathcal{L}(T(I \times I))$



$\mathcal{H}$-matrices

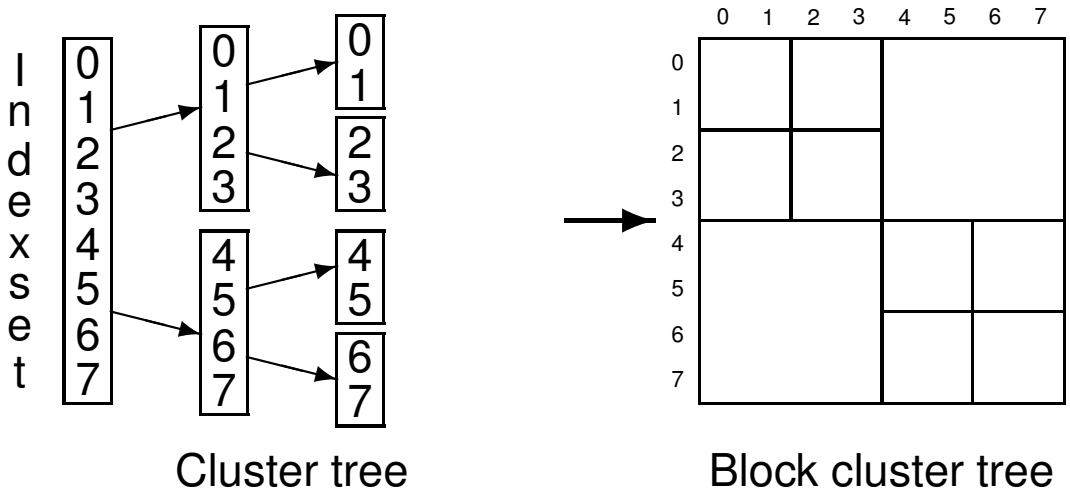
- Index set $I = \{0, \cdots, n-1\}$
- Cluster tree $T(I)$ constructed by *binary space partitioning*,
- depth$(T(I)) = \log_2 n$
- Block cluster tree $T(I \times I)$ with standard admissibility ($\eta = 0.5$)
- Leafs of block cluster tree: $\mathcal{L}(T(I \times I))$



Cluster tree Block cluster tree

$\mathcal{H}$-matrices

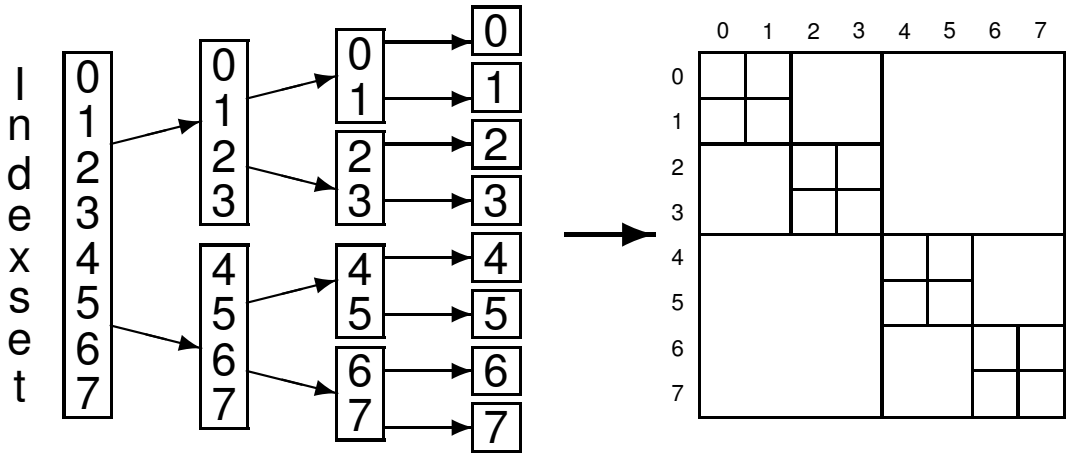
- Index set $I = \{0, \cdots, n-1\}$
- Cluster tree $T(I)$ constructed by *binary space partitioning*,
- depth$(T(I)) = \log_2 n$
- Block cluster tree $T(I \times I)$ with standard admissibility ($\eta = 0.5$)
- Leafs of block cluster tree: $\mathcal{L}(T(I \times I))$



Cluster tree Block cluster tree

$\mathcal{H}$-matrices

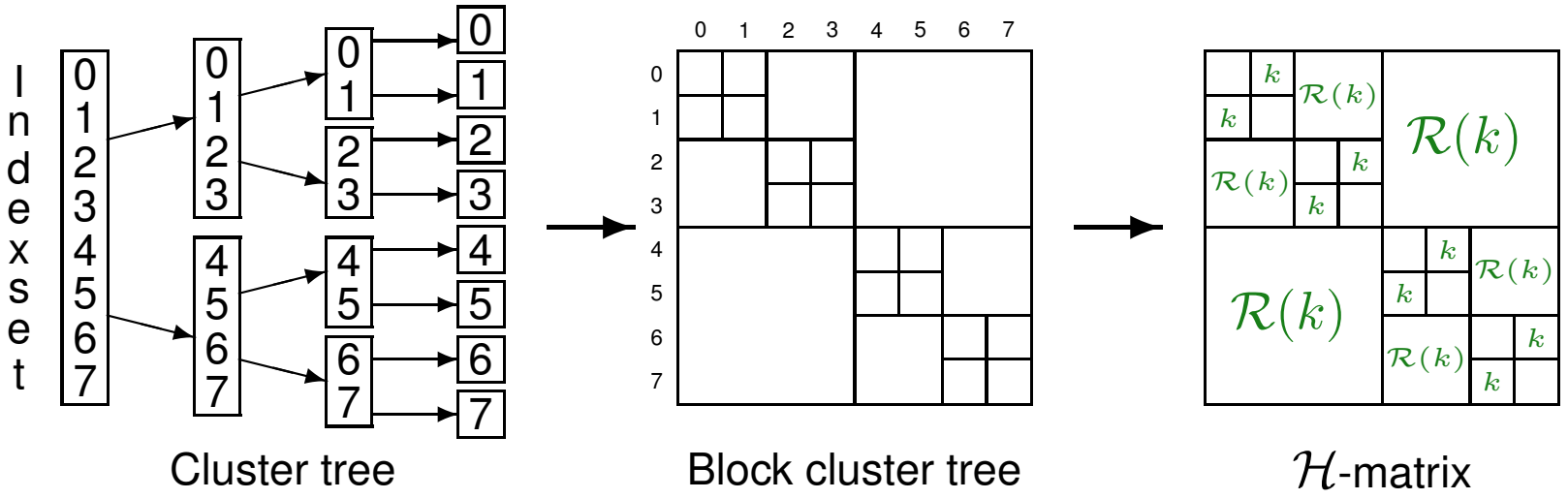
- Index set $I = \{0, \cdots, n-1\}$
- Cluster tree $T(I)$ constructed by *binary space partitioning*,
- depth$(T(I)) = \log_2 n$
- Block cluster tree $T(I \times I)$ with standard admissibility ($\eta = 0.5$)
- Leafs of block cluster tree: $\mathcal{L}(T(I \times I))$



Cluster tree Block cluster tree

$\mathcal{H}$-matrices

- Index set $I = \{0, \cdots, n-1\}$
- Cluster tree $T(I)$ constructed by *binary space partitioning*,
- depth$(T(I)) = \log_2 n$
- Block cluster tree $T(I \times I)$ with standard admissibility ($\eta = 0.5$)
- Leafs of block cluster tree: $\mathcal{L}(T(I \times I))$



Model Problem

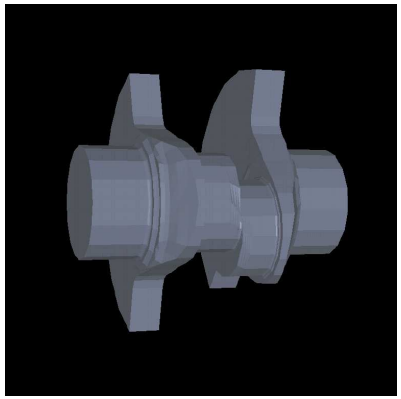
- Problem for all numerical examples: BEM, single layer potential

Model Problem

- Problem for all numerical examples: BEM, single layer potential
- representation by $\mathcal{H}$-matrices, rank-k blocks computed with *ACA*

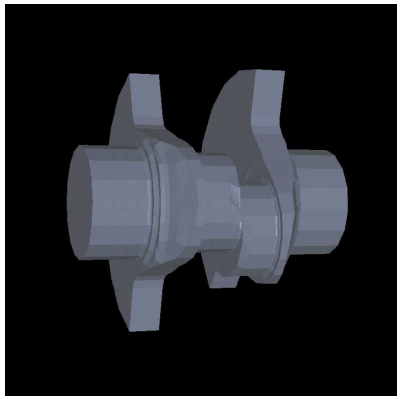
Model Problem

- Problem for all numerical examples: BEM, single layer potential
- representation by $\mathcal{H}$-matrices, rank-k blocks computed with *ACA*
- different geometries; Example:



Model Problem

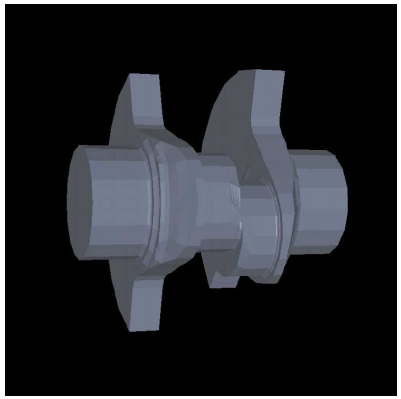
- Problem for all numerical examples: BEM, single layer potential
- representation by $\mathcal{H}$-matrices, rank-k blocks computed with *ACA*
- different geometries; Example:



- computed on parallel system with p processors

Model Problem

- Problem for all numerical examples: BEM, single layer potential
- representation by $\mathcal{H}$-matrices, rank-k blocks computed with *ACA*
- different geometries; Example:



- computed on parallel system with p processors
- Problem chosen because:
  - matrix creation is expensive (compared to FEM)
  - all matrix entries are non-zero (simpler cost function)

Matrix Building

Basic 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

Basic 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

Matrix Building on Shared Memory Systems

Matrix Building on Shared Memory Systems

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

Matrix Building on Shared Memory Systems

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

Matrix Building on Shared Memory Systems

- sufficient to 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;

Matrix Building on Shared Memory Systems

- sufficient to 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: $\frac{t(1)}{t(p)} \geq \frac{p}{\left(2-\frac{1}{p}\right)}$

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

Parallel Algorithm 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 $(\tau, \sigma) \in \mathcal{L}(T(I \times I))$ **do**
    run (build_matrix((τ, σ)));
endfor;

sync_all ();

Numerical Results (Shared Memory)

Fixed accuracy ($\varepsilon = 10^{-4}$)

Time and Parallel Efficiency $E(p) = t(1)/(p \cdot t(p))$:

n	$t(1)$	$E(4)$	$E(8)$	$E(12)$	$E(16)$
4 416	54.4 s	99.3 %	98.6 %	98.6 %	94.4 %
16 128	177.0 s	99.2 %	98.3 %	96.4 %	93.8 %
89 412	2097.9 s	99.2 %	96.6 %	96.7 %	94.1 %

Numerical Results (Shared Memory)

Fixed accuracy ($\varepsilon = 10^{-4}$)

Time and Parallel Efficiency $E(p) = t(1)/(p \cdot t(p))$:

n	$t(1)$	$E(4)$	$E(8)$	$E(12)$	$E(16)$
4 416	54.4 s	99.3 %	98.6 %	98.6 %	94.4 %
16 128	177.0 s	99.2 %	98.3 %	96.4 %	93.8 %
89 412	2097.9 s	99.2 %	96.6 %	96.7 %	94.1 %

- high efficiency even for small problems

Numerical Results (Shared Memory)

Fixed accuracy ($\varepsilon = 10^{-4}$)

Time and Parallel Efficiency $E(p) = t(1)/(p \cdot t(p))$:

n	$t(1)$	$E(4)$	$E(8)$	$E(12)$	$E(16)$
4 416	54.4 s	99.3 %	98.6 %	98.6 %	94.4 %
16 128	177.0 s	99.2 %	98.3 %	96.4 %	93.8 %
89 412	2097.9 s	99.2 %	96.6 %	96.7 %	94.1 %

- high efficiency even for small problems
- low overhead of thread pool

Matrix Building on Distributed Memory Systems

Matrix Building on Distributed Memory Systems

- Goal: no communication needed

Matrix Building on Distributed Memory Systems

- Goal: no communication needed
- using *offline* scheduling: load balancing before computation

Matrix Building on Distributed Memory Systems

- Goal: no communication needed
- using *offline* scheduling: load balancing before computation
- cost function needed:

$$c_k(\tau,\sigma) = \begin{cases} k \cdot (|\tau| + |\sigma|), & (\tau,\sigma) \text{ is admissible} \\ |\tau| \cdot |\sigma|, & \text{else} \end{cases}$$

Matrix Building on Distributed Memory Systems

- Goal: no communication needed
- using *offline* scheduling: load balancing before computation
- cost function needed:

$$c_k(\tau,\sigma) = \begin{cases} k \cdot (|\tau| + |\sigma|), & (\tau,\sigma) \text{ is admissible} \\ |\tau| \cdot |\sigma|, & \text{else} \end{cases}$$

- refined version of list scheduling possible: *Longest Process Time* (*LPT*) scheduling (list scheduling with ordering)

Matrix Building on Distributed Memory Systems

- Goal: no communication needed
- using *offline* scheduling: load balancing before computation
- cost function needed:

$$c_k(\tau,\sigma) = \begin{cases} k \cdot (|\tau| + |\sigma|), & (\tau,\sigma) \text{ is admissible} \\ |\tau| \cdot |\sigma|, & \text{else} \end{cases}$$

- refined version of list scheduling possible: *Longest Process Time* (*LPT*) scheduling (list scheduling with ordering)
- Parallel Speedup:

$$\frac{t(1)}{t(p)} \geq \frac{p}{\left(\frac{4}{3} - \frac{1}{3p}\right)}$$

LPT algorithm:

---

procedure LPT($\mathcal{L}(T(I \times I))$)

$V = \mathcal{L}(T(I \times I))$;

$C_{0 \leq i < p} = 0$;

while $V \neq \emptyset$ **do**

take $v \in V$, s.t. $c_k(v) = \max_{v \in V} c_k(v)$;

choose $i \in \{0, \ldots, p-1\}$, s.t. $C_i = \min_{0 \leq j < p} C_j$;

assign v to processor i;

$C_i = C_i + c_k(v); V = V \setminus \{v\}$;

endwhile;

end;

LPT algorithm:

---

procedure LPT($\mathcal{L}(T(I \times I))$)

$V = \mathcal{L}(T(I \times I))$;

$C_{0 \leq i < p} = 0$;

while $V \neq \emptyset$ **do**

take $v \in V$, s.t. $c_k(v) = \max_{v \in V} c_k(v)$;

choose $i \in \{0, \ldots, p-1\}$, s.t. $C_i = \min_{0 \leq j < p} C_j$;

assign v to processor i;

$C_i = C_i + c_k(v); V = V \setminus \{v\}$;

endwhile;

end;

For fixed accuracy approximation:

- Problem: no fixed rank given

LPT algorithm:

---

procedure LPT($\mathcal{L}(T(I \times I))$)

$V = \mathcal{L}(T(I \times I))$;

$C_{0 \leq i < p} = 0$;

while $V \neq \emptyset$ **do**

take $v \in V$, s.t. $c_k(v) = \max_{v \in V} c_k(v)$;

choose $i \in \{0, \ldots, p-1\}$, s.t. $C_i = \min_{0 \leq j < p} C_j$;

assign v to processor i;

$C_i = C_i + c_k(v); V = V \setminus \{v\}$;

endwhile;

end;

For fixed accuracy approximation:

- Problem: no fixed rank given
- instead choosing an average rank k_{avg} (depending on problem)

Numerical Results (Distributed Memory)

Fixed Rank ($k = 10$)

N	$t(1)$	$E(4)$	$E(8)$	$E(12)$	$E(16)$
4 416	65.1 s	98.6 %	98.0 %	96.9 %	94.6 %
16 128	242.2 s	98.5 %	98.9 %	98.0 %	93.4 %
89 412	*2862.4* s	–	*95.5* %	*96.2* %	*93.3* %

Numerical Results (Distributed Memory)

Fixed Rank ($k = 10$)

N	$t(1)$	$E(4)$	$E(8)$	$E(12)$	$E(16)$
4 416	65.1 s	98.6 %	98.0 %	96.9 %	94.6 %
16 128	242.2 s	98.5 %	98.9 %	98.0 %	93.4 %
89 412	*2862.4* s	–	*95.5* %	*96.2* %	*93.3* %

Fixed Accuracy ($\varepsilon = 10^{-4}$, $k_{\text{avg}} = 10$)

n	$t(1)$	$E(4)$	$E(8)$	$E(12)$	$E(16)$
4 416	63.5 s	98.6 %	98.0 %	96.2 %	92.3 %
16 128	207.6 s	98.5 %	95.1 %	93.5 %	91.4 %
89 412	*2512.0* s	–	*94.0* %	*96.2* %	*91.3* %

Bulk Synchronous Parallel Machine

Bulk Synchronous Parallel Machine

Sequential Systems

Computer model: von Neumann machine

Bulk Synchronous Parallel Machine

Sequential Systems

Computer model: von Neumann machine

- CPU: executes instructions

Bulk Synchronous Parallel Machine

Sequential Systems

Computer model: von Neumann machine

- CPU: executes instructions
- Memory: stores data and programs

Bulk Synchronous Parallel Machine

Sequential Systems

Computer model: von Neumann machine

- CPU: executes instructions
- Memory: stores data and programs
- Bus: for transfer of data and programs between CPU and Memory

Bulk Synchronous Parallel Machine

Sequential Systems

Computer model: von Neumann machine

- CPU: executes instructions
- Memory: stores data and programs
- Bus: for transfer of data and programs between CPU and Memory

Hence

- programming for all systems follows same rules

Bulk Synchronous Parallel Machine

Sequential Systems

Computer model: von Neumann machine

- CPU: executes instructions
- Memory: stores data and programs
- Bus: for transfer of data and programs between CPU and Memory

Hence

- programming for all systems follows same rules
- performance of programs is predictable: by speed of CPU and memory

Parallel Systems

No similar model for parallel computers available

Parallel Systems

No similar model for parallel computers available

Instead:

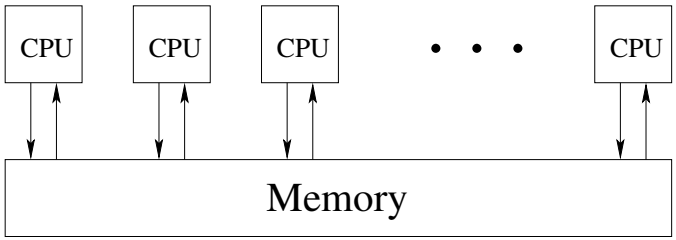
- direct parallelisation: *PVM*, *MPI* etc.
  - Advantage: for many systems available
  - Disadvantage: complicated programming, good knowledge of hardware necessary

Parallel Systems

No similar model for parallel computers available

Instead:

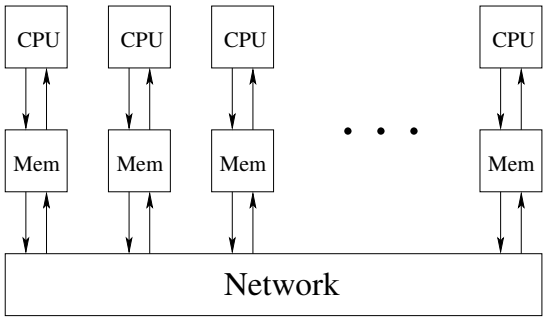
- direct parallelisation: *PVM*, *MPI* etc.
  - Advantage: for many systems available
  - Disadvantage: complicated programming, good knowledge of hardware necessary
- PRAM: p CPUs, global memory shared by all processors



  - Advantage: simple programming, no communication
  - Disadvantage: only for real shared memory systems (not found in practice)

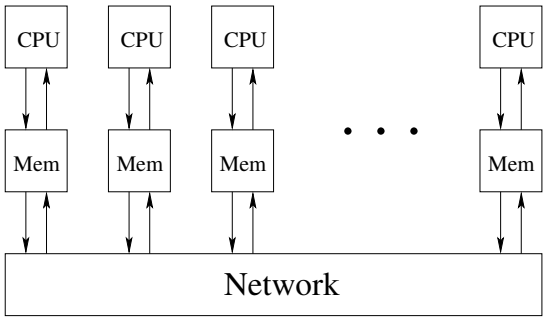
Bulk Synchronous Parallel (BSP) Machine

Most computer systems (shared and distributed memory) have local memory and a network:



Bulk Synchronous Parallel (BSP) Machine

Most computer systems (shared and distributed memory) have local memory and a network:



BSP model based on 3 parameters:

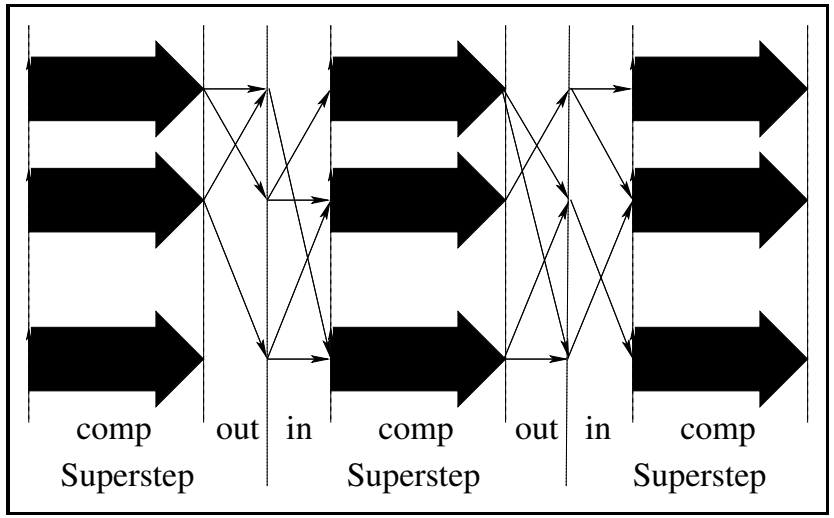
p = the number of processors

g = speed of all CPUs / speed of network ($\sim$ bandwidth)

l = time for a global synchronisation

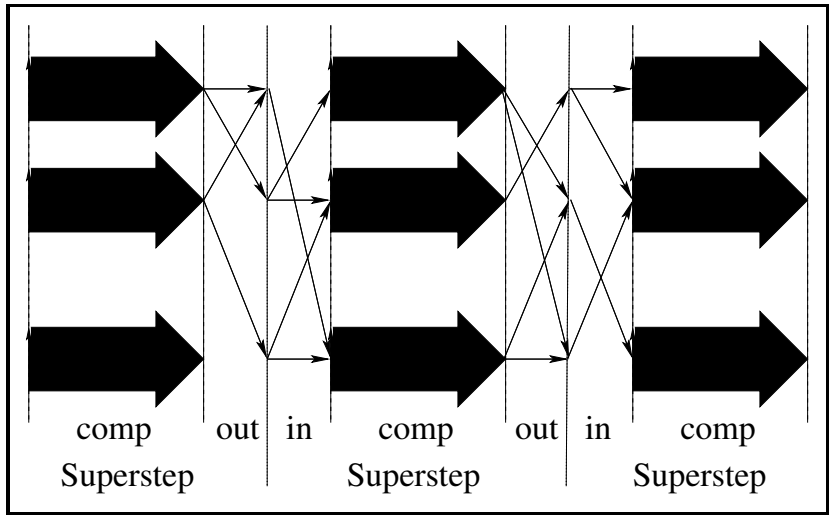
BSP computation

Sequence of *Supersteps*:



BSP computation

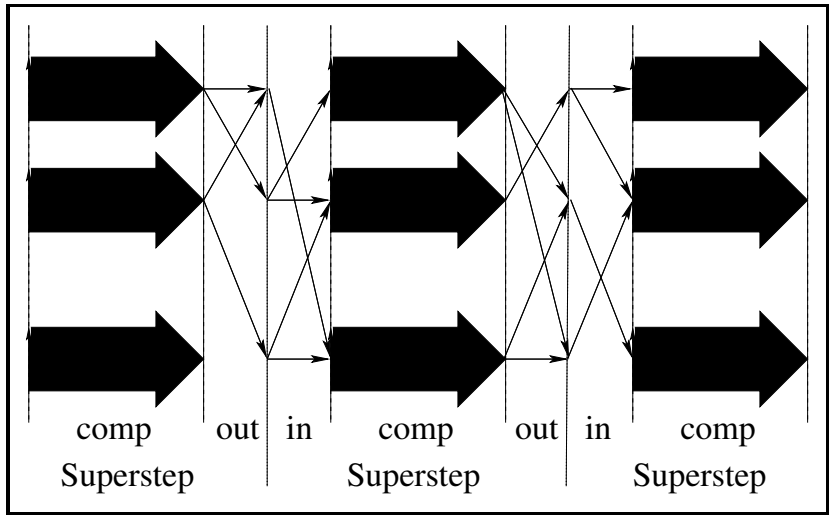
Sequence of *Supersteps*:



Analysis of BSP algorithms:

BSP computation

Sequence of *Supersteps*:

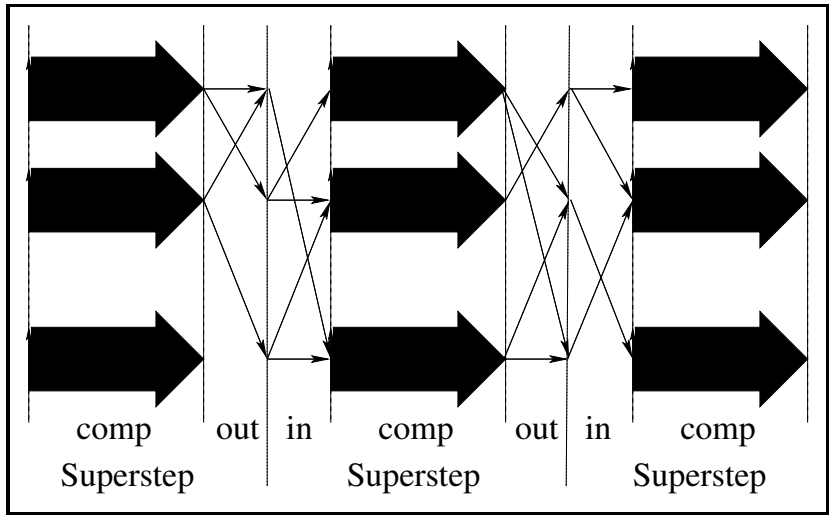


Analysis of BSP algorithms:

- for each superstep:
  - amount of work w, maximal size of data sent/received h

BSP computation

Sequence of *Supersteps*:

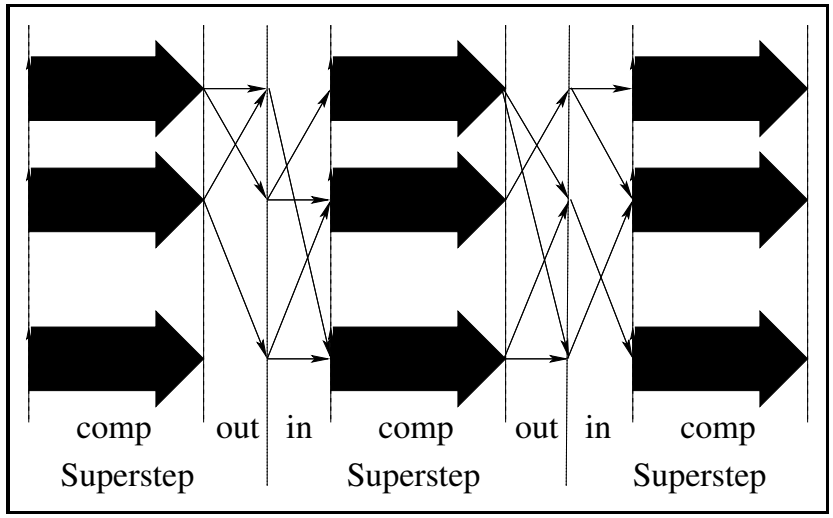


Analysis of BSP algorithms:

- for each superstep:
  - amount of work w, maximal size of data sent/received h
  - costs per step: $w + g \cdot h + l$

BSP computation

Sequence of *Supersteps*:



Analysis of BSP algorithms:

- for each superstep:
  - amount of work w, maximal size of data sent/received h
  - costs per step: $w + g \cdot h + l$
- summing up S supersteps yields expression:

$$W + g \cdot H + l \cdot S$$

Matrix-Vector Multiplication (Distributed Memory)

Multiplication for Dense Matrices

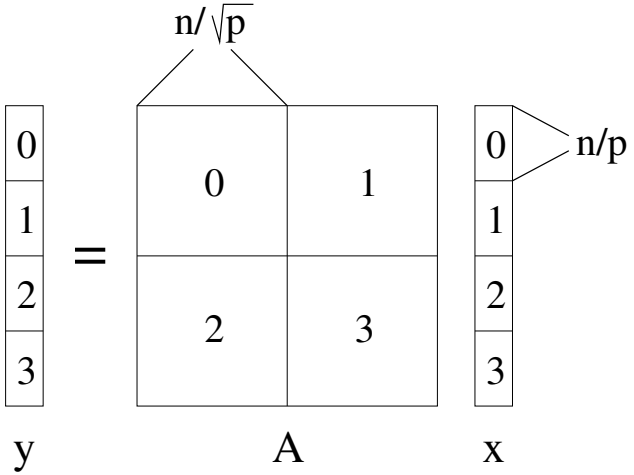
Matrix-Vector Multiplication (Distributed Memory)

Multiplication for Dense Matrices

considered: $y = \alpha Ax + \beta y$

data distribution (per processor i):

- x_i, y_i: n/p elements
- A: n^2/p elements



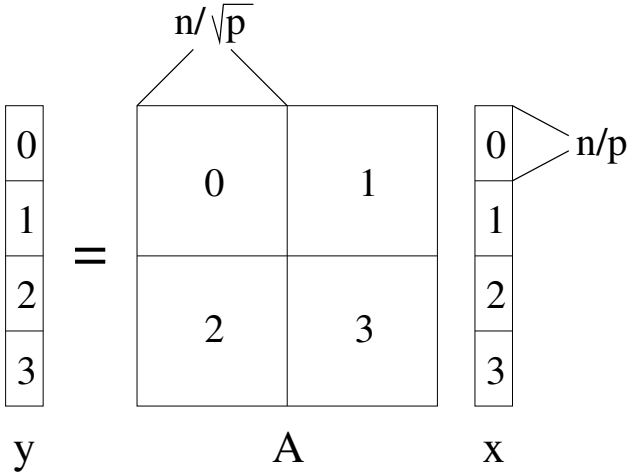
Matrix-Vector Multiplication (Distributed Memory)

Multiplication for Dense Matrices

considered: $y = \alpha Ax + \beta y$

data distribution (per processor i):

- x_i, y_i: n/p elements
- A: n^2/p elements



procedure dense_mul(α, A, x, β, y)

{ *first step* }

$y_i = \beta \cdot y_i$;

send x_i to all processors sharing it;

sync();

{ *second step* }

$y'_i = \alpha A x_i$;

send y'_i to all processors sharing it;

sync();

{ *third step* }

$y_i = y_i + \sum y'_j$;

sync();

end;

Complexity of dense Matrix-Vector Multiplication:

$$\mathcal{W}(n,p) = \mathcal{O}\left(\frac{n^2}{p} + \frac{n}{\sqrt{p}}\right) + g \cdot \mathcal{O}\left(\frac{n}{\sqrt{p}}\right) + 3 \cdot l$$

Complexity of dense Matrix-Vector Multiplication:

$$\mathcal{W}(n,p) = \mathcal{O}\left(\frac{n^2}{p} + \frac{n}{\sqrt{p}}\right) + g \cdot \mathcal{O}\left(\frac{n}{\sqrt{p}}\right) + 3 \cdot l$$

Multiplication for $\mathcal{H}$-matrices

Complexity of dense Matrix-Vector Multiplication:

$$\mathcal{W}(n,p) = \mathcal{O}\left(\frac{n^2}{p} + \frac{n}{\sqrt{p}}\right) + g \cdot \mathcal{O}\left(\frac{n}{\sqrt{p}}\right) + 3 \cdot l$$

Multiplication for $\mathcal{H}$-matrices

- Idea: same algorithm, different data distribution

Complexity of dense Matrix-Vector Multiplication:

$$\mathcal{W}(n,p) = \mathcal{O}\left(\frac{n^2}{p} + \frac{n}{\sqrt{p}}\right) + g \cdot \mathcal{O}\left(\frac{n}{\sqrt{p}}\right) + 3 \cdot l$$

Multiplication for $\mathcal{H}$-matrices

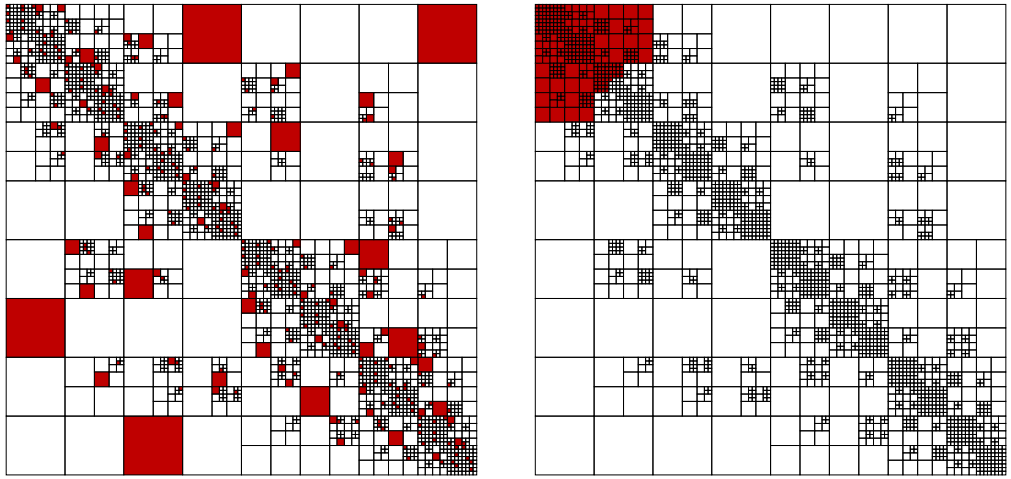
- Idea: same algorithm, different data distribution
- Problem: How to distribute A ?

Complexity of dense Matrix-Vector Multiplication:

$$\mathcal{W}(n,p) = \mathcal{O}\left(\frac{n^2}{p} + \frac{n}{\sqrt{p}}\right) + g \cdot \mathcal{O}\left(\frac{n}{\sqrt{p}}\right) + 3 \cdot l$$

Multiplication for $\mathcal{H}$-matrices

- Idea: same algorithm, different data distribution
- Problem: How to distribute A ?



Sharing Constant

Definition: Let $\mathcal{L}(T, q)$ denote the leaves associated to processor q. For $i \in I$ let the *sharing constant* $c_{\text{sh}}(i)$ be defined as

$$c_{\text{sh}}(i) = \max\{c^{\tau}_{\text{sh}}(i), c^{\sigma}_{\text{sh}}(i)\}$$

with

$$\begin{aligned} c^{\tau}_{\text{sh}}(i) &= |\{q \,|\, \exists(\tau, \sigma) \in \mathcal{L}(T, q) : i \in \tau\}| \quad \text{and} \\ c^{\sigma}_{\text{sh}}(i) &= |\{q \,|\, \exists(\tau, \sigma) \in \mathcal{L}(T, q) : i \in \sigma\}|. \end{aligned}$$

For the index set I let the sharing constant be: $c_{\text{sh}}(I) = \max_{i \in I} c_{\text{sh}}(i)$.

Sharing Constant

Definition: Let $\mathcal{L}(T, q)$ denote the leaves associated to processor q. For $i \in I$ let the *sharing constant* $c_{\text{sh}}(i)$ be defined as

$$c_{\text{sh}}(i) = \max\{c_{\text{sh}}^{\tau}(i), c_{\text{sh}}^{\sigma}(i)\}$$

with

$$\begin{aligned} c_{\text{sh}}^{\tau}(i) &= |\{q \,|\, \exists(\tau, \sigma) \in \mathcal{L}(T, q) : i \in \tau\}| \quad \text{and} \\ c_{\text{sh}}^{\sigma}(i) &= |\{q \,|\, \exists(\tau, \sigma) \in \mathcal{L}(T, q) : i \in \sigma\}|. \end{aligned}$$

For the index set I let the sharing constant be: $c_{\text{sh}}(I) = \max_{i \in I} c_{\text{sh}}(i)$.

(Rewritten) Complexity of dense Matrix-Vector multiplication ($c_{\text{sh}} = \sqrt{p}$):

$$\mathcal{W}(n, p) = \mathcal{O}\left(\frac{n^2}{p} + \frac{c_{\text{sh}} n}{p}\right) + g \cdot \mathcal{O}\left(\frac{c_{\text{sh}} n}{p}\right) + 3 \cdot l$$

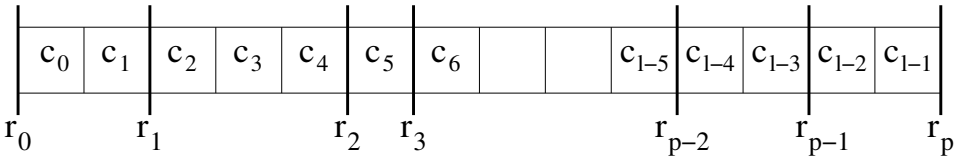
Load Balancing

Goal: minimise c_{sh}.

Load Balancing

Goal: minimise c_{sh}.

Sequence Partitioning



For (ordered) list of costs $c_0, c_1, \ldots, c_{l-1}$ find set of delimiters $r_0 = 0, \ldots, r_p = l - 1$, s.t.

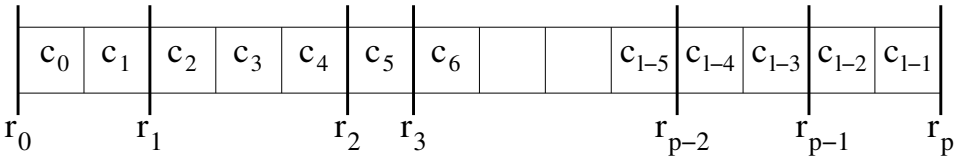
$$\max_{0 \leq i < p} \sum_{j=r_i}^{r_{i+1}} c_j$$

is minimised.

Load Balancing

Goal: minimise c_{sh}.

Sequence Partitioning



For (ordered) list of costs $c_0, c_1, \ldots, c_{l-1}$ find set of delimiters $r_0 = 0, \ldots, r_p = l - 1$, s.t.

$$\max_{0 \leq i < p} \sum_{j=r_i}^{r_{i+1}} c_j$$

is minimised.

Sequence partitioning problem can be solved in: $\mathcal{O}(l \cdot p)$.

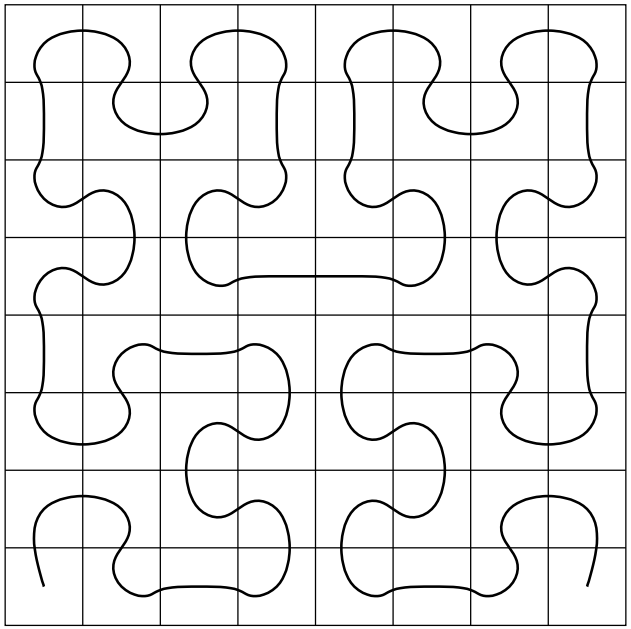
Space-filling Curves

Ordering for $\mathcal{L}(T(I \times I))$ is done by *space-filling curves*.

Space-filling Curves

Ordering for $\mathcal{L}(T(I \times I))$ is done by *space-filling curves*.

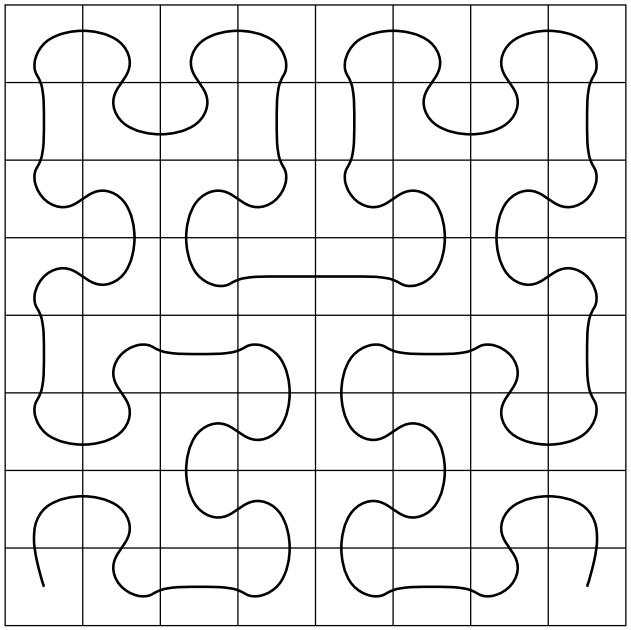
Hilbert curve



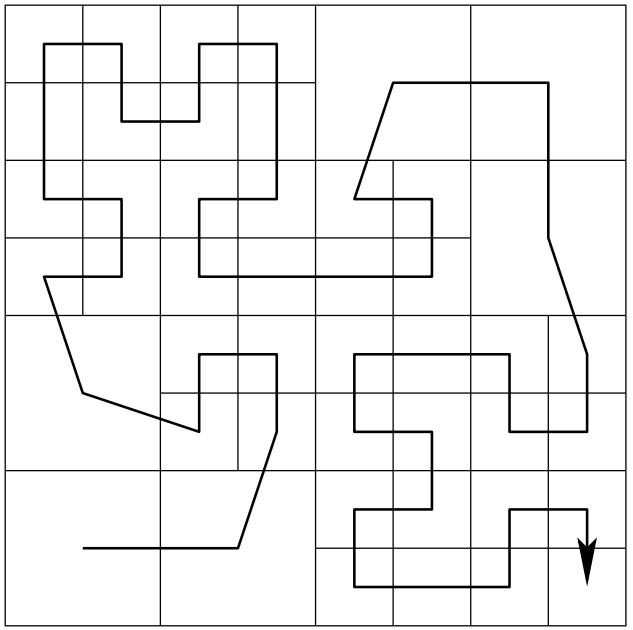
Space-filling Curves

Ordering for $\mathcal{L}(T(I \times I))$ is done by *space-filling curves*.

Hilbert curve



Applied to $\mathcal{L}(T(I \times I))$

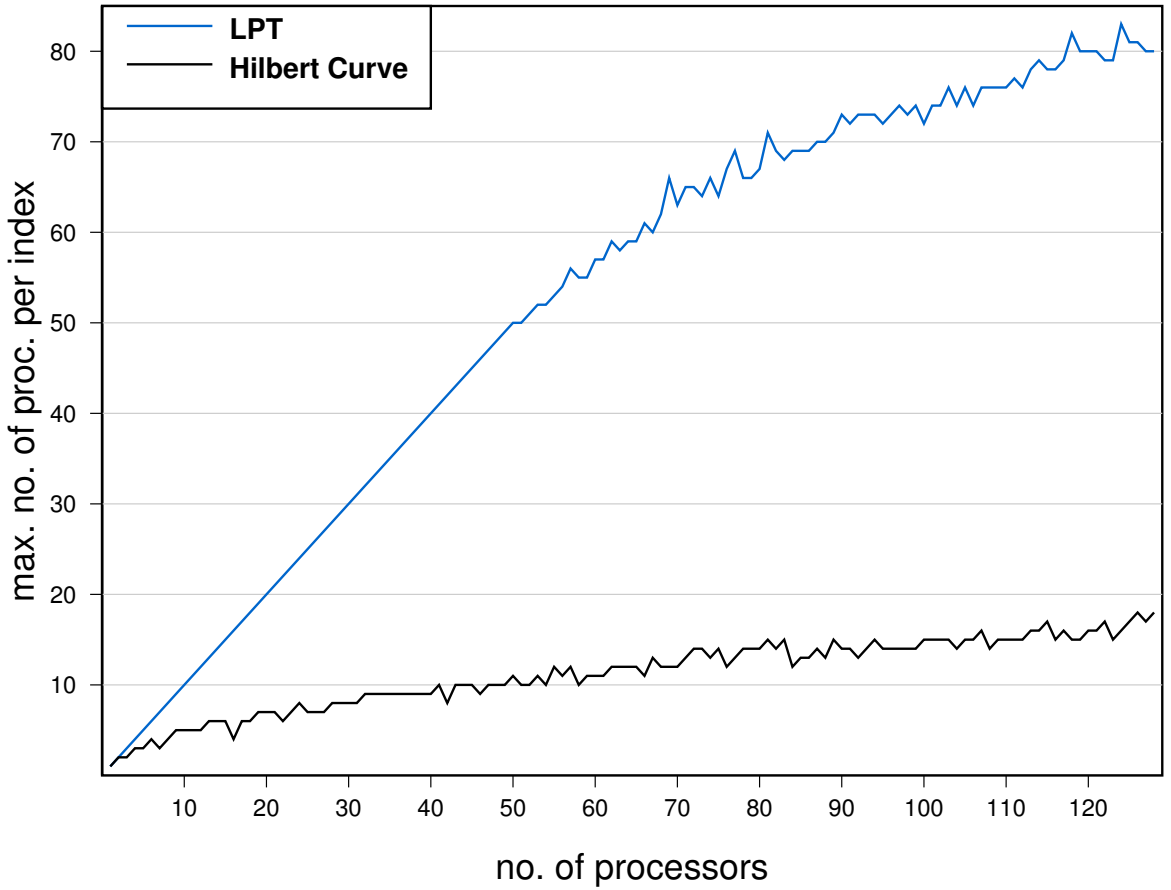


Numerical Results

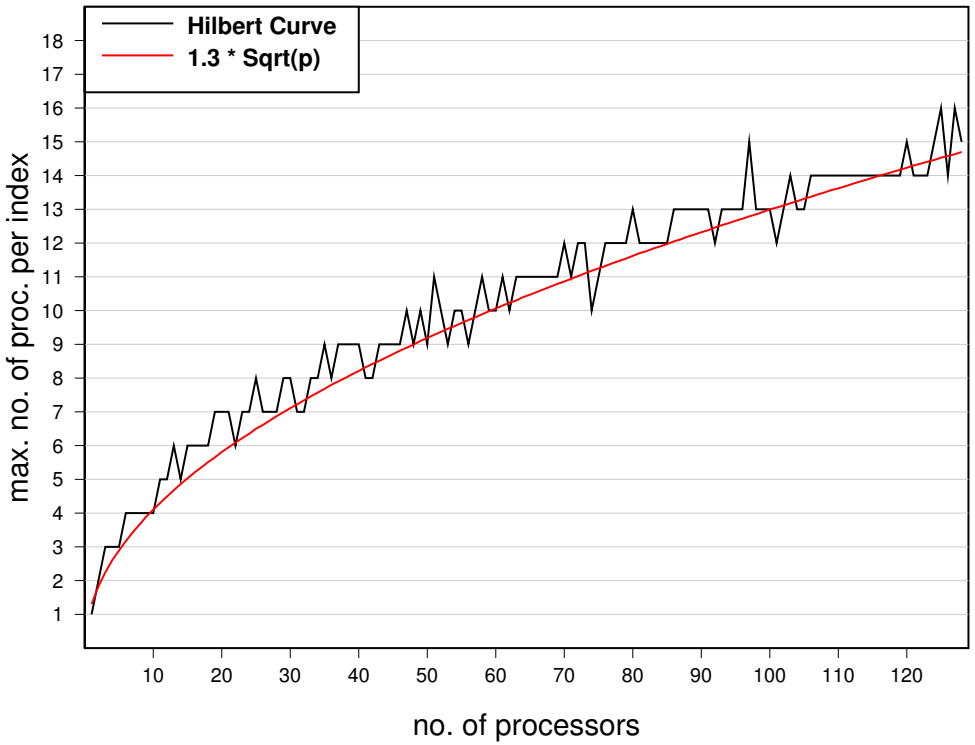
Numerical Results

Values of c_{sh}

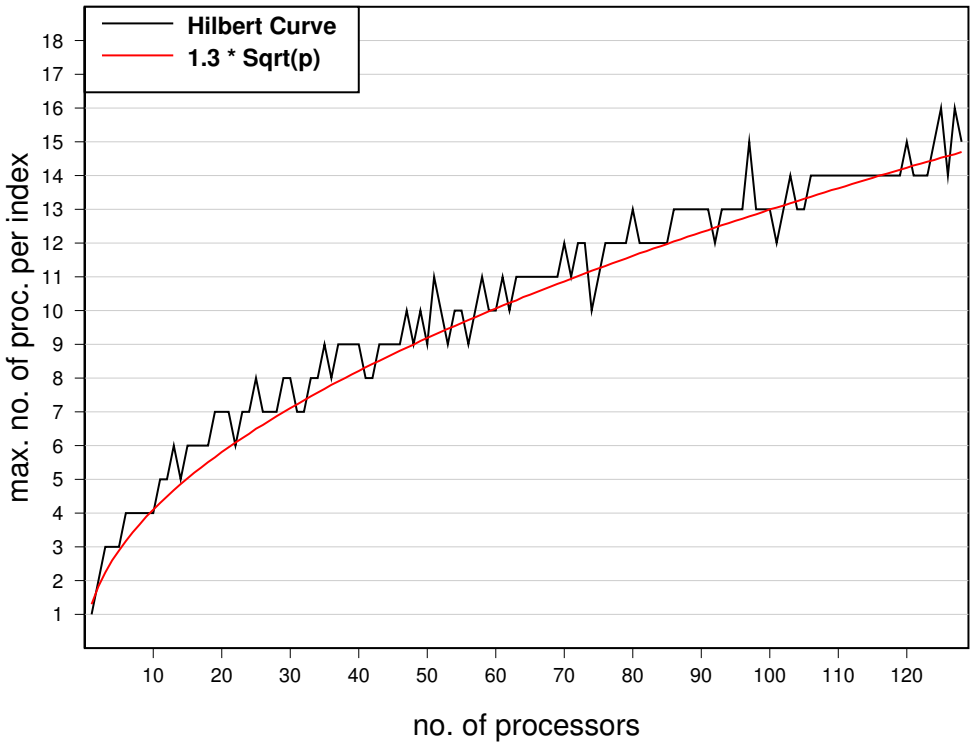
LPT scheduling vs. sequence part. with space-filling curves:



Hilbert curve:

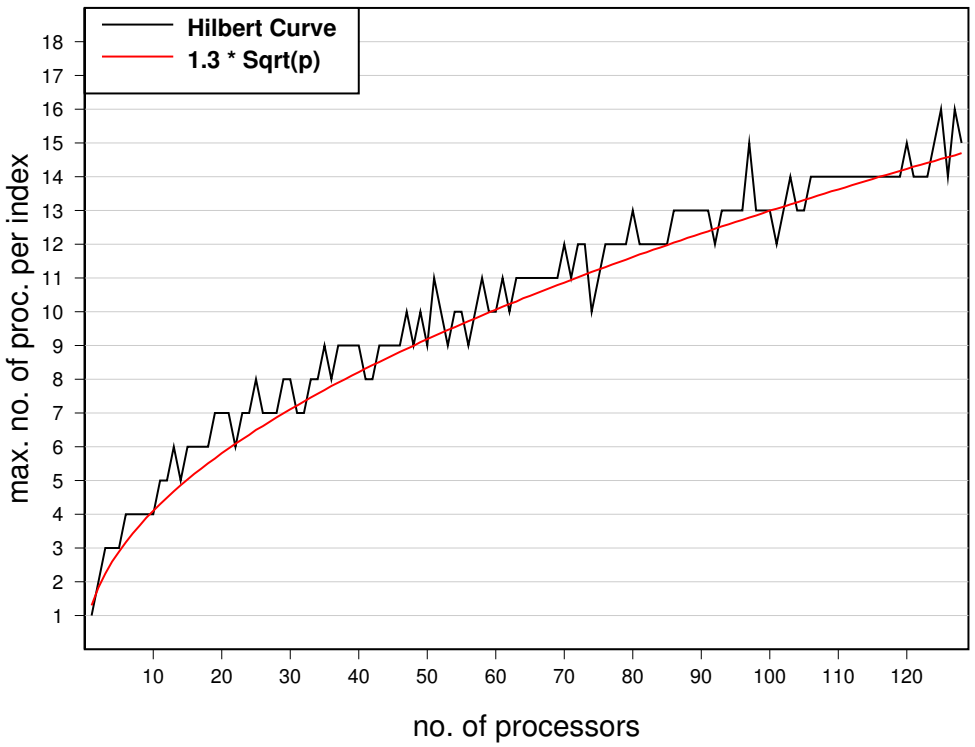


Hilbert curve:



With space-filling curves and sequence partitioning: $c_{\text{sh}} = \mathcal{O}\left(\sqrt{p}\right)$ and

Hilbert curve:



With space-filling curves and sequence partitioning: $c_{\text{sh}} = \mathcal{O}\left(\sqrt{p}\right)$ and

$$\mathcal{W}(n,p) = \mathcal{O}\left(\frac{n \log n}{p} + \frac{n}{\sqrt{p}}\right) + g \cdot \mathcal{O}\left(\frac{n}{\sqrt{p}}\right) + 3 \cdot l$$

Time and Parallel Efficiency

Fixed rank ($k = 10$):

n	$t(1)$	$E(4)$	$E(8)$	$E(12)$	$E(16)$
4 416	0.33 s	90.4 %	82.9 %	74.2 %	69.1 %
16 128	1.20 s	87.7 %	84.1 %	79.6 %	76.0 %
89 412	–	–	1.98 s	*97.5* %	*95.9* %

Time and Parallel Efficiency

Fixed rank ($k = 10$):

n	$t(1)$	$E(4)$	$E(8)$	$E(12)$	$E(16)$
4 416	0.33 s	90.4 %	82.9 %	74.2 %	69.1 %
16 128	1.20 s	87.7 %	84.1 %	79.6 %	76.0 %
89 412	–	–	1.98 s	*97.5* %	*95.9* %

Fixed accuracy ($\varepsilon = 10^{-4}, k_{\text{avg}} = 10$):

n	$t(1)$	$E(4)$	$E(8)$	$E(12)$	$E(16)$
4 416	0.32 s	90.1 %	82.8 %	74.6 %	68.2 %
16 128	1.03 s	83.3 %	87.2 %	79.6 %	75.8 %
89 412	–	–	1.69 s	*96.1* %	*93.0* %

Matrix-Vector Multiplication (Shared Memory)

- On shared memory systems: no communication needed

Matrix-Vector Multiplication (Shared Memory)

- On shared memory systems: no communication needed
- focus on matrix-vector multiplication and vector summation, not vector transfer

Matrix-Vector Multiplication (Shared Memory)

- On shared memory systems: no communication needed
- focus on matrix-vector multiplication and vector summation, not vector transfer
- hence minimise c_{sh}^{τ}, not c_{sh}

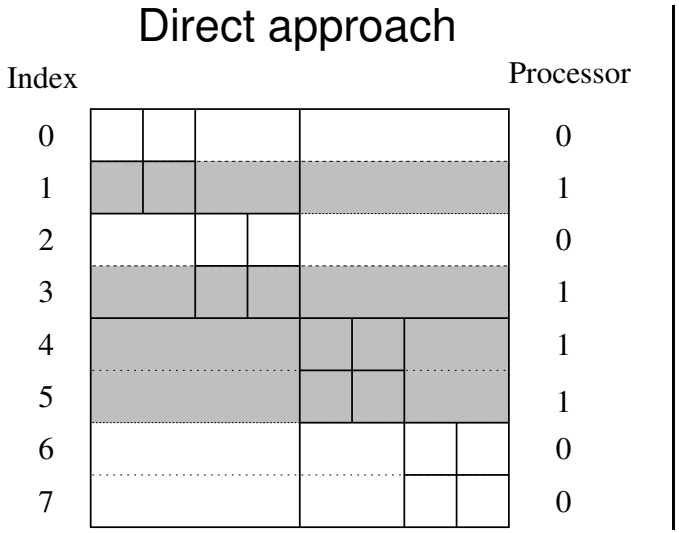
Matrix-Vector Multiplication (Shared Memory)

- On shared memory systems: no communication needed
- focus on matrix-vector multiplication and vector summation, not vector transfer
- hence minimise c_{sh}^{τ}, not c_{sh} (optimal value: $c_{\text{sh}}^{\tau} = 1$)

Matrix-Vector Multiplication (Shared Memory)

- On shared memory systems: no communication needed
- focus on matrix-vector multiplication and vector summation, not vector transfer
- hence minimise c_{sh}^{τ}, not c_{sh} (optimal value: $c_{\text{sh}}^{\tau} = 1$)

Resulting distribution: based on cluster tree $T(I)$

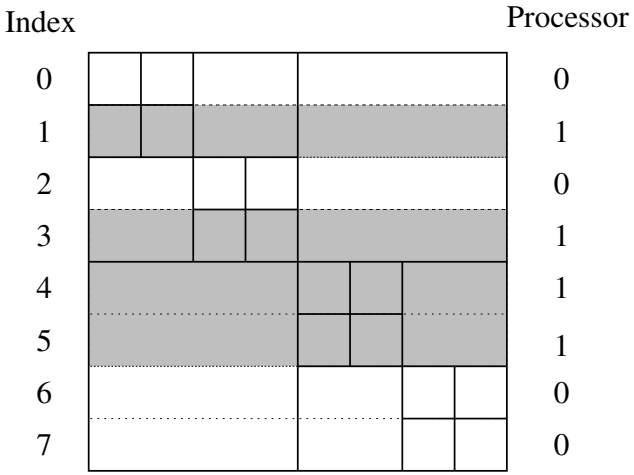


Matrix-Vector Multiplication (Shared Memory)

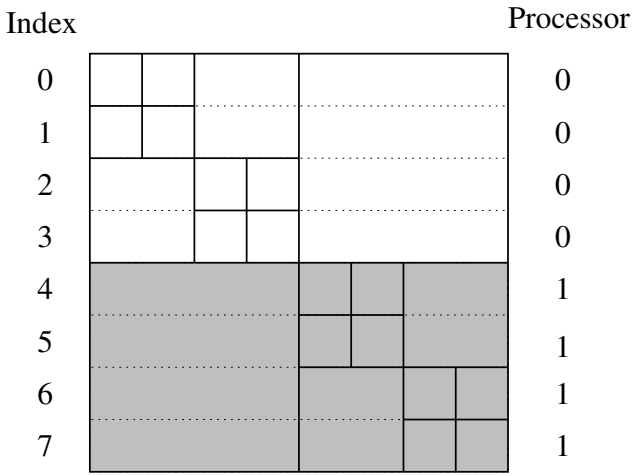
- On shared memory systems: no communication needed
- focus on matrix-vector multiplication and vector summation, not vector transfer
- hence minimise c_{sh}^{τ}, not c_{sh} (optimal value: $c_{\text{sh}}^{\tau} = 1$)

Resulting distribution: based on cluster tree $T(I)$

Direct approach



Contiguous index sets



Shared Rank-k Matrix Blocks

- rank-k block represented as $M = A \cdot B^T$

Shared Rank-k Matrix Blocks

- rank-k block represented as $M = A \cdot B^T$
- but load balancing w.r.t. c^{τ}_{sh}, hence, for $M \cdot x$,

  $B^T \cdot x$ is execute on each processor sharing M

Shared Rank-k Matrix Blocks

- rank-k block represented as $M = A \cdot B^T$
- but load balancing w.r.t. c^{τ}_{sh}, hence, for $M \cdot x$,

  $B^T \cdot x$ is execute on each processor sharing M
- for good parallel efficiency: distribute partial multiplications $B^T \cdot x$

Shared Rank-k Matrix Blocks

- rank-k block represented as $M = A \cdot B^T$
- but load balancing w.r.t. c^{τ}_{sh}, hence, for $M \cdot x$,

  $B^T \cdot x$ is execute on each processor sharing M
- for good parallel efficiency: distribute partial multiplications $B^T \cdot x$

Load Balancing

- use sequence partitioning for cluster tree $T(I)$: contiguous index sets per processor

Shared Rank-k Matrix Blocks

- rank-k block represented as $M = A \cdot B^T$
- but load balancing w.r.t. c^{τ}_{sh}, hence, for $M \cdot x$,

  $B^T \cdot x$ is execute on each processor sharing M
- for good parallel efficiency: distribute partial multiplications $B^T \cdot x$

Load Balancing

- use sequence partitioning for cluster tree $T(I)$: contiguous index sets per processor
- use LPT scheduling for partial multiplications

BSP

procedure shm_mul($i, \alpha, A, x, \beta, y$)

{ *step 1* }

$y_i = \beta \cdot y_i$;

send x_i to all processors sharing it;

sync();

{ *step 2* }

compute partial rank-k products on proc i;

send results to all sharing processors;

sync();

{ *step 3* }

$y'_i = \alpha A x_i$;

send y'_i to all processors sharing it;

sync();

{ *step 4* }

$y_i = y_i + \sum y'_j$ (disjoint summation);

sync();

end;

BSP

procedure shm_mul($i, \alpha, A, x, \beta, y$)

{ *step 1* }

$y_i = \beta \cdot y_i$;

send x_i to all processors sharing it;

sync();

{ *step 2* }

compute partial rank-k products on proc i;

send results to all sharing processors;

sync();

{ *step 3* }

$y'_i = \alpha A x_i$;

send y'_i to all processors sharing it;

sync();

{ *step 4* }

$y_i = y_i + \sum y'_j$ (disjoint summation);

sync();

end;

Thread Pool

procedure step_1 (i, β, y, A)

$y_i = \beta \cdot y_i$;

compute partial products on proc i;

end;

procedure step_2 (i, α, A, x, y)

$y_i = y_i + \alpha A x_i$;

end;

procedure shm_mul_tp($i, \alpha, A, x, \beta, y$)

for $0 \leq i < p$ **do**

run(step_1(i, β, y, A));

sync_all();

for $0 \leq i < p$ **do**

run(step_2(i, α, A, x, y));

sync_all();

end;

Complexity of Shared Memory Multiplication

Complexity of BSP algorithm:

$$\mathcal{W}(n,p) = \mathcal{O}\left(\frac{n \log n}{p}\right) + g \cdot \mathcal{O}(n) + 4 \cdot l$$

Complexity of Shared Memory Multiplication

Complexity of BSP algorithm:

$$\mathcal{W}(n,p) = \mathcal{O}\left(\frac{n \log n}{p}\right) + g \cdot \mathcal{O}(n) + 4 \cdot l$$

Complexity of thread pool algorithm:

$$\mathcal{W}(n,p) = \mathcal{O}\left(\frac{n \log n}{p}\right)$$

Complexity of Shared Memory Multiplication

Complexity of BSP algorithm:

$$\mathcal{W}(n,p) = \mathcal{O}\left(\frac{n \log n}{p}\right) + g \cdot \mathcal{O}(n) + 4 \cdot l$$

Complexity of thread pool algorithm:

$$\mathcal{W}(n,p) = \mathcal{O}\left(\frac{n \log n}{p}\right)$$

Numerical Results

Fixed accuracy ($\varepsilon = 10^{-4}$, no k_{avg} necessary !):

n	$t(1)$	$E(4)$	$E(8)$	$E(12)$	$E(16)$
4 416	0.19 s	92.2 %	87.0 %	–	–
16 128	0.62 s	90.0 %	87.4 %	–	–
89 412	6.64 s	90.7 %	88.8 %	85.5 %	84.0 %