Computing with limited memory

Loris Marchal (CNRS, Lyon, France)
loris.marchal@ens-lyon.fr

November 19, 2013

Outline

Introduction and motivation

Minimize I/O in out-of-core matrix computations
- Naïve and optimized algorithms for matrix product
- Lower bound on the I/O volume
- Extending lower bounds to other operations
- Cache-oblivious algorithms

Memory-Aware DAGs scheduling
- Pebble game
- Optimal depth-first and general traversals
- Complexity of parallel tree processing
- Practical solutions for limited memory

Conclusion

Outline

Introduction

Usual performance metric: makespan
(or other time-related metric)

Today: focus on memory

- Workflows with large temporary data
- Bad evolution of perf. for computation vs. communication: $1/\text{Flops} \ll 1/\text{bandwidth} \ll \text{latency}$
- Gap between processing power and communication cost increasing exponentially

	annual improvements
Flops rate	59%
mem. bandwidth	26%
mem. latency	5%

- Avoid communications (I/O)
- Restrict to in-core memory (out-of-core is expensive)

Introduction

Usual performance metric: makespan
(or other time-related metric)

Today: focus on memory

- Workflows with large temporary data
- Bad evolution of perf. for computation vs. communication: $1/\text{Flops} \ll 1/\text{bandwidth} \ll \text{latency}$
- Gap between processing power and communication cost increasing exponentially

	annual improvements
Flops rate	59%
mem. bandwidth	26%
mem. latency	5%

- Avoid communications (I/O)
- Restrict to in-core memory (out-of-core is expensive)

Introduction

Usual performance metric: makespan
(or other time-related metric)

Today: focus on memory

- Workflows with large temporary data
- Bad evolution of perf. for computation vs. communication: $1/\text{Flops} \ll 1/\text{bandwidth} \ll \text{latency}$
- Gap between processing power and communication cost increasing exponentially

	annual improvements
Flops rate	59%
mem. bandwidth	26%
mem. latency	5%

- Avoid communications (I/O)
- Restrict to in-core memory (out-of-core is expensive)

Outline

Model

Out-of-core execution:

- Fast memory of size M
- M is to small to accomodate all data
- Unlimited disk space
- Disk access are slow: minimize read/write (I/O)

Applies to other two-level systems:

- Fast but limited cache / Large and slower memory
- Fast but limited L1 cache / Large and slower L2/L3 cache

Model

Out-of-core execution:

- Fast memory of size M
- M is to small to accomodate all data
- Unlimited disk space
- Disk access are slow: minimize read/write (I/O)

Applies to other two-level systems:

- Fast but limited cache / Large and slower memory
- Fast but limited L1 cache / Large and slower L2/L3 cache

Outline

Introduction and motivation

Minimize I/O in out-of-core matrix computations

- Naïve and optimized algorithms for matrix product
- Lower bound on the I/O volume
- Extending lower bounds to other operations
- Cache-oblivious algorithms

Memory-Aware DAGs scheduling

- Pebble game
- Optimal depth-first and general traversals
- Complexity of parallel tree processing
- Practical solutions for limited memory

Conclusion

Basic matrix-product algorithm: analysis

```
naive-matrix-multiply(n,C,A,B)
for i = 1 to n
  for j = 1 to n C[i,j] = 0
    for k = 1 to n
      C[i,j] = C[i,j] + A[i,k] * B[k,j]
    end for
  end for
end for
```

- how many I/O operations with a memory of size M
- assumption: $M < n^2/2$
- all B elements accessed during outer loop: at least $n^2/2$ reads
- total: at least $n^3/2$ read (at most $4n^3$ read/write)

Basic matrix-product algorithm: analysis

```
naive-matrix-multiply(n,C,A,B)
for i = 1 to n
  for j = 1 to n C[i,j] = 0
    for k = 1 to n
      C[i,j] = C[i,j] + A[i,k] * B[k,j]
    end for
  end for
end for
```

- how many I/O operations with a memory of size M
- assumption: $M < n^2/2$
- all B elements accessed during outer loop: at least $n^2/2$ reads
- total: at least $n^3/2$ read (at most $4n^3$ read/write)

Basic matrix-product algorithm: analysis

```
naive-matrix-multiply(n,C,A,B)
for i = 1 to n
  for j = 1 to n C[i,j] = 0
    for k = 1 to n
      C[i,j] = C[i,j] + A[i,k] * B[k,j]
    end for
  end for
end for
```

- how many I/O operations with a memory of size M
- assumption: $M < n^2/2$
- all B elements accessed during outer loop: at least $n^2/2$ reads
- total: at least $n^3/2$ read (at most $4n^3$ read/write)

Basic matrix-product algorithm: analysis

```
naive-matrix-multiply(n,C,A,B)
for i = 1 to n
  for j = 1 to n C[i,j] = 0
    for k = 1 to n
      C[i,j] = C[i,j] + A[i,k] * B[k,j]
    end for
  end for
end for
```

- how many I/O operations with a memory of size M
- assumption: $M < n^2/2$
- all B elements accessed during outer loop: at least $n^2/2$ reads
- total: at least $n^3/2$ read (at most $4n^3$ read/write)

Matrix-product algorithm: how to do better ?

Idea: use blocks of size $\sqrt{M}/3$

```
blocked-matrix-multiply(n,C,A,B)
b = square root of (memory size/3)
for i = 1 to n step b
  for j = 1 to n step b
    fill C[i:i+b-1,j:j+b-1] with zeros
    for k = 1 to n step b
      naive-matrix-multiply(b,C[i:i+b-1,j:j+b-1],
                              A[i:i+b-1,k:k+b-1],
                              B[k:k+b-1,j:j+b-1])
    end for
  end for
end for
```

- each iteration of the inner loop accesses only $3b^2 = M$ data: each data is read/written only once
- bound on the number of transfers: $(n/b)^3 \times 2M = (n/\sqrt{M/3})^3 \times 2M = O(n^3/\sqrt{M})$

Matrix-product algorithm: how to do better ?

Idea: use blocks of size $\sqrt{M}/3$

```
blocked-matrix-multiply(n,C,A,B)
b = square root of (memory size/3)
for i = 1 to n step b
  for j = 1 to n step b
    fill C[i:i+b-1,j:j+b-1] with zeros
    for k = 1 to n step b
      naive-matrix-multiply(b,C[i:i+b-1,j:j+b-1],
                              A[i:i+b-1,k:k+b-1],
                              B[k:k+b-1,j:j+b-1])
    end for
  end for
end for
```

- each iteration of the inner loop accesses only $3b^2 = M$ data: each data is read/written only once
- bound on the number of transfers: $(n/b)^3 \times 2M = (n/\sqrt{M/3})^3 \times 2M = O(n^3/\sqrt{M}))$

Outline

Matrix-product algorithm: can we do even better?

- Consider a “normal” matrix-product algorithm (not Strassen)
- Decompose a schedule into *phases* that transfer exactly M data
- $c_{i,j}$ is *alive* in phase p is it computes $a_{i,k}b_{k,j}$ for some k
- alive $c_{i,j}$ either in memory or written: at most $2M$ alive $c_{i,j}$ in a phase
- at most $2M$ elements of A (B) in memory during phase p: A_p (B_p)
- S_p^1: set of rows of A with $\sqrt{M}$ or more elements in A_p ($|S_p^1| \leq 2\sqrt{M}$)
  - each row used in at most $|B_p| \leq 2M$ products
  - at most $4M^{3/2}$ multiplications with elements from S_p^1
- S_p^2: set of rows of A with fewer elements in A_p
  - each row used for a different *alive* $c_{i,j}$
  - at most $\sqrt{M} \times 2M$ multiplications with elements from S_p^2
- total: at most $6M^{3/2}$ per phase
- number of full phases $= \lfloor n^3/6M^{3/2} \rfloor \geq n^3/6M^{3/2} - 1$
- number of transfers $\geq \frac{n^3}{6\sqrt{M}} - M$

Matrix-product algorithm: can we do even better?

- Consider a "normal" matrix-product algorithm (not Strassen)
- Decompose a schedule into *phases* that transfer exactly M data
- $c_{i,j}$ is *alive* in phase p is it computes $a_{i,k}b_{k,j}$ for some k
- alive $c_{i,j}$ either in memory or written: at most $2M$ alive $c_{i,j}$ in a phase
- at most $2M$ elements of A (B) in memory during phase p: A_p (B_p)
- S_p^1: set of rows of A with $\sqrt{M}$ or more elements in A_p ($|S_p^1| \leq 2\sqrt{M}$)
  - each row used in at most $|B_p| \leq 2M$ products
  - at most $4M^{3/2}$ multiplications with elements from S_p^1
- S_p^2: set of rows of A with fewer elements in A_p
  - each row used for a different *alive* $c_{i,j}$
  - at most $\sqrt{M} \times 2M$ multiplications with elements from S_p^2
- total: at most $6M^{3/2}$ per phase
- number of full phases $= \lfloor n^3/6M^{3/2} \rfloor \geq n^3/6M^{3/2} - 1$
- number of transfers $\geq \frac{n^3}{6\sqrt{M}} - M$

Matrix-product algorithm: can we do even better?

- Consider a “normal” matrix-product algorithm (not Strassen)
- Decompose a schedule into *phases* that transfer exactly M data
- $c_{i,j}$ is *alive* in phase p is it computes $a_{i,k}b_{k,j}$ for some k
- alive $c_{i,j}$ either in memory or written: at most $2M$ alive $c_{i,j}$ in a phase
- at most $2M$ elements of A (B) in memory during phase p: A_p (B_p)
- S_p^1: set of rows of A with $\sqrt{M}$ or more elements in A_p ($|S_p^1| \leq 2\sqrt{M}$)
  - each row used in at most $|B_p| \leq 2M$ products
  - at most $4M^{3/2}$ multiplications with elements from S_p^1
- S_p^2: set of rows of A with fewer elements in A_p
  - each row used for a different *alive* $c_{i,j}$
  - at most $\sqrt{M} \times 2M$ multiplications with elements from S_p^2
- total: at most $6M^{3/2}$ per phase
- number of full phases $= \lfloor n^3/6M^{3/2} \rfloor \geq n^3/6M^{3/2} - 1$
- number of transfers $\geq \frac{n^3}{6\sqrt{M}} - M$

Matrix-product algorithm: can we do even better?

- Consider a “normal” matrix-product algorithm (not Strassen)
- Decompose a schedule into *phases* that transfer exactly M data
- $c_{i,j}$ is *alive* in phase p is it computes $a_{i,k}b_{k,j}$ for some k
- alive $c_{i,j}$ either in memory or written: at most $2M$ alive $c_{i,j}$ in a phase
- at most $2M$ elements of A (B) in memory during phase p: A_p (B_p)
- S_p^1: set of rows of A with $\sqrt{M}$ or more elements in A_p ($|S_p^1| \leq 2\sqrt{M}$)
  - each row used in at most $|B_p| \leq 2M$ products
  - at most $4M^{3/2}$ multiplications with elements from S_p^1
- S_p^2: set of rows of A with fewer elements in A_p
  - each row used for a different *alive* $c_{i,j}$
  - at most $\sqrt{M} \times 2M$ multiplications with elements from S_p^2
- total: at most $6M^{3/2}$ per phase
- number of full phases $= \lfloor n^3/6M^{3/2} \rfloor \geq n^3/6M^{3/2} - 1$
- number of transfers $\geq \frac{n^3}{6\sqrt{M}} - M$

Matrix-product algorithm: can we do even better?

- Consider a "normal" matrix-product algorithm (not Strassen)
- Decompose a schedule into *phases* that transfer exactly M data
- $c_{i,j}$ is *alive* in phase p is it computes $a_{i,k}b_{k,j}$ for some k
- alive $c_{i,j}$ either in memory or written: at most $2M$ alive $c_{i,j}$ in a phase
- at most $2M$ elements of A (B) in memory during phase p: A_p (B_p)
- S_p^1: set of rows of A with $\sqrt{M}$ or more elements in A_p ($|S_p^1| \leq 2\sqrt{M}$)
  - each row used in at most $|B_p| \leq 2M$ products
  - at most $4M^{3/2}$ multiplications with elements from S_p^1
- S_p^2: set of rows of A with fewer elements in A_p
  - each row used for a different *alive* $c_{i,j}$
  - at most $\sqrt{M} \times 2M$ multiplications with elements from S_p^2
- total: at most $6M^{3/2}$ per phase
- number of full phases $= \lfloor n^3/6M^{3/2} \rfloor \geq n^3/6M^{3/2} - 1$
- number of transfers $\geq \frac{n^3}{6\sqrt{M}} - M$

Matrix-product algorithm: can we do even better?

- Consider a "normal" matrix-product algorithm (not Strassen)
- Decompose a schedule into *phases* that transfer exactly M data
- $c_{i,j}$ is *alive* in phase p is it computes $a_{i,k}b_{k,j}$ for some k
- alive $c_{i,j}$ either in memory or written: at most $2M$ alive $c_{i,j}$ in a phase
- at most $2M$ elements of A (B) in memory during phase p: A_p (B_p)
- S_p^1: set of rows of A with $\sqrt{M}$ or more elements in A_p ($|S_p^1| \leq 2\sqrt{M}$)
  - each row used in at most $|B_p| \leq 2M$ products
  - at most $4M^{3/2}$ multiplications with elements from S_p^1
- S_p^2: set of rows of A with fewer elements in A_p
  - each row used for a different *alive* $c_{i,j}$
  - at most $\sqrt{M} \times 2M$ multiplications with elements from S_p^2
- total: at most $6M^{3/2}$ per phase
- number of full phases $= \lfloor n^3/6M^{3/2} \rfloor \geq n^3/6M^{3/2} - 1$
- number of transfers $\geq \frac{n^3}{6\sqrt{M}} - M$

Matrix-product algorithm: can we do even better?

- Consider a "normal" matrix-product algorithm (not Strassen)
- Decompose a schedule into *phases* that transfer exactly M data
- $c_{i,j}$ is *alive* in phase p is it computes $a_{i,k}b_{k,j}$ for some k
- alive $c_{i,j}$ either in memory or written: at most $2M$ alive $c_{i,j}$ in a phase
- at most $2M$ elements of A (B) in memory during phase p: A_p (B_p)
- S_p^1: set of rows of A with $\sqrt{M}$ or more elements in A_p ($|S_p^1| \leq 2\sqrt{M}$)
  - each row used in at most $|B_p| \leq 2M$ products
  - at most $4M^{3/2}$ multiplications with elements from S_p^1
- S_p^2: set of rows of A with fewer elements in A_p
  - each row used for a different *alive* $c_{i,j}$
  - at most $\sqrt{M} \times 2M$ multiplications with elements from S_p^2
- total: at most $6M^{3/2}$ per phase
- number of full phases $= \lfloor n^3/6M^{3/2} \rfloor \geq n^3/6M^{3/2} - 1$
- number of transfers $\geq \frac{n^3}{6\sqrt{M}} - M$

Matrix-product algorithm: can we do even better?

- Consider a "normal" matrix-product algorithm (not Strassen)
- Decompose a schedule into *phases* that transfer exactly M data
- $c_{i,j}$ is *alive* in phase p is it computes $a_{i,k}b_{k,j}$ for some k
- alive $c_{i,j}$ either in memory or written: at most $2M$ alive $c_{i,j}$ in a phase
- at most $2M$ elements of A (B) in memory during phase p: A_p (B_p)
- S_p^1: set of rows of A with $\sqrt{M}$ or more elements in A_p ($|S_p^1| \leq 2\sqrt{M}$)
  - each row used in at most $|B_p| \leq 2M$ products
  - at most $4M^{3/2}$ multiplications with elements from S_p^1
- S_p^2: set of rows of A with fewer elements in A_p
  - each row used for a different *alive* $c_{i,j}$
  - at most $\sqrt{M} \times 2M$ multiplications with elements from S_p^2
- total: at most $6M^{3/2}$ per phase
- number of full phases $= \lfloor n^3/6M^{3/2} \rfloor \geq n^3/6M^{3/2} - 1$
- number of transfers $\geq \frac{n^3}{6\sqrt{M}} - M$

Matrix-product algorithm: can we do even better?

- Consider a "normal" matrix-product algorithm (not Strassen)
- Decompose a schedule into *phases* that transfer exactly M data
- $c_{i,j}$ is *alive* in phase p is it computes $a_{i,k}b_{k,j}$ for some k
- alive $c_{i,j}$ either in memory or written: at most $2M$ alive $c_{i,j}$ in a phase
- at most $2M$ elements of A (B) in memory during phase p: A_p (B_p)
- S_p^1: set of rows of A with $\sqrt{M}$ or more elements in A_p $(|S_p^1| \leq 2\sqrt{M})$
  - each row used in at most $|B_p| \leq 2M$ products
  - at most $4M^{3/2}$ multiplications with elements from S_p^1
- S_p^2: set of rows of A with fewer elements in A_p
  - each row used for a different *alive* $c_{i,j}$
  - at most $\sqrt{M} \times 2M$ multiplications with elements from S_p^2
- total: at most $6M^{3/2}$ per phase
- number of full phases $= \lfloor n^3/6M^{3/2} \rfloor \geq n^3/6M^{3/2} - 1$
- number of transfers $\geq \frac{n^3}{6\sqrt{M}} - M$

Matrix-product algorithm: can we do even better?

- Consider a "normal" matrix-product algorithm (not Strassen)
- Decompose a schedule into *phases* that transfer exactly M data
- $c_{i,j}$ is *alive* in phase p is it computes $a_{i,k}b_{k,j}$ for some k
- alive $c_{i,j}$ either in memory or written: at most $2M$ alive $c_{i,j}$ in a phase
- at most $2M$ elements of A (B) in memory during phase p: A_p (B_p)
- S_p^1: set of rows of A with $\sqrt{M}$ or more elements in A_p ($|S_p^1| \leq 2\sqrt{M}$)
  - each row used in at most $|B_p| \leq 2M$ products
  - at most $4M^{3/2}$ multiplications with elements from S_p^1
- S_p^2: set of rows of A with fewer elements in A_p
  - each row used for a different *alive* $c_{i,j}$
  - at most $\sqrt{M} \times 2M$ multiplications with elements from S_p^2
- total: at most $6M^{3/2}$ per phase
- number of full phases $= \lfloor n^3/6M^{3/2} \rfloor \geq n^3/6M^{3/2} - 1$
- number of transfers $\geq \frac{n^3}{6\sqrt{M}} - M$

Matrix-product algorithm: can we do even better?

- Consider a "normal" matrix-product algorithm (not Strassen)
- Decompose a schedule into *phases* that transfer exactly M data
- $c_{i,j}$ is *alive* in phase p is it computes $a_{i,k}b_{k,j}$ for some k
- alive $c_{i,j}$ either in memory or written: at most $2M$ alive $c_{i,j}$ in a phase
- at most $2M$ elements of A (B) in memory during phase p: A_p (B_p)
- S_p^1: set of rows of A with $\sqrt{M}$ or more elements in A_p ($|S_p^1| \leq 2\sqrt{M}$)
  - each row used in at most $|B_p| \leq 2M$ products
  - at most $4M^{3/2}$ multiplications with elements from S_p^1
- S_p^2: set of rows of A with fewer elements in A_p
  - each row used for a different *alive* $c_{i,j}$
  - at most $\sqrt{M} \times 2M$ multiplications with elements from S_p^2
- total: at most $6M^{3/2}$ per phase
- number of full phases $= \lfloor n^3/6M^{3/2} \rfloor \geq n^3/6M^{3/2} - 1$
- number of transfers $\geq \frac{n^3}{6\sqrt{M}} - M$

Matrix-product algorithm: can we do even better?

- Consider a "normal" matrix-product algorithm (not Strassen)
- Decompose a schedule into *phases* that transfer exactly M data
- $c_{i,j}$ is *alive* in phase p is it computes $a_{i,k}b_{k,j}$ for some k
- alive $c_{i,j}$ either in memory or written: at most $2M$ alive $c_{i,j}$ in a phase
- at most $2M$ elements of A (B) in memory during phase p: A_p (B_p)
- S_p^1: set of rows of A with $\sqrt{M}$ or more elements in A_p ($|S_p^1| \leq 2\sqrt{M}$)
  - each row used in at most $|B_p| \leq 2M$ products
  - at most $4M^{3/2}$ multiplications with elements from S_p^1
- S_p^2: set of rows of A with fewer elements in A_p
  - each row used for a different *alive* $c_{i,j}$
  - at most $\sqrt{M} \times 2M$ multiplications with elements from S_p^2
- total: at most $6M^{3/2}$ per phase
- number of full phases $= \lfloor n^3/6M^{3/2} \rfloor \geq n^3/6M^{3/2} - 1$
- number of transfers $\geq \frac{n^3}{6\sqrt{M}} - M$

Matrix-product algorithm: can we do even better?

- Consider a "normal" matrix-product algorithm (not Strassen)
- Decompose a schedule into *phases* that transfer exactly M data
- $c_{i,j}$ is *alive* in phase p is it computes $a_{i,k}b_{k,j}$ for some k
- alive $c_{i,j}$ either in memory or written: at most $2M$ alive $c_{i,j}$ in a phase
- at most $2M$ elements of A (B) in memory during phase p: A_p (B_p)
- S_p^1: set of rows of A with $\sqrt{M}$ or more elements in A_p ($|S_p^1| \leq 2\sqrt{M}$)
  - each row used in at most $|B_p| \leq 2M$ products
  - at most $4M^{3/2}$ multiplications with elements from S_p^1
- S_p^2: set of rows of A with fewer elements in A_p
  - each row used for a different *alive* $c_{i,j}$
  - at most $\sqrt{M} \times 2M$ multiplications with elements from S_p^2
- total: at most $6M^{3/2}$ per phase
- number of full phases $= \lfloor n^3/6M^{3/2} \rfloor \geq n^3/6M^{3/2} - 1$
- number of transfers $\geq \frac{n^3}{6\sqrt{M}} - M$

Matrix-product algorithm: can we do even better?

- Consider a "normal" matrix-product algorithm (not Strassen)
- Decompose a schedule into *phases* that transfer exactly M data
- $c_{i,j}$ is *alive* in phase p is it computes $a_{i,k}b_{k,j}$ for some k
- alive $c_{i,j}$ either in memory or written: at most $2M$ alive $c_{i,j}$ in a phase
- at most $2M$ elements of A (B) in memory during phase p: A_p (B_p)
- S_p^1: set of rows of A with $\sqrt{M}$ or more elements in A_p ($|S_p^1| \leq 2\sqrt{M}$)
  - each row used in at most $|B_p| \leq 2M$ products
  - at most $4M^{3/2}$ multiplications with elements from S_p^1
- S_p^2: set of rows of A with fewer elements in A_p
  - each row used for a different *alive* $c_{i,j}$
  - at most $\sqrt{M} \times 2M$ multiplications with elements from S_p^2
- total: at most $6M^{3/2}$ per phase
- number of full phases $= \lfloor n^3/6M^{3/2} \rfloor \geq n^3/6M^{3/2} - 1$
- number of transfers $\geq \frac{n^3}{6\sqrt{M}} - M$

Matrix-product algorithm: better bound

Lemma (Loomis-Whitney inequality).

With N_A, N_B, N_C elements of A, B, C, we can perform at most $\sqrt{N_A N_B N_C}$ elementary multiplications.

- in each phase of the previous proof: $N_A, N_B, N_C \leq 2M$
- at most $2\sqrt{2}M^{3/2}$ products
- number of transfers: $\geq \frac{n^3}{2\sqrt{2M}} - M$

Further improvement:

- $N_A = N_A^{\text{received}} + N_A^{\text{cached}}$
- $N_A^{\text{received}} + N_B^{\text{received}} + N_C^{\text{received}} \leq M$
- $N_A^{\text{cached}} + N_B^{\text{cached}} + N_C^{\text{cached}} \leq M$
- $N_A + N_B + N_C \leq 2M$
- $\sqrt{N_A N_B N_C} \leq (2M/3)^{3/2}$
- number of transfers: $\geq \frac{27}{8}\frac{n^3}{\sqrt{M}} - M$

Matrix-product algorithm: better bound

Lemma (Loomis-Whitney inequality).

With N_A, N_B, N_C elements of A, B, C, we can perform at most $\sqrt{N_A N_B N_C}$ elementary multiplications.

- in each phase of the previous proof: $N_A, N_B, N_C \leq 2M$
- at most $2\sqrt{2}M^{3/2}$ products
- number of transfers: $\geq \frac{n^3}{2\sqrt{2M}} - M$

Further improvement:

- $N_A = N_A^{\text{received}} + N_A^{\text{cached}}$
- $N_A^{\text{received}} + N_B^{\text{received}} + N_C^{\text{received}} \leq M$
- $N_A^{\text{cached}} + N_B^{\text{cached}} + N_C^{\text{cached}} \leq M$
- $N_A + N_B + N_C \leq 2M$
- $\sqrt{N_A N_B N_C} \leq (2M/3)^{3/2}$
- number of transfers: $\geq \frac{27}{8}\frac{n^3}{\sqrt{M}} - M$

Matrix-product algorithm: better bound

Lemma (Loomis-Whitney inequality).

With N_A, N_B, N_C elements of A, B, C, we can perform at most $\sqrt{N_A N_B N_C}$ elementary multiplications.

- in each phase of the previous proof: $N_A, N_B, N_C \leq 2M$
- at most $2\sqrt{2}M^{3/2}$ products
- number of transfers: $\geq \frac{n^3}{2\sqrt{2M}} - M$

Further improvement:

- $N_A = N_A^{\text{received}} + N_A^{\text{cached}}$
- $N_A^{\text{received}} + N_B^{\text{received}} + N_C^{\text{received}} \leq M$
- $N_A^{\text{cached}} + N_B^{\text{cached}} + N_C^{\text{cached}} \leq M$
- $N_A + N_B + N_C \leq 2M$
- $\sqrt{N_A N_B N_C} \leq (2M/3)^{3/2}$
- number of transfers: $\geq \frac{27}{8} \frac{n^3}{\sqrt{M}} - M$

Matrix-product algorithm: parallel processing

Bounds on the number of transfers:

- For a processor computing W products:

$$I/O_W \geq \frac{W}{2\sqrt{2M}} - M$$

- If we use P processors, one of them computes at least n^3/P products

$$I/O \geq \frac{n^3}{2\sqrt{2M}P} - M$$

Example: 2D algorithms (Cannon, SUMMA, ...):

- 2D block distributions on a grid $\sqrt{P} \times \sqrt{P}$
- store A, B and C: $3n^2/P$ elements on each processor
- at each step, each processors receives a block of A and B
- storage per processor: $O(n^2/P)$
- communication volume per processor: $(n/\sqrt{P})^2 \times \sqrt{P} = n^2/\sqrt{P}$

Outline

Generalized expression and model

Generalized matrix computation:

$$C(i,j) = f_{i,j}(g_{i,j,k}(A(i,k), B(k,j)) \text{ for } k \in S_{i,j}, K)$$

where

- $A(i,j)$, $B(i,j)$, $C(i,j)$ are any reordering of A,B,C
- K represents any other arguments
- $f_{i,j}$, $g_{i,j,k}$ depends non-trivially on their arguments
- A, B and C may overlap

Trivial application to matrix product:

- $g_{i,j,k}$: product
- $S_{i,j} = \{(i,j,k) \text{ for } k = 1 \ldots n\}$
- $f_{i,j}$: sum

I/O analysis for extended model

- As previously, decompose into phases of M transfers
- consider operands (of A, B or C) in memory during a phase
- Root: how it came to be in memory?
  - R1: already in memory at the beginning of the phase, or read during the phase (at most $2M$)
  - R2: created during the phase (not bounded)
- Destination: what happens when it disappears?
  - D1: still in memory at the end of the phase, or written during the phase (at most $2M$)
  - D2: discarded (not bounded)
- Discard R2/D2 for now
- *Alive* values of A in a phase $\leq 4M$ (= R1/* + */D1)
- Using Loomis-Whitney inequality: at most $\sqrt{(4M)^3}$ computations in a phase
- For a computation of size G: at least $G/(8\sqrt{M}) - M$ transfers

I/O analysis for extended model

- As previously, decompose into phases of M transfers
- consider operands (of A, B or C) in memory during a phase
- Root: how it came to be in memory?
  - R1: already in memory at the beginning of the phase, or read during the phase (at most $2M$)
  - R2: created during the phase (not bounded)
- Destination: what happens when it disappears?
  - D1: still in memory at the end of the phase, or written during the phase (at most $2M$)
  - D2: discarded (not bounded)
- Discard R2/D2 for now
- *Alive* values of A in a phase $\leq 4M$ (= R1/* + */D1)
- Using Loomis-Whitney inequality: at most $\sqrt{(4M)^3}$ computations in a phase
- For a computation of size G: at least $G/(8\sqrt{M}) - M$ transfers

Extending to solving linear equations

- TRSM kernel ($C = A^1 B$) for A upper triangular (solve linear equations)

$$C_{i,j} = (B_{i,j} - \sum_{k=i+1}^{n} A_{i,k} \cdot C_{k,j})/A_{i,i}$$

  (any order of j, decreasing i)

- May be transformed to

$$C(i,j) = f_{i,j}(g_{i,j,k}(A(i,k), B(k,j)) \text{ for } k \in S_{i,j}, K)$$

  with:
  - $C = B$
  - $g_{i,j,k}$ multiplies $A_{i,k} \cdot C_{k,j}$
  - $f_{i,j}$ performs the sum, subtracts from B_i, j divides by $A_{i,i}$
- Same bound as for matrix multiplication!
- Achieved by some algorithms

Extending to LU factorization

- Gaussian elimination: $A = L \cdot U$ where L is lower triangular, U is upper triangular

$$
\begin{aligned}
L_{i,j} &= (A_{i,j} - \sum_{k<j} L_{i,k} \cdot U_{k,j})/U_{j,j} \text{ for } i > j \\
U_{i,j} &= A_{i,j} - \sum_{k<i} L_{i,k} \cdot U_{k,j} \text{ for } i \leq j
\end{aligned}
$$

- May be transformed to

$$C(i,j) = f_{i,j}(g_{i,j,k}(A(i,k), B(k,j)) \text{ for } k \in S_{i,j}, K)$$

with:

  - $A = B = C$
  - $g_{i,j,k}$ multiplies $L_{i,k} \cdot U_{k,j}$
  - $f_{i,j}$ performs the sum, subtracts from A_i, j (divides by $U_{j,j}$)
- Same bound
- Achieved by some algorithms

Outline

What if we don't know the memory size M ?

- Back to the matrix product (square matrix of size $n \times n$)

$$C = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix} = A \cdot B = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$

- Recursive matrix multiplication algorithm:

```
RMM(n,A,B)
if n == 1 then C=A*B else {
  C_11 = RMM(n/2,A_11,B_11) + RMM(n/2,A_12,B_21)
  C_12 = RMM(n/2,A_11,B_12) + RMM(n/2,A_12,B_22)
  C_21 = RMM(n/2,A_21,B_11) + RMM(n/2,A_22,B_21)
  C_22 = RMM(n/2,A_21,B_12) + RMM(n/2,A_22,B_22)
return C
```

Analysis of the recursive algorithm

```
RMM(n,A,B)
if n == 1 then C=A*B else {
  C_11 = RMM(n/2,A_11,B_11) + RMM(n/2,A_12,B_21)
  C_12 = RMM(n/2,A_11,B_12) + RMM(n/2,A_12,B_22)
  C_21 = RMM(n/2,A_21,B_11) + RMM(n/2,A_22,B_21)
  C_22 = RMM(n/2,A_21,B_12) + RMM(n/2,A_22,B_22)
return C
```

- $C(n)$: Number of arithmetic operations in RMM(n,A,B)

$$C(n) = 8\,C(n/2) + 4\,(n/2)^2 \text{ if } n > 1 \text{ otherwise } 1$$
$$C(n) = 2n^3 \text{ ... as usual, in different order}$$

- $T(n)$: Number of transfers RMM(n,A,B) with memory M

$$T(n) = 8\,T(n/2) + 12\,(n/2)^2 \text{ if } 3n^2 > M \text{ otherwise } 3n^2$$
$$T(n) = O(n^3/\sqrt{M} + n^2) \text{ ... same as blocked version}$$

Analysis of the recursive algorithm

```
RMM(n,A,B)
if n == 1 then C=A*B else {
  C_11 = RMM(n/2,A_11,B_11) + RMM(n/2,A_12,B_21)
  C_12 = RMM(n/2,A_11,B_12) + RMM(n/2,A_12,B_22)
  C_21 = RMM(n/2,A_21,B_11) + RMM(n/2,A_22,B_21)
  C_22 = RMM(n/2,A_21,B_12) + RMM(n/2,A_22,B_22)
return C
```

- $C(n)$: Number of arithmetic operations in RMM(n,A,B)

$$C(n) = 8\ C(n/2) + 4\ (n/2)^2 \text{ if } n > 1 \text{ otherwise } 1$$
$$C(n) = 2n^3 \ldots \text{as usual, in different order}$$

- $T(n)$: Number of transfers RMM(n,A,B) with memory M

$$T(n) = 8\ T(n/2) + 12\ (n/2)^2 \text{ if } 3n^2 > M \text{ otherwise } 3n^2$$
$$T(n) = O(n^3/\sqrt{M} + n^2) \ldots \text{same as blocked version}$$

Summary on cache-oblivious algorithms

- Designed for unknown cache (or memory) size
- Works well for operations naturally expressed by divide-and-conquer algorithms (matrix multiplication, FFT, sorting, matrix transposition, . . .)
- Asymptotically optimal algorithms
- Well adapted to memory/cache hierarchies:
  L3 (large, slow) $\rightarrow$ L2 (avg. size, avg. speed) $\rightarrow$ L1 (small, fast)
- Extensions exist for parallel machines: Parallel External Memory (PEM)

- In practice for matrix computations, usually outperformed by optimized blocked algorithms

References

- Foundation paper: Hong & Kung: "I/0 Complexity: The Red-Blue Pebble Game" (STOC 1981)
- Communication lower bounds revisited by Irony, Toledo, Tiskin (JPDC 2004)
- Application to numerical linear algebra: Ballard, Demmel, Holtz (SIAM. J. Matrix Anal. & Appl 2011)
  - Development of *communication-avoiding algorithms*
- Cache-oblivious algorithms: Frigo, Leiserson, Prokop, Ramachandran (FOCS 1999), ...

Outline

Introduction

- Directed Acyclic Graphs: express task dependencies
  - nodes: computational tasks
  - edges: dependencies (data = output of a task = input of another task)
- Formalism proposed long ago in scheduling
- Back into fashion thanks to task based runtimes

Here, we focus on task *trees*:

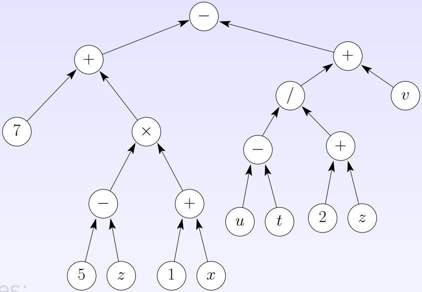
- Arise in multifrontal sparse matrix factorization
- Assembly/Elimination tree: application task graph is a tree
- Large temporary data
- Memory usage becomes a bottleneck

Outline

Related Work: Register Allocation & Pebble Game

How to efficiently compute the following arithmetic expression with the minimum number of registers ?

$$7 + (1 + x)(5 - z) - ((u - t)/(2 + z)) + v$$



Pebble-game rules:

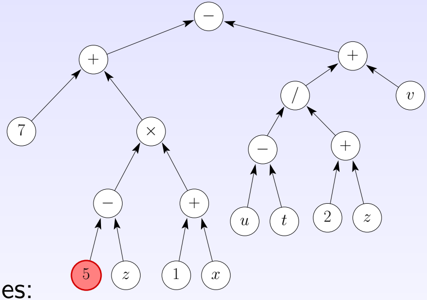
- Inputs can be pebbled anytime
- If all ancestors are pebbled, a node can be pebbled
- A pebble may be removed anytime

Objective: pebble root node using minimum number of pebbles

Related Work: Register Allocation & Pebble Game

How to efficiently compute the following arithmetic expression with the minimum number of registers ?

$$7 + (1 + x)(5 - z) - ((u - t)/(2 + z)) + v$$



Pebble-game rules:

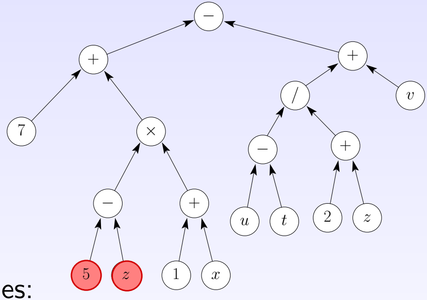
- Inputs can be pebbled anytime
- If all ancestors are pebbled, a node can be pebbled
- A pebble may be removed anytime

Objective: pebble root node using minimum number of pebbles

Related Work: Register Allocation & Pebble Game

How to efficiently compute the following arithmetic expression with the minimum number of registers ?

$$7 + (1 + x)(5 - z) - ((u - t)/(2 + z)) + v$$



Pebble-game rules:

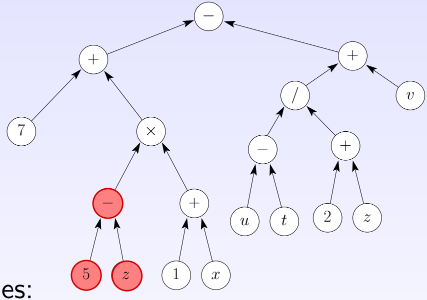
- Inputs can be pebbled anytime
- If all ancestors are pebbled, a node can be pebbled
- A pebble may be removed anytime

Objective: pebble root node using minimum number of pebbles

Related Work: Register Allocation & Pebble Game

How to efficiently compute the following arithmetic expression with the minimum number of registers ?

$$7 + (1 + x)(5 - z) - ((u - t)/(2 + z)) + v$$



Pebble-game rules:

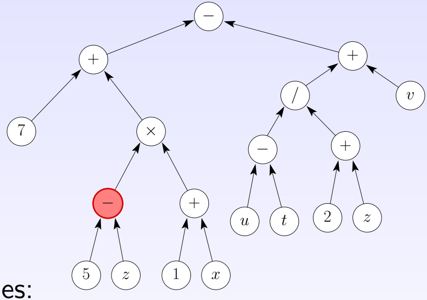
- Inputs can be pebbled anytime
- If all ancestors are pebbled, a node can be pebbled
- A pebble may be removed anytime

Objective: pebble root node using minimum number of pebbles

Related Work: Register Allocation & Pebble Game

How to efficiently compute the following arithmetic expression with the minimum number of registers ?

$$7 + (1 + x)(5 - z) - ((u - t)/(2 + z)) + v$$



Pebble-game rules:

- Inputs can be pebbled anytime
- If all ancestors are pebbled, a node can be pebbled
- A pebble may be removed anytime

Objective: pebble root node using minimum number of pebbles

Related Work: Register Allocation & Pebble Game

How to efficiently compute the following arithmetic expression with the minimum number of registers ?

$$7 + (1 + x)(5 - z) - ((u - t)/(2 + z)) + v$$

Complexity results

Problem on trees:

- Polynomial algorithm [Sethi & Ullman, 1970]

General problem on DAGs (common subexpressions):

- P-Space complete [Gilbert, Lengauer & Tarjan, 1980]
- Without re-computation: NP-complete [Sethi, 1973]

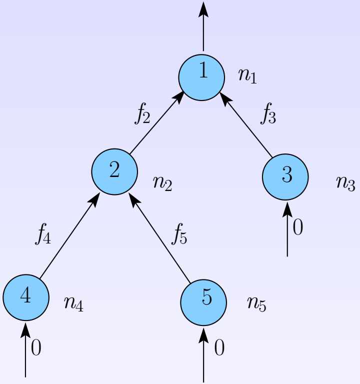
Pebble-game rules:

- Inputs can be pebbled anytime
- If all ancestors are pebbled, a node can be pebbled
- A pebble may be removed anytime

Objective: pebble root node using minimum number of pebbles

Outline

Notations: Tree-Shaped Task Graphs



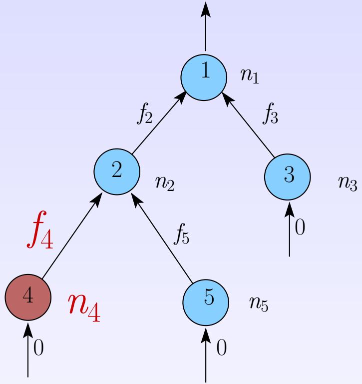
- In-tree of n nodes
- Output data of size f_i
- Execution data of size n_i
- Input data of leaf nodes have null size

- Memory for node i: $MemReq(i) = \left(\sum_{j \in Children(i)} f_j \right) + n_i + f_i$

Two existing sequential algorithms:

- Best traversal [J. Liu, 1987]
- Best post-order traversal [J. Liu, 1986]

Notations: Tree-Shaped Task Graphs



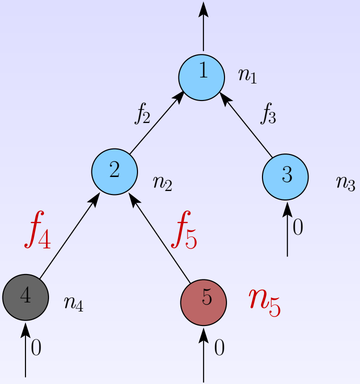
- In-tree of n nodes
- Output data of size f_i
- Execution data of size n_i
- Input data of leaf nodes have null size

- Memory for node i: $MemReq(i) = \left(\sum_{j \in Children(i)} f_j \right) + n_i + f_i$

Two existing sequential algorithms:

- Best traversal [J. Liu, 1987]
- Best post-order traversal [J. Liu, 1986]

Notations: Tree-Shaped Task Graphs



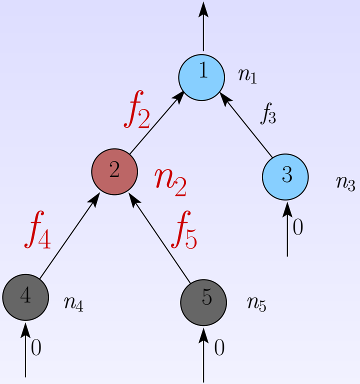
- In-tree of n nodes
- Output data of size f_i
- Execution data of size n_i
- Input data of leaf nodes have null size

- Memory for node i: $MemReq(i) = \left(\sum_{j \in Children(i)} f_j \right) + n_i + f_i$

Two existing sequential algorithms:

- Best traversal [J. Liu, 1987]
- Best post-order traversal [J. Liu, 1986]

Notations: Tree-Shaped Task Graphs



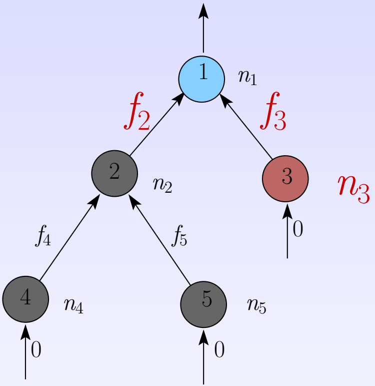
- In-tree of n nodes
- Output data of size f_i
- Execution data of size n_i
- Input data of leaf nodes have null size

- Memory for node i: $MemReq(i) = \left(\sum_{j \in Children(i)} f_j \right) + n_i + f_i$

Two existing sequential algorithms:

- Best traversal [J. Liu, 1987]
- Best post-order traversal [J. Liu, 1986]

Notations: Tree-Shaped Task Graphs



- In-tree of n nodes
- Output data of size f_i
- Execution data of size n_i
- Input data of leaf nodes have null size

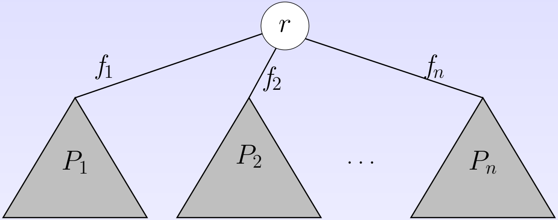
- Memory for node i: $MemReq(i) = \left(\sum_{j \in Children(i)} f_j \right) + n_i + f_i$

Two existing sequential algorithms:

- Best traversal [J. Liu, 1987]
- Best post-order traversal [J. Liu, 1986]

Liu's Best Post-Order Traversal for Trees

Post-Order: entirely process one subtree after the other (DFS)



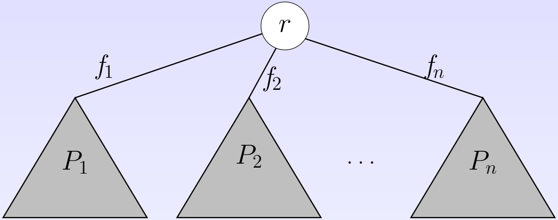
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \dots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3,\ \dots, \sum_{i<n} f_i + P_n,\ \sum f_i + n_r + f_r\}$$

- Optimal order:
- Post-Order traversals are dominant for unit-weight trees

Liu's Best Post-Order Traversal for Trees

Post-Order: entirely process one subtree after the other (DFS)



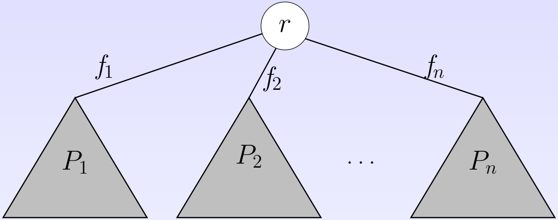
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \dots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3,\ \dots, \sum_{i<n} f_i + P_n,\ \sum f_i + n_r + f_r\}$$

- Optimal order:
- Post-Order traversals are dominant for unit-weight trees

Liu's Best Post-Order Traversal for Trees

Post-Order: entirely process one subtree after the other (DFS)



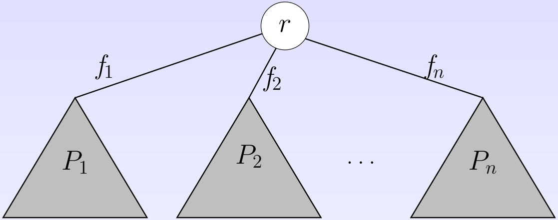
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \dots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3,\ \dots, \sum_{i<n} f_i + P_n,\ \sum f_i + n_r + f_r\}$$

- Optimal order:
- Post-Order traversals are dominant for unit-weight trees

Liu's Best Post-Order Traversal for Trees

Post-Order: entirely process one subtree after the other (DFS)



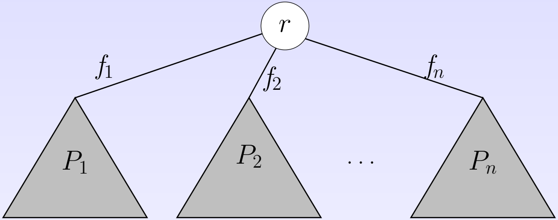
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \ldots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3,\ \ldots, \sum_{i<n} f_i + P_n,\ \sum f_i + n_r + f_r\}$$

- Optimal order:
- Post-Order traversals are dominant for unit-weight trees

Liu's Best Post-Order Traversal for Trees

Post-Order: entirely process one subtree after the other (DFS)



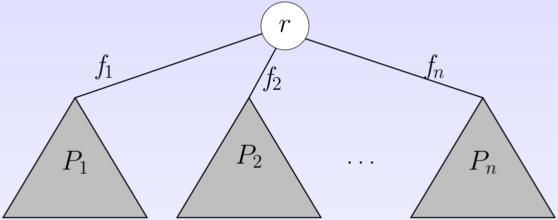
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \dots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3,\ \dots, \sum_{i<n} f_i + P_n,\ \sum f_i + n_r + f_r\}$$

- Optimal order:
- Post-Order traversals are dominant for unit-weight trees

Liu's Best Post-Order Traversal for Trees

Post-Order: entirely process one subtree after the other (DFS)



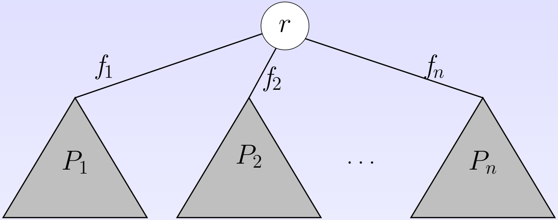
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \ldots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3,\ \ldots, \sum_{i<n} f_i + P_n,\ \sum f_i + n_r + f_r\}$$

- Optimal order: non-increasing $P_i - f_i$
- Post-Order traversals are dominant for unit-weight trees

Liu's Best Post-Order Traversal for Trees

Post-Order: entirely process one subtree after the other (DFS)



- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \ldots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3,\ \ldots, \sum_{i<n} f_i + P_n,\ \sum f_i + n_r + f_r\}$$

- Optimal order: non-increasing $P_i - f_i$
- Post-Order traversals are dominant for unit-weight trees

Proof for best post-order

Theorem (Best Post-Order).

The best post-order traversal is obtain by processing subtrees in non-increasing order $P_i - f_i$.

Proof:

- Consider an optimal traversal which does not respect the order:
  - subtree j is processed right before subtree k
  - $P_k - f_k \geq P_j - f_j$

	peak when j, then k	peak when k, then j
during first subtree	$mem\_before + P_j$	$mem\_before + P_k$
during second subtree	$mem\_before + f_j + P_k$	$mem\_before + f_k + P_j$

- $f_k + P_j \leq f_j + P_k$
- Transform the schedule step by step without increasing the memory.

Proof for best post-order

Theorem (Best Post-Order).

The best post-order traversal is obtain by processing subtrees in non-increasing order $P_i - f_i$.

Proof:

- Consider an optimal traversal which does not respect the order:
  - subtree j is processed right before subtree k
  - $P_k - f_k \geq P_j - f_j$

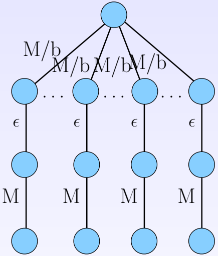
	peak when j, then k	peak when k, then j
during first subtree	$mem\_before + P_j$	$mem\_before + P_k$
during second subtree	$mem\_before + f_j + P_k$	$mem\_before + f_k + P_j$

- $f_k + P_j \leq f_j + P_k$
- Transform the schedule step by step without increasing the memory.

Post-Order is not optimal...

Post-Order traversals are arbitrarily bad in the general case

There is no constant k such that the best post-order traversal is a k-approximation.



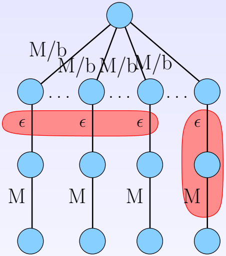
- Minimum peak memory: $M_{\min} = M + \epsilon + (b-1)\epsilon$
- Minimum post-order peak memory: $M_{\min} = M + \epsilon + (b-1)M/b$

	actual assembly trees	random trees
Non optimal traversals	4.2%	61%
Maximum increase compared to optimal	18%	22%
Average increased compared to optimal	**1%**	12%

Post-Order is not optimal...

Post-Order traversals are arbitrarily bad in the general case

There is no constant k such that the best post-order traversal is a k-approximation.



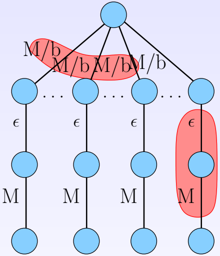
- Minimum peak memory: $M_{\min} = M + \epsilon + (b-1)\epsilon$
- Minimum post-order peak memory: $M_{\min} = M + \epsilon + (b-1)M/b$

	actual assembly trees	random trees
Non optimal traversals	4.2%	61%
Maximum increase compared to optimal	18%	22%
Average increased compared to optimal	**1%**	12%

Post-Order is not optimal...

Post-Order traversals are arbitrarily bad in the general case

There is no constant k such that the best post-order traversal is a k-approximation.



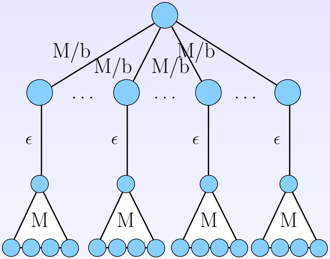
- Minimum peak memory: $M_{\min} = M + \epsilon + (b-1)\epsilon$
- Minimum post-order peak memory: $M_{\min} = M + \epsilon + (b-1)M/b$

	actual assembly trees	random trees
Non optimal traversals	4.2%	61%
Maximum increase compared to optimal	18%	22%
Average increased compared to optimal	**1%**	12%

Post-Order is not optimal...

Post-Order traversals are arbitrarily bad in the general case

There is no constant k such that the best post-order traversal is a k-approximation.



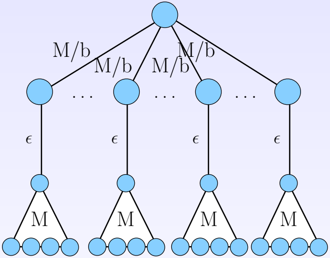
- Minimum peak memory: $M_{\min} = M + \epsilon + 2(b-1)\epsilon$
- Minimum post-order peak memory: $M_{\min} = M + \epsilon + 2(b-1)M/b$

	actual assembly trees	random trees
Non optimal traversals	4.2%	61%
Maximum increase compared to optimal	18%	22%
Average increased compared to optimal	**1%**	12%

Post-Order is not optimal...but almost!

Post-Order traversals are arbitrarily bad in the general case

There is no constant k such that the best post-order traversal is a k-approximation.



- Minimum peak memory: $M_{\min} = M + \epsilon + (b-1)\epsilon$
- Minimum post-order peak memory: $M_{\min} = M + \epsilon + (b-1)M/b$

	actual assembly trees	random trees
Non optimal traversals	4.2%	61%
Maximum increase compared to optimal	18%	22%
Average increased compared to optimal	**1%**	12%

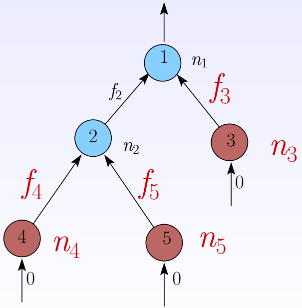
Liu's optimal traversal – sketch

- Recursive algorithm: at each step, merge the optimal ordering of each subtree (sequence)
- Sequence: divided into segments:
  - H_1: maximum over the whole sequence (hill)
  - V_1: minimum after H_1 (valley)
  - H_2: maximum after H_1
  - V_2: minimum after H_2
  - ...
  - The valleys V_is are the boundaries of the segments
- Combine the sequences by non-increasing $H - V$
- Complex proof based on a partial order on the cost-sequences: $(H_1, V_1, H_2, V_2, \ldots, H_r, V_r) \prec (H'_1, V'_1, H'_2, V'_2, \ldots, H'_{r'}, V'_{r'})$ if for each $1 \leq i \leq r$, there exists $1 \leq j \leq r'$ with $H_i \leq H'_j$ and $V_i \leq V'_j$.

Outline

Model for Parallel Tree Processing

- p uniform processors
- Shared memory of size M
- Task i has execution times p_i
- Parallel processing of nodes $\Rightarrow$ larger memory
- Trade-off time vs. memory



NP-Completeness in the Pebble Game Model

Background:

- Makespan minimization NP-complete for trees ($P|trees|C_{\max}$)
- Polynomial when unit-weight tasks ($P|p_i = 1, trees|C_{\max}$)
- Pebble game polynomial on trees

Pebble game model:

- Unit execution time: $p_i = 1$
- Unit memory costs: $n_i = 0, f_i = 1$
  (pebble edges, equivalent to pebble game for trees)

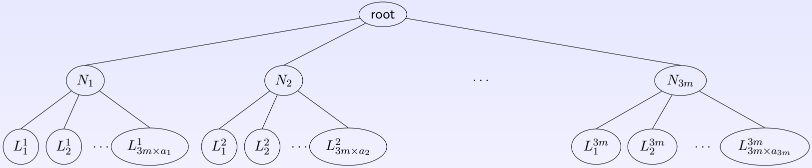
Theorem

Deciding whether a tree can be scheduled using at most B pebbles in at most C steps is NP-complete.

NP-Completeness – Proof

Reduction from 3-Partition:

- $3m$ integers a_i and B with $\sum ai = mB$,
- find m subsets S_k of 3 elements with $\sum_{i \in S_k} a_i = B$



Schedule the tree using:

- $p = 3mB$ processors,
- at most $B = 3m \times B + 3m$ pebbles,
- at most $C = 2m + 1$ steps.

Space-Time Tradeoff

Not possible to get a guarantee on both memory and time simultaneously:

Theorem 1

There is no algorithm that is both an α-approximation for makespan minimization and a β-approximation for memory peak minimization when scheduling tree-shaped task graphs.

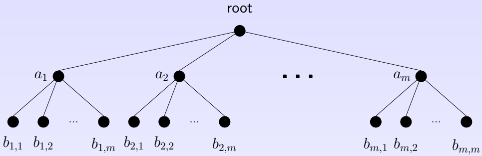
Lemma

For a schedule with peak memory M and makespan $C_{\max}$,

$$M \times C_{\max} \geq 2(n-1)$$

Proof: each edge stays in memory for at least 2 steps.

Space-Time Tradeoff – Proof



- With m^2 processors: $C^*_{\max} = 3$
- With 1 processor, sequentialize the a_i subtrees: $M^* = 2m$
- By contradiction, approximating both objectives: $C_{\max} \leq 3\alpha$ and $M \leq 2m\beta$
- But $M \times C_{\max} \geq 2(n-1) = 2m^2 + 2m$
- $2m^2 + 2m \leq 6m\alpha\beta$
- Contradiction for a sufficiently large value of m

Outline

Practical solutions for limited memory

- In practice: physical bound on the memory
- How to cope with this bound, and guarantee completion?
- Two approaches:
  - Sequential activation order
  - Memory booking

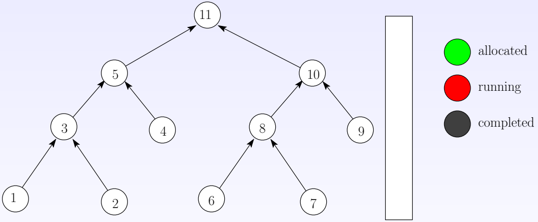
Sequential activation order

Idea (Sequential Task Flow model):

- activate tasks using a prescribed order
  (memory allocation: $f_i + n_i$)
- schedule active (and ready) tasks using another order/priority

When a node completes:

- Allocate as many tasks as possible
- Then, start processing allocated tasks



- ☺ minimum memory requirement: memory peak of the activation traversal
- ☹ no memory reuse

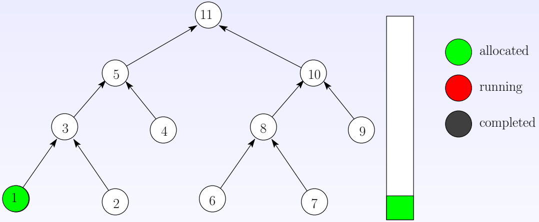
Sequential activation order

Idea (Sequential Task Flow model):

- activate tasks using a prescribed order (memory allocation: $f_i + n_i$)
- schedule active (and ready) tasks using another order/priority

When a node completes:

- Allocate as many tasks as possible
- Then, start processing allocated tasks



- ☺ minimum memory requirement: memory peak of the activation traversal
- ☹ no memory reuse

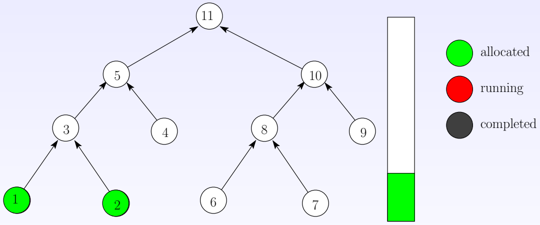
Sequential activation order

Idea (Sequential Task Flow model):

- activate tasks using a prescribed order
  (memory allocation: $f_i + n_i$)
- schedule active (and ready) tasks using another order/priority

When a node completes:

- Allocate as many tasks as possible
- Then, start processing allocated tasks



- ☺ minimum memory requirement: memory peak of the activation traversal
- ☹ no memory reuse

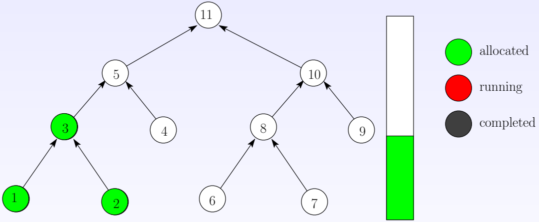
Sequential activation order

Idea (Sequential Task Flow model):

- activate tasks using a prescribed order (memory allocation: $f_i + n_i$)
- schedule active (and ready) tasks using another order/priority

When a node completes:

- Allocate as many tasks as possible
- Then, start processing allocated tasks



- ☺ minimum memory requirement: memory peak of the activation traversal
- ☹ no memory reuse

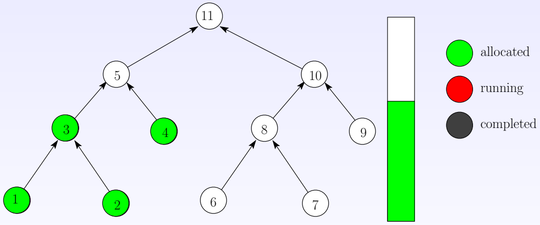
Sequential activation order

Idea (Sequential Task Flow model):

- activate tasks using a prescribed order (memory allocation: $f_i + n_i$)
- schedule active (and ready) tasks using another order/priority

When a node completes:

- Allocate as many tasks as possible
- Then, start processing allocated tasks



- ☺ minimum memory requirement: memory peak of the activation traversal
- ☹ no memory reuse

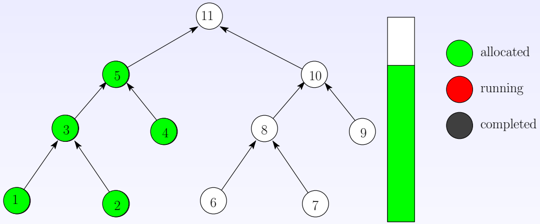
Sequential activation order

Idea (Sequential Task Flow model):

- activate tasks using a prescribed order
  (memory allocation: $f_i + n_i$)
- schedule active (and ready) tasks using another order/priority

When a node completes:

- Allocate as many tasks as possible
- Then, start processing allocated tasks



- ☺ minimum memory requirement: memory peak of the activation traversal
- ☹ no memory reuse

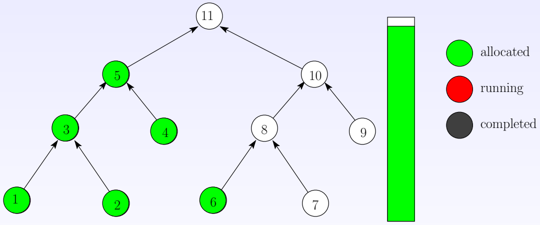
Sequential activation order

Idea (Sequential Task Flow model):

- activate tasks using a prescribed order (memory allocation: $f_i + n_i$)
- schedule active (and ready) tasks using another order/priority

When a node completes:

- Allocate as many tasks as possible
- Then, start processing allocated tasks



- ☺ minimum memory requirement: memory peak of the activation traversal
- ☹ no memory reuse

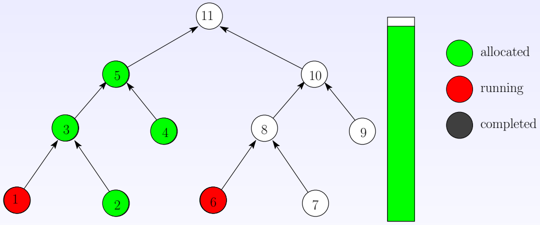
Sequential activation order

Idea (Sequential Task Flow model):

- activate tasks using a prescribed order
  (memory allocation: $f_i + n_i$)
- schedule active (and ready) tasks using another order/priority

When a node completes:

- Allocate as many tasks as possible
- Then, start processing allocated tasks



- ☺ minimum memory requirement: memory peak of the activation traversal
- ☹ no memory reuse

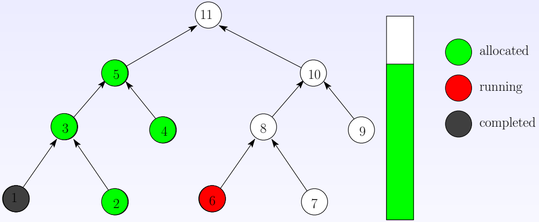
Sequential activation order

Idea (Sequential Task Flow model):

- activate tasks using a prescribed order
  (memory allocation: $f_i + n_i$)
- schedule active (and ready) tasks using another order/priority

When a node completes:

- Allocate as many tasks as possible
- Then, start processing allocated tasks



- ☺ minimum memory requirement: memory peak of the activation traversal
- ☹ no memory reuse

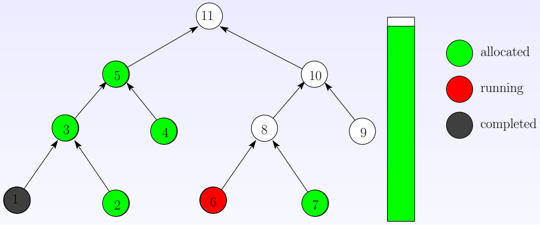
Sequential activation order

Idea (Sequential Task Flow model):

- activate tasks using a prescribed order
  (memory allocation: $f_i + n_i$)
- schedule active (and ready) tasks using another order/priority

When a node completes:

- Allocate as many tasks as possible
- Then, start processing allocated tasks



- ☺ minimum memory requirement: memory peak of the activation traversal
- ☹ no memory reuse

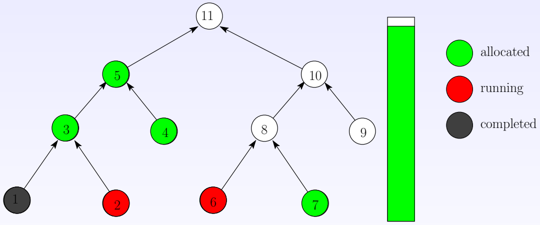
Sequential activation order

Idea (Sequential Task Flow model):

- activate tasks using a prescribed order
  (memory allocation: $f_i + n_i$)
- schedule active (and ready) tasks using another order/priority

When a node completes:

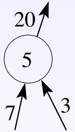
- Allocate as many tasks as possible
- Then, start processing allocated tasks



- ☺ minimum memory requirement: memory peak of the activation traversal
- ☹ no memory reuse

Heuristic design: memory booking

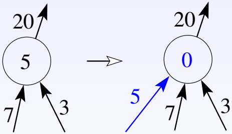
- Design of scheduling heuristics with guaranteed peak memory
- Idea: re-use memory for parents, grand-parents, ...
- Book memory only when starting new leaves
- Stronger assumptions:
  - Reduction tree: $\sum_{j \in Children(i)} f_j \geq f_i$
  - No extra memory cost for task execution
- For trees that do not respect these constraints, add fictitious nodes



- ☺ memory reuse
- ☹ extra memory weights

Heuristic design: memory booking

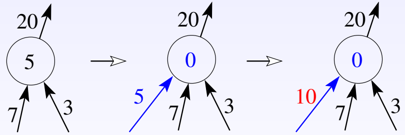
- Design of scheduling heuristics with guaranteed peak memory
- Idea: re-use memory for parents, grand-parents, ...
- Book memory only when starting new leaves
- Stronger assumptions:
  - Reduction tree: $\sum_{j \in Children(i)} f_j \geq f_i$
  - No extra memory cost for task execution
- For trees that do not respect these constraints, add fictitious nodes



- ☺ memory reuse
- ☹ extra memory weights

Heuristic design: memory booking

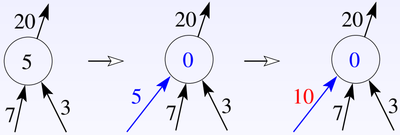
- Design of scheduling heuristics with guaranteed peak memory
- Idea: re-use memory for parents, grand-parents, ...
- Book memory only when starting new leaves
- Stronger assumptions:
  - Reduction tree: $\sum_{j \in Children(i)} f_j \geq f_i$
  - No extra memory cost for task execution
- For trees that do not respect these constraints, add fictitious nodes



- ☺ memory reuse
- ☹ extra memory weights

Heuristic design: memory booking

- Design of scheduling heuristics with guaranteed peak memory
- Idea: re-use memory for parents, grand-parents, ...
- Book memory only when starting new leaves
- Stronger assumptions:
  - Reduction tree: $\sum_{j \in Children(i)} f_j \geq f_i$
  - No extra memory cost for task execution
- For trees that do not respect these constraints, add fictitious nodes



- ☺ memory reuse
- ☹ extra memory weights

Outline

Conclusion

- Memory, I/O and cache impact performance
- Avoid data movement, re-use data as much as possible
- Many different approaches, depending on the target application model:
    - Cache-oblivious algorithms (recursive computations)
    - Communication-avoiding algorithms (numerical algebra)
    - Memory-Aware scheduling (task graphs)