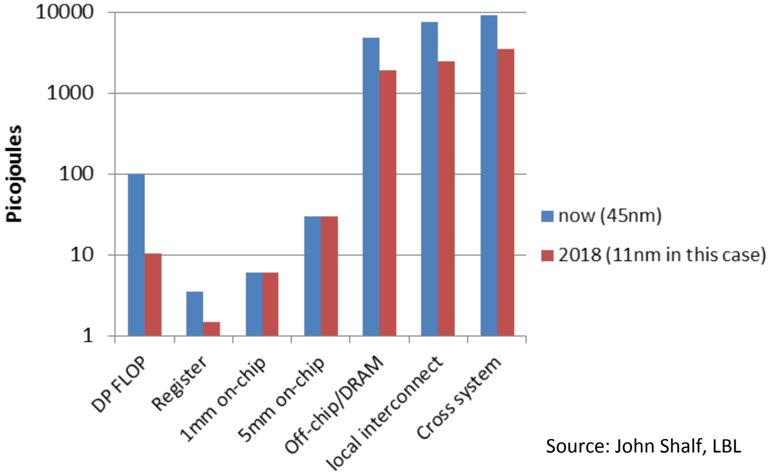
Part 4: Communication Avoiding Algorithms

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

October 21, 2020

Yet Another Motivation. . .

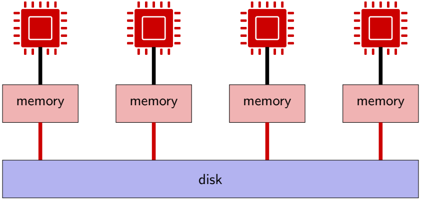
. . . for limiting communications



Source: John Shalf, LBL

Communication Avoiding Algorithms

Context: Distributed Memory



Communications: Data movements between:

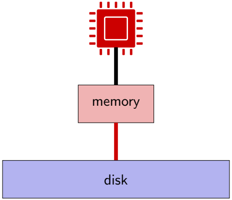
- one processor and its memory
- different processors/memories

Objective:

- Derive communication lower bounds for many linear algebra operations
- Design communication-optimal algorithms

Reminder: Matrix Product Lower Bound

Context: Single processor + Memory (size M)



- Analysis in phases of M I/O operations
- Bound on the number of elementary product in each phase: $F = O(M^{3/2})$
  Geometric argument: Loomis-Whitney inequality
- At least n^3/F phases, of M I/Os, in total: $\Theta(n^3/\sqrt{M})$ I/Os

Part 4: Communication Avoiding Algorithms

Generalization to other Linear Algebra Algorithms
- Generalized Matrix Computations
- I/O Analysis
- Application to LU Factorization

Analysis and Lower Bounds for Parallel Algorithms
- Matrix Multiplication Lower Bound for P processors
- 2D and 3D Algorithms for Matrix Multiplication
- 2.5D Algorithm for Matrix Multiplication

Conclusion

Part 4: Communication Avoiding Algorithms

Generalization to other Linear Algebra Algorithms
Generalized Matrix Computations
I/O Analysis
Application to LU Factorization

Analysis and Lower Bounds for Parallel Algorithms
Matrix Multiplication Lower Bound for P processors
2D and 3D Algorithms for Matrix Multiplication
2.5D Algorithm for Matrix Multiplication

Conclusion

Generalization to other Linear Algebra Algorithms

- Inputs/Ouput: $n \times n$ matrices A,B,C
- Any mapping of the matrices to the memory (possibly overlapping)

Generalization to other Linear Algebra Algorithms

- Inputs/Ouput: $n \times n$ matrices A, B, C
- Any mapping of the matrices to the memory (possibly overlapping)

General computation

For all $(i, j) \in S_c$,

$$C_{i,j} \leftarrow f_{i,j}\Big(g_{i,j,k}(A_{i,k} B_{k,j}) \text{ for } k \in S_{i,j}, \text{ any other arguments}\Big)$$

Generalization to other Linear Algebra Algorithms

- Inputs/Ouput: $n \times n$ matrices A, B, C
- Any mapping of the matrices to the memory (possibly overlapping)

General computation

For all $(i,j) \in S_c$,

$$C_{i,j} \leftarrow f_{i,j}\Big(g_{i,j,k}(A_{i,k} B_{k,j}) \text{ for } k \in S_{i,j}, \text{ any other arguments}\Big)$$

- For matrix multiplication:

Generalization to other Linear Algebra Algorithms

- Inputs/Ouput: $n \times n$ matrices A, B, C
- Any mapping of the matrices to the memory (possibly overlapping)

General computation

For all $(i, j) \in S_c$,

$$C_{i,j} \leftarrow f_{i,j}\Big(g_{i,j,k}(A_{i,k}B_{k,j}) \text{ for } k \in S_{i,j}, \text{ any other arguments}\Big)$$

- For matrix multiplication:
  - $f_{i,j}$: summation, $g_{i,j,k}$: product
  - $S_{i,j} = [1, n]$, $S_C = [1, n] \times [1, n]$

Generalized Matrix Computations

General computation

For all $(i, j) \in S_c$,

$$C_{i,j} \leftarrow f_{i,j}\Big(g_{i,j,k}(A_{i,k}B_{k,j}) \text{ for } k \in S_{i,j}, \text{ any other arguments}\Big)$$

- f_i, j and $g_{i,j,k}$ non-trivial:
  - $g_{i,j,k}$ needs to the value of $A_{i,k}$ and $B_{k,j}$ in memory
  - f_i, j needs at least an "accumulator" while results of $g_{i,j,k}(\dots)$ are loaded/computed in memory one after the other
- S_C, $S_{i,j}$, $f_{i,j}$, $g_{i,j,k}$ possibly determined at runtime
- Correct computations may require special ordering of computations: no such constraint needed for the lower bound:
  - any order for computing the $g_{i,j,k}$
  - any order for computing and storing the $f_{i,j}$

Generalized Matrix Computations

General computation

For all $(i, j) \in S_c$,

$$C_{i,j} \leftarrow f_{i,j}\Big(g_{i,j,k}(A_{i,k}B_{k,j}) \text{ for } k \in S_{i,j}, \text{ any other arguments}\Big)$$

- f_i, j and $g_{i,j,k}$ non-trivial:
  - $g_{i,j,k}$ needs to the value of $A_{i,k}$ and $B_{k,j}$ in memory
  - f_i, j needs at least an "accumulator" while results of $g_{i,j,k}(\dots)$ are loaded/computed in memory one after the other
- S_C, $S_{i,j}$, $f_{i,j}$, $g_{i,j,k}$ possibly determined at runtime
- Correct computations may require special ordering of computations: no such constraint needed for the lower bound:
  - any order for computing the $g_{i,j,k}$
  - any order for computing and storing the $f_{i,j}$

Generalized Matrix Computations

General computation

For all $(i, j) \in S_c$,

$$C_{i,j} \leftarrow f_{i,j}\Big(g_{i,j,k}(A_{i,k}B_{k,j}) \text{ for } k \in S_{i,j},\ \text{any other arguments}\Big)$$

- f_i, j and $g_{i,j,k}$ non-trivial:
  - $g_{i,j,k}$ needs to the value of $A_{i,k}$ and $B_{k,j}$ in memory
  - f_i, j needs at least an "accumulator" while results of $g_{i,j,k}(\dots)$ are loaded/computed in memory one after the other
- S_C, $S_{i,j}$, $f_{i,j}$, $g_{i,j,k}$ possibly determined at runtime
- Correct computations may require special ordering of computations: no such constraint needed for the lower bound:
  - any order for computing the $g_{i,j,k}$
  - any order for computing and storing the $f_{i,j}$

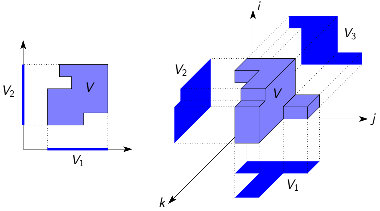
Geometric analysis

Analysis based on Loomis-Whitney inequality:

Theorem (Discrete Loomis-Whitney Inequality).

Let V be a finite subset of $\mathbb{Z}^D$ and V_1, V_2, V_3 denotes the orthogonal projections of V on each coordinate planes, we have:

$$|V|^2 \leq |V_1| \cdot |V_2| \cdot |V_3|,$$



I/O Analysis

One phase: M I/Os operations (loads and stores)

Classify operands based on their root and destination:

- R1: operands present in fast memory at the beginning of the phase or loaded (at most $2M$ such operands)
- R2: operands computed during the phase
- D1: operands left in fast memory at the end of the phase or written (at most $2M$ such operands)
- D2: operands discarded

- Forget about R2/D2 operands
- At most 4M operands available in one phase, for each matrix
- Loomis-Whitney $\Rightarrow$ at most $F = \sqrt{(4M)^3}$ computations of g
- Total number of loads and stores:

$$M \left\lfloor \frac{G}{F} \right\rfloor = M \left\lfloor \frac{G}{\sqrt{(4M)^3}} \right\rfloor \geq \frac{G}{8\sqrt{M}} - M$$

I/O Analysis

One phase: M I/Os operations (loads and stores)

Classify operands based on their root and destination:

- R1: operands present in fast memory at the beginning of the phase or loaded (at most $2M$ such operands)
- R2: operands computed during the phase
- D1: operands left in fast memory at the end of the phase or written (at most $2M$ such operands)
- D2: operands discarded

I/O Analysis

One phase: M I/Os operations (loads and stores)

Classify operands based on their root and destination:

- R1: operands present in fast memory at the beginning of the phase or loaded (at most $2M$ such operands)
- R2: operands computed during the phase
- D1: operands left in fast memory at the end of the phase or written (at most $2M$ such operands)
- D2: operands discarded

- Forget about R2/D2 operands
- At most 4M operands available in one phase, for each matrix
- Loomis-Whitney $\Rightarrow$ at most $F = \sqrt{(4M)^3}$ computations of g
- Total number of loads and stores:

$$M \left\lfloor \frac{G}{F} \right\rfloor = M \left\lfloor \frac{G}{\sqrt{(4M)^3}} \right\rfloor \geq \frac{G}{8\sqrt{M}} - M$$

I/O Analysis

One phase: M I/Os operations (loads and stores)

Classify operands based on their root and destination:

- R1: operands present in fast memory at the beginning of the phase or loaded (at most $2M$ such operands)
- R2: operands computed during the phase
- D1: operands left in fast memory at the end of the phase or written (at most $2M$ such operands)
- D2: operands discarded

- Forget about R2/D2 operands
- At most 4M operands available in one phase, for each matrix
- Loomis-Whitney $\Rightarrow$ at most $F = \sqrt{(4M)^3}$ computations of g
- Total number of loads and stores:

$$M \left\lfloor \frac{G}{F} \right\rfloor = M \left\lfloor \frac{G}{\sqrt{(4M)^3}} \right\rfloor \geq \frac{G}{8\sqrt{M}} - M$$

I/O Analysis

One phase: M I/Os operations (loads and stores)

Classify operands based on their root and destination:

- R1: operands present in fast memory at the beginning of the phase or loaded (at most $2M$ such operands)
- R2: operands computed during the phase
- D1: operands left in fast memory at the end of the phase or written (at most $2M$ such operands)
- D2: operands discarded

- Forget about R2/D2 operands
- At most 4M operands available in one phase, for each matrix
- Loomis-Whitney $\Rightarrow$ at most $F = \sqrt{(4M)^3}$ computations of g
- Total number of loads and stores:

$$M \left\lfloor \frac{G}{F} \right\rfloor = M \left\lfloor \frac{G}{\sqrt{(4M)^3}} \right\rfloor \geq \frac{G}{8\sqrt{M}} - M$$

I/O Analysis

One phase: M I/Os operations (loads and stores)

Classify operands based on their root and destination:

- R1: operands present in fast memory at the beginning of the phase or loaded (at most $2M$ such operands)
- R2: operands computed during the phase
- D1: operands left in fast memory at the end of the phase or written (at most $2M$ such operands)
- D2: operands discarded

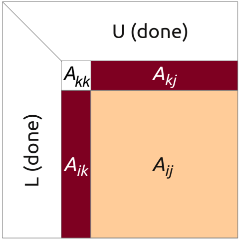
- Forget about R2/D2 operands
- At most 4M operands available in one phase, for each matrix
- Loomis-Whitney $\Rightarrow$ at most $F = \sqrt{(4M)^3}$ computations of g
- Total number of loads and stores:

$$M \left\lfloor \frac{G}{F} \right\rfloor = M \left\lfloor \frac{G}{\sqrt{(4M)^3}} \right\rfloor \geq \frac{G}{8\sqrt{M}} - M$$

Application to LU Factorization (1/2)

LU factorization (Gaussian elimination):

- Convert a matrix A into product $L \times U$
- L is lower triangular with diagonal 1
- U is upper triangular
- $(L - D + U)$ stored in place with A



LU Algorithm

For $k = 1 \dots n - 1$:

- For $i = k + 1 \dots n$,
  $A_{i,k} \leftarrow a_{i,k} / a_{k,k}$ (column/panel preparation)
- For $i = k + 1 \dots n$,
  For $j = k + 1 \dots n$,
  $A_{i,j} \leftarrow A_{i,j} - A_{i,k} A_{k,j}$ (update)

Application to LU Factorization (2/2)

Can be expressed as follows:

$$U_{i,j} = A_{i,j} - \sum_{k<i} L_{i,k} \cdot U_{k,j} \qquad \text{for } i \leq j$$

$$L_{i,j} = (A_{i,j} - \sum_{k<j} L_{i,k} \cdot U_{k,j})/U_{j,j} \qquad \text{for } i > j$$

Fits the generalized matrix computations:

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

with:

Application to LU Factorization (2/2)

Can be expressed as follows:

$$U_{i,j} = A_{i,j} - \sum_{k<i} L_{i,k} \cdot U_{k,j} \qquad \text{for } i \leq j$$

$$L_{i,j} = (A_{i,j} - \sum_{k<j} L_{i,k} \cdot U_{k,j}) / U_{j,j} \qquad \text{for } i > j$$

Fits the generalized matrix computations:

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

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 $U_{j,j}$)
- I/O lower bound: $O(G/\sqrt{M}) = O(n^3/\sqrt{M})$
- Some algorithms attain this bound (hard because of pivoting)

Last homework (due Nov. 2nd)

We consider the following algorithm for computing the solution of a linear system of equations $Ax = b$ where A is a lower triangular matrix (of size $n \times n$) and x and b are two vectors (of size n):

for $i = 1 \ldots n$ **do**
  $x_i \leftarrow b_i$

for $i = 1 \ldots n$ **do**
  $x_i \leftarrow \dfrac{x_i}{A_{i,i}}$ **for** $k = i + 1 \ldots n$ **do**
    $x_k \leftarrow x_k - x_i \times A_{k,i}$

Questions:

1. Show how this computation can be modeled as a generalized matrix computation. In particular, exhibit $A, B, C, f_{i,j}, g_{i,j,k}, S_{i,j}$ and possibly other arguments.
2. Extend the previous lower bound on the total volume of communication to this problem.

Part 4: Communication Avoiding Algorithms

Generalization to other Linear Algebra Algorithms

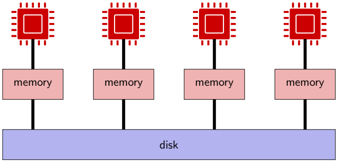
- Generalized Matrix Computations
- I/O Analysis
- Application to LU Factorization

Analysis and Lower Bounds for Parallel Algorithms

- Matrix Multiplication Lower Bound for P processors
- 2D and 3D Algorithms for Matrix Multiplication
- 2.5D Algorithm for Matrix Multiplication

Conclusion

Matrix Multiplication Lower Bound for P processors



Lemma.

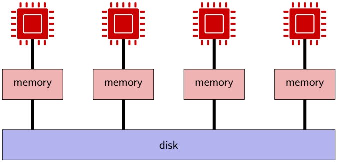
Consider a conventional $N \times N$ matrix multiplication performed on P processors with distributed memory. A processor with memory M that perform W elementary products must send or receive at least $\frac{W}{2\sqrt{2}\sqrt{M}} - M$ elements.

Theorem.

Consider a conventional $N \times N$ matrix multiplication on P processors, each with a memory M. Some processor has a volume of I/O at least $\frac{N^3}{2\sqrt{2}P\sqrt{M}} - M$.

NB: bound useful only when $M < N^2/(2P^{3/2})$

Matrix Multiplication Lower Bound for P processors



Lemma.

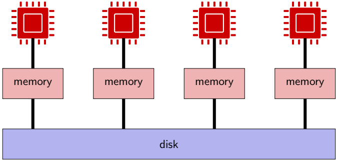
Consider a conventional $N \times N$ matrix multiplication performed on P processors with distributed memory. A processor with memory M that perform W elementary products must send or receive at least $\frac{W}{2\sqrt{2}\sqrt{M}} - M$ elements.

Theorem.

Consider a conventional $N \times N$ matrix multiplication on P processors, each with a memory M. Some processor has a volume of I/O at least $\frac{N^3}{2\sqrt{2}P\sqrt{M}} - M$.

NB: bound useful only when $M < N^2/(2P^{3/2})$

Matrix Multiplication Lower Bound for P processors



Lemma.

Consider a conventional $N \times N$ matrix multiplication performed on P processors with distributed memory. A processor with memory M that perform W elementary products must send or receive at least $\frac{W}{2\sqrt{2}\sqrt{M}} - M$ elements.

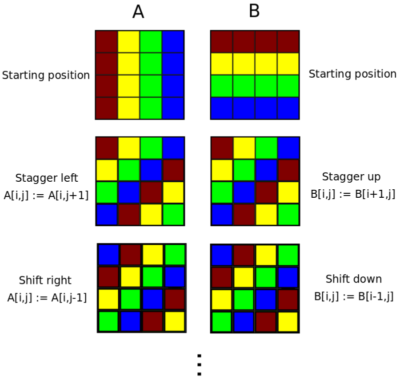
Theorem.

Consider a conventional $N \times N$ matrix multiplication on P processors, each with a memory M. Some processor has a volume of I/O at least $\frac{N^3}{2\sqrt{2}P\sqrt{M}} - M$.

NB: bound useful only when $M < N^2/(2P^{3/2})$

Cannon's 2D algorithm

- Processors organized on a square 2D grid of size $\sqrt{P} \times \sqrt{P}$
- A, B, C matrices distributed by blocks of size $N/\sqrt{P} \times N/\sqrt{P}$
  Processor $P_{i,j}$ initially holds matrices $A_{i,j}$, $B_{i,j}$, computes $C_{i,j}$
- At each step, each proc. performs a $A_{i,k} \times B_{k,j}$ block product



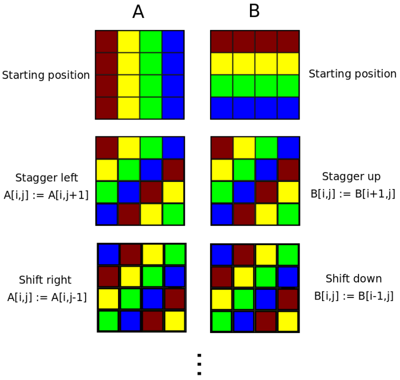
(color $= k$)

- First reallign matrices:
  - Shift $A_{i,j}$ blocks to the left by i (wraparound)
  - Shift $B_{i,j}$ blocks to the top by j (wraparound)

  Then $P_{i,j}$ holds blocks $A_{i,i+j}$ and $B_{i+j,j}$
- At each step:
  - Compute one block product
  - shift A blocks right
  - shift B blocks down
- Total I/O cost: $O(N^2\sqrt{P})$
- Storage $O(N^2/P)$ per proc.

Cannon's 2D algorithm

- Processors organized on a square 2D grid of size $\sqrt{P} \times \sqrt{P}$
- A, B, C matrices distributed by blocks of size $N/\sqrt{P} \times N/\sqrt{P}$
  Processor $P_{i,j}$ initially holds matrices $A_{i,j}$, $B_{i,j}$, computes $C_{i,j}$
- At each step, each proc. performs a $A_{i,k} \times B_{k,j}$ block product



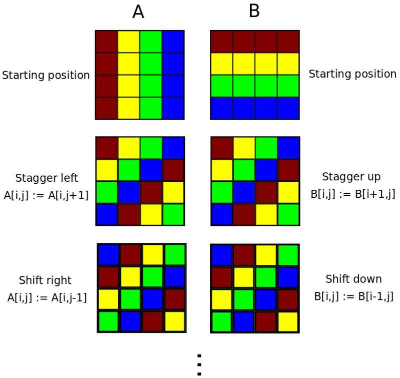
(color $= k$)

- First reallign matrices:
  - Shift $A_{i,j}$ blocks to the left by i (wraparound)
  - Shift $B_{i,j}$ blocks to the top by j (wraparound)

  Then $P_{i,j}$ holds blocks $A_{i,i+j}$ and $B_{i+j,j}$
- At each step:
  - Compute one block product
  - shift A blocks right
  - shift B blocks down
- Total I/O cost: $O(N^2\sqrt{P})$
- Storage $O(N^2/P)$ per proc.

Cannon's 2D algorithm

- Processors organized on a square 2D grid of size $\sqrt{P} \times \sqrt{P}$
- A, B, C matrices distributed by blocks of size $N/\sqrt{P} \times N/\sqrt{P}$
  Processor $P_{i,j}$ initially holds matrices $A_{i,j}$, $B_{i,j}$, computes $C_{i,j}$
- At each step, each proc. performs a $A_{i,k} \times B_{k,j}$ block product



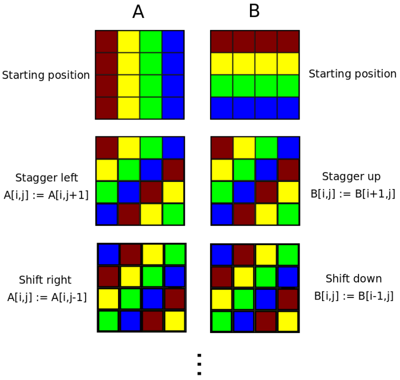
(color $= k$)

- First reallign matrices:
  - Shift $A_{i,j}$ blocks to the left by i (wraparound)
  - Shift $B_{i,j}$ blocks to the top by j (wraparound)

  Then $P_{i,j}$ holds blocks $A_{i,i+j}$ and $B_{i+j,j}$
- At each step:
  - Compute one block product
  - shift A blocks right
  - shift B blocks down
- Total I/O cost: $O(N^2\sqrt{P})$
- Storage $O(N^2/P)$ per proc.

Cannon's 2D algorithm

- Processors organized on a square 2D grid of size $\sqrt{P} \times \sqrt{P}$
- A, B, C matrices distributed by blocks of size $N/\sqrt{P} \times N/\sqrt{P}$
  Processor $P_{i,j}$ initially holds matrices $A_{i,j}$, $B_{i,j}$, computes $C_{i,j}$
- At each step, each proc. performs a $A_{i,k} \times B_{k,j}$ block product



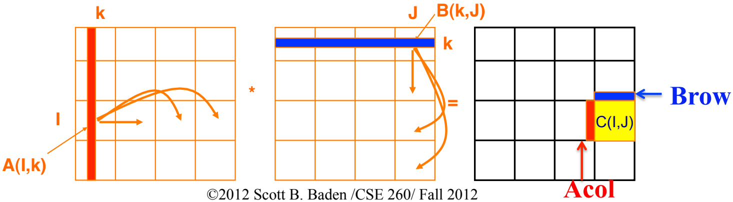
(color $= k$)

- First reallign matrices:
  - Shift $A_{i,j}$ blocks to the left by i (wraparound)
  - Shift $B_{i,j}$ blocks to the top by j (wraparound)

  Then $P_{i,j}$ holds blocks $A_{i,i+j}$ and $B_{i+j,j}$
- At each step:
  - Compute one block product
  - shift A blocks right
  - shift B blocks down
- Total I/O cost: $O(N^2\sqrt{P})$
- Storage $O(N^2/P)$ per proc.

Other 2D Algorithm: SUMMA

- SUMMA: Scalable Universal Matrix Multiplication Algorithm
- Same 2D grid distribution: $P_{i,j}$ holds $A_{i,j}$, $B_{i,j}$, computes $C_{i,j}$
- At each step k, column k of A and row k of B are broadcasted (from processors owning the data)
- Each processor computes a local contribution (outer-product)



©2012 Scott B. Baden /CSE 260/ Fall 2012

- Smaller communications ⇒ smaller temporary storage
- Same I/O volume: $O(N^2\sqrt{P})$

I/O Lower Bound for 2D algorithms

Theorem.

Consider a conventional matrix multiplication on P processors each with $O(N^2/P)$ storage, some processor has a I/O volume at least $\Theta(N^2/\sqrt{P})$.

Proof: Previous result: $O(N^3/P\sqrt{M})$ with $M = N^2/P$.

- When balanced, total I/O volume: $\Theta(N^2\sqrt{P})$
- Both Cannon's algorithm and SUMMA are optimal

Can we do better?

I/O Lower Bound for 2D algorithms

Theorem.

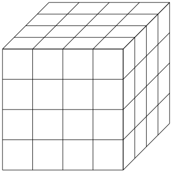
Consider a conventional matrix multiplication on P processors each with $O(N^2/P)$ storage, some processor has a I/O volume at least $\Theta(N^2/\sqrt{P})$.

Proof: Previous result: $O(N^3/P\sqrt{M})$ with $M = N^2/P$.

- When balanced, total I/O volume: $\Theta(N^2\sqrt{P})$
- Both Cannon's algorithm and SUMMA are optimal among 2D algorithms (memory limited to $O(N^2/P)$)

Can we do better?

3D Algorithm

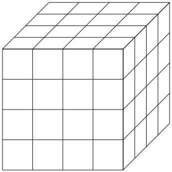


- Consider 3D grid of processor: $q \times q \times q$ ($q = P^{1/3} = \sqrt[3]{P}$)
- Processor i, j, k owns blocks $A_{i,k}, B_{k,j}, C^{(k)}_{i,j}$
- Matrices are replicated (including C)
- Each processor computes its local contribution
- Then summation of the various $C^{(k)}_{i,j}$ for all k
- Memory needed: $O(N^2/q^2) = O(N^2/P^{2/3})$ per processor
- Total I/O volume: $O(N^2/q^2 \times q^3) = O(N^2 q) = O(N^2 \sqrt[3]{P})$

Lower Bound:

- Previous theorem does not give useful bound (only when $M < N^2/2/P^{2/3}$)
- More complex analysis shows that the I/O volume on some processor is $\Theta(N^2/P^{2/3})$
- In total, when balanced $\Theta(N^2 \sqrt[3]{P}) \Rightarrow$ 3D algo. is optimal
- Can we do better?

3D Algorithm

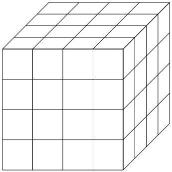


- Consider 3D grid of processor: $q \times q \times q$ ($q = P^{1/3} = \sqrt[3]{P}$)
- Processor i, j, k owns blocks $A_{i,k}, B_{k,j}, C_{i,j}^{(k)}$
- Matrices are replicated (including C)
- Each processor computes its local contribution
- Then summation of the various $C_{i,j}^{(k)}$ for all k
- Memory needed: $O(N^2/q^2) = O(N^2/P^{2/3})$ per processor
- Total I/O volume: $O(N^2/q^2 \times q^3) = O(N^2 q) = O(N^2 \sqrt[3]{P})$

Lower Bound:

- Previous theorem does not give useful bound (only when $M < N^2/2/P^{2/3}$)
- More complex analysis shows that the I/O volume on some processor is $\Theta(N^2/P^{2/3})$
- In total, when balanced $\Theta(N^2 \sqrt[3]{P}) \Rightarrow$ 3D algo. is optimal
- Can we do better?

3D Algorithm

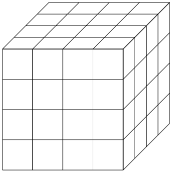


- Consider 3D grid of processor: $q \times q \times q$ ($q = P^{1/3} = \sqrt[3]{P}$)
- Processor i, j, k owns blocks $A_{i,k}, B_{k,j}, C_{i,j}^{(k)}$
- Matrices are replicated (including C)
- Each processor computes its local contribution
- Then summation of the various $C_{i,j}^{(k)}$ for all k
- Memory needed: $O(N^2/q^2) = O(N^2/P^{2/3})$ per processor
- Total I/O volume: $O(N^2/q^2 \times q^3) = O(N^2 q) = O(N^2 \sqrt[3]{P})$

Lower Bound:

- Previous theorem does not give useful bound (only when $M < N^2/2/P^{2/3}$)
- More complex analysis shows that the I/O volume on some processor is $\Theta(N^2/P^{2/3})$
- In total, when balanced $\Theta(N^2 \sqrt[3]{P}) \Rightarrow$ 3D algo. is optimal
- Can we do better?

3D Algorithm



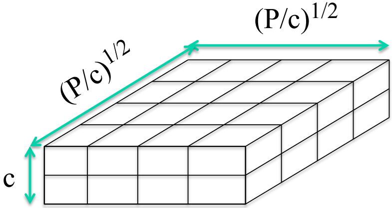
- Consider 3D grid of processor: $q \times q \times q$ ($q = P^{1/3} = \sqrt[3]{P}$)
- Processor i, j, k owns blocks $A_{i,k}, B_{k,j}, C_{i,j}^{(k)}$
- Matrices are replicated (including C)
- Each processor computes its local contribution
- Then summation of the various $C_{i,j}^{(k)}$ for all k
- Memory needed: $O(N^2/q^2) = O(N^2/P^{2/3})$ per processor
- Total I/O volume: $O(N^2/q^2 \times q^3) = O(N^2 q) = O(N^2 \sqrt[3]{P})$

Lower Bound:

- Previous theorem does not give useful bound (only when $M < N^2/2/P^{2/3}$)
- More complex analysis shows that the I/O volume on some processor is $\Theta(N^2/P^{2/3})$
- In total, when balanced $\Theta(N^2 \sqrt[3]{P}) \Rightarrow$ 3D algo. is optimal
- Can we do better?

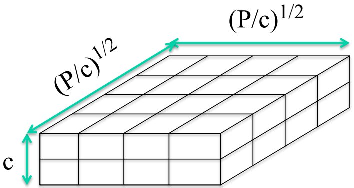
2.5D Algorithm (1/2)

- 3D algorithm requires large memory on each processor ($\sqrt[3]{P}$ copies of each matrices)
- What if we have space for only $1 < c < \sqrt[3]{P}$ copies ?
- Assume each processor has a memory $M = O(c \cdot N^2/P)$
- Arrange processors in $\sqrt{P/c} \times \sqrt{P/c} \times c$ grid:
  c layers, each layer with P/c processors in square grid
- A, B, C distributed by blocks of size $N\sqrt{c/P} \times N\sqrt{c/P}$, replicated on each layer



- NB: $c = 1$ gets 2D, $c = P^{1/3}$ gives 3D

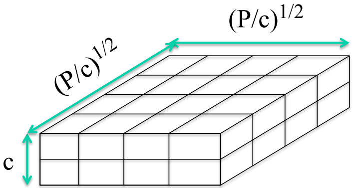
2.5D Algorithm (2/2)



- Each layer responsible for a fraction $1/c$ of Cannon's alg.: Different initial shifts of A and B
- Finally, sum C over layers
- Total I/O volume: $O(N^2/\sqrt{P/c})$
  - Replication, initial shift, final sum: $O(N^2 c)$
  - c layers of fraction $1/c$ of Cannon's alg. with grid size $\sqrt{P/c}$: $O\left(N^2\sqrt{P/c}\right)$
- Reaches lower bound on I/Os per processor:

$$O\left(\frac{N^3}{P\sqrt{M}}\right) = O\left(\frac{N^3}{P\sqrt{cN^2/P}}\right) = O(N^2/\sqrt{cP})$$

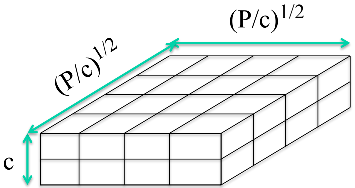
2.5D Algorithm (2/2)



- Each layer responsible for a fraction $1/c$ of Cannon's alg.: Different initial shifts of A and B
- Finally, sum C over layers
- Total I/O volume: $O(N^2/\sqrt{P/c})$
  - Replication, initial shift, final sum: $O(N^2c)$
  - c layers of fraction $1/c$ of Cannon's alg. with grid size $\sqrt{P/c}$: $O\left(N^2\sqrt{P/c}\right)$
- Reaches lower bound on I/Os per processor:

$$O\left(\frac{N^3}{P\sqrt{M}}\right) = O\left(\frac{N^3}{P\sqrt{cN^2/P}}\right) = O(N^2/\sqrt{cP})$$

2.5D Algorithm (2/2)

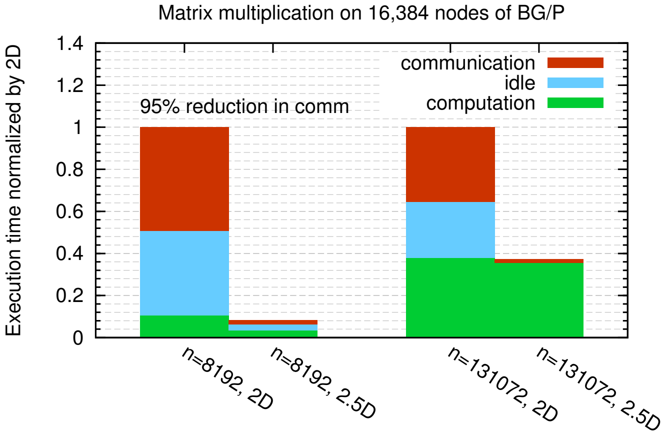


- Each layer responsible for a fraction $1/c$ of Cannon's alg.: Different initial shifts of A and B
- Finally, sum C over layers
- Total I/O volume: $O(N^2/\sqrt{P/c})$
  - Replication, initial shift, final sum: $O(N^2 c)$
  - c layers of fraction $1/c$ of Cannon's alg. with grid size $\sqrt{P/c}$: $O\left(N^2\sqrt{P/c}\right)$
- Reaches lower bound on I/Os per processor:

$$O\left(\frac{N^3}{P\sqrt{M}}\right) = O\left(\frac{N^3}{P\sqrt{cN^2/P}}\right) = O(N^2/\sqrt{cP})$$

Performance on Blue Gene P

C=16



Source Jim Demmel

Part 4: Communication Avoiding Algorithms

Generalization to other Linear Algebra Algorithms
Generalized Matrix Computations
I/O Analysis
Application to LU Factorization

Analysis and Lower Bounds for Parallel Algorithms
Matrix Multiplication Lower Bound for P processors
2D and 3D Algorithms for Matrix Multiplication
2.5D Algorithm for Matrix Multiplication

Conclusion

Conclusion

Generalized I/O lower bound for matrix computations:

- Apply to most linear algebra algorithms
- Design of I/O-optimal algorithms

Parallel algorithms with distributed memory:

- Adapted I/O lower bounds (depends on M on each processor)
- Asymptotically optimal algorithm for matrix multiplication…
  …and many other matrix computations
  "communication-avoiding algorithms"

- Here: focus on the total I/O volume
- Similar lower bound and analysis for the number of messages: also important factor for performance
- Variant: Write-avoiding algorithms for NVRAMs (writes more expensive than reads)