Parallel Dense Matrix Algorithms

Chris Kauffman

Last Updated:
Sun Oct 3 10:02:31 PM CDT 2021

Logistics

Assignments

- A1 grading in progress, look for results in the next day
- A2 will go up around ~~Friday~~ Thursday and feature MPI Coding

Today

- Overview of some Linear Algebra Libraries
- Mini-Exam 1

Recall Matrix Transpose

- Common operation on matrices is a **transpose** notated A^T
- Interchanges rows/columns of A: $a_{ij} \rightarrow a_{ji}$
- Diagonal elements stay the same
- Algorithms that perform operations on A can often be performed on A^T without re-arranging A - how? *Hint: consider summing rows of A vs summing rows of A^T*

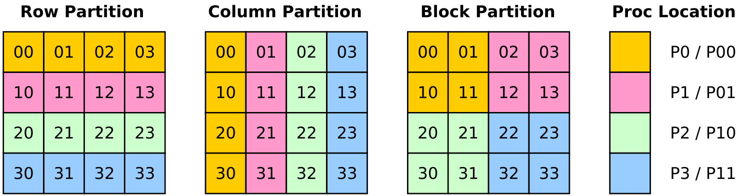
Original matrix A

0	5	10	15
20	25	30	35
40	45	50	55
60	65	70	75

transpose(A)

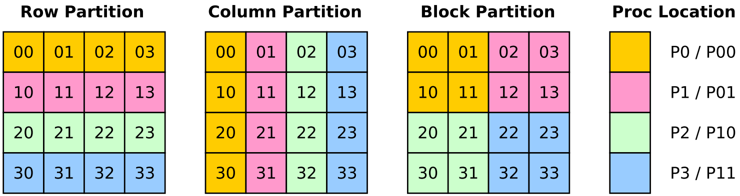
0	20	40	60
5	25	45	65
10	30	50	70
15	35	55	75

Exercise: Matrix Partitioning Across Processors



- Recall several ways to partition matrices across processors
- Diagram shows these
  - Entry `ij` may be an individual element OR...
  - Entry `ij` may be a **Block**: ex. Block (2,3) is the submatrix from rows 200-299 and cols 300-399
- Assume **square** matrices : #rows = #cols
- Common to multiply to compute product: $C = A \times B$
- Ideal partitioning for A and B?
- Ideal partitioning for $C = A^T \times B$
- Ideal partitioning for $C = A \times B^T$

Answers: Matrix Partitioning Across Processors



- $C = A \times B$
  - Ideally A is row-partitioned, B is column partitioned
  - Then block-partitioned C could be computed w/o communication
- $C = A^T \times B$
  - Ideally A and B column-partitioned
- $C = A \times B^T$
  - Ideally A and B row-partitioned
- Block-partitioning often used: not ideal for any version but less communication required when both A and A^T will b used

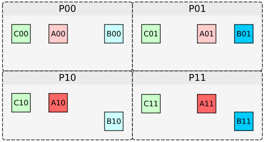
Naive Parallel Dense Multiplication: Overview

- Most applications start with block-partitioned matrices
- To compute Matrix-Matrix multiply, procs must have (eventually) multiply full rows by full columns to compute an output block
- Naive method: each Proc stores full rows/columns needed for it to independently compute output block which it stores

Naive Parallel Dense Multiplication: Demo

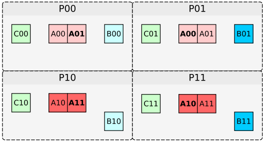
1. Initial data layout: each Proc holds a block of A, B, and C respectively. Processors are arragned in a logical grid that reflects their initial data.

In subsequent steps, received / computed data is bolded.



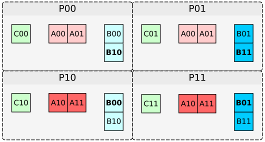
2. Each proc participates in an All-to-All sharing of data for the Row it is in.

This leaves each proc with entire rows of A.

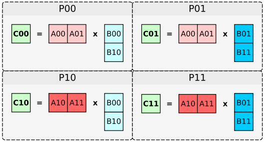


3. Each proc participates in an All-to-All sharing of data for the Column it is in.

This leaves each row with complete columns as well.



4. Each Proc now has a unique set of complete Rows and Columns and can independently compute a block of output matrix C through block multiplication.



Exercise: Analysis of Naive Dense Mult.

Assumptions

- Matrices A and B are size $N \times N$ so N^2 elements
- P processors with block partitioning: initially $N^2/\sqrt{P}$ elements of A, B on each proc (assume P is a perfect square)
- Simplified communication cost for All-to-All on a Ring:

$$t_{comm} = (p-1)(t_s + t_w M)$$

with p #procs in ring, t_s comm startup time, t_w per word transfer rate, M message size.

Questions

1. What is communication cost of this algorithm?
2. How much time does the final block matrix multiply take?
3. What is the memory requirement for each proc?
4. What do you see as the biggest disadvantage for this algorithm?

Answers: Analysis of Naive Dense Mult.

1. What is communication cost of this algorithm?
   - #Procs in rows/cols is $\sqrt{P} \sim$ ring size
   - $M = N^2/\sqrt{P}$: message size is num elements on each proc
   - 2 All-to-All shares : 1 for rows, 1 for cols

   $$t_{comm} = 2(\sqrt{P} - 1) \times (t_s + t_w(N^2/\sqrt{P}))$$

2. What is the memory requirement for each proc? E.g. how many submatrices of A,B are on each proc?
   - Full rows/cols on each proc
   - Requires $\sqrt{P}$ submatrices for each Proc
3. How much time does the final block matrix multiply take?
   - Each proc has $\sqrt{P}$ submats of A,B to multiply
   - Each submat is size $N/\sqrt{O}$ with size s requiring $O(s^3)$ opts

   $$t_{mult} = O((\sqrt{P}) \times ((N/\sqrt{P})^3)) = O(N^3/P)$$

4. What do you see as the biggest disadvantage for this algorithm?
   - Major: The need to store $\sqrt{P}$ sub matrices on all procs may be prohibitive
   - Minor: Not much chance to overlap communication / computation in the algorithm

Cannon's Algorithm

- Proposed in Lynn Elliot Carter's 1969 thesis
- Target was very small parallel machines implementing a Kalman Filter algorithm in hardware
- "Communication" happening between small Procs with data in registers
- Scales nicely to large machines and overcomes the large memory requirement of the Naive Mat-Mult Algorithm

A CELLULAR COMPUTER TO IMPLEMENT

THE KALMAN FILTER ALGORITHM

by

LYNN ELLIOT CANNON

By the conventional definition of matrix product, if A is multiplied by B, the result, call it C, is given by

$$C = A \times B = \begin{bmatrix} a_1b_1+a_2b_2+a_3b_3 & a_1b_4+a_2b_5+a_3b_6 & a_1b_7+a_2b_8+a_3b_9 \\ a_4b_1+a_5b_2+a_6b_3 & a_4b_4+a_5b_5+a_6b_6 & a_4b_7+a_5b_8+a_6b_9 \\ a_7b_1+a_8b_2+a_9b_3 & a_7b_4+a_8b_5+a_9b_6 & a_7b_7+a_8b_8+a_9b_9 \end{bmatrix}.$$

The symmetry of this product can be seen by comparing the ij^{th} element with the ji^{th} element and noticing that one is obtained from

-24-

A. 1. The first row of A is left alone.

2. The second row of A is shifted left one column.

3. The third row of A is shifted left two columns.

(Note, in general the i^{th} row of A is shifted left $i-1$ columns for $i = 1, \ldots, n$).

B. 1. The first column of B is left alone.

2. The second column of B is shifted up one row

3. The third column of B is shifted up two rows.

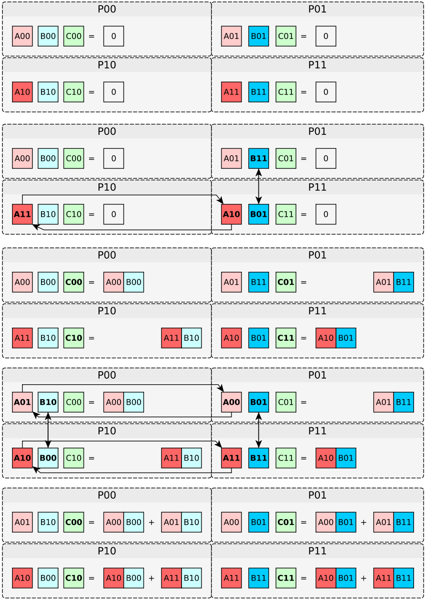
(Note, in general the j^{th} column of B is shifted up $j-1$ rows for $j = 1, \ldots, n$)

Once the registers have been shifted the multiplication pr

Demo

Cannon's Algorithm for Parallel Matrix Multiply: Demo for 2x2 block arrangment

$$\begin{bmatrix} C00 & C01 \\ C10 & C11 \end{bmatrix} = \begin{bmatrix} A00 & A01 \\ A10 & A11 \end{bmatrix} \times \begin{bmatrix} B00 & B01 \\ B10 & B11 \end{bmatrix}$$



1. Initial data layout.
Each Proc holds a block of A, B, and C respectively. Processors are arragned in a logical grid that reflects their initial data.

In subsequent steps, received / computed data is bolded.

2. Initial communication.
Row I: Rotate A blocks left by I procs
Col J: Rotate B blocks down by J procs

3. Multiply Step
Each Proc has a submatrix of A and B that allows partial computation of its output block of C. Stored blocks of A and B are multiplied and added onto contents of C.

4. Subsequent communication.
Row I: Rotate A blocks left by 1.
Col J: Rotate B blocks down by 1.

5. Multiply Step (final).
Each Proc has a submatrix of A and B that allows partial computation of its output block of C

Algorithm terminates after sqrt(P) Communicate / Multiply steps.

Cannon's Algorithm Pseudocode

```
procedure Cannon(i , j, N){
  PE(i,j) has blocks A1=A(i,j) and B1=B(i,j)
  Allocate space A2, B2, Cij sized as A1

  rsend_col = (j-i+N % N)
  rrecv_col = (j+i+N % N)
  doboth send A1 to   PE(i,rsend_col)
         recv A2 from PE(i,rsend_col)

  csend_row = (i-j+N % N)
  crecv_row = (i+j+N % N)
  doboth send B1 to   PE(csend_row,j)
         recv B2 from PE(csend_row,j)

  for(i=1 to N){
    swap A1,A2 and B1,B2
    Cij += A1 * B1

    doboth send A1 to   PE(i, j-1+N % N)
           recv A2 from PE(i, j+1+N % N)
    doboth send B1 to   PE(i-1+N % N, j)
           recv B2 from PE(i+1+N % N, j)
  }
}
```

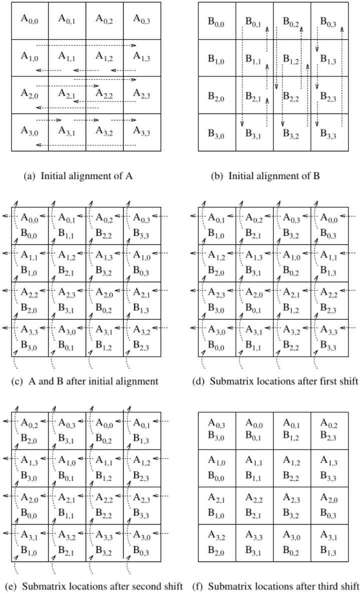


Figure 8.3 The communication steps in Cannon's algorithm on 16 processes.