Dense Matrix Algorithms
(Chapter 8)

Alexandre David
B2-206
Dense Matrix Algorithm

- Dense or full matrices: few known zeros.
  - Other algorithms for sparse matrix.
- Square matrices for pedagogical purposes only – can be generalized.
- Natural to have data decomposition.
  - 3.2.2 input/output/intermediate data.
  - 3.4.1 mapping schemes based on data partitioning.
Today

- Matrix*Vector
- Matrix*Matrix
- Solving systems of linear equations.

The chapter is not just about giving algorithms but also about analyzing them.
Recall: See chapter 5 on analysis. $W = \text{problem size} = \text{number of basic computation in the best sequential algorithm to solve the problem.}$
There are better algorithms but the sake of simplicity we consider the straightforward computation, which is still a good algorithm, though not the best.
Matrix*Vector – Serial Algorithm

\[ y_i = \sum_{k=1}^{n} a_{ik} x_k \]

**procedure** MAT_Vec(A,x,y)
  for i := 0 to n-1 do
    y[i] := 0
  for k := 0 to n-1 do
    y[i] := y[i] + A[i,k]*x[k]
  done
  done
endproc

How to parallelize?

How to parallelize: Row-wise 1-D, column-wise 1-D, or 2-D partitioning.
Matrix*Matrix – Serial Algorithm

\[ c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \]

\[ \text{procedure MAT_MULT}(A,B,C) \]
\[ \quad \text{for } i := 0 \text{ to } n-1 \text{ do} \]
\[ \quad \quad \text{for } j := 0 \text{ to } n-1 \text{ do} \]
\[ \quad \quad \quad C[i,j] := 0 \]
\[ \quad \quad \quad \text{for } k := 0 \text{ to } n-1 \text{ do} \]
\[ \quad \quad \quad \quad C[i,j] := C[i,j] + A[i,k]B[k,j] \]
\[ \quad \quad \text{done} \]
\[ \quad \text{done} \]
\[ \text{endproc} \]

How to parallelize?

Similar partitioning as for the matrix*vector and block partitioning.
Matrix*Vector – Row-wise 1-D Partitioning

- Initial distribution:
  - Each process has a row of the $n \times n$ matrix.
  - Each process has an element of the $n \times 1$ vector.
  - Each process is responsible for computing one element of the result.
Matrix*Vector – 1-D

n processes

\[ \begin{align*} \begin{array}{ccc} A & x & y \\ \times & = & = \\ \end{array} \end{align*} \]

But every process needs the entire vector \( \Rightarrow \) all-to-all broadcast.
All-to-All Broadcast
Parallel Computation

\[ y_i = \sum_{k=1}^{n} a_{ik} x_k \] in parallel on the \( n \) processes.
Example Matrix*Vector (Program 6.4)

Partition on rows.

Allgather (All-to-all broadcast)

Multiply

04-04-2006  Alexandre David, MVP'06
Analysis

- All-to-all broadcast & multiplications in $\Theta(n)$.
- For $n$ processes $W=n^2=\alpha n T_p$
  $\Rightarrow$ The algorithm is cost-optimal.

A parallel system is cost-optimal iff
$p T_p=\Theta(W)$. 
Performance Metrics

- Efficiency $E = S/p$.
  - Measure time spent in doing useful work.
  - Previous sum example: $E = \Theta(1/\log n)$.
- Cost $C = pT_P$
  - A.k.a. work or processor-time product.
  - Note: $E = T_s/C$.
  - Cost optimal if $E$ is a constant.

Reminder from lecture 8.
Using Fewer Processes

- Brent’s scheduling principle: It’s possible.
- Using \( p \) processes:
  - \( n/p \) rows per process.
  - Communication time = \( t_s \log p + t_w (n/p)(p-1) \)
    \( \sim t_s \log p + t_w n = \Theta(n) \).
  - Computation: \( n \times n/p \).
    \[ \Rightarrow pT_p = \Theta(n^2) = W \Rightarrow \text{It is cost optimal.} \]
Scalability – Recall

- Efficiency increases with the size of the problem.
- Efficiency decreases with the number of processors.
- Scalability measures the ability to increase speedup in function of $p$. 
Isoefficiency Function

For scalable systems efficiency can be kept constant if \( T_0/W \) is kept constant.

For a target \( E \)

\[
E = \frac{1}{1 + \frac{T_0(W,p)}{W}}
\]

Keep this constant

\[
\frac{T_0(W,p)}{W} = \frac{1 - E}{E},
\]

Isoefficiency function

\[
W = \frac{E}{1 - E} T_0(W,p).
\]

What it means: The isoeficiency function determines the ease with which a parallel system can maintain its efficiency in function of the number of processors. A small function means that small increments of the problem size are enough (to compensate the increase of \( p \)), i.e., the system is scalable. A large function means the problem size must be incremented dramatically to compensate \( p \), i.e., the system is poorly scalable.

Unscalable system do not have an isoeficiency function.

Isoeficiency function is in function of \( p \).
Is Our Algorithm Scalable?

- \( T_0 = pT_P - W \Rightarrow T_0 = tp \log p + t_w np. \)
- We want to determine \( W = KT_0. \) Try with both terms separately:
  - \( W = Kt_P \log p. \)
  - \( W = Kt_w np = n^2 \Rightarrow W = (Kt_w p)^2. \)
- Bound from concurrency: \( p = O(n) \Rightarrow W = \Omega(p^2). \)
- \( W = \Theta(p^2) \): asymptotic isoefficiency function. Rate to increase the problem size (in function of \( p \)) to maintain a fixed efficiency: \( p = \Theta(n). \)
Matrix*Vector – 2-D

- Matrix $n\times n$ partitioned on $n\times n$ processes.
- Vector $n\times 1$ distributed in the last (or 1st column).
- Similarly we want fewer processes: blocks of $(n/\sqrt{p})^2$ elements.
Matrix*Vector – 2-D

$\begin{array}{c}
\begin{array}{c}
A \\
\times
\end{array}
\end{array}$

$\begin{array}{c}
\begin{array}{c}
x \\
= \\
y
\end{array}
\end{array}$

Processes in column $i$ need element of the vector in row $i$.

1. Distribute on diagonal.
2. One-to-all broadcast on columns.
3. Multiplication.
4. All-to-one reduction (+).
Example Matrix*Vector (Program 6.8)

Partition.

Distribute vector.

Sum reduce on rows.

Row sub-topology.

Colum sub-topology.

Local multiplication.
Which one is better? 1-D or 2-D?
Analysis

Communications:
- one-to-one $\Theta(1)$ +
- one-to-all broadcast $\Theta(\log n)$ +
- all-to-one reduction $\Theta(\log n)$.
- + multiplications $\Theta(1)$.
- $T_p=\Theta(n^2 \log n) \Rightarrow$ not cost-optimal.
- Brent’s scheduling principle?
Using Fewer Processes

- Blocks of \( (n/\sqrt{p})^2 \) elements. Costs:
  - one to one in \( t_s + t_w n/\sqrt{p} \) +
  - one-to-all broadcast in \( (t_s + t_w n/\sqrt{p}) \log \sqrt{p} \) +
  - all-to-one reduction in \( (t_s + t_w n/\sqrt{p}) \log \sqrt{p} \) +
  - computations in \( (n/\sqrt{p})^2 \).
- Total \( \sim n^2/p + t_s \log p + (t_w n/\sqrt{p}) \log p \).
- \( pT_p = \Theta(n^2) \Rightarrow \text{cost-optimal.} \)
Scalability Analysis

- \[ T_0 = pT_p - W = t_s \log p + t_w n \sqrt{p \log p}. \]
- As before, isoefficiency analysis:
  - \[ W = K t_s p \log p. \]
  - \[ W = K t_w n \sqrt{p \log p} = n^2 \Rightarrow W = (K t_w \sqrt{p \log p})^2. \]
  - Bound from concurrency: \( p = O(n^2) \Rightarrow W = \Omega(p). \)
  - \[ W = \Theta(p \log^2 p). \]
  - \( p = f(n) \) ? \( p \log^2 p = \Theta(n^2) \) ... \( p = \Theta(n^2 / \log^2 n). \)
Which One Is Better?

- 1-D: \( T_p \sim \frac{n^2}{p} + t_s \log p + t_w n. \)
- 2-D: \( T_p \sim \frac{n^2}{p} + t_s \log p + (t_w n/\sqrt{p}) \log p. \)

- 1-D: \( W = \Theta(p^2). \)
- 2-D: \( W = \Theta(p \log^2 p). \)

- Degree of concurrency...
procedure BLOCK_MAT_MULT(A,B,C)
    for i := 0 to q-1 do
        for j := 0 to q-1 do
            C[i,j] := 0
            for k := 0 to q-1 do
                C[i,j] := C[i,j] + A[i,k]*B[k,j]
            done
        done
    done
endproc

q*q blocks of (n/q)*(n/q) submatrices. Still $n^3$ additions & multiplications.
A Simple Parallel Algorithm

- Map the algorithm to $p=q^2$ processes.
- We need all $A[i,k]$ and $B[k,j]$ to compute the $C[i,j]$.

Steps:
- All-to-all broadcast of $A[i,k]$ on rows.
- All-to-all broadcast of $B[k,j]$ on columns.
- Local multiplications.
Analysis

Costs:
- all-to-all \( \sqrt{p} \) broadcasts of \( n^2/p \) elements
  \[ = t_s \log \sqrt{p} + t_w(n^2/p)(\sqrt{p}-1) \times 2 \]
- + computations = \( \sqrt{p} \) multiplications of \( (n/\sqrt{p}) \times (n/\sqrt{p}) \) matrices cost \( n^3/p \).
- \( pT_p = \Theta(n^3) \) for \( p = \Omega(n^2) \) \( \Rightarrow \) cost-optimal.
- Isoefficiency \( W = \Theta(p^{3/2}) \).
- Drawback: memory requirement in \( n^2 \sqrt{p} \).

Better?

One multiplication of a \( (n/\sqrt{p}) \times (n/\sqrt{p}) \) matrix costs \( (n/\sqrt{p})^3 \) and we have \( \sqrt{p} \) of them (per process).
Cannon’s Algorithm

- Idea: re-schedule computations to avoid contention.
  - Processes on rows \( i \) hold a different \( A[i,k] \).
  - Processes on columns \( j \) hold a different \( B[k,j] \).
  - Rotate the matrices \( \Rightarrow \) we need only 2 sub-matrices per process at any time.
    \( \Rightarrow \) memory efficient in \( O(n^2) \).
Align A & B

(a) Initial alignment of A

(b) Initial alignment of B
Figure 8.3 The communication steps in Cannon's algorithm on 16 processes.
### Analysis

- **Costs:**
  - $2 \times (A \& B) \sqrt{p}$-single step shifts = $2(t_s + t_w n^2/p) \sqrt{p}$ +
  - $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ submatrices = $n^3/p$.
  - Cost-optimal, same isoefficiency function as previously.
The DNS Algorithm

- 3-D partitioning!
- Cube with faces corresponding to A, B, C.
- Internal nodes correspond to multiply operations $P_{i,j,k}$.
  - Multiplications in time $\Theta(1)$.
  - Additions in time $\Theta(\log n)$.
  - Communication...
- Can use up to $n^3$ processes – better concurrency.

DNS: Dekel, Nassimi, and Sahni.
(a) Initial distribution of $A$ and $B$

(b) After moving $A_{i,j}$ from $P_{i,j}$ to $P_{i,j}$
Figure 8.4 The communication steps in the DNS algorithm while multiplying $4 \times 4$ matrices $A$ and $B$ on 64 processes. The shaded processes in part (c) store elements of the first row of $A$ and the shaded processes in part (d) store elements of the first column of $B$. 
Communication Steps

- Move the columns of A & rows of B.
- One-to-all broadcast along j & i axis.
- All-to-one reduction (+) along k axis.
- Communication on groups of \( n \) processes, in time \( \Theta(\log n) \).
- Not cost optimal for \( n^3 \) processes.
Brent’s Scheduling Principle

**Theorem**

If a parallel computation consists of \( k \) phases taking time \( t_1, t_2, \ldots, t_k \) using \( a_1, a_2, \ldots, a_k \) processors in phases \( 1, 2, \ldots, k \), then the computation can be done in time \( \mathcal{O}(a/p + t) \) using \( p \) processors where \( t = \text{sum}(t_i) \), \( a = \text{sum}(a_i t_i) \).

What it means: same time as the original plus an overhead. If the number of processors increases then we decrease the overhead. The overhead corresponds to simulating the \( a_i \) with \( p \). What it **really** means: It is possible to make algorithms optimal with the right amount of processors (provided that \( t^p \) has the same order of magnitude of \( t_{\text{sequential}} \)). That gives you a bound on the number of needed processors.

It’s a **scheduling** principle to reduce the number of physical processors needed by the algorithm and increase utilization. It does not do miracles.

Proof: \( i \)’th phase, \( p \) processors simulate \( a_i \) processors. Each of them simulate at most \( \text{ceil}(a/p) \leq a/p + 1 \), which consumes time \( t_i \) at a constant factor for each of them.
Look At One Dimension

- $k$ phases = $\log n$.
- $t_i =$ constant time.
- $a_i =$ $n/2, n/4, \ldots, 1$ processors.
- With $q$ processors we can use time $O(\log n + n/p)$.
- Choose $q = O(n/\log n) \rightarrow$ time $O(\log n)$ and this is optimal!

3-D: use $p = O(n^3/\log^3 n)$

Note: $n$ is a power of 2 to simplify. Recall the definition of optimality to conclude that it is optimal indeed. This does not give us an implementation, but almost.

Typo p6 “using $O(n/\log n)$ processors”. Divide and conquer same as compress and iterate for the exercise.
Systems of Linear Equations

\[ A \times x = b \]

\[ a_{0,0}x_0 + a_{0,1}x_1 + \ldots + a_{0,n-1}x_{n-1} = b_0, \]

\[ \ldots \]

\[ a_{n-1,0}x_0 + a_{n-1,1}x_1 + \ldots + a_{n-1,n-1}x_{n-1} = b_{n-1} \]
Solving Systems of Linear Equations

- Step 1: Reduce the original system to

\[ U \times x = y \]

- Step 2:
  Solve & back-substitute from \( x_{n-1} \) to \( x_0 \).
Technical Issues

- Non singular matrices.
- Numerical precision (is the solution numerically stable) $\rightarrow$ permute columns.
  - In particular no division by zero, thanks.
  - Procedure known as Gaussian elimination with partial pivoting.
Gaussian Elimination

1. procedure GAUSSIAN_ELIMINATION (A, b, y)
2. begin
3. for k := 0 to n - 1 do /* Outer loop */
4. begin
5. for j := k + 1 to n - 1 do
7. v[k] := b[k] / A[k, k];
8. A[k, k] := 1;
9. for i := k + 1 to n - 1 do
10. begin
11. for j := k + 1 to n - 1 do
13. b[i] := b[i] - A[i, k] * v[k];
15. endfor; /* Line 9 */
16. endfor; /* Line 3 */
17. end GAUSSIAN_ELIMINATION
Parallel Gaussian Elimination

- 1-D partitioning:
  - 1 process/row.
  - Process j computes $A[*,j]$.
  - Cost (+communication) = $\Theta(n^3 \log n)$ not cost optimal.

- All processes work on the same iteration.
  - $k+1$ iteration starts when $k^{th}$ iteration is complete.
  - Improve: pipelined/asynchronous version.
Pipelined Version

1. procedure GAUSSIAN_ELIMINATION (A, b, y)
2. begin
3.   for k := 0 to n - 1 do /* Outer loop */
4.     begin
5.       for j := k + 1 to n - 1 do
7.         y[k] := b[k] / A[k, k];
8.         A[k, k] := 1;
9.       for i := k + 1 to n - 1 do
10.      begin
11.        for j := k + 1 to n - 1 do
13.          b[i] := b[i] - A[i, k] * y[k];
15.      endfor; /* Line 9 */
16.    endfor; /* Line 3 */
17. end GAUSSIAN_ELIMINATION

P_k forwards & does not wait.
P_j's forward & do not wait.
Pipelined Gaussian Elimination

- No \( \log n \) for communication (no broadcast) and the rest of the computations are the same.
- The pipelined version is cost-optimal.
- Fewer processes:
  - Block 1-D partitioning, loss of efficiency due to idle processes (load balance problem).
  - Cyclic 1-D partitioning better.
Gaussian Elimination – 2-D
Partitioning

- Similar as before.
- Pipelined version cost-optimal.
- More scalable than 1-D.
Finally Back-Substitution

1. procedure BACK_SUBSTITUTION (U, x, y)
2. begin
3. for k := n - 1 downto 0 do /* Main loop */
4. begin
5. x[k] := y[k];
6. for i := k - 1 downto 0 do
7. y[i] := y[i] - x[k] * U[i, k];
8. endfor;
9. end BACK_SUBSTITUTION

Algorithm 8.5 A serial algorithm for back-substitution entries of the principal diagonal equal to one, and all

Intrinsically serial algorithm. Pipelined parallel version not cost optimal. Does not matter because of lower order of magnitude.