Parallel Processing

Parallel Matrix Multiplication

Felicja Okulicka-Dłużewska

F.Okulicka@mini.pw.edu.pl

Warsaw University of Technology

Faculty of Mathematics and Information Science

Go to full-screen mode now by hitting CTRL-L

Overview

- Cannon's algorithm
- Strassen's algorithm
- DNS's algorithm

Parallel Matrix Multiplication

The operation of matrix multiplication has the following properties:

- Matrix multiplication is associative:

$$A(BC) = (AB)C$$

- Matrix multiplication is distributive over matrix addition:
  - $A(B + C) = AB + AC$
  - $(A + B)C = AC + BC$

Parallel Matrix Multiplication

The operation of matrix multiplication has the following properties:

- If the matrix is defined over a field (for example, over the Real or Complex fields), then it is compatible with scalar multiplication in that field. If c is a scalar we have:
  - $c(AB) = (cA)B$
  - $(Ac)B = A(cB)$
  - $(AB)c = A(Bc)$

These properties help to create the algorithms for parallel matrix multiplication on the distributed machines.

Parallel Matrix Multiplication

We will show how to implement matrix multiplication

$$C = C + A * B$$

on several different kinds of architectures (shared memory or different kinds of nets).
Let A, B and C are dense matrices of size $n * n$ (a dense matrix is a matrix in which most of the entries are nonzero).

The standard algorithm requires $2 * n^3$ arithmetic operations. Hence the optimal parallel time on p processors will be $2 * n^3/p$ time steps (arithmetic operations). Scheduling these operations is the interesting part of the algorithm design.

Parallel Matrix Multiplication

For the **shared memory** machines the **BLAS** procedures are used. The **Level 3 BLAS** contains the efficient procedures for matrix multiplication. For distributed architecture the PBLAS procedures can be used. For **distributed memory** the **data layout** is significant. The two most basic layouts are:

- 1D blocked
- 2D blocked

BLAS

The BLAS (**B**asic **L**inear **A**lgebra **S**ubprograms) include subroutines for common linear algebra computations such as dot-products, matrix-vector multiplication, and matrix-matrix multiplication. Using matrix-matrix operations (in particular, matrix multiplication) tuned for a particular architecture can mask the effects of the memory hierarchy (cache misses, TLB misses, etc.) and permit floating-point operations to be performed near peak speed of the machine.

An important aim of the BLAS is to provide a portability layer for computation.

BLAS

Basic Linear Algebra Subprograms

The BLAS are relatively low-level linear algebra operations:

- L-norm of a vector
- Inner (dot) product of two vectors
- Position of the maximum absolute element of a vector
- Matrix-vector product
- Matrix- matrix product

BLAS1

BLAS1 = Level 1 BLAS Vector - vector operations :

O(n) operations on O(n) data (each datum is used once only).

It was formulated in the 1970s for traditional serial machines

BLAS1

Name	Operation	Prefixes
_ROTG	Generate plane rotation	S, D
_ROTMG	Generate modified plane rotation	S, D
_ROT	Apply lane rotation	S, D
_ROTM	Apply modified plane rotation	S, D
_SWAP	$x \leftrightarrow y$	S, D, C, Z
_SCAL	$x \leftarrow \alpha x$	S, D, C, Z, CS, ZD
_COPY	$y \leftarrow x$	S, D, C, Z
_AXPY	$y \leftarrow \alpha x + y$	S, D, C, Z
_DOT	$dot \leftarrow x^T y$	S, D, DS
_DOTU	$dot \leftarrow x^T y$	C, Z

BLAS1

Name	Operation	Prefixes
_DOTC	$dot \leftarrow x^H y$	C, Z
_DOT	$dot \leftarrow \alpha + x^T y$	SDS
_NRM2	$nrm2 \leftarrow \|\|x\|\|_2$	S, D, SC, DZ
_ASUM	$asum \leftarrow \|\|re(x)\|\|_1 + \|\|im(x)\|\|_1$	S, D, SC, DZ
I_AMAX	$amax \leftarrow 1^{st} k \ni \|re(x) + \|im(x)\|$	S, D, C, Z

Meaning of prefixes:

S - REAL

D - DOUBLE PRECISION

C - COMPLEX

Z - COMPLEX*16

D,Z - may not be supported by all machines

BLAS2

BLAS2 - vector-matrix operations

Name	Operation	Prefixes
_GEMV	$y \leftarrow \alpha A x + \beta y$	S, D, C, Z
	$y \leftarrow \alpha A^T x + \beta y$	S, D, C, Z
	$y \leftarrow \alpha A^H x + \beta y$	S, D, C, Z
	$A - m \times n$	
_GBMV	$y \leftarrow \alpha A x + \beta y,$	S, D, C, Z
	$y \leftarrow \alpha A^T x + \beta y$	S, D, C, Z
	$y \leftarrow \alpha A^H x + \beta y$	S, D, C, Z
	$A - m \times n$	

BLAS2

Name	Operation	Prefixes
_HEMV	$y \leftarrow \alpha Ax + \beta y$	C, Z
_HBMV	$y \leftarrow \alpha Ax + \beta y$	C, Z
_HPMV	$y \leftarrow \alpha Ax + \beta y$	C, Z
_SYMV	$y \leftarrow \alpha Ax + \beta y$	S, D
_SBMV	$y \leftarrow \alpha Ax + \beta y$	S, D
_SPMV	$y \leftarrow \alpha Ax + \beta y$	S, D

BLAS2

Name	Operation	Prefixes
_TRMV	$x \leftarrow \alpha A x, x \leftarrow \alpha A^T x, x \leftarrow \alpha A^H x$	S, D, C, Z
_TBMV	$x \leftarrow \alpha A x, x \leftarrow \alpha A^T x, x \leftarrow \alpha A^H x$	S, D, C, Z
_TPMV	$x \leftarrow \alpha A x, x \leftarrow \alpha A^T x, x \leftarrow \alpha A^H x$	S, D, C, Z
_TRSV	$x \leftarrow \alpha A^{-1} x, x \leftarrow \alpha A^{-T} x, x \leftarrow \alpha A^{-H} x$	S, D, C, Z
_TBSV	$x \leftarrow \alpha A^{-1} x, x \leftarrow \alpha A^{-T} x, x \leftarrow \alpha A^{-H} x$	S, D, C, Z
_TPSV	$x \leftarrow \alpha A^{-1} x, x \leftarrow \alpha A^{-T} x, x \leftarrow \alpha A^{-H} x$	S, D, C, Z

BLAS3

Name	Operation	Prefixes
_GER	$A \leftarrow \alpha x y^T + A,\ A - m \times n$	S, D
_GERU	$A \leftarrow \alpha x y^T + A,\ A - m \times n$	C, Z
_GERC	$A \leftarrow \alpha x y^H + A,\ A - m \times n$	C, Z
_HER	$A \leftarrow \alpha x x^H + A$	C, Z
_HPR	$A \leftarrow \alpha x x^H + A$	C, Z
_HER2	$A \leftarrow \alpha x y^H + y(\alpha x)^H + A$	C, Z
_HPR2	$A \leftarrow \alpha x y^H + y(\alpha x)^H + A$	C, Z

BLAS3

Name	Operation	Prefixes
_SYR	$A \leftarrow \alpha x x^T + A$	S, D
_SPR	$A \leftarrow \alpha x x^T + A$	S, D
_SYR2	$A \leftarrow \alpha x y^T + \alpha y x^T + A$	S, D
_SPR2	$A \leftarrow \alpha x y^T + \alpha y x^T + A$	S, D

BLAS3

Matrix types:

GE - GEneral	SY - SYmmetric
GB - General Band	SB - Symmetric Band
	SP - Symmetric Packed
HE - Hermitian	TR - TRiangular
HB - Hermitian Band	TB - Triangular Band
HP - Hermitian Packed	TP - Triangular Packed

Gauss using BLAS

Following BLAS procedures can be used to implement the Gauss elimination:

- Reciprocal Scale x=x/alpha : RSCALE
- Scaled vector accumulation y=alpha*x+beta*y : AXPBY
- Rank one updates A=alpha*x*trans(y)+beta*A : SGER, DGER
- Triangular solver : TRSV, TBSV, TPSV, TRSM

Methods for the Shared Memory Machine

Matrix Multiplication on the **Shared Memory Machine** bases on the **loop parallelization**.

The capacity of the slow and fast memories should to be taken under consideration when we implement and run the application.

The data should be reused optimally.

Optimality of the method depends on the amount of the number of references to the slow memory.

Such algorithms are **easier to implement** than in the distributed memory case.

Methods for the Distributed Machine

The two most popular algorithms for matrix multiplication on distributed memory machine were invented by Cannon and by Strassen.
Cannon's algorithm is for the **Cartesian architecture**.
Strassen's one is for the **graph architecture**.

The third method that will be described is the **DNS's** algorithm. All of them base on the partitioning of the matrix in parts which are put in separate memories.

We partition matrices A and B into m square blocks for **distributed memory** the **2D blocked data layout** . The matrix multiplication formula is:

$$C_{i,j} = \sum_{k=1}^{m} A(i,k) * B(k,j)$$

where:

$$C = \begin{bmatrix} C(1,1) & ... & C(1,m) \\ C(2,1) & ... & C(2,m) \\ ... & ... & ... \\ C(m,1) & ... & C(m,m) \end{bmatrix}$$

$$A = \begin{bmatrix} A(1,1) & ... & A(1,m) \\ A(2,1) & ... & A(2,m) \\ ... & ... & ... \\ A(m,1) & ... & A(m,m) \end{bmatrix}$$

$$B = \begin{bmatrix} B(1,1) & ... & B(1,m) \\ B(2,1) & ... & B(2,m) \\ ... & ... & ... \\ B(m,1) & ... & B(m,m) \end{bmatrix}$$

Cannon's Algorithm on 2D grid

We partition matrices A and B in p square blocks. The processes are working on the Cartesian grid and are labeled from $P(0,0)$ to $P(\sqrt{p-1}, \sqrt{p-1})$ and initially assign submatrices $A(i,j)$ and $B(i,j)$ to process $P(i,j)$. Although every process in the $i-th$ row requires $sqrt(p)$ submatrices $A(i,k)$ ($0 < k < sqrt(p)$) it is possible to schedule the computation of the $sqrt(p)$ processes in the $i-th$ row such that , at any given time, each process is using a different $A(i,k)$.

These block can be systematically rotated among the processes after every submatrix multiplication so that every process gets a fresh $A(i,k)$ after each rotation. An identical schedule is applied to the columns, then no process holds more than one block of each matrix at any time, and the total memory requirement of the algorithm over all the processes is $O(n^2)$. Cannon algorithm is based on this idea. The first communication step of the algorithm aligns the blocks of A and B in such a way that each process multiplies its local submatrices. This alignment is achieved for matrix A by shifting all submatrices $A(i,j)$ to the left (with wraparound) by i steps. All submatrices $B(i,j)$ are shifted up by j steps. These are circular shift operations.

After first shift we have the following data distribution:

$$A = \begin{bmatrix} A(1,1) & A(1,2) & ... & A(1,m) \\ A(2,2) & A(2,3) & ... & A(2,1) \\ ... & ... & ... & \\ A(m,m) & A(m,1) & ... & A(m,m-1) \end{bmatrix}$$

$$B = \begin{bmatrix} B(1,1) & B(2,2) & ... & B(m,m) \\ B(2,1) & B(3,2) & ... & B(1,m) \\ ... & ... & ... & ... \\ B(m,1) & B(1,2) & ... & B(m-1,m) \end{bmatrix}$$

The diagonal blocks of A are in the first column and the diagonal blocks of B are in the first row.

After a submatrices multiplication step , each block of A moves one step left and each block of B moves one step up (with wraparound). A sequence of $sqrt(p)$ such submatrices multiplications and single step shifts pairs up each $A(i,k)$ and $B(k,j)$ for k ($0 \leq k < sqrt(p)$) at $P(i,j)$ completes the multiplication of matrices A and B.

Example of Cannon's algorithm

The communication steps in Cannon's algorithm on 9 processes.

The initial alignment looks as follows:

$$\begin{bmatrix} A(0,0) & A(0,1) & A(0,2) \\ B(0,0) & B(0,1) & B(0,2) \\ \\ A(1,0) & A(1,1) & A(1,2) \\ B(1,0) & B(1,1) & B(1,2) \\ \\ A(2,0) & A(2,1) & A(2,2) \\ B(2,0) & B(2,1) & B(2,2) \end{bmatrix}$$

Example of Cannon's algorithm

After the initial alignment the diagonal blocks of A are in the first column and the diagonal blocks of B are in the first row.

$$\begin{bmatrix} A(0,0) & A(0,1) & A(0,2) \\ B(0,0) & B(1,1) & B(2,2) \\ \\ A(1,1) & A(1,2) & A(1,0) \\ B(1,0) & B(2,1) & B(0,2) \\ \\ A(2,2) & A(2,0) & A(2,1) \\ B(2,0) & B(0,1) & B(1,2) \end{bmatrix}$$

Example of Cannon's algorithm

The multiplications are performed by the processors:

$$\begin{bmatrix} A(0,0) * B(0,0) & A(0,1) * B(1,1) & A(0,2) * B(2,2) \\ A(1,1) * B(1,0) & A(1,2) * B(2,1) & A(1,0) * B(0,2) \\ A(2,2) * B(2,0) & A(2,0) * B(0,1) & A(2,1) * B(1,2) \end{bmatrix}$$

After first shift A is rotated right and B is rotated up.

The diagonal blocks of A are in the last column.

The diagonal blocks of B are in the last row.

Example of Cannon's algorithm

Submatrices location after first shift looks as follows:

$$\begin{bmatrix} A(0,1) & A(0,2) & A(0,0) \\ B(1,0) & B(2,1) & B(0,2) \\ \\ A(1,2) & A(1,0) & A(1,1) \\ B(2,0) & B(0,1) & B(1,2) \\ \\ A(2,0) & A(2,1) & A(2,2) \\ B(0,0) & B(1,1) & B(2,2) \end{bmatrix}$$

Example of Cannon's algorithm

The multiplication gives the partial result on each processor - the first elements of the series:

$$\begin{bmatrix} A(0,1) * B(1,0) & A(0,2) * B(2,1) & A(0,0) * B(0,2) \\ A(1,2) * B(2,0) & A(1,0) * B(0,1) & A(1,1) * B(1,2) \\ A(2,0) * B(0,0) & A(2,1) * B(1,1) & A(2,2) * B(2,2) \end{bmatrix}$$

After the submatrices multiplication after first shift and the addition to the previous the results on each processor are:

$$\begin{bmatrix} A(0,0) * B(0,0) + A(0,1) * B(1,0) & A(0,1) * B(1,1) + A(0,2) * B(2, \\ A(1,1) * B(1,0) + A(1,2) * B(2,0) & A(1,2) * B(2,1) + A(1,0) * B(0, \\ A(2,2) * B(2,0) + A(2,0) * B(0,0) & A(2,0) * B(0,1) + A(2,1) * B(1, \end{bmatrix}$$

Example of Cannon's algorithm

Submatrices location after second shift:

$$\begin{bmatrix} A(0,2) & A(0,0) & A(0,1) \\ B(2,0) & B(0,1) & B(1,2) \\ \\ A(1,0) & A(1,1) & A(1,2) \\ B(0,0) & B(1,1) & B(2,2) \\ \\ A(2,1) & A(2,2) & A(2,0) \\ B(1,0) & B(2,1) & B(0,2) \end{bmatrix}$$

Example of Cannon's algorithm

The result on each processor after second shift and multiplications gives the final result:

$$\begin{bmatrix} A(0,0)*B(0,0)+A(0,1)*B(1,0)+A(0,2)*B(2,0) & \ldots & \ldots \\ A(1,1)*B(1,0)+A(1,2)*B(2,0)+A(1,0)*B(0,0) & \ldots & \ldots \\ A(2,2)*B(2,0)+A(2,0)*B(0,0)+A(2,1)*B(1,0) & \ldots & \ldots \end{bmatrix}$$

The arguments against the Cannon's algorithm are following:

- The number of processors p must be a perfect square,
- A and B must be square,
- The use of the block-cyclic layouts complicates the computation,
- A and B are not 'aligned' in the way they are stored on processors,
- Extra copies of the local matrices are needed.

1D Cannon's Algorithm by columns

The communication steps in Cannon's algorithm on 3 processes with 1D layout by columns is presented. The initial alignment looks as follows:

$$\left\|\begin{array}{c|c|c} A(0,0) & A(0,1) & A(0,2) \\ B(0,0) & B(0,1) & B(0,2) \\ & & \\ A(1,0) & A(1,1) & A(1,2) \\ B(1,0) & B(1,1) & B(1,2) \\ & & \\ A(2,0) & A(2,1) & A(2,2) \\ B(2,0) & B(2,1) & B(2,2) \end{array}\right\|$$

1D Cannon's Algorithm

The multiplications are performed by the processors:

$$\left\| A(*,0)^* B(0,0) \right\| A(*,1)^* B(1,1) \left\| A(*,2)^* B(2,2) \right\|$$

A after first shift - A is rotated right.

$$\left\| \begin{matrix} A(0,1) \\ A(1,1) \\ A(2,1) \end{matrix} \right\| \begin{matrix} A(0,2) \\ A(1,2) \\ A(2,2) \end{matrix} \left\| \begin{matrix} A(0,0) \\ A(1,0) \\ A(2,0) \end{matrix} \right\|$$

B is not needed to be rotated up. Submatrices multiplication after first shift of A:

$$\left\| A(*,1)^* B(1,0) \right\| A(*,2)^* B(2,1) \left\| A(*,0)^* B(0,2) \right\|$$

1D Cannon's Algorithm

The result on each processor:

$$\left\|\begin{array}{c|c|c} A(0,0)*B(0,0)+ & A(0,1)*B(1,1)+ & A(0,2)*B(2,2)+ \\ A(0,1)*B(1,0) & A(0,2)*B(2,1) & A(0,0)*B(0,2) \\ \hline A(1,0)*B(0,0)+ & A(1,1)*B(1,1)+ & A(1,2)*B(2,2)+ \\ A(1,1)*B(1,0) & A(1,2)*B(2,1) & A(1,0)*B(0,2) \\ \hline A(2,0)*B(0,0)+ & A(2,1)*B(1,1)+ & A(2,2)*B(2,2)+ \\ A(2,1)*B(1,0) & A(2,2)*B(2,1) & A(2,0)*B(0,2) \end{array}\right\|$$

$$= \left\|\begin{array}{c|c|c} A(*,0)^*\ B(0,0) & A(*,1)^*\ B(1,1) & A(*,2)^*\ B(2,2) \\ +A(*,1)^*\ B(1,0) & +\ A(*,2)^*\ B(2,1) & +A(*,0)^*\ B(0,2) \end{array}\right\|$$

1D Cannon's Algorithm

The matrix A after second shift - A is rotated right:

$$\left\| \begin{matrix} A(0,2) \\ A(1,2) \\ A(2,2) \end{matrix} \right\| \left\| \begin{matrix} A(0,0) \\ A(1,0) \\ A(2,0) \end{matrix} \right\| \left\| \begin{matrix} A(0,1) \\ A(1,1) \\ A(2,1) \end{matrix} \right\| = \left\| \; A(*,2) \; \right\| \; A(*,0) \; \left\| \; A(*,1) \; \right\|$$

B is not needed to be rotated up.

Submatrixes multiplication after second shift of A:

$$\left\| \; A(*,2)^* \; B(2,0) \; \right\| \; A(*,0)^* \; B(0,1) \; \left\| \; A(*,1)^* \; B(1,2) \; \right\|$$

1D Cannon's Algorithm

The result on each processor:

$A(0,0) * B(0,0) +$ $A(0,1) * B(1,0) +$ $A(0,2) * B(2,0)$	$A(0,1) * B(1,1) +$ $A(0,2) * B(2,1) +$ $A(0,0) * B(0,1)$	$A(0,2) * B(2,2) +$ $A(0,0) * B(0,2) +$ $A(0,1) * B(1,2)$
$A(1,0) * B(0,0) +$ $A(1,1) * B(1,0) +$ $A(1,2) * B(2,0)$	$A(1,1) * B(1,1) +$ $A(1,2) * B(2,1) +$ $A(1,0) * B(0,1)$	$A(1,2) * B(2,2) +$ $A(1,0) * B(0,2) +$ $A(1,1) * B(1,2)$
$A(2,0) * B(0,0) +$ $A(2,1) * B(1,0) +$ $A(2,2) * B(2,0)$	$A(2,1) * B(1,1) +$ $A(2,2) * B(2,1) +$ $A(2,0) * B(0,1)$	$A(2,2) * B(2,2) +$ $A(2,0) * B(0,2) +$ $A(2,1) * B(1,2)$

1D Cannon's Algorithm

Finally:

$$\left[\!\!\left[\begin{array}{c|c|c} A(0,*)*B(*,0) & A(0,*)*B(*,1) & A(0,*)*B(*,2) \\ \hline A(1,*)*B(*,0) & A(1,*)*B(*,1) & A(1,*)*B(*,2) \\ \hline A(2,*)*B(*,0) & A(2,*)*B(*,1) & A(2,*)*B(*,2) \end{array}\right]\!\!\right] =$$

$$\left[\!\!\left[A*B(*,0) \;\middle\|\; A*B(*,1) \;\middle\|\; A*B(*,2) \right]\!\!\right] = A*B$$

1D Cannon's Algorithm by rows

When the matrices A, B ,C are divided by rows, the matrix A is not rotated. The matrix B is rotated up.

The initial alignment on 1D layout by rows on 3 processors looks as follows:

$$\left[\begin{array}{ccc} A(0,0) & A(0,1) & A(0,2) \\ B(0,0) & B(0,1) & B(0,2) \\ \hline\hline A(1,0) & A(1,1) & A(1,2) \\ B(1,0) & B(1,1) & B(1,2) \\ \hline\hline A(2,0) & A(2,1) & A(2,2) \\ B(2,0) & B(2,1) & B(2,2) \end{array}\right]$$

1D Cannon's Algorithm by rows

The first multiplication:

$$\begin{bmatrix} A(0,0)*B(0,0) & A(0,0)*B(0,1) & A(0,0)*B(0,2) \\ \hline\hline A(1,1)*B(1,0) & A(1,1)*B(1,1) & A(1,1)*B(1,2) \\ \hline\hline A(2,2)*B(2,0) & A(2,2)*B(2,1) & A(2,2)*B(2,2) \end{bmatrix} =$$

$$\begin{bmatrix} A(0,0)*B(0,*) \\ \hline\hline A(1,1)*B(1,*) \\ \hline\hline A(2,2)*B(2,*) \end{bmatrix}$$

1D Cannon's Algorithm by rows

Second multiplication after the shifting of B:

$$\left[\begin{array}{ccc} A(0,1)*B(1,0) & A(0,1)*B(1,1) & A(0,1)*B(1,2) \\ \hline\hline A(1,2)*B(2,0) & A(1,2)*B(2,1) & A(1,2)*B(2,2) \\ \hline\hline A(2,0)*B(0,0) & A(2,0)*B(0,1) & A(2,0)*B(0,2) \end{array}\right] =$$

$$\left[\begin{array}{c} A(0,1)*B(1,*) \\ \hline\hline A(1,2)*B(2,*) \\ \hline\hline A(2,0)*B(0,*) \end{array}\right]$$

1D Cannon's Algorithm by rows

3-rd multiplication:

$$\left[\begin{array}{ccc} A(0,2)*B(2,0) & A(0,2)*B(2,1) & A(0,2)*B(2,2) \\ \hline\hline A(1,0)*B(0,0) & A(1,0)*B(0,1) & A(1,0)*B(0,2) \\ \hline\hline A(2,1)*B(1,0) & A(2,1)*B(1,1) & A(2,1)*B(1,2) \end{array}\right] =$$

$$\left[\begin{array}{c} A(0,2)*B(2,*) \\ \hline\hline A(1,0)*B(0,*) \\ \hline\hline A(2,1)*B(1,*) \end{array}\right]$$

1D Cannon's Algorithm by rows

Finally:

$$\left[\begin{array}{c} A(0,0)*B(0,*)+\ A(0,1)*B(1,*)\ +\ A(0,2)*B(2,*) \\ \hline\hline A(1,1)*B(1,*)+\ A(1,2)*B(2,*)\ +\ A(1,0)*B(0,*) \\ \hline\hline A(2,2)*B(2,*)+\ A(2,0)*B(0,*)+\ A(2,1)*B(1,*) \end{array}\right]$$

$$= \left[\begin{array}{c} A(0,*)*B \\ \hline\hline A(1,*)*B \\ \hline\hline A(2,*)*B \end{array}\right] = \text{A*B}$$

Strassen's Algorithm

We divide the matrices A,B,C on 4 blocks each. The formula for the multiplication is:

$$C(1,1) = A(1,1) * B(1,1) + A(1,2) * B(2,1)$$
$$C(1,2) = A(1,1) * B(1,2) + A(1,2) * B(2,2)$$
$$C(2,1) = A(2,1) * B(1,1) + A(2,2) * B(2,1)$$
$$C(2,2) = A(2,1) * B(1,2) + A(2,2) * B(2,2)$$

This involves 8 matrix multiplications (and 4 additions). Matrix multiplication requires n^3 floating point operations (flops), whereas matrix addition requires n^2 floating point operations (flops).

Strassen's Algorithm

This is a divide-and-conquer algorithm: $(2 * n) * (2 * n)$ matrix multiplication has been divided into $(8 * n * n)$ matrix multiplications (and $(4 * n * n)$ additions). Clearly each of these "divided" problems ($n * n$ matrix multiplications) itself could be further subdivided into $(8 * \frac{n}{2} * \frac{n}{2})$ matrix multiplications, and so on. For this particular approach to matrix-matrix multiplication there may be nothing to gain by posing the algorithm as a divide-and-conquer algorithm.

Strassen's Algorithm

However, if we define the intermediate matrixes P_i as follows:

$$P_1 = (A(1,1) + A(2,2)) * (B(1,1) + B(2,2))$$
$$P_2 = (A(2,1) + A(2,2)) * B(1,1)$$
$$P_3 = A(1,1) * (B(1,2) - B(2,2))$$
$$P_4 = A(2,2) * (B(2,1) - B(1,1))$$
$$P_5 = (A(1,1) + A(2,2)) * B(2,2)$$
$$P_6 = (A(2,1) - A(1,1)) * (B(1,1) + B(2,2))$$
$$P_7 = (A(1,2) - A(2,2)) * (B(2,1) + B(2,2))$$

Then we have:

$$C(1,1) = P_1 + P_4 - P_5 + P_7$$
$$C(1,2) = P_3 + P_5$$
$$C(2,1) = P_2 + P_4$$
$$C(2,2) = P_1 + P_3 - P_2 + P_6$$

This involves 7 matrix multiplications (and 18 matrix aditions).

Strassen's Algorithm

This is again divide-and-conquer algorithm, but now the $(2*n)*(2*n)$ matrix multiplication has been replaced by **7*n*n** matrix multiplications (and 18 matrix additions), which is worthwhile saving for large matrices The algorithm is applied recursively. So that the $7 * n * n$ matrix multiplications are replaced by $(49 * \frac{n}{2} * \frac{n}{2})$ matrix multiplications, and so on. The recursion is continued until the "divided" matrixes are sufficiently small that the standard multiplication algorithm is more efficient that this recursive approach. The algorithm was suggested by Strasses (1969). The algorithm is not as strongly stable (in the numerical sense) as the conventional algorithm, but it is sufficiently stable for many applications.

DNS Algorithm (Dekel, Nassimi, Sahni)

- The DNS algorithm based on partitioning intermediate data that can use up to n^3 processes and performs matrix multiplication in time $O(log(n))$ by using $\Omega(n^3/log(n))$ processes.
- Assume that n^3 processes are available for multiplying two $n \times n$ matrices. These processes are arranged in a three-dimensional $n \times n \times n$ logical array.
- Since the matrix multiplication algorithm performs n^3 scalar multiplications, each of the n^3 processes is assigned a single scalar multiplication.
- The processes are labeled according to their location in the array, and the multiplication $A[i,k] * B[k,j]$ is assigned to process $P[i,j,k]$ ($0 \leq i,j,k < n$).

DNS's Algorithm

- After each process performs a single multiplication, the contents of $P[i,j,0]$,$P[i,j,1]$,...,$P[i,j,n-1]$ are added to obtain $C[i,j]$.
- The addition for all $C[i,j]$ can be carried out simultaneously in

$$log(n)$$

steps each. Thus, it takes one step to multiply and $log(n)$ steps to add. It takes time

$$O(log(n))$$

to multiply the $n \times n$ matrix.

DNS's Algorithm

Initial distribution of A and B make the levels $3, 2, 1$ empty. Matrices are put at level 0:

Level $K = 0$:

$$\begin{array}{cccc} A_{0,0} & A_{1,0} & A_{2,0} & A_{3,0} \\ A_{1,0} & A_{1,1} & A_{1,1} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{array} \qquad \begin{array}{cccc} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{array}$$

By "..." we denote the memories of the processors, which are empty.

DNS's Algorithm

After moving $A[i,j]$ from $P[i,j,0]$ to $P[i,j,j]$ and $B[i,j]$ from $P[i,j,0]$ to $P[i,j,i]$:

Level $K = 0$:

$$\begin{array}{cccc} A_{0,0} & \ldots & \ldots & \ldots \\ A_{1,0} & \ldots & \ldots & \ldots \\ A_{2,0} & \ldots & \ldots & \ldots \\ A_{3,0} & \ldots & \ldots & \ldots \end{array} \qquad \begin{array}{cccc} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \end{array}$$

Level $K = 1$:

$$\begin{array}{cccc} \ldots & A_{0,1} & \ldots & \ldots \\ \ldots & A_{1,1} & \ldots & \ldots \\ \ldots & A_{2,1} & \ldots & \ldots \\ \ldots & A_{3,1} & \ldots & \ldots \end{array} \qquad \begin{array}{cccc} \ldots & \ldots & \ldots & \ldots \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \end{array}$$

DNS's Algorithm

Level $K = 2$:

$$\begin{array}{cccc|cccc} \ldots & \ldots & A_{0,2} & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & A_{1,2} & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & A_{2,2} & \ldots & B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ \ldots & \ldots & A_{3,2} & \ldots & \ldots & \ldots & \ldots & \ldots \end{array}$$

Level $K = 3$:

$$\begin{array}{ccc|cccccc} \ldots & \ldots & \ldots & A_{0,3} & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & A_{1,3} & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & A_{2,3} & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & A_{3,3} & B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{array}$$

DNS's Algorithm

After broadcasting $A[i,j]$ along j axis and $B[i,j]$ along i axis:

Level $K = 0$:

$$\begin{array}{cccc|cccc} A_{0,0} & A_{0,0} & A_{0,0} & A_{0,0} & B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ A_{1,0} & A_{1,0} & A_{1,0} & A_{1,0} & B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ A_{2,0} & A_{2,0} & A_{2,0} & A_{2,0} & B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ A_{3,0} & A_{3,0} & A_{3,0} & A_{3,0} & B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \end{array}$$

Level $K = 1$:

$$\begin{array}{cccc|cccc} A_{0,1} & A_{0,1} & A_{0,1} & A_{0,1} & B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ A_{1,1} & A_{1,1} & A_{1,1} & A_{1,1} & B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ A_{2,1} & A_{2,1} & A_{2,1} & A_{2,1} & B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ A_{3,1} & A_{3,1} & A_{3,1} & A_{3,1} & B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \end{array}$$

DNS's Algorithm

Level $K = 2$:

$$\begin{array}{cccc} A_{0,2} & A_{0,2} & A_{0,2} & A_{0,2} \\ A_{1,2} & A_{1,2} & A_{1,2} & A_{1,2} \\ A_{2,2} & A_{2,2} & A_{2,2} & A_{2,2} \\ A_{3,2} & A_{3,2} & A_{3,2} & A_{3,2} \end{array} \qquad \begin{array}{cccc} B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \end{array}$$

Level $K = 3$:

$$\begin{array}{cccc} A_{0,3} & A_{0,3} & A_{0,3} & A_{0,3} \\ A_{1,3} & A_{1,3} & A_{1,3} & A_{1,3} \\ A_{2,3} & A_{2,3} & A_{2,3} & A_{2,3} \\ A_{3,3} & A_{3,3} & A_{3,3} & A_{3,3} \end{array} \qquad \begin{array}{cccc} B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{array}$$

DNS's Algorithm

The vertical column of processes $P[i, j, *]$ computes the dot product of row $A[i, *]$ and column $B[*, j]$.

The DNS algorithm has three main communication steps:

(1) moving the columns of A and the rows of B to their respective places,

(2) performing one-to-all broadcast along the j axis for A and along the i axis for B,

(3) all-to-one reduction along the k axis.

Final remarks

Practical Parallel Software for matrix multiplication exists:

- PBLAS: Parallel Basic Linear Algebra Subroutines,
- PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers,
- SUMMA: Scalable Universal Matrix Multiplication Algorithms,
- BLACS,
- BLAS.

Thank you for your attention!

Any questions are welcome.