

A Variety of Parallel Matrix Multiplication Algorithms for Teaching Parallel Algorithms and Programming

Peter Sobe

Dresden University of Applied Sciences
Faculty of Computer Science and Mathematics

32th PARS Workshop, Innsbruck, September 2026

Contents

- 1 Motivation
- 2 Algorithms
- 3 Comparison
- 4 Summary

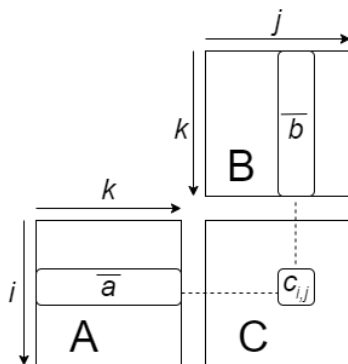
Motivation

Needed for Teaching:

- Systematic comparison of different parallel version of matrix multiplication
- Relations to concepts of parallel computing
- Addressing methods for parallelization
- Using common criteria to compare algorithms

Matrix Multiplication

Principle:



```
for i:=0 to n-1
  for j:=0 to n-1
    c[i,j]:=0
    for k:=0 to n-1
      c[i,j] := c[i,j] +
        a[j,k] * b[k,i]
    endfor
  endfor
endfor
}
```

Parallel Algorithms

Shared, global memory:

- PRAM algorithm
- OpenMP (original, sequential code with OpenMP pragma)

Distributed memory (MPI):

- One-dimensional and two-dimensional partitioning
- Cannon Algorithm

Parallel Algorithms: PRAM, global memory

PRAM: hypothetical Architecture for algorithm design and evaluation

- as many processors as algorithm can utilize,
- global memory,
- step-synchronized execution and memory access

```
1: C' (i, j, l) := A(i, l) * B(l, j)
2: for h=1 to ld n do
    if (l<=n/pow2(h)) then
        C' (i, j, l) := C' (i, j, 2l-1) + C' (i, j, 2l)
3: if (l=1) then C(i, j) := C' (i, j, l)
```

Number of processors: $p = n^3$

Speedup: $S = \frac{T_{seq}}{T_{par}} = \frac{2 \cdot n^3}{2+ld\ n}$

Efficiency: $E = \frac{S}{p} = \frac{2 \cdot n^3}{(2+ld\ n) \cdot n^3} = \frac{2}{2+ld\ n}$

Relative memory overhead: $M = \frac{3 \cdot n^2 + (n^3 - n^2)}{3 \cdot n^2} = \frac{2 \cdot n^2 + n^3}{3 \cdot n^2} = \frac{2+n}{3}$

OpenMP, shared memory

OpenMP: induces multithreading regions, controlled by pragma directives in the code

```
void mult_function_omp( double** a, double** b,
                        double** c, int n)
{
#pragma omp parallel for
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
      { c[i][j]=0.0;
        for (int k=0;k<n;k++)
          c[i][j] += a[i][k] * b[k][j];
      }
}
```

Number of processors: $p \leq n$

Speedup and Efficiency: $S = p$, $E = 1$ (ideal conditions)

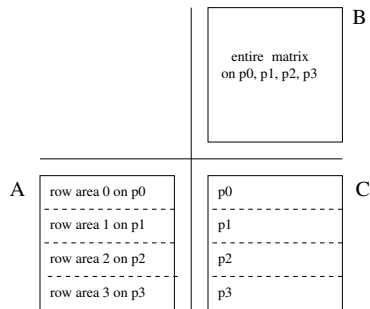
Relative memory overhead: $M = 1$

MPI, Partitioning on distributed memory

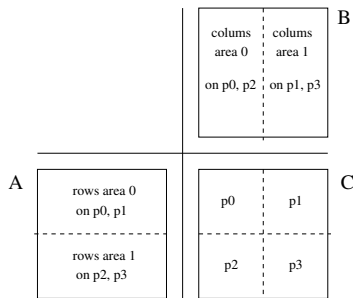
Two options:

- one-dimensional partitioning, either along rows or along columns
- two-dimensional partitioning

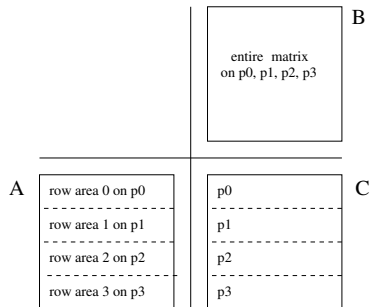
one-dimensional



two-dimensional



MPI, One-dimensional partitioning



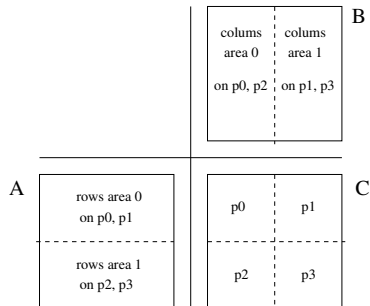
```
void mult_1d_part(int start_row, int end_row,
                  double** a_region, double** b,
                  double** c_region, int n)
{
    int i, j, k;
    for (i=0; i<end_row-start_row; i++)
        for (j=0; j<n; j++)
        { c_region[i][j]=0.0;
          for (k=0; k<n; k++)
              c_region[i][j] += a_region[i][k] * b[k][j];
        }
}
```

Number of processors: $p \leq n$

Speedup and Efficiency: $S = p$, $E = 1$ (ideal conditions)

Relative memory overhead: $M = \frac{2+p}{3}$

MPI, Two-dimensional Partitioning



```
void mult_2d_part(int start_row, int end_row,
                  int start_column, int end_column,
                  double** a_region, double** b_region,
                  double** c_tile, int n)
{
    int i, j, k;
    for (i=0; i<end_row-start_row; i++)
        for (j=0; j<end_column-start_column; j++)
        {
            c_tile[i][j]=0.0;
            for (k=0; k<n; k++)
                c_tile[i][j] += a_region[i][k] *
                                b_region[k][j];
        }
}
```

Number of processors: $p \leq n^2$

Speedup and Efficiency: $S = p$, $E = 1$ (ideal conditions)

Relative memory overhead: $M = \frac{1+2\sqrt{p}}{3}$

MPI, Cannon Algorithm on distributed memory

The Cannon algorithm was developed for a cellular system in 1969

Principle:

- Matrices are initially distributed element-wise on a mesh of processing units
- alternating calculations and shift operation allow to obtain the scalar products for each $c_{i,j}$ element at mesh position i,j

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

MPI, Cannon Algorithm on distributed memory

The Cannon algorithm was developed for a cellular system in 1969

Principle:

- Matrices are initially distributed element-wise on a mesh of processing units
- alternating calculations and shift operation allow to obtain the scalar products for each $c_{i,j}$ element at mesh position i, j

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

MPI, Cannon Algorithm on distributed memory

The Cannon algorithm was developed for a cellular system in 1969

Principle:

- Matrices are initially distributed element-wise on a mesh of processing units
- alternating calculations and shift operation allow to obtain the scalar products for each $c_{i,j}$ element at mesh position i,j

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

MPI, Cannon Algorithm on distributed memory

The Cannon algorithm was developed for a cellular system in 1969

Principle:

- Matrices are initially distributed element-wise on a mesh of processing units
- alternating calculations and shift operation allow to obtain the scalar products for each $c_{i,j}$ element at mesh position i,j

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

MPI, Cannon Algorithm on distributed memory

The Cannon algorithm was developed for a cellular system in 1969

Principle:

- Matrices are initially distributed element-wise on a mesh of processing units
- alternating calculations and shift operation allow to obtain the scalar products for each $c_{i,j}$ element at mesh position i, j

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

MPI, Cannon Algorithm on distributed memory

setup phase:

start with the initial tile-wise distribution

(1.) move elements in A to start position: shift i -th row i elements in left direction

(2.) move elements in B to start position: shift j -th column j elements upwards

calculation phase:

On processor i, j :

set $c=0$

For $k=0$ to $n-1$ do

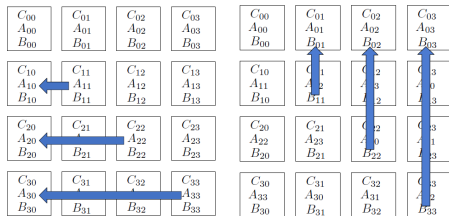
in each step k arrange

$a = A[i, (i+j+k)]$, $b = B[(i+j+k), j]$ to be on processor i, j

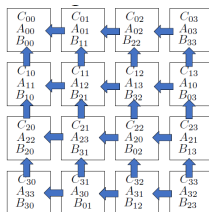
$c = c + a \cdot b$

Set $C[i, j] = c$

setup phase

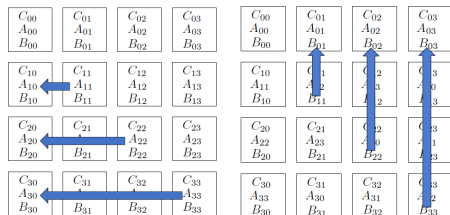


calculation phase

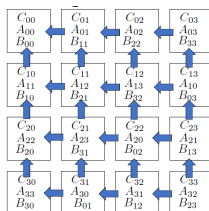


MPI, Cannon Algorithm on distributed memory

setup phase



calculation phase



Number of processors: $p \leq n^2$

Speedup and Efficiency: S near p , E near 1 (ideal conditions)

Relative memory overhead: $M = 1$

Comparison: Estimations

Estimated properties compared.

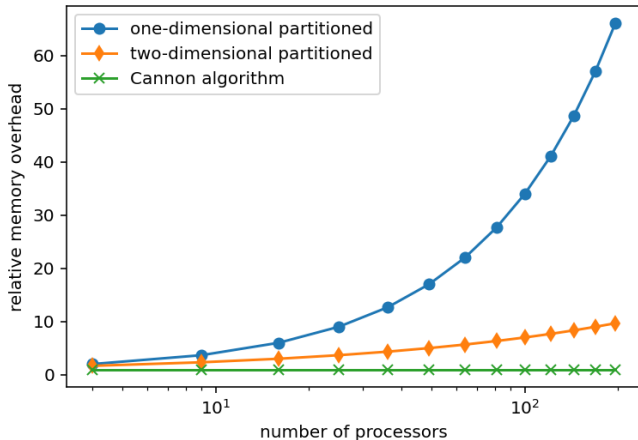
p : the number of processors

n : for the problem size, assuming multiplication of $n \times n$ matrices.

	PRAM	shared mem. OpenMP	distributed memory		
			one-dim. partitioning	two-dim. partitioning	Cannon algorithm
Processors utilized	$p = n^3$	$p \leq n$ (typical)	$p \leq n$ (limited)	$p \leq n^2$ (limited)	$p \leq n^2$ (limited)
Speedup	$\frac{2n^3}{2+d \cdot n}$	p	p	p	<i>near</i> p
Efficiency	$\frac{2}{(2+d \cdot n)}$	1	1	1	<i>near</i> 1
Relative memory overhead	$\frac{2+n}{3}$	1	$\frac{2+p}{3}$	$\frac{1+2\sqrt{p}}{3}$	1

Comparison: Estimations

Relative memory overhead for different processor numbers (p).
Only square p numbers are provided, a requirement for the Cannon algorithm.



Note the logarithmic x axis.

Comparison: Performance Analysis

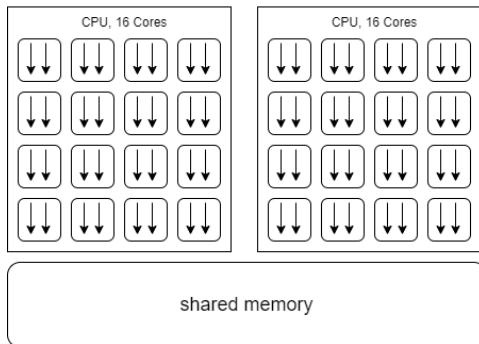
System for performance comparison:



Core Intel XEON
Silver 4314 CPU @ 2.4 GHz
2x SMT

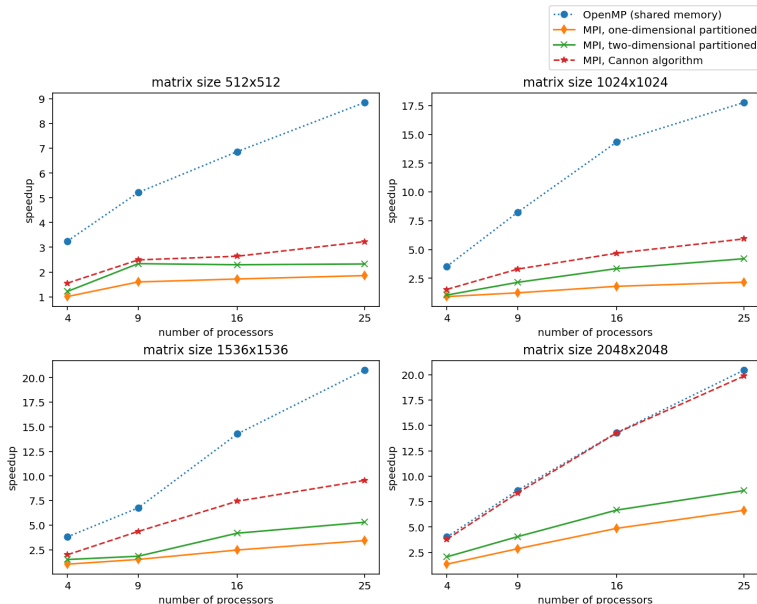
DELL PowerEdge R450 Rack
Intel XEON, 2 CPU sockets,
shared memory,
32 cores, SMT factor 2

Ubuntu with gcc 13.3.0
and OpenMPI 4.1.6.



Comparison: Performance Analysis

Speedup reached by different algorithms with different problem size.



Comparison: Performance Analysis

conclusions from performance analysis:

- shared memory and OpenMP provides the best improvement by parallelization.
- data transfer for a distributed memory implementation has high cost and limits the gain by parallelization.
- a two-dimensional partitioning is beneficial over the one-dimensional in terms of lower memory overhead.
Accordingly, better performance was observed.
- among the distributed memory variants, the Cannon algorithm performs best.

Summary

- 5 algorithms that are well known and dedicated to different systems
- Comparison of calculated properties:
Utilized processors, Speedup, Efficiency, Memory overhead
 - High memory overhead on distributed systems, except Cannon algorithm
 - Two-dimensional partitioning better than one-dimensional
 - PRAM: lower speedup in relation to p , but profits from huge p -values
- Performance evaluation (Speedup) with 4 algorithms, excluded PRAM algorithm
 - Top 1: OpenMP on shared memory
 - Cannon algorithm better than 1-dim and 2-dim partitioned variants
- Lessons to learn (for teaching):
 - on shared memory the basis form of matrix multiplication is a good choice
 - on distributed memory architectures the basis form, particularly one-dim. partitioning is a bad choice. Use two-dim. partitioning or Cannon algorithm!