Algorithms for Dense Matrices, Vectors, and Arrays

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu
Topics for Today

• Matrix-vector multiplication
  — 1D and 2D partitionings

• Matrix-matrix multiplication
  — 2D partitioning and Cannon’s algorithm
  — 2.5D matrix multiplication

• Exotic distributions for dense arrays
  — generalized multipartitioning for line sweep computations
Matrix Algorithms

- Regular structure
- Well-suited to static partitioning of computation
- Computation typically partitioned based on the data
  - input, output, intermediate
- Typical data partitionings for matrix algorithms
  - 1D and 2D block, cyclic, block-cyclic
- Exotic partitionings with special properties
  - multipartitioning
Matrix-Vector Multiplication

• Problem:
  — multiply a dense $n \times n$ matrix $A$ with an $n \times 1$ vector $x$
  — yield the $n \times 1$ result vector $y$

• How many multiplications and additions in a serial implementation?
  • Serial algorithm requires $n^2$ multiplications and additions
First, consider simple case: $p = n$

- $p \times p$ matrix is partitioned among $p$ processors
  - each processor stores a complete row of the matrix
- $p \times 1$ vector $x$ is also partitioned
  - each process owns one element of $x$

- Given this distribution, how do we proceed with the multiplication?
Computation

- Use all-to-all broadcast to distribute all of $x$ to each process

\[ y[i] = \sum_{j=0}^{p-1} A[i,j] \times x[j] \]

- Process $P_i$ computes

\[ y[i] = \sum_{j=0}^{p-1} A[i,j] \times x[j] \]
Matrix-Vector Multiplication: 1D Partitioning

Analysis for $p = n$

- All-to-all broadcast: $\Theta(p) = \Theta(n)$ time
  —each process must receive vector $x$, which is of length $n$
- Computation of $y[i]$: $\Theta(p) = \Theta(n)$ time
  —each process performs dot product of length $n$
- Overall algorithm: $\Theta(p) = \Theta(n)$ time
  —both communication and computation are $\Theta(n)$
- Total parallel work = $\Theta(p^2) = \Theta(n^2)$
Consider $p < n$

- Initially, each process stores
  - $n/p$ complete rows of the matrix $A$
  - $n/p$ elements of the vector $x$
- Distribute all of vector to each process
  - all-to-all broadcast among $p$ processes, messages of size $n/p$
- On each processor: $n/p$ local dot products on rows of length $n$

- Parallel run time
  $$T_p = \frac{n^2}{p} + t_s \log p + t_w n.$$  
  (for large $p$ on hypercube)
Matrix-Vector Multiplication: 2D Partitioning

Consider when $p = n^2$

- $n \times n$ matrix $A$ is partitioned among $n^2$ processors
  —each processor owns a single element
- $n \times 1$ vector $x$ is distributed in last column of $n$ processors
- What is a good algorithm given this distribution?
Matrix-Vector Multiplication: 2D Partitioning

Computation sketch

• Align the vector with the matrix
  — first communication step
    – align the vector $x$ along the main diagonal of the matrix
  — second communication step
    – copy vector element from each diagonal process to all processes in its corresponding column
      use $n$ independent broadcasts
      one among all processors in each column

• Compute the result vector
  — perform all-to-one reduction along the rows
Matrix-Vector Multiplication: 2-D Partitioning

For the one-element-per-process case, \( p = n^2 \) if matrix size is \( n \times n \)
Matrix-Vector Multiplication: 2-D Partitioning

Analysis when $p = n^2$

- Three communication operations used in this algorithm
  - one-to-one communication: align vector along the main diagonal
  - one-to-all broadcast: vector element to $n$ processes in column
  - all-to-one reduction in each row

- Time complexity of communication
  - the one-to-all and all-to-one operations take $\Theta(\log n)$ time

- Parallel time is $\Theta(\log n)$

- The cost (process-time product) is $\Theta(n^2 \log n)$ (recall $p = n^2$)
Matrix-Matrix Multiplication

- **Compute** $C = A \times B$ for $A, B, C$ $n \times n$ dense, square matrices

- **Serial complexity is** $O(n^3)$
  - we don’t explicitly consider better serial algorithms (e.g., Strassen)
    - they can be used as serial kernels in the parallel algorithms

- **Block operations are a useful concept**
  - an $n \times n$ matrix $A$ can be regarded as a $q \times q$ array of blocks
  - $A_{i,j}$ $(0 \leq i, j < q)$, each block is an $(n/q) \times (n/q)$ submatrix

- **Blocked computation**
  - $q^3$ matrix multiplications, each using $(n/q) \times (n/q)$ matrices
Matrix-Matrix Multiplication

Consider two $n \times n$ matrices $A$ and $B$ partitioned into $p$ blocks

- $A_{i,j}$ and $B_{i,j}$ ($0 \leq i, j < \sqrt{p}$) of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each

- Process $P_{i,j}$
  - initially stores $A_{i,j}$ and $B_{i,j}$
  - computes block $C_{i,j}$ of the result matrix.

- Computing submatrix $C_{i,j}$
  - requires all submatrices $A_{i,k}$ and $B_{k,j}$ for $0 \leq k < \sqrt{p}$
Matrix Multiplication

Consider data needed for output matrix block shown in purple.
Matrix-Matrix Multiplication

- **All-to-all broadcast**
  - blocks of $A$ along rows
  - blocks of $B$ along columns

- Perform local submatrix multiplication

- **Two all-to-all broadcasts take time**
  \[ 2(t_s \log(\sqrt{p}) + t_w(n^2/p)(\sqrt{p} - 1)) \]

- **Computation requires $\sqrt{p}$ multiplications of**
  \( (n/\sqrt{p}) \times (n/\sqrt{p}) \) submatrices

- **Parallel run time**
  \[ T_P = n^3/p + 2t_s \log \sqrt{p} + 2t_w(n^2/p)(\sqrt{p} - 1) \]

- **Major drawback of the algorithm: memory requirements**
  - each process needs space for a block of rows and a block of columns
Memory efficient algorithm idea

• Approach
  —schedule the computations of the $\sqrt{p}$ processes of each row
  —at any given time, each process is using different blocks

• Systematically rotate blocks among the processes
  —rotate after every submatrix multiplication
  —each process get fresh blocks in each rotation
Matrix-Matrix Multiplication: Cannon's Algorithm

Aim: align the blocks of $A$ and $B$ so that each process multiplies its local submatrices

- Initial alignment
  - shift all submatrices $A_{i,j}$ to the left (with wraparound) by $i$ steps
  - shift all submatrices $B_{i,j}$ up (with wraparound) by $j$ steps

- Perform local block multiplication

- Rotate
  - move each block of $A$ one step left (with wraparound)
  - move each block of $B$ moves one step up (with wraparound)

- Perform next block multiplication

- Add to partial result

- Repeat until all $\sqrt{p}$ blocks have been multiplied.
Cannon’s Matrix Multiplication

Initial State

\[ \begin{array}{c}
\text{Initial State}
\end{array} \]
Cannon’s Matrix Multiplication

Perform Alignment

shift $A_{ij}$ left by $l$

shift $B_{ij}$ up by $j$

Alignment step 0
Cannon’s Matrix Multiplication

Perform Alignment

- Shift $A_{ij}$ left by $l$
- Shift $B_{ij}$ up by $j$

Alignment step 1

\[ \times \]
Cannon’s Matrix Multiplication

Perform Alignment

Shift $A_{ij}$ left by $l$

Shift $B_{ij}$ up by $j$

Alignment step 2

$\times$
Cannon’s Matrix Multiplication

Perform Alignment

shift $A_{ij}$ left by $l$

shift $B_{ij}$ up by $j$

Alignment step 3
Cannon’s Matrix Multiplication

Perform Multiplication; then

shift \( A_{ij} \) left by 1

shift \( B_{ij} \) up by 1

Multiplication step 1
Cannon’s Matrix Multiplication

Perform Multiplication; then

- shift $A_{ij}$ left by 1
- shift $B_{ij}$ up by 1

Multiplication step 2
Cannon’s Matrix Multiplication

Perform Multiplication; then
shift $A_{ij}$ left by 1
shift $B_{ij}$ up by 1

Multiplication step 3
Cannon’s Matrix Multiplication

Perform Multiplication; then

shift $A_{ij}$ left by 1

shift $B_{ij}$ up by 1
Cannon’s Matrix Multiplication

Perform Multiplication; then

shift $A_{ij}$ left by 1

shift $B_{ij}$ up by 1

Multiplication step 5
Cannon’s Matrix Multiplication

Perform Multiplication; then

shift $A_{ij}$ left by 1

shift $B_{ij}$ up by 1

Multiplication step 6
Matrix-Matrix Multiplication: Cannon's Algorithm

- **Alignment step**
  - maximum distance over which a block shifts is \( \sqrt{p} - 1 \)
  - two shift operations require a total of \( 2(t_s + t_w n^2 / p) \) time.

- **Compute-shift phase**
  - \( \sqrt{p} \) single-step shifts
  - each shift takes \( t_s + t_w n^2 / p \) time

- **Computation time**
  - multiplying \( \sqrt{p} \) matrices of size \( (n/\sqrt{p}) \times (n/\sqrt{p}) \) is \( n^3 / p \)

- The parallel time is approximately
  \[
  T_P = \frac{n^3}{p} + 2\sqrt{p} t_s + 2t_w \frac{n^2}{\sqrt{p}}.
  \]

- More messages, but Cannon’s is memory optimal
Cannon’s Algorithm in Use


This paper provides an overview of a program synthesis system for a class of quantum chemistry computations. These computations are expressible as a set of tensor contractions and arise in electronic structure modeling. The input to the system is a high-level specification of the computation, from which the system can synthesize high-performance parallel code tailored to the characteristics of the target architecture. Several components of the synthesis system are described, focusing on performance optimization issues that they address.

Keywords—Communication minimization, compiler optimizations, data locality optimization, domain-specific languages, high-level programming languages, memory-constrained optimization, tensor contraction expressions
3D Matrix Multiplication

- What if a processor can hold a larger block of the matrix?

\[
\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}} \longrightarrow \frac{n}{3\sqrt{p}} \times \frac{n}{3\sqrt{p}}
\]

Each point on the cube is a processor.

Each green circle represents a block of the A matrix of size:

\[
\frac{n}{3\sqrt{p}} \times \frac{n}{3\sqrt{p}}
\]
2.5D Matrix Multiplication

- What if we do not have as many processors/memory?

\[
\left(\frac{n}{\sqrt[3]{p}}\right) \times \left(\frac{n}{\sqrt[3]{p}}\right) \rightarrow \frac{n}{\sqrt{p/c}} \times \frac{n}{\sqrt{p/c}}
\]

Each point on the cube is a processor.

Each green circle represents a block of the A matrix of size:

\[
\frac{n}{\sqrt{p/c}} \times \frac{n}{\sqrt{p/c}}
\]
**2.5D Matrix Multiplication Algorithm**

**Algorithm 2:** $[C] = 2.5D$-matrix-multiply($A,B,n,p,c$)

**Input:** square $n$-by-$n$ matrices $A$, $B$ distributed so that $P_{ij0}$ owns $\frac{n}{\sqrt{p/c}}$-by-$\frac{n}{\sqrt{p/c}}$ blocks $A_{ij}$ and $B_{ij}$ for each $i,j$

**Output:** square $n$-by-$n$ matrix $C = A \cdot B$ distributed so that $P_{ij0}$ owns $\frac{n}{\sqrt{p/c}}$-by-$\frac{n}{\sqrt{p/c}}$ block $C_{ij}$ for each $i,j$

```plaintext
/* do in parallel with all processors */
forall $i,j \in \{0,1,...,\sqrt{p/c} - 1\}, k \in \{0,1,...,c - 1\}$ do
    $P_{ij0}$ broadcasts $A_{ij}$ and $B_{ij}$ to all $P_{ijk}$
    $s := \text{mod} (j - i + k\sqrt{p/c^3}, \sqrt{p/c})$
    $P_{ijk}$ sends $A_{ij}$ to $A_{local}$ on $P_{isk}$
    $s' := \text{mod} (i - j + k\sqrt{p/c^3}, \sqrt{p/c})$
    $P_{ijk}$ sends $B_{ij}$ to $B_{local}$ on $P_{s'jk}$
    $C_{ijk} := A_{local} \cdot B_{local}$
    $s := \text{mod} (j + 1, \sqrt{p/c})$
    $s' := \text{mod} (i + 1, \sqrt{p/c})$
    for $t = 1$ to $\sqrt{p/c^3} - 1$ do
        $P_{ijk}$ sends $A_{local}$ to $P_{isk}$
        $P_{ijk}$ sends $B_{local}$ to $P_{s'jk}$
        $C_{ijk} := C_{ijk} + A_{local} \cdot B_{local}$
    end
end
```

$P_{ijk}$ contributes $C_{ijk}$ to a sum-reduction to $P_{ij0}$
Further, the cost of sacrificing flops for latency is large. Namely, if we see that the algorithmic costs are large, it is best to pick all block factorsizations of these blocks are on the critical path and must be done in strict sequence.

Given this dependency path (shown in Figure 2), we can lower bound the complexity of the algorithm by counting the complexity along this path. The latency cost is a function of the blocks be such algorithm must compute a sequence of diagonal blocks are on the critical path and must be done in strict sequence.

We now lower bound the communication cost for any algorithm that follows the above restrictions. Any algorithm must compute a sequence of diagonal blocks such that one block must be factorized sequentially such that such block can be factorized (it can be updated but Gaussian elimination cannot start).

Given a parallel LU factorization algorithm, we assume the algorithm must uphold the following properties:

- Factorize sequentially such that blocks must be factorized (it can be updated but Gaussian elimination cannot start).
- Communicate with the back plane of the 3D cube.

4 2.5D LU communication lower bound

We argue that for Gaussian-elimination style LU algorithms that achieve the bandwidth lower bound, the latency lower bound is actually much higher, namely 2.5D-matrix-multiply(a, b).

/* do in parallel with all processors */
forall i, j ∈ {0, 1, ..., √p/c - 1}, k ∈ {0, 1, ..., c - 1} do

input matrices A, B distributed so that P_{i0} owns n√p/c - by- n√p/c blocks A_{ij} and B_{ij} for each i, j

output: square n-by-n matrix C = A · B distributed so that P_{i0} owns n√p/c - by- n√p/c block C_{ij} for each i, j

Pass my blocks to the back planes of the cube (3D matrix multiplication step)

/* initial circular shift on B */

/* rightwards circular shift on A */
/* downwards circular shift on B */

end
2.5D Matrix Multiplication Algorithm

Algorithm 2: \([C] = 2.5\text{D}-\text{matrix-multiply}(A,B,n,p,c)\)

**Input:** square \(n\)-by-\(n\) matrices \(A, B\) distributed so that \(P_{ij0}\) owns \(\frac{n}{p/c}\text{-by-}\frac{n}{p/c}\) blocks \(A_{ij}\) and \(B_{ij}\) for each \(i, j\)

**Output:** square \(n\)-by-\(n\) matrix \(C = A \cdot B\) distributed so that \(P_{ij0}\) owns \(\frac{n}{p/c}\text{-by-}\frac{n}{p/c}\) block \(C_{ij}\) for each \(i, j\)

```plaintext
/* do in parallel with all processors */
forall \(i, j \in \{0, 1, ..., \sqrt{p/c} - 1\}, k \in \{0, 1, ..., c - 1\}\) do
    \(P_{ij0}\) broadcasts \(A_{ij}\) and \(B_{ij}\) to all \(P_{ijk}\)
    \(s := \text{mod} (j - i + k\sqrt{p/c^3}, \sqrt{p/c})\)
    \(P_{ijk}\) sends \(A_{ij}\) to \(A_{\text{local}}\) on \(P_{isk}\)
    \(s' := \text{mod} (i - j + k\sqrt{p/c^3}, \sqrt{p/c})\)
    \(P_{ijk}\) sends \(B_{ij}\) to \(B_{\text{local}}\) on \(P_{s'jk}\)
    \(C_{ijk} := A_{\text{local}} \cdot B_{\text{local}}\)
    \(s := \text{mod} (j + 1, \sqrt{p/c})\)
    \(s' := \text{mod} (i + 1, \sqrt{p/c})\)
    for \(t = 1\) to \(\sqrt{p/c^3} - 1\) do
        \(P_{ijk}\) sends \(A_{\text{local}}\) to \(P_{isk}\)
        \(P_{ijk}\) sends \(B_{\text{local}}\) to \(P_{s'jk}\)
        \(C_{ijk} := C_{ijk} + A_{\text{local}} \cdot B_{\text{local}}\)
    end
    \(P_{ijk}\) contributes \(C_{ijk}\) to a sum-reduction to \(P_{ij0}\)
end
```

Cannon’s Equivalent Alignment Step

/* replicate input matrices */
/* rightwards circular shift on A */
/* downwards circular shift on B */
Given a parallel LU factorization algorithm, we assume the algorithm must uphold the following properties:

We argue that for Gaussian-elimination style LU algorithms that achieve the bandwidth lower bound, the 2.5D LU communication lower bound

\[ \begin{align*}
&\text{Algorithm 2: } [C] = 2.5\text{D-matrix-multiply}(A,B,n,p,c) \\
&\text{Input: square } n\text{-by-}n \text{ matrices } A, B \text{ distributed so that } P_{ij0} \text{ owns } \frac{n}{\sqrt{p/c}}\text{-by-}\frac{n}{\sqrt{p/c}} \text{ blocks } A_{ij} \text{ and } B_{ij} \text{ for each } i, j \\
&\text{Output: square } n\text{-by-}n \text{ matrix } C = A \cdot B \text{ distributed so that } P_{ij0} \text{ owns } \frac{n}{\sqrt{p/c}}\text{-by-}\frac{n}{\sqrt{p/c}} \text{ block } C_{ij} \text{ for each } i, j \\
\end{align*} \]

/* do in parallel with all processors */

forall \( i, j \in \{0, 1, ..., \sqrt{p/c} - 1\} \), \( k \in \{0, 1, ..., c - 1\} \) do

\( P_{ij0} \) broadcasts \( A_{ij} \) and \( B_{ij} \) to all \( P_{ijk} \)

\( s := \text{mod} \ (j - i + k\sqrt{p/c^3}, \sqrt{p/c}) \)

\( P_{ijk} \) sends \( A_{ij} \) to \( A_{\text{local}} \) on \( P_{isk} \)

\( s' := \text{mod} \ (i - j + k\sqrt{p/c^3}, \sqrt{p/c}) \)

\( P_{ijk} \) sends \( B_{ij} \) to \( B_{\text{local}} \) on \( P_{s'jk} \)

\( C_{ijk} := A_{\text{local}} \cdot B_{\text{local}} \)

\( s := \text{mod} \ (j + 1, \sqrt{p/c}) \)

\( s' := \text{mod} \ (i + 1, \sqrt{p/c}) \)

for \( t = 1 \) to \( \sqrt{p/c^3} - 1 \) do

\( P_{ijk} \) sends \( A_{\text{local}} \) to \( P_{isk} \)

\( P_{ijk} \) sends \( B_{\text{local}} \) to \( P_{s'jk} \)

\( C_{ijk} := C_{ijk} + A_{\text{local}} \cdot B_{\text{local}} \)

end

\( P_{ijk} \) contributes \( C_{ijk} \) to a sum-reduction to \( P_{ij0} \)

end

\[ \text{Cannon’s Equivalent Multiply and Shift Step} \]

---

2.5D Matrix Multiplication Algorithm
Matrix Multiplication Costs

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Bandwidth (W)</th>
<th>Latency (S)</th>
<th>Memory per processor (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cannon’s MM</td>
<td>$\theta(n^2/\sqrt{p})$</td>
<td>$\theta(\sqrt{p})$</td>
<td>$\approx 3n^2/p$</td>
</tr>
<tr>
<td>3D MM</td>
<td>$\theta(n^2/p^{2/3})$</td>
<td>$\theta(\log p)$</td>
<td>$\approx 3n^2/p^{2/3}$</td>
</tr>
<tr>
<td>2.5D MM</td>
<td>$\Omega(n^2/\sqrt{cp})$</td>
<td>$\theta(\sqrt{p}/\sqrt{c^3})$</td>
<td>$\approx 3cn^2/p$</td>
</tr>
</tbody>
</table>

$W =$ number of words sent or received along critical path
$S =$ number of messages sent or received along critical path

2.5dMM
Reduces bandwidth cost by $\sqrt{c}$ compared to Cannon’s algorithm
Reduces latency cost by $\sqrt{c^3}$ compared to Cannon’s algorithm

*matrix size: $n \times n$, number of processors: $p$
2.5D Matrix Multiplication Algorithm

Speed up for best choice of $c$ in comparison to Cannon’s algorithm.

“On 16,384 nodes of BG/P, our 2.5D algorithm multiplies a small square matrix ($n = 8192$), 2.6X faster than Cannon's algorithm.”

Figure Credits: Technical Report No. UCB/EECS-2011-10, Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms.
Linear Algebra for Multicore

- Multicore is a disruptive technology for software
- Must rethink and rewrite applications, algorithms and software
  — as before with cluster computing and message passing
- Numerical libraries need to change
- Recent research
  — event-driven DAG scheduled computations
    - direct solvers (LU) on distributed memory using UPC
    - PLASMA software framework dense linear algebra for multicore
PLASMA: Parallel Linear Algebra for Multicore

• Objectives
  — parallel performance
    – high utilization of each core
    – scaling to large numbers of cores
  — any memory model
    – shared memory: symmetric or non-symmetric
    – distributed memory
    – GPUs

• Solution properties
  — asynchrony: avoid fork-join (bulk synchronous design)
  — dynamic scheduling: out-of-order execution
  — fine granularity: independent block operations
  — locality of reference: store data using block data layout

A community effort led by Tennessee and Berkeley
Computations as DAGs

Reorganize algorithms and software to work on tiles that are scheduled based on the directed acyclic graph of the computation.

Cholesky
4 x 4

QR
4 x 4
Cholesky using PLASMA

PLASMA
Arbitrary DAG
Fully dynamic scheduling

Nested fork-join parallelism (e.g., Cilk, TBB)
PLASMA Provides Highest Performance
## Over 40 Years of Dense Matrix Algorithms

<table>
<thead>
<tr>
<th>Approach</th>
<th>Reduce Communication</th>
<th>Length of Critical Path</th>
<th>Memory Footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cannon 1969</td>
<td>communication intensive</td>
<td>(\sqrt{p}) shifts</td>
<td>optimal</td>
</tr>
<tr>
<td>Multipartitioning 2002</td>
<td>“just enough cuts” for balanced parallelism without excessive msg volume</td>
<td>full parallelism using skewed cyclic block distribution</td>
<td></td>
</tr>
<tr>
<td>PLASMA 2009</td>
<td></td>
<td>DAG + priorities avoids waiting</td>
<td></td>
</tr>
<tr>
<td>2.5D MM 2011</td>
<td>reduce message volume along critical path</td>
<td>reduce number of messages along critical path</td>
<td>maintain ‘c’ copies of data</td>
</tr>
</tbody>
</table>
References


• http://icl.cs.utk.edu/plasma/index.html

• http://www.cs.utexas.edu/~flame