Parallel Numerical Algorithms

Chapter 3 – Dense Linear Systems
Section 3.1 – Vector and Matrix Products

Michael T. Heath and Edgar Solomonik

Department of Computer Science
University of Illinois at Urbana-Champaign

CS 554 / CSE 512

Outline

Basic Linear Algebra Subprograms

- Basic Linear Algebra Subprograms (BLAS) are building blocks for many other matrix computations
- BLAS encapsulate basic operations on vectors and matrices so they can be optimized for particular computer architecture while high-level routines that call them remain portable
- BLAS offer good opportunities for optimizing utilization of memory hierarchy
- Generic BLAS are available from `netlib`, and many computer vendors provide custom versions optimized for their particular systems

Examples of BLAS

Level	Work	Examples	Function
1	$\mathcal{O}(n)$	`daxpy`	Scalar $\times$ vector $+$ vector
		`ddot`	Inner product
		`dnrm2`	Euclidean vector norm
2	$\mathcal{O}(n^2)$	`dgemv`	Matrix-vector product
		`dtrsv`	Triangular solve
		`dger`	Outer-product
3	$\mathcal{O}(n^3)$	`dgemm`	Matrix-matrix product
		`dtrsm`	Multiple triangular solves
		`dsyrk`	Symmetric rank-k update

$$\underbrace{\text{BLAS 1 effective sec/flop}}_{\gamma_1} > \underbrace{\text{BLAS 2 effective sec/flop}}_{\gamma_2} \gg \underbrace{\text{BLAS 3 effective sec/flop}}_{\gamma_3}$$

Inner Product

- Inner product of two n-vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ given by

$$\boldsymbol{x}^T \boldsymbol{y} = \sum_{i=1}^{n} x_i \, y_i$$

- Computation of inner product requires n multiplications and $n - 1$ additions

$$M_1 = \Theta(n), \quad Q_1 = \Theta(n), \quad T_1 = \Theta(\gamma \, n)$$

- Effectively as hard as scalar reduction, can be done via binary or binomial tree summation

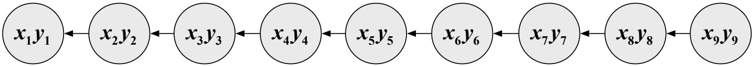
Parallel Algorithm

Partition

- For $i = 1, \ldots, n$, fine-grain task i stores x_i and y_i, and computes their product $x_i\, y_i$

Communicate

- Sum reduction over n fine-grain tasks



Fine-Grain Parallel Algorithm

$z_i = x_i y_i$ { local scalar product }

reduce z_i across all tasks $i = 1, ..., n$ { sum reduction }

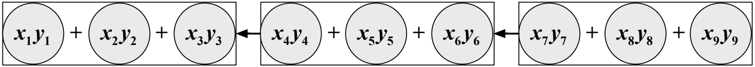
Agglomeration and Mapping

Agglomerate

- Combine k components of both $\boldsymbol{x}$ and $\boldsymbol{y}$ to form each coarse-grain task, which computes inner product of these subvectors
- Communication becomes sum reduction over n/k coarse-grain tasks

Map

- Assign $(n/k)/p$ coarse-grain tasks to each of p processors, for total of n/p components of $\boldsymbol{x}$ and $\boldsymbol{y}$ per processor



Coarse-Grain Parallel Algorithm

$z_i = \boldsymbol{x}_{[i]}^T \boldsymbol{y}_{[i]}$ { local inner product }

reduce z_i across all processors $i = 1, ..., p$ { sum reduction }

[$\boldsymbol{x}_{[i]}$ – subvector of $\boldsymbol{x}$ assigned to processor i]

Performance

The parallel costs (L_p, W_p, F_p) for the inner product are given by

- Computational cost $F_p = \Theta(n/p)$ regardless of network
- The latency and bandwidth costs depend on network:
  - 1-D mesh: $L_p, W_p = \Theta(p)$
  - 2-D mesh: $L_p, W_p = \Theta(\sqrt{p})$
  - hypercube: $L_p, W_p = \Theta(\log p)$
- For a hypercube or fully-connected network time is

$$T_p = \alpha L_p + \beta W_p + \gamma F_p = \Theta(\alpha \log(p) + \gamma n/p)$$

- Efficiency and scaling are the same as for binary tree sum

Outer Product

- Outer product of two n-vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is $n \times n$ matrix $\boldsymbol{Z} = \boldsymbol{x}\boldsymbol{y}^T$ whose (i, j) entry $z_{ij} = x_i \, y_j$
- For example,

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}^T = \begin{bmatrix} x_1y_1 & x_1y_2 & x_1y_3 \\ x_2y_1 & x_2y_2 & x_2y_3 \\ x_3y_1 & x_3y_2 & x_3y_3 \end{bmatrix}$$

- Computation of outer product requires n^2 multiplications,

$$M_1 = \Theta(n^2), \quad Q_1 = \Theta(n^2), \quad T_1 = \Theta(\gamma n^2)$$

(in this case, we should treat M_1 as output size or define the problem as in the BLAS: $\boldsymbol{Z} = \boldsymbol{Z}_{\text{input}} + \boldsymbol{x}\boldsymbol{y}^T$)

Parallel Algorithm

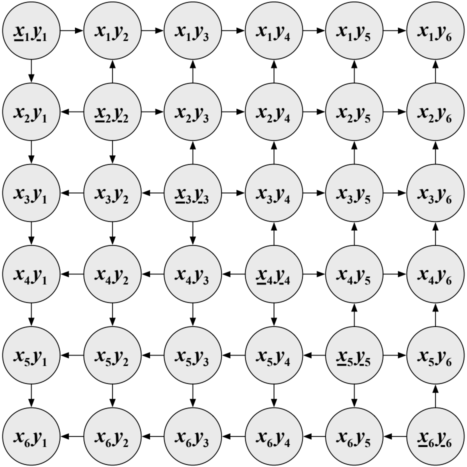
Partition

- For $i, j = 1, \ldots, n$, fine-grain task (i, j) computes and stores $z_{ij} = x_i\, y_j$, yielding 2-D array of n^2 fine-grain tasks
- Assuming no replication of data, at most $2n$ fine-grain tasks store components of $\boldsymbol{x}$ and $\boldsymbol{y}$, say either
  - for some j, task (i, j) stores x_i and task (j, i) stores y_i, or
  - task (i, i) stores both x_i and y_i, $i = 1, \ldots, n$

Communicate

- For $i = 1, \ldots, n$, task that stores x_i broadcasts it to all other tasks in ith task row
- For $j = 1, \ldots, n$, task that stores y_j broadcasts it to all other tasks in jth task column

Fine-Grain Tasks and Communication



Fine-Grain Parallel Algorithm

broadcast x_i to tasks $(i, k),\ k = 1, \ldots, n$ { horizontal broadcast }

broadcast y_j to tasks $(k, j),\ k = 1, \ldots, n$ { vertical broadcast }

$z_{ij} = x_i y_j$ { local scalar product }

Agglomeration

Agglomerate

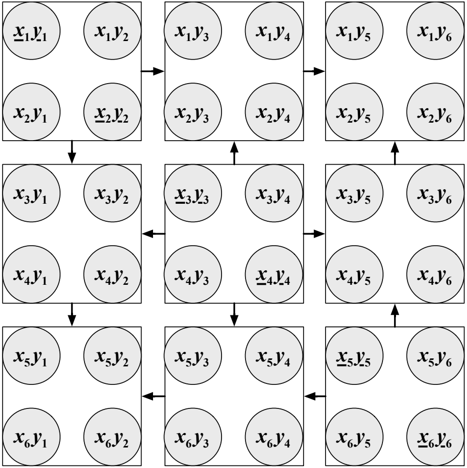
With $n \times n$ array of fine-grain tasks, natural strategies are

- 2-D: Combine $k \times k$ subarray of fine-grain tasks to form each coarse-grain task, yielding $(n/k)^2$ coarse-grain tasks
- 1-D column: Combine n fine-grain tasks in each column into coarse-grain task, yielding n coarse-grain tasks
- 1-D row: Combine n fine-grain tasks in each row into coarse-grain task, yielding n coarse-grain tasks

2-D Agglomeration

- Each task that stores portion of x must broadcast its subvector to all other tasks in its task row
- Each task that stores portion of y must broadcast its subvector to all other tasks in its task column

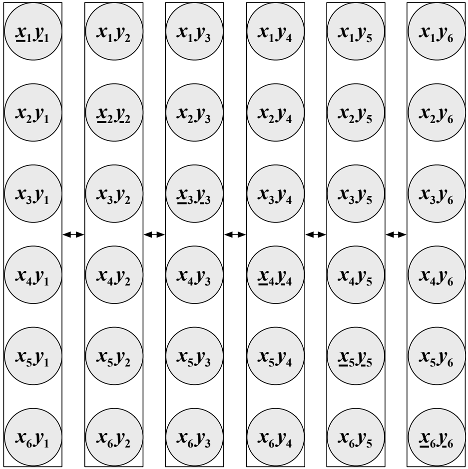
2-D Agglomeration



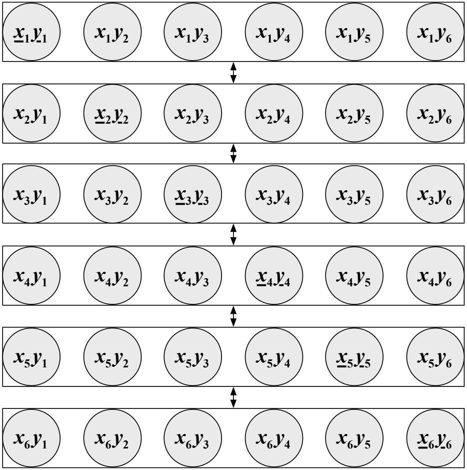
1-D Agglomeration

- If either x or y stored in one task, then broadcast required to communicate needed values to all other tasks
- If either x or y distributed across tasks, then multinode broadcast required to communicate needed values to other tasks

1-D Column Agglomeration



1-D Row Agglomeration

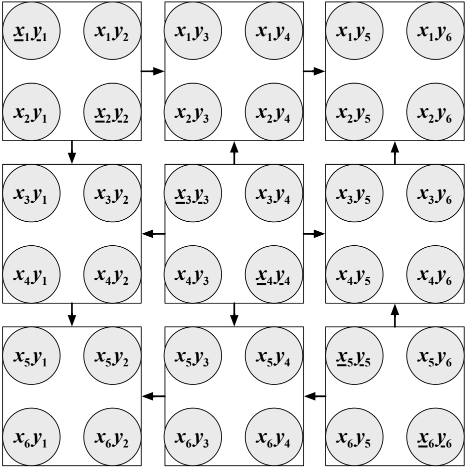


Mapping

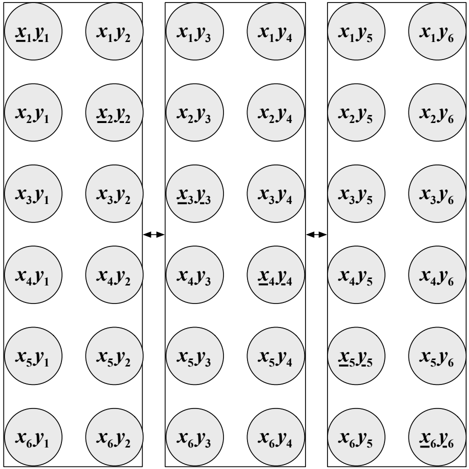
Map

- 2-D: Assign $(n/k)^2/p$ coarse-grain tasks to each of p processors using any desired mapping in each dimension, treating target network as 2-D mesh
- 1-D: Assign n/p coarse-grain tasks to each of p processors using any desired mapping, treating target network as 1-D mesh

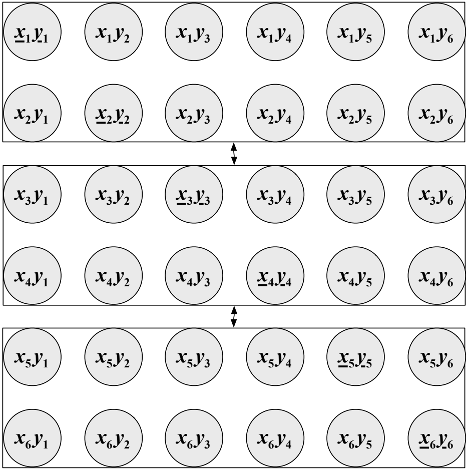
2-D Agglomeration with Block Mapping



1-D Column Agglomeration with Block Mapping



1-D Row Agglomeration with Block Mapping



Coarse-Grain Parallel Algorithm

broadcast $\boldsymbol{x}_{[i]}$ to ith process row { horizontal broadcast }

broadcast $\boldsymbol{y}_{[j]}$ to jth process column { vertical broadcast }

$\boldsymbol{Z}_{[i][j]} = \boldsymbol{x}_{[i]}\boldsymbol{y}_{[j]}^T$ { local outer product }

[$\boldsymbol{Z}_{[i][j]}$ means submatrix of $\boldsymbol{Z}$ assigned to process (i, j) by mapping]

Performance

The parallel costs (L_p, W_p, F_p) for the outer product are

- Computational cost $F_p = \Theta(n^2/p)$ regardless of network
- The latency and bandwidth costs can be derived from the cost of broadcast/allgather
  - 1-D agglomeration: $L_p = \Theta(\log p), W_p = \Theta(n)$
  - 2-D agglomeration: $L_p = \Theta(\log p), W_p = \Theta(n/\sqrt{p})$
- For 1-D agglomeration, execution time is

$$T_p^{\text{1-D}} = T_p^{\text{allgather}}(n) + \Theta(\gamma n^2/p) = \Theta(\alpha \log(p) + \beta n + \gamma n^2/p)$$

- For 2-D agglomeration, execution time is

$$T_p^{\text{2-D}} = 2T_{\sqrt{p}}^{\text{bcast}}(n/\sqrt{p}) + \Theta(\gamma n^2/p) = \Theta(\alpha \log(p) + \beta n/\sqrt{p} + \gamma n^2/p)$$

Outer Product Strong Scaling

- 1-D agglomeration is strongly scalable to

$$p_s = \Theta(\min((\gamma/\alpha)n^2/\log((\gamma/\alpha)n^2), (\gamma/\beta)n))$$

processors, since

$$E_{p_s}^{\text{1-D}} = \Theta(1/(1 + (\alpha/\gamma)\log(p_s)p_s/n^2 + (\beta/\gamma)p_s/n))$$

- 2-D agglomeration is strongly scalable to

$$p_s = \Theta(\min((\gamma/\alpha)n^2/\log((\gamma/\alpha)n^2), (\gamma/\beta)^2 n^2))$$

processors, since

$$E_{p_s}^{\text{2-D}} = \Theta(1/(1 + (\alpha/\gamma)\log(p_s)p_s/n^2 + (\beta/\gamma)\sqrt{p_s}/n))$$

Outer Product Weak Scaling

- 1-D agglomeration is weakly scalable to

$$p_w = \Theta(\min(2^{(\gamma/\alpha)n^2}, (\gamma/\beta)^2 n^2))$$

processors, since

$$E_{p_w}^{\text{1-D}}(\sqrt{p_w}n) = \Theta(1/(1 + (\alpha/\gamma)\log(p_w)/n^2 + (\beta/\gamma)\sqrt{p_w}/n))$$

- 2-D agglomeration is weakly scalable to

$$p_w = \Theta(2^{(\gamma/\alpha)n^2})$$

processors, since

$$E_{p_w}^{\text{2-D}}(\sqrt{p_w}n) = \Theta(1/(1 + (\alpha/\gamma)\log(p_w)/n^2 + (\beta/\gamma)/n))$$

Memory Requirements

- The memory requirements are dominated by storing Z, which in practice means the outer-product is a poor primitive (local *flop-to-byte ratio* of 1)
- If possible, Z should be operated on as it is computed, e.g. if we really need

$$v_i = \sum_j f(x_i y_j) \quad \text{for some scalar function } f$$

- If Z does not need to be stored, vector storage dominates
- Without further modification, 1-D algorithm requires $M_p = \Theta(np)$ overall storage for vector
- Without further modification, 2-D algorithm requires $M_p = \Theta(n\sqrt{p})$ overall storage for vector

Matrix-Vector Product

- Consider matrix-vector product

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$$

where $\boldsymbol{A}$ is $n \times n$ matrix and $\boldsymbol{x}$ and $\boldsymbol{y}$ are n-vectors

- Components of vector $\boldsymbol{y}$ are given by inner products:

$$y_i = \sum_{j=1}^{n} a_{ij}\, x_j, \quad i = 1, \ldots, n$$

- The sequential memory, work, and time are

$$M_1 = \Theta(n^2), \quad Q_1 = \Theta(n^2), \quad T_1 = \Theta(\gamma n^2)$$

Parallel Algorithm

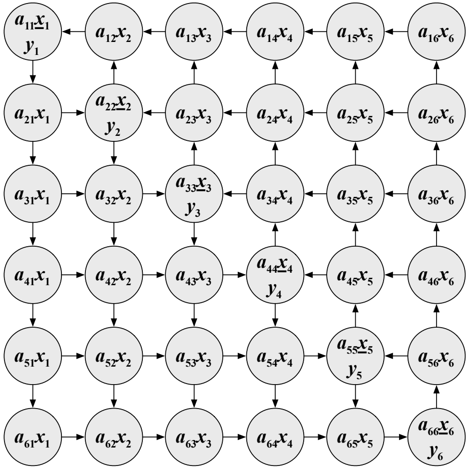
Partition

- For $i, j = 1, \ldots, n$, fine-grain task (i, j) stores a_{ij} and computes $a_{ij}\, x_j$, yielding 2-D array of n^2 fine-grain tasks
- Assuming no replication of data, at most $2n$ fine-grain tasks store components of $\boldsymbol{x}$ and $\boldsymbol{y}$, say either
    - for some j, task (j, i) stores x_i and task (i, j) stores y_i, or
    - task (i, i) stores both x_i and y_i, $i = 1, \ldots, n$

Communicate

- For $j = 1, \ldots, n$, task that stores x_j broadcasts it to all other tasks in jth task column
- For $i = 1, \ldots, n$, sum reduction over ith task row gives y_i

Fine-Grain Tasks and Communication



Fine-Grain Parallel Algorithm

broadcast x_j to tasks $(k, j),\ k = 1, \ldots, n$ { vertical broadcast }

$y_i = a_{ij} x_j$ { local scalar product }

reduce y_i across tasks $(i, k),\ k = 1, \ldots, n$ { horizontal sum reduction }

Agglomeration

Agglomerate

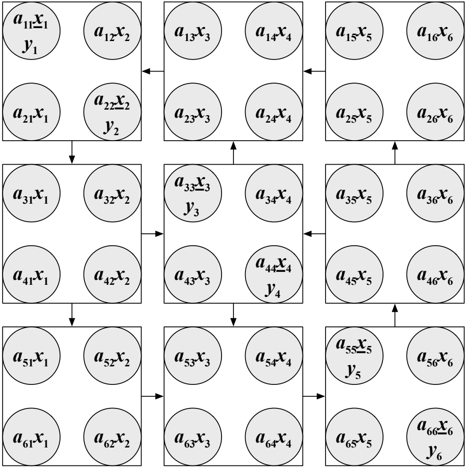
With $n \times n$ array of fine-grain tasks, natural strategies are

- 2-D: Combine $k \times k$ subarray of fine-grain tasks to form each coarse-grain task, yielding $(n/k)^2$ coarse-grain tasks
- 1-D column: Combine n fine-grain tasks in each column into coarse-grain task, yielding n coarse-grain tasks
- 1-D row: Combine n fine-grain tasks in each row into coarse-grain task, yielding n coarse-grain tasks

2-D Agglomeration

- Subvector of x broadcast along each task column
- Each task computes local matrix-vector product of submatrix of A with subvector of x
- Sum reduction along each task row produces subvector of result y

2-D Agglomeration



1-D Agglomeration

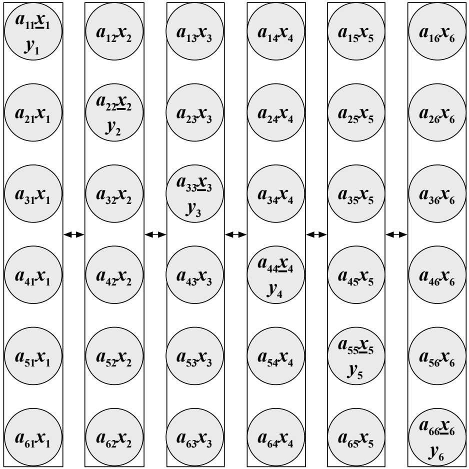
1-D column agglomeration

- Each task computes product of its component of x times its column of matrix, with no communication required
- Sum reduction across tasks then produces y

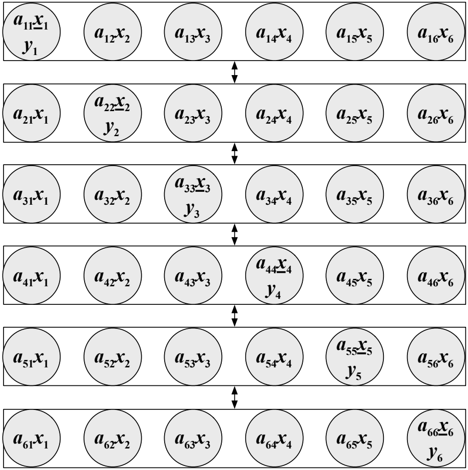
1-D row agglomeration

- If x stored in one task, then broadcast required to communicate needed values to all other tasks
- If x distributed across tasks, then multinode broadcast required to communicate needed values to other tasks
- Each task computes inner product of its row of A with *entire* vector x to produce its component of y

1-D Column Agglomeration



1-D Row Agglomeration



1-D Agglomeration

Column and row algorithms are dual to each other

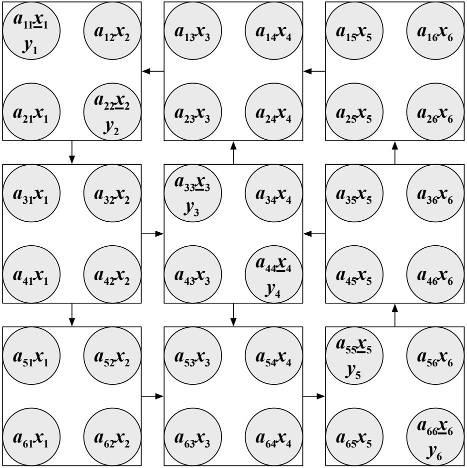
- Column algorithm begins with communication-free local vector scaling (daxpy) computations combined across processors by a reduction
- Row algorithm begins with broadcast followed by communication-free local inner-product (ddot) computations

Mapping

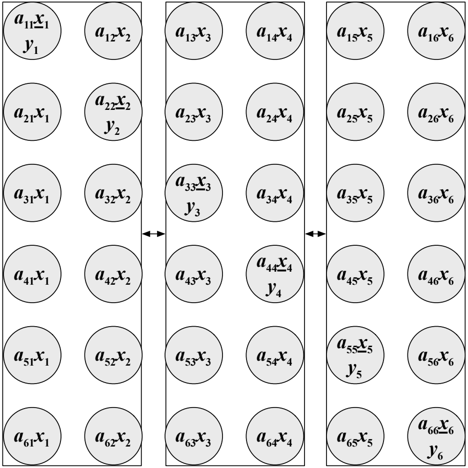
Map

- 2-D: Assign $(n/k)^2/p$ coarse-grain tasks to each of p processes using any desired mapping in each dimension, treating target network as 2-D mesh
- 1-D: Assign n/p coarse-grain tasks to each of p processes using any desired mapping, treating target network as 1-D mesh

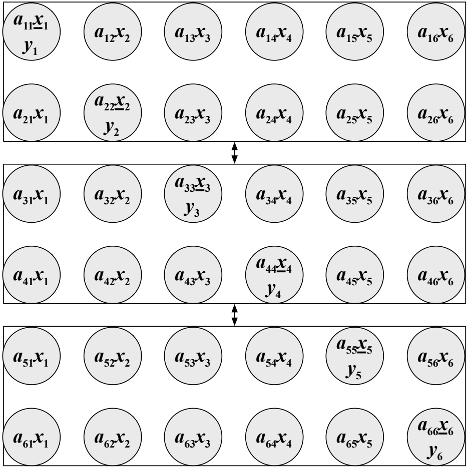
2-D Agglomeration with Block Mapping



1-D Column Agglomeration with Block Mapping



1-D Row Agglomeration with Block Mapping



Coarse-Grain Parallel Algorithm

broadcast $\boldsymbol{x}_{[\,j\,]}$ to jth process column { vertical broadcast }

$\boldsymbol{y}_{[i\,]} = \boldsymbol{A}_{[i\,][\,j\,]}\boldsymbol{x}_{[\,j\,]}$ { local matrix-vector product }

reduce $\boldsymbol{y}_{[i\,]}$ across ith process row { horizontal sum reduction }

Performance

The parallel costs (L_p, W_p, F_p) for the matrix-vector product are

- Computational cost $F_p = \Theta(n^2/p)$ regardless of network
- Communication costs can be derived from the cost of collectives
    - 1-D agglomeration: $L_p = \Theta(\log p), W_p = \Theta(n)$
    - 2-D agglomeration: $L_p = \Theta(\log p), W_p = \Theta(n/\sqrt{p})$
- For 1-D row agglomeration, perform allgather,

$$T_p^{\text{1-D}} = T_p^{\text{allgather}}(n) + \Theta(\gamma n^2/p) = \Theta(\alpha \log(p) + \beta n + \gamma n^2/p)$$

- For 2-D agglomeration, perform broadcast and reduction,

$$\begin{aligned} T_p^{\text{2-D}} &= T_{\sqrt{p}}^{\text{bcast}}(n/\sqrt{p}) + T_{\sqrt{p}}^{\text{reduce}}(n/\sqrt{p}) + \Theta(\gamma n^2/p) \\ &= \Theta(\alpha \log(p) + \beta n/\sqrt{p} + \gamma n^2/p) \end{aligned}$$

Matrix-Matrix Product

- Consider matrix-matrix product

$$\boldsymbol{C} = \boldsymbol{A}\,\boldsymbol{B}$$

  where $\boldsymbol{A}$, $\boldsymbol{B}$, and result $\boldsymbol{C}$ are $n \times n$ matrices

- Entries of matrix $\boldsymbol{C}$ are given by

$$c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj}, \quad i, j = 1, \ldots, n$$

- Each of n^2 entries of $\boldsymbol{C}$ requires n multiply-add operations, so model serial time as

$$T_1 = \gamma\, n^3$$

Matrix-Matrix Product

- Matrix-matrix product can be viewed as
  - n^2 inner products, or
  - sum of n outer products, or
  - n matrix-vector products

  and each viewpoint yields different algorithm

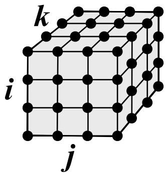
- One way to derive parallel algorithms for matrix-matrix product is to apply parallel algorithms already developed for inner product, outer product, or matrix-vector product

- However, considering the problem as a whole yields the best algorithms

Parallel Algorithm

Partition

- For $i, j, k = 1, \ldots, n$, fine-grain task (i, j, k) computes product $a_{ik}\, b_{kj}$, yielding 3-D array of n^3 fine-grain tasks
- Assuming no replication of data, at most $3n^2$ fine-grain tasks store entries of $\boldsymbol{A}$, $\boldsymbol{B}$, or $\boldsymbol{C}$, say task (i, j, j) stores a_{ij}, task (i, j, i) stores b_{ij}, and task (i, j, k) stores c_{ij} for $i, j = 1, \ldots, n$ and some fixed k



- We refer to subsets of tasks along i, j, and k dimensions as rows, columns, and layers, respectively, so kth column of $\boldsymbol{A}$ and kth row of $\boldsymbol{B}$ are stored in kth layer of tasks

Parallel Algorithm

Communicate

- Broadcast entries of jth column of $\boldsymbol{A}$ horizontally along each task row in jth layer
- Broadcast entries of ith row of $\boldsymbol{B}$ vertically along each task column in ith layer
- For $i, j = 1, \ldots, n$, result c_{ij} is given by sum reduction over tasks (i, j, k), $k = 1, \ldots, n$

Fine-Grain Algorithm

broadcast a_{ik} to tasks $(i, q, k),\ q = 1, \ldots, n$ { horizontal broadcast }

broadcast b_{kj} to tasks $(q, j, k),\ q = 1, \ldots, n$ { vertical broadcast }

$c_{ij} = a_{ik} b_{kj}$ { local scalar product }

reduce c_{ij} across tasks $(i, j, q),\ q = 1, \ldots, n$ { lateral sum reduction }

Agglomeration

Agglomerate

With $n \times n \times n$ array of fine-grain tasks, natural strategies are

- 3-D: Combine $q \times q \times q$ subarray of fine-grain tasks
- 2-D: Combine $q \times q \times n$ subarray of fine-grain tasks, eliminating sum reductions
- 1-D column: Combine $n \times 1 \times n$ subarray of fine-grain tasks, eliminating vertical broadcasts and sum reductions
- 1-D row: Combine $1 \times n \times n$ subarray of fine-grain tasks, eliminating horizontal broadcasts and sum reductions

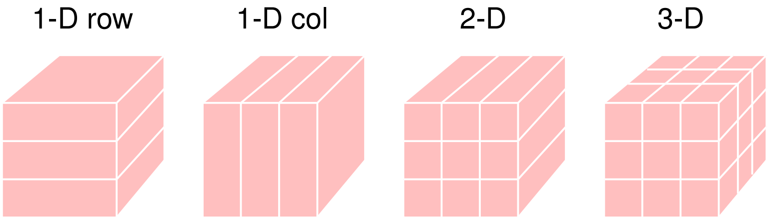
Mapping

Map

Corresponding mapping strategies are

- 3-D: Assign $(n/q)^3/p$ coarse-grain tasks to each of p processors using any desired mapping in each dimension, treating target network as 3-D mesh
- 2-D: Assign $(n/q)^2/p$ coarse-grain tasks to each of p processors using any desired mapping in each dimension, treating target network as 2-D mesh
- 1-D: Assign n/p coarse-grain tasks to each of p processors using any desired mapping, treating target network as 1-D mesh

Agglomeration with Block Mapping



Coarse-Grain 3-D Parallel Algorithm

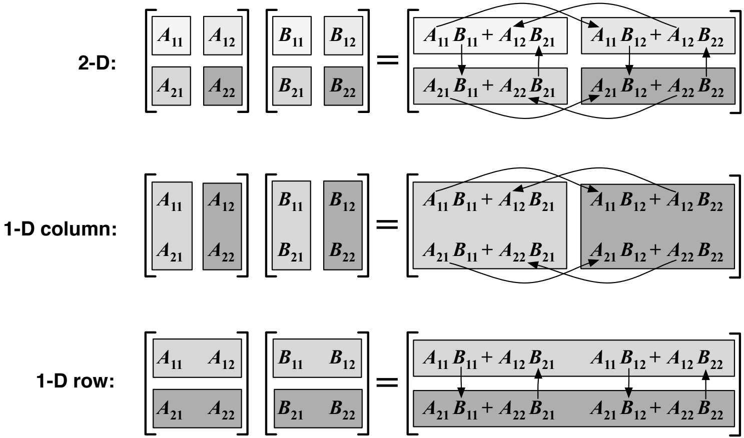
broadcast $\boldsymbol{A}_{[i][k]}$ to ith processor row { horizontal broadcast }

broadcast $\boldsymbol{B}_{[k][j]}$ to jth processor column { vertical broadcast }

$\boldsymbol{C}_{[i][j]} = \boldsymbol{A}_{[i][k]}\boldsymbol{B}_{[k][j]}$ { local matrix product }

reduce $\boldsymbol{C}_{[i][j]}$ across processor layers { lateral sum reduction }

Agglomeration with Block Mapping



Coarse-Grain 2-D Parallel Algorithm

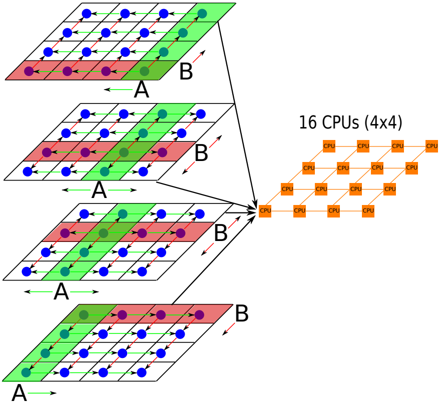
allgather $\boldsymbol{A}_{[i][j]}$ in ith processor row { horizontal broadcast }
allgather $\boldsymbol{B}_{[i][j]}$ in jth processor column { vertical broadcast }
$\boldsymbol{C}_{[i][j]} = \boldsymbol{0}$
for $k = 1, \ldots, \sqrt{p}$
$\quad \boldsymbol{C}_{[i][j]} = \boldsymbol{C}_{[i][j]} + \boldsymbol{A}_{[i][k]}\boldsymbol{B}_{[k][j]}$ { sum local products }
end

SUMMA Algorithm

- Algorithm just described requires excessive memory, since each process accumulates $\sqrt{p}$ blocks of both $\boldsymbol{A}$ and $\boldsymbol{B}$
- One way to reduce memory requirements is to
  - broadcast blocks of $\boldsymbol{A}$ successively across processor rows
  - broadcast blocks of $\boldsymbol{B}$ successively across processor cols

$\boldsymbol{C}_{[i][j]} = \boldsymbol{0}$
for $k = 1, \ldots, \sqrt{p}$
    broadcast $\boldsymbol{A}_{[i][k]}$ in ith processor row { horizontal broadcast }
    broadcast $\boldsymbol{B}_{[k][j]}$ in jth processor column { vertical broadcast }
    $\boldsymbol{C}_{[i][j]} = \boldsymbol{C}_{[i][j]} + \boldsymbol{A}_{[i][k]}\boldsymbol{B}_{[k][j]}$ { sum local products }
end

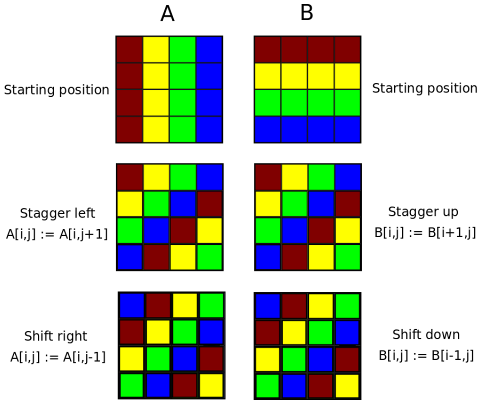
SUMMA Algorithm



Cannon Algorithm

- Another approach, due to Cannon (1969), is to circulate blocks of B vertically and blocks of A horizontally in ring fashion
- Blocks of both matrices must be initially aligned using circular shifts so that correct blocks meet as needed
- Requires less memory than SUMMA and replaces line broadcasts with shifts, lowering the number of messages

Cannon Algorithm



⋮

Fox Algorithm

- It is possible to mix techniques from SUMMA and Cannon's algorithm:
  - circulate blocks of B in ring fashion vertically along processor columns step by step so that each block of B comes in conjunction with appropriate block of A broadcast at that same step
- This algorithm is due to Fox et al.
- Shifts, especially in Cannon's algorithm, are harder to generalize to nonsquare processor grids than collectives in algorithms like SUMMA

Execution Time for 3-D Agglomeration

- For 3-D agglomeration, computing each of p blocks $\boldsymbol{C}_{[i][j]}$ requires matrix-matrix product of two $(n/\sqrt[3]{p}\,) \times (n/\sqrt[3]{p}\,)$ blocks, so

$$F_p = (n/\sqrt[3]{p}\,)^3 = n^3/p$$

- On 3-D mesh, each broadcast or reduction takes time

$$T_{p^{1/3}}^{\text{bcast}}((n/p^{1/3})^2) = O(\alpha \log p + \beta n^2/p^{2/3})$$

- Total time is therefore

$$T_p = O(\alpha \log p + \beta\, n^2/p^{2/3} + \gamma\, n^3/p)$$

- However, overall memory footprint is

$$M_p = \Theta(p(n/p^{1/3})^2) = \Theta(p^{1/3}n^2)$$

Execution Time for 2-D Agglomeration

- For 2-D agglomeration (SUMMA), computation of each block $C_{[i][j]}$ requires $\sqrt{p}$ matrix-matrix products of $(n/\sqrt{p}\,) \times (n/\sqrt{p}\,)$ blocks, so

$$F_p = \sqrt{p}\,(n/\sqrt{p}\,)^3 = n^3/p$$

- For broadcast among rows and columns of processor grid, communication time is

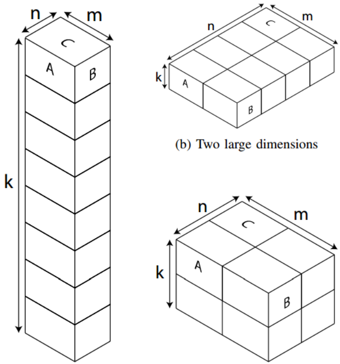
$$2\sqrt{p}T^{\text{bcast}}_{\sqrt{p}}(n^2/p) \quad = \Theta(\alpha\sqrt{p}\log(p) + \beta n^2/\sqrt{p})$$

- Total time is therefore

$$T_p = O(\alpha\sqrt{p}\log(p) + \beta n^2/\sqrt{p} + \gamma\, n^3/p)$$

- The algorithm is memory-efficient, $M_p = \Theta(n^2)$

Rectangular Matrix Multiplication



(a) One large dimension

(b) Two large dimensions

(c) Three large dimensions

If $\boldsymbol{C}$ is $m \times n$, $\boldsymbol{A}$ is $m \times k$, and $\boldsymbol{B}$ is $k \times n$, choosing a 3D grid that optimizes volume-to-surface-area ratio yields bandwidth cost...

$$W_p(m,n,k) = O\left(\min_{p_1p_2p_3=p} \left[\frac{mk}{p_1p_2} + \frac{kn}{p_1p_3} + \frac{mn}{p_2p_3} \right] \right)$$

Communication vs. Memory Tradeoff

- Communication cost for 2-D algorithms for matrix-matrix product is optimal, assuming no replication of storage
- If explicit replication of storage is allowed, then lower communication volume is possible via 3-D algorithm
- Generally, we assign $n/p_1 \times n/p_2 \times n/p_3$ computation subvolume to each processor
- The largest face of the subvolume gives communication cost, the smallest face gives minimal memory usage
  - can keep smallest face local while successively computing slices of subvolume

Leveraging Additional Memory in Matrix Multiplication

- Provided $\bar{M}$ total available memory, 2-D and 3-D algorithms achieve bandwidth cost

$$W_p(n, \bar{M}) = \begin{cases} \infty & : \bar{M} < n^2 \\ n^2/\sqrt{p} & : \bar{M} = \Theta(n^2) \\ n^2/p^{2/3} & : \bar{M} = \Theta(n^2 p^{1/3}) \end{cases}$$

- For general $\bar{M}$, possible to pick processor grid to achieve

$$W_p(n, \bar{M}) = O(n^3/(\sqrt{p}\bar{M}^{1/2}) + n^2/p^{2/3})$$

- and an overall execution time of

$$T_p(n, \bar{M}) = O(\alpha(\log p + n^3\sqrt{p}/\bar{M}^{3/2}) + \beta W_p(n, \bar{M}) + \gamma n^3/p)$$

References

- R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, A three-dimensional approach to parallel matrix multiplication, *IBM J. Res. Dev.*, 39:575-582, 1995
- J. Berntsen, Communication efficient matrix multiplication on hypercubes, *Parallel Comput.* 12:335-342, 1989
- J. Demmel, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, Communication-optimal parallel recursive rectangular matrix multiplication, IPDPS, 2013
- J. W. Demmel, M. T. Heath, and H. A. van der Vorst, Parallel numerical linear algebra, *Acta Numerica* 2:111-197, 1993

References

- R. Dias da Cunha, A benchmark study based on the parallel computation of the vector outer-product $A = uv^T$ operation, *Concurrency: Practice and Experience* 9:803-819, 1997
- G. C. Fox, S. W. Otto, and A. J. G. Hey, Matrix algorithms on a hypercube I: matrix multiplication, *Parallel Comput.* 4:17-31, 1987
- D. Irony, S. Toledo, and A. Tiskin, Communication lower bounds for distributed-memory matrix multiplication, *J. Parallel Distrib. Comput.* 64:1017-1026, 2004.
- S. L. Johnsson, Communication efficient basic linear algebra computations on hypercube architectures, *J. Parallel Distrib. Comput.* 4(2):133-172, 1987

References

- S. L. Johnsson, Minimizing the communication time for matrix multiplication on multiprocessors, *Parallel Comput.* 19:1235-1257, 1993
- B. Lipshitz, Communication-avoiding parallel recursive algorithms for matrix multiplication, Tech. Rept. UCB/EECS-2013-100, University of California at Berkeley, May 2013.
- O. McBryan and E. F. Van de Velde, Matrix and vector operations on hypercube parallel processors, *Parallel Comput.* 5:117-126, 1987
- R. A. Van De Geijn and J. Watts, SUMMA: Scalable universal matrix multiplication algorithm, *Concurrency: Practice and Experience* 9(4):255-274, 1997