Parallel Numerical Algorithms
Chapter 5 – Vector and Matrix Products

Prof. Michael T. Heath

Department of Computer Science
University of Illinois at Urbana-Champaign

CS 554 / CSE 512

Outline

1. Inner Product
2. Outer Product
3. Matrix-Vector Product
4. Matrix-Matrix Product

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	$\mathcal{O}(n)$	`saxpy`	Scalar $\times$ vector $+$ vector
		`sdot`	Inner product
		`snrm2`	Euclidean vector norm
2	$\mathcal{O}(n^2)$	`sgemv`	Matrix-vector product
		`strsv`	Triangular solution
		`sger`	Rank-one update
3	$\mathcal{O}(n^3)$	`sgemm`	Matrix-matrix product
		`strsm`	Multiple triang. solutions
		`ssyrk`	Rank-k update

Simplifying Assumptions

- For problem of dimension n using p processes, assume p (or in some cases $\sqrt{p}$) divides n
- For 2-D mesh, assume p is perfect square and mesh is $\sqrt{p} \times \sqrt{p}$
- For hypercube, assume p is power of two
- Assume matrices are square, $n \times n$, not rectangular
- Dealing with general cases where these assumptions do not hold is straightforward but tedious, and complicates notation
- Caveat: your mileage may vary, depending on assumptions about target system, such as level of concurrency in communication

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
- For simplicity, model serial time as

$$T_1 = t_c \, n$$

where t_c is time for one scalar multiply-add operation

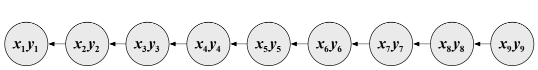
Parallel Algorithm

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

$z = x_i y_i$	{ local scalar product }
reduce z across all tasks	{ sum reduction }

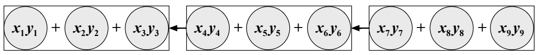
Agglomeration and Mapping

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 processes, for total of n/p components of $\boldsymbol{x}$ and $\boldsymbol{y}$ per process



Coarse-Grain Parallel Algorithm

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

reduce z across all processes { sum reduction }

[$\boldsymbol{x}_{[i]}$ means subvector of $\boldsymbol{x}$ assigned to process i by mapping]

Performance

- Time for computation phase is

$$T_{\text{comp}} = t_c\, n/p$$

  regardless of network
- Depending on network, time for communication phase is
  - 1-D mesh: $T_{\text{comm}} = (t_s + t_w)\,(p-1)$
  - 2-D mesh: $T_{\text{comm}} = (t_s + t_w)\,2(\sqrt{p}-1)$
  - hypercube: $T_{\text{comm}} = (t_s + t_w)\,\log p$
- For simplicity, ignore cost of additions in reduction, which is usually negligible

Scalability for 1-D Mesh

- For 1-D mesh, total time is

$$T_p = t_c\, n/p + (t_s + t_w)\,(p-1)$$

- To determine isoefficiency function, set

$$\begin{aligned} T_1 &\approx E\,(p\,T_p) \\ t_c\, n &\approx E\,(t_c\, n + (t_s + t_w)\, p\,(p-1)) \end{aligned}$$

  which holds if $n = \Theta(p^2)$, so isoefficiency function is $\Theta(p^2)$, since $T_1 = \Theta(n)$

Scalability for 2-D Mesh

- For 2-D mesh, total time is

$$T_p = t_c\, n/p + (t_s + t_w)\,2(\sqrt{p}-1)$$

- To determine isoefficiency function, set

$$t_c\, n \approx E\,(t_c\, n + (t_s + t_w)\, p\, 2(\sqrt{p}-1))$$

  which holds if $n = \Theta(p^{3/2})$, so isoefficiency function is $\Theta(p^{3/2})$, since $T_1 = \Theta(n)$

Scalability for Hypercube

- For hypercube, total time is

$$T_p = t_c\, n/p + (t_s + t_w)\,\log p$$

- To determine isoefficiency function, set

$$t_c\, n \approx E\,(t_c\, n + (t_s + t_w)\, p\,\log p)$$

  which holds if $n = \Theta(p\,\log p)$, so isoefficiency function is $\Theta(p\,\log p)$, since $T_1 = \Theta(n)$

Optimality for 1-D Mesh

- To determine optimal number of processes for given n, take p to be continuous variable and minimize T_p with respect to p
- For 1-D mesh

$$\begin{aligned} T_p' &= \frac{d}{dp}\left[t_c\, n/p + (t_s + t_w)\,(p-1)\right] \\ &= -t_c\, n/p^2 + (t_s + t_w) = 0 \end{aligned}$$

  implies that optimal number of processes is

$$p \approx \sqrt{\frac{t_c\, n}{t_s + t_w}}$$

Optimality for 1-D Mesh

- If $n < (t_s + t_w)/t_c$, then only one process should be used
- Substituting optimal p into formula for T_p shows that optimal time to compute inner product grows as $\sqrt{n}$ with increasing n on 1-D mesh

Optimality for Hypercube

- For hypercube

$$\begin{aligned} T_p' &= \frac{d}{dp}\Big[t_c\, n/p + (t_s + t_w)\, \log p\Big] \\ &= -t_c\, n/p^2 + (t_s + t_w)/p = 0 \end{aligned}$$

implies that optimal number of processes is

$$p \approx \frac{t_c\, n}{t_s + t_w}$$

and optimal time grows as $\log n$ with increasing n

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, so model serial time as

$$T_1 = t_c\, n^2$$

where t_c is time for one scalar multiplication

Parallel Algorithm

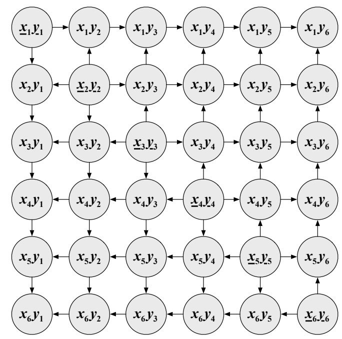
Partition

- 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

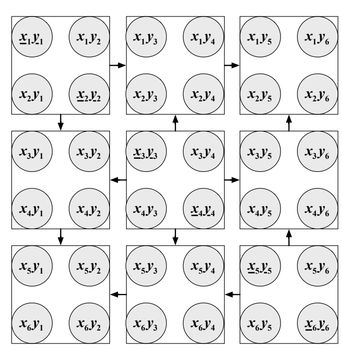
With $n \times n$ array of fine-grain tasks, natural strategies are

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

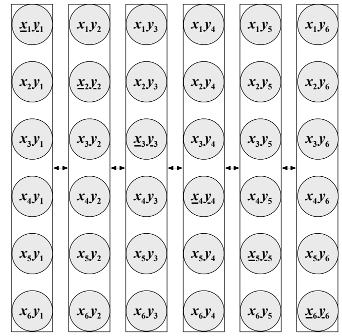
2-D Agglomeration



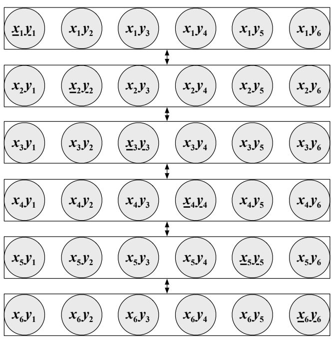
1-D Agglomeration

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

1-D Column Agglomeration



1-D Row Agglomeration

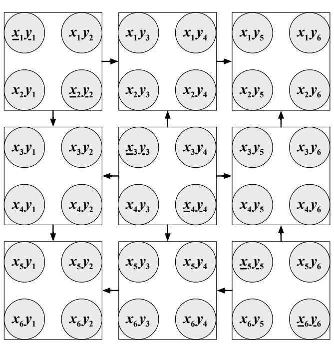


Mapping

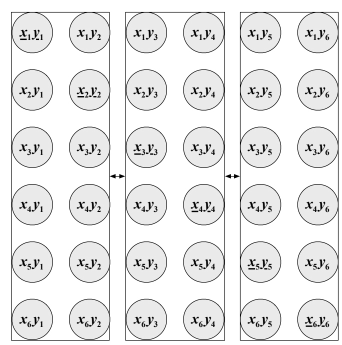
Map

- 2-D: Assign $(n/k)^2/p$ coarse-grain tasks to each of p 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

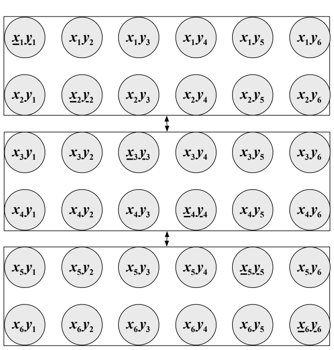
2-D Agglomeration with Block Mapping



1-D Column Agglomeration with Block Mapping



1-D Row Agglomeration with Block Mapping



Coarse-Grain Parallel Algorithm

broadcast $\boldsymbol{x}_{[i]}$ 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]

Scalability

- Time for computation phase is

$$T_{\text{comp}} = t_c\, n^2/p$$

regardless of network or agglomeration scheme

- For 2-D agglomeration on 2-D mesh, communication time is at least

$$T_{\text{comm}} = (t_s + t_w\, n/\sqrt{p}\,)\,(\sqrt{p} - 1)$$

assuming broadcasts can be overlapped

Scalability for 2-D Mesh

- Total time for 2-D mesh is at least

$$\begin{aligned} T_p &= t_c\, n^2/p + (t_s + t_w\, n/\sqrt{p}\,)\,(\sqrt{p} - 1) \\ &\approx t_c\, n^2/p + t_s\sqrt{p} + t_w\, n \end{aligned}$$

- To determine isoefficiency function, set

$$t_c\, n^2 \approx E\,(t_c\, n^2 + t_s\, p^{3/2} + t_w\, n\, p)$$

which holds for large p if $n = \Theta(p)$, so isoefficiency function is $\Theta(p^2)$, since $T_1 = \Theta(n^2)$

Scalability for Hypercube

- Total time for hypercube is at least

$$\begin{aligned} T_p &= t_c\, n^2/p + (t_s + t_w\, n/\sqrt{p}\,)\,(\log p)/2 \\ &= t_c\, n^2/p + t_s\,(\log p)/2 + t_w\, n\,(\log p)/(2\sqrt{p}\,) \end{aligned}$$

- To determine isoefficiency function, set

$$t_c\, n^2 \approx E\,(t_c\, n^2 + t_s\, p\,(\log p)/2 + t_w\, n\,\sqrt{p}\,(\log p)/2)$$

which holds for large p if $n = \Theta(\sqrt{p}\,\log p)$, so isoefficiency function is $\Theta(p\,(\log p)^2)$, since $T_1 = \Theta(n^2)$

Scalability for 1-D mesh

- Depending on network, time for communication phase with 1-D agglomeration is at least
  - 1-D mesh: $T_{\text{comm}} = (t_s + t_w\, n/p)\,(p - 1)$
  - 2-D mesh: $T_{\text{comm}} = (t_s + t_w\, n/p)\,2(\sqrt{p} - 1)$
  - hypercube: $T_{\text{comm}} = (t_s + t_w\, n/p)\,\log p$

  assuming broadcasts can be overlapped

Scalability for 1-D Mesh

- For 1-D mesh, total time is at least

$$\begin{aligned} T_p &= t_c\, n^2/p + (t_s + t_w\, n/p)\,(p - 1) \\ &\approx t_c\, n^2/p + t_s\, p + t_w\, n \end{aligned}$$

- To determine isoefficiency function, set

$$t_c\, n^2 \approx E\,(t_c\, n^2 + t_s\, p^2 + t_w\, n\, p)$$

which holds if $n = \Theta(p)$, so isoefficiency function is $\Theta(p^2)$, since $T_1 = \Theta(n^2)$

Memory Requirements

- With either 1-D or 2-D algorithm, straightforward broadcasting of $\boldsymbol{x}$ or $\boldsymbol{y}$ could require as much total memory as replication of entire vector in all processes
- Memory requirements can be reduced by circulating portions of $\boldsymbol{x}$ or $\boldsymbol{y}$ through processes in ring fashion, with each process using each portion as it passes through, so that no process need store entire vector at once

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

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

- Each of n components requires n multiply-add operations, so model serial time as

$$T_1 = t_c\, n^2$$

Parallel Algorithm

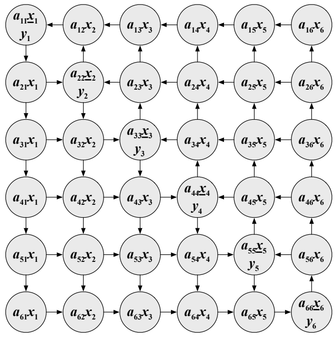
Partition

- 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

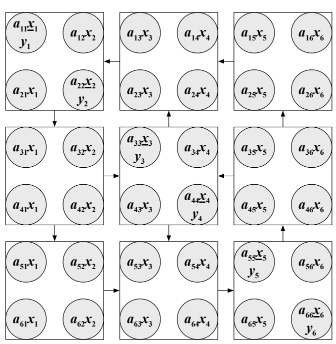
With $n \times n$ array of fine-grain tasks, natural strategies are

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

2-D Agglomeration



1-D Agglomeration

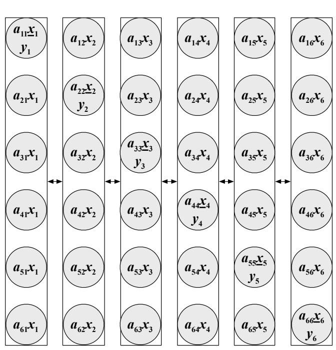
1-D column agglomeration

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

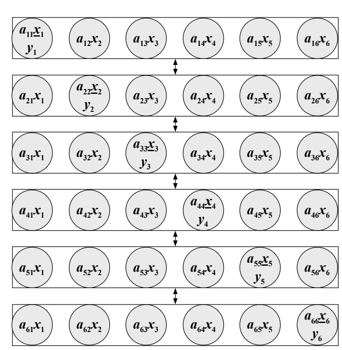
1-D row agglomeration

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

1-D Column Agglomeration



1-D Row Agglomeration



1-D Agglomeration

Column and row algorithms are dual to each other

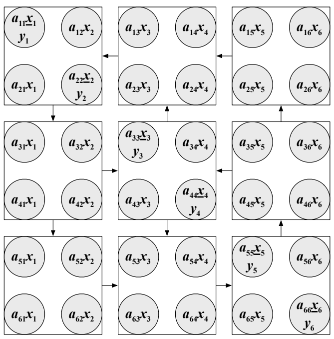
- Column algorithm begins with communication-free local `saxpy` computations followed by sum reduction
- Row algorithm begins with broadcast followed by communication-free local `sdot` computations

Mapping

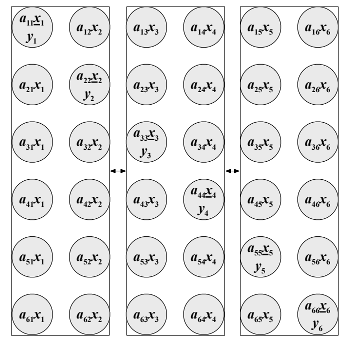
Map

- 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

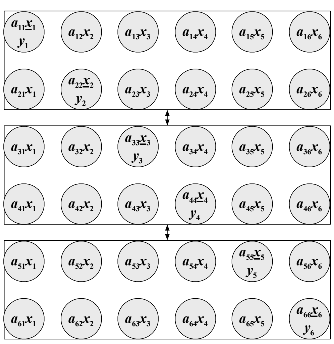
2-D Agglomeration with Block Mapping



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 }

Scalability

- Time for computation phase is

$$T_{\text{comp}} = t_c\, n^2/p$$

regardless of network or agglomeration scheme

- For 2-D agglomeration on 2-D mesh, each of two communication phases requires time

$$(t_s + t_w\, n/\sqrt{p}\,)\,(\sqrt{p} - 1) \approx t_s\,\sqrt{p} + t_w\, n$$

so total time is

$$T_p \approx t_c\, n^2/p + 2(t_s\,\sqrt{p} + t_w\, n)$$

Scalability for 2-D Mesh

- To determine isoefficiency function, set

$$t_c\, n^2 \approx E\,(t_c\, n^2 + 2(t_s\, p^{3/2} + t_w\, n\, p))$$

which holds if $n = \Theta(p)$, so isoefficiency function is $\Theta(p^2)$, since $T_1 = \Theta(n^2)$

Scalability for Hypercube

- Total time for hypercube is

$$T_p = t_c\, n^2/p + (t_s + t_w\, n/\sqrt{p}\,)\, \log p$$
$$= t_c\, n^2/p + t_s\, \log p + t_w\, n\, (\log p)/\sqrt{p}$$

- To determine isoefficiency function, set

$$t_c\, n^2 \approx E\,(t_c\, n^2 + t_s\, p\, \log p + t_w\, n\, \sqrt{p}\, \log p)$$

which holds for large p if $n = \Theta(\sqrt{p}\, \log p)$, so isoefficiency function is $\Theta(p\,(\log p)^2)$, since $T_1 = \Theta(n^2)$

Scalability for 1-D Mesh

- Depending on network, time for communication phase with 1-D agglomeration is at least
  - 1-D mesh: $T_{\text{comm}} = (t_s + t_w\, n/p)\,(p-1)$
  - 2-D mesh: $T_{\text{comm}} = (t_s + t_w\, n/p)\, 2(\sqrt{p} - 1)$
  - hypercube: $T_{\text{comm}} = (t_s + t_w\, n/p)\, \log p$
- For 1-D agglomeration on 1-D mesh, total time is at least

$$T_p = t_c\, n^2/p + (t_s + t_w\, n/p)\,(p-1)$$
$$\approx t_c\, n^2/p + t_s\, p + t_w\, n$$

Scalability for 1-D Mesh

- To determine isoefficiency function, set

$$t_c\, n^2 \approx E\,(t_c\, n^2 + t_s\, p^2 + t_w\, n\, p)$$

which holds if $n = \Theta(p)$, so isoefficiency function is $\Theta(p^2)$, since $T_1 = \Theta(n^2)$

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

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

Matrix-Matrix Product

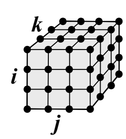
- Matrix-matrix product can be viewed as
  - n^2 inner products, or
  - sum of n outer products, or
  - n matrix-vector products

  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
- We will develop parallel algorithms for this problem directly, however

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 processes 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 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

Agglomeration with Block Mapping

1-D row 1-D col 2-D 3-D

Coarse-Grain 3-D Parallel Algorithm

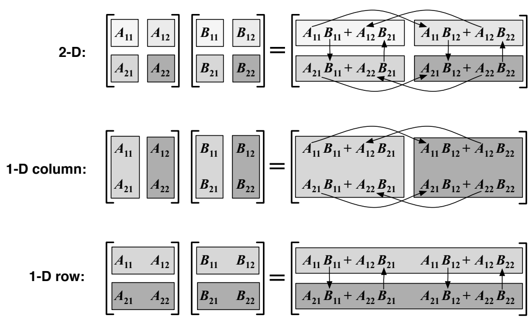
broadcast $A_{[i][k]}$ to ith process row { horizontal broadcast }

broadcast $B_{[k][j]}$ to jth process column { vertical broadcast }

$C_{[i][j]} = A_{[i][k]} B_{[k][j]}$ { local matrix product }

reduce $C_{[i][j]}$ across process layers { lateral sum reduction }

Agglomeration with Block Mapping



Coarse-Grain 2-D Parallel Algorithm

all-to-all bcast $A_{[i][j]}$ in ith process row { horizontal broadcast }
all-to-all bcast $B_{[i][j]}$ in jth process column { vertical broadcast }
$C_{[i][j]} = O$
for $k = 1, \ldots, \sqrt{p}$
  $C_{[i][j]} = C_{[i][j]} + A_{[i][k]} B_{[k][j]}$ { sum local products }
end

Fox Algorithm

- Algorithm just described requires excessive memory, since each process accumulates $\sqrt{p}$ blocks of both A and B
- One way to reduce memory requirements is to
  - broadcast blocks of A successively across process rows, and
  - circulate blocks of B in ring fashion vertically along process 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.

Cannon Algorithm

- Another approach, due to Cannon, 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 even less memory than Fox algorithm, but trickier to program because of shifts required
- Performance and scalability of Fox and Cannon algorithms are not significantly different from that of previous 2-D algorithm, but memory requirements are much less

Scalability 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

$$T_{\text{comp}} = t_c\,(n/\sqrt[3]{p}\,)^3 = t_c\,n^3/p$$

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

$$(t_s + t_w\,(n/\sqrt[3]{p})^2)\,(\sqrt[3]{p} - 1) \approx t_s\,p^{1/3} + t_w\,n^2/p^{1/3}$$

- Total time is therefore

$$T_p = t_c\,n^3/p + 3t_s\,p^{1/3} + 3t_w\,n^2/p^{1/3}$$

Scalability for 3-D Agglomeration

- To determine isoefficiency function, set

$$t_c\,n^3 \approx E\,(t_c\,n^3 + 3t_s\,p^{4/3} + 3t_w\,n^2 p^{2/3})$$

which holds for large p if $n = \Theta(p^{2/3})$, so isoefficiency function is $\Theta(p^2)$, since $T_1 = \Theta(n^3)$

- For hypercube, total time becomes

$$T_p = t_c\,n^3/p + t_s\,\log p + t_w\,n^2(\log p)/p^{2/3}$$

which leads to isoefficiency function of $\Theta(p\,(\log p)^3)$

Scalability for 2-D Agglomeration

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

$$T_{\text{comp}} = t_c\sqrt{p}\,(n/\sqrt{p}\,)^3 = t_c\,n^3/p$$

- For 2-D mesh, communication time for broadcasts along rows and columns is

$$\begin{aligned} T_{\text{comm}} &= (t_s + t_w\,n^2/p)(\sqrt{p} - 1) \\ &\approx t_s\sqrt{p} + t_w\,n^2/\sqrt{p} \end{aligned}$$

assuming horizontal and vertical broadcasts can overlap (multiply by two otherwise)

Scalability for 2-D Agglomeration

- Total time for 2-D mesh is

$$T_p \approx t_c\,n^3/p + t_s\sqrt{p} + t_w\,n^2/\sqrt{p}$$

- To determine isoefficiency function, set

$$t_c\,n^3 \approx E\,(t_c\,n^3 + t_s\,p^{3/2} + t_w\,n^2\sqrt{p}\,)$$

which holds for large p if $n = \Theta(\sqrt{p}\,)$, so isoefficiency function is $\Theta(p^{3/2})$, since $T_1 = \Theta(n^3)$

Scalability for 1-D Agglomeration

- For 1-D agglomeration on 1-D mesh, total time is

$$\begin{aligned} T_p &= t_c\,n^3/p + (t_s + t_w\,n^2/p)\,(p - 1) \\ &\approx t_c\,n^3/p + t_s\,p + t_w\,n^2 \end{aligned}$$

- To determine isoefficiency function, set

$$t_c\,n^3 \approx E\,(t_c\,n^3 + t_s\,p^2 + t_w\,n^2\,p)$$

which holds for large p if $n = \Theta(p)$, so isoefficiency function is $\Theta(p^3)$ since $T_1 = \Theta(n^3)$

Communication vs. Memory Tradeoff

- Communication volume 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
- Block-recursive 3-D algorithm can reduce communication volume by factor of $p^{-1/6}$ while increasing memory usage by factor of $p^{1/3}$
- Recently, "2.5-D" algorithms have been developed that interpolate between 2-D and 3-D algorithms, using partial storage replication to reduce communication volume to whatever extent available memory allows

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