

Parallel scalability

Victor Eijkhout

PCSE 2016

High performance linear algebra

Justification

Bringing architecture-awareness to linear algebra, we discuss how high performance results from using the right formulation and implementation of algorithms.

Table of Contents

- 1 Collectives as building blocks; complexity
- 2 Scalability analysis of dense matrix-vector product
- 3 Sparse matrix-vector product
- 4 Latency hiding / communication minimizing
- 5 Multicore block algorithms

Simple model of parallel computation

- α : message latency
- β : time per word (inverse of bandwidth)
- γ : time per floating point operation

Send n items and do m operations:

$$cost = \alpha + \beta \cdot n + \gamma \cdot m$$

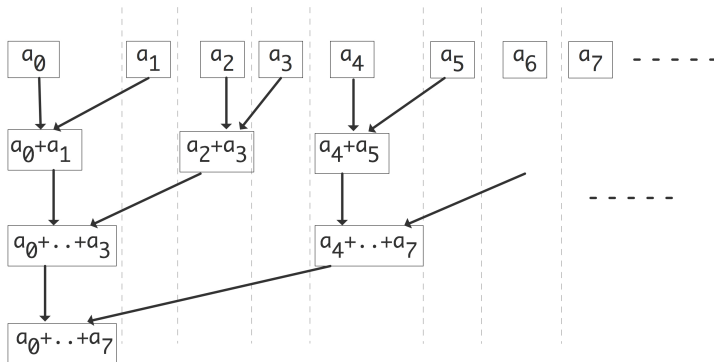
Pure sends: no γ term,

pure computation: no α, β terms,

sometimes mixed: reduction

Model for collectives

- One simultaneous send and receive:
- doubling of active processors
- collectives have a $\alpha \log_2 p$ cost component



Broadcast

	$t = 0$	$t = 1$	$t = 2$
p_0	$x_0 \downarrow, x_1 \downarrow, x_2 \downarrow, x_3 \downarrow$	$x_0 \downarrow, x_1 \downarrow, x_2 \downarrow, x_3 \downarrow$	x_0, x_1, x_2, x_3
p_1		$x_0 \downarrow, x_1 \downarrow, x_2 \downarrow, x_3 \downarrow$	x_0, x_1, x_2, x_3
p_2			x_0, x_1, x_2, x_3
p_3			x_0, x_1, x_2, x_3

On $t = 0$, p_0 sends to p_1 ; on $t = 1$ p_0, p_1 send to p_2, p_3 .

Optimal complexity:

$$\lceil \log_2 p \rceil \alpha + n\beta.$$

Actual complexity:

$$\lceil \log_2 p \rceil (\alpha + n\beta).$$

Good enough for short vectors.

Reduce

Optimal complexity:

$$\lceil \log_2 p \rceil \alpha + n\beta + \frac{p-1}{p} \gamma n.$$

Spanning tree algorithm:

	$t = 1$	$t = 2$	$t = 3$
p_0	$x_0^{(0)}, x_1^{(0)}, x_2^{(0)}, x_3^{(0)}$	$x_0^{(0:1)}, x_1^{(0:1)}, x_2^{(0:1)}, x_3^{(0:1)}$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$
p_1	$x_0^{(1)} \uparrow, x_1^{(1)} \uparrow, x_2^{(1)} \uparrow, x_3^{(1)} \uparrow$		
p_2	$x_0^{(2)}, x_1^{(2)}, x_2^{(2)}, x_3^{(2)}$	$x_0^{(2:3)} \uparrow, x_1^{(2:3)} \uparrow, x_2^{(2:3)} \uparrow, x_3^{(2:3)} \uparrow$	
p_3	$x_0^{(3)} \uparrow, x_1^{(3)} \uparrow, x_2^{(3)} \uparrow, x_3^{(3)} \uparrow$		

Running time

$$\lceil \log_2 p \rceil \left(\alpha + n\beta + \frac{p-1}{p} \gamma n \right).$$

Good enough for short vectors.

Long vector broadcast

Combine scatter and bucket-allgather:

	$t = 0$	$t = 1$	<i>etcetera</i>
p_0	$x_0 \downarrow$	x_0	$x_3 \downarrow$ x_0, x_2, x_3
p_1	$x_1 \downarrow$	$x_0 \downarrow, x_1$	x_0, x_1, x_3
p_2	$x_2 \downarrow$	$x_1 \downarrow, x_2$	x_0, x_1, x_2
p_3	$x_3 \downarrow$	$x_2 \downarrow, x_3$	x_1, x_2, x_3

Complexity becomes

$$p\alpha + \beta n(p-1)/p$$

better if n large

Allgather

Gather n elements: each processor owns n/p ;
optimal running time

$$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p} n \beta.$$

	$t = 1$	$t = 2$	$t = 3$
p_0	$x_0 \downarrow$	$x_0 x_1 \downarrow$	$x_0 x_1 x_2 x_3$
p_1	$x_1 \uparrow$	$x_0 x_1 \downarrow$	$x_0 x_1 x_2 x_3$
p_2	$x_2 \downarrow$	$x_2 x_3 \uparrow$	$x_0 x_1 x_2 x_3$
p_3	$x_3 \uparrow$	$x_2 x_3 \uparrow$	$x_0 x_1 x_2 x_3$

Same time as gather, half of gather-and-broadcast.

Reduce-scatter

	$t = 1$	$t = 2$	$t = 3$
p_0	$x_0^{(0)}, x_1^{(0)}, x_2^{(0)} \downarrow, x_3^{(0)} \downarrow$	$x_0^{(0:2:2)}, x_1^{(0:2:2)} \downarrow$	$x_0^{(0:3)}$
p_1	$x_0^{(1)}, x_1^{(1)}, x_2^{(1)} \downarrow, x_3^{(1)} \downarrow$	$x_0^{(1:3:2)} \uparrow, x_1^{(1:3:2)}$	$x_1^{(0:3)}$
p_2	$x_0^{(2)} \uparrow, x_1^{(2)} \uparrow, x_2^{(2)}, x_3^{(2)}$	$x_2^{(0:2:2)}, x_3^{(0:2:2)} \downarrow$	$x_2^{(0:3)}$
p_3	$x_0^{(3)} \uparrow, x_1^{(3)} \uparrow, x_2^{(3)}, x_3^{(3)}$	$x_0^{(1:3:2)} \uparrow, x_1^{(1:3:2)}$	$x_3^{(0:3)}$

$$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p} n(\beta + \gamma).$$

Table of Contents

- 1 Collectives as building blocks; complexity
- 2 Scalability analysis of dense matrix-vector product
- 3 Sparse matrix-vector product
- 4 Latency hiding / communication minimizing
- 5 Multicore block algorithms

Parallel matrix-vector product; general

- Assume a division by block rows
- Every processor p has a set of row indices I_p

Mvp on processor p :

$$\forall_i: y_i = \sum_j a_{ij} x_j$$

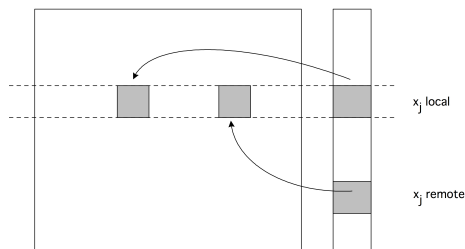
$$\forall_i: y_i = \sum_q \sum_{j \in I_q} a_{ij} x_j$$

Local and remote operations

Local and remote parts:

$$\forall_i: y_i = \sum_{j \in I_p} a_{ij} x_j + \sum_{q \neq p} \sum_{j \in I_q} a_{ij} x_j$$

Local part I_p can be executed right away, I_q requires communication.



Combine:
communication and computation;
only used in the sparse case

Note possible overlap

Exercise

How much can overlap help you?

Dense MVP

- Separate communication and computation:
- first allgather
- then matrix-vector product

Cost computation 1.

Algorithm:

Step	Cost (lower bound)
Allgather x_i so that x is available on all nodes	
Locally compute $y_i = A_i x$	$\approx 2 \frac{n^2}{P} \gamma$

Allgather

Assume that data arrives over a binary tree:

- latency $\alpha \log_2 P$
- transmission time, receiving n/P elements from $P - 1$ processors

Algorithm with cost:

Step	Cost (lower bound)
Allgather x_i so that x is available on all nodes	$\lceil \log_2(P) \rceil \alpha + \frac{P-1}{P} n\beta \approx \log_2(P)\alpha + n\beta$
Locally compute $y_i = A_i x$	$\approx 2 \frac{n^2}{P} \gamma$

Parallel efficiency

$$E_p^{1\text{D-row}}(n) = \frac{S_p^{1\text{D-row}}(n)}{p} = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{p}{2n} \frac{\beta}{\gamma}}.$$

Strong scaling, weak scaling?

Optimistic scaling

Processors fixed, problem grows:

$$E_p^{\text{1D-row}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{p}{2n} \frac{\beta}{\gamma}}.$$

Roughly $E_p \sim 1 - n^{-1}$

Strong scaling

Problem fixed, $p \rightarrow \infty$

$$E_p^{\text{1D-row}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{p}{2n} \frac{\beta}{\gamma}}.$$

Strong scaling

Problem fixed, $p \rightarrow \infty$

$$E_p^{\text{1D-row}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{p}{2n} \frac{\beta}{\gamma}}.$$

Roughly $E_p \sim p^{-1}$

Weak scaling

Memory fixed:

$$M = n^2/p$$

$$E_p^{\text{1D-row}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{p}{2n} \frac{\beta}{\gamma}} = \frac{1}{1 + \frac{\log_2(p)}{2M} \frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2\sqrt{M}} \frac{\beta}{\gamma}}$$

Weak scaling

Memory fixed:

$$M = n^2/p$$

$$E_p^{\text{1D-row}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{p}{2n} \frac{\beta}{\gamma}} = \frac{1}{1 + \frac{\log_2(p)}{2M} \frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2\sqrt{M}} \frac{\beta}{\gamma}}$$

Does not scale: $E_p \sim 1/\sqrt{p}$

problem in β term: too much communication

Two-dimensional partitioning

x_0 a_{00} a_{10} a_{20} a_{30}	a_{01} a_{11} a_{21} a_{31}	a_{02} a_{12} a_{22} a_{32}	y_0	x_3 a_{03} a_{13} a_{23} a_{33}	a_{04} a_{14} a_{24} a_{34}	a_{05} a_{15} a_{25} a_{35}	y_1	x_6 a_{06} a_{16} a_{26} a_{37}	a_{07} a_{17} a_{27} a_{37}	a_{08} a_{18} a_{28} a_{38}	y_2	x_9 a_{09} a_{19} a_{29} a_{39}	$a_{0,10}$ $a_{1,10}$ $a_{2,10}$ $a_{3,10}$	$a_{0,11}$ $a_{1,11}$ $a_{2,11}$ $a_{3,11}$	y_3
x_1 a_{40} a_{50} a_{60} a_{70}	a_{41} a_{51} a_{61} a_{71}	a_{42} a_{52} a_{62} a_{72}	y_4	x_4 a_{43} a_{53} a_{63} a_{73}	a_{44} a_{54} a_{64} a_{74}	a_{45} a_{55} a_{65} a_{75}	y_5	x_7 a_{46} a_{56} a_{66} a_{77}	a_{47} a_{57} a_{67} a_{77}	a_{48} a_{58} a_{68} a_{78}	y_6	x_{10} a_{49} a_{59} a_{69} a_{79}	$a_{4,10}$ $a_{5,10}$ $a_{6,10}$ $a_{7,10}$	$a_{4,11}$ $a_{5,11}$ $a_{6,11}$ $a_{7,11}$	y_7
x_2 a_{80} a_{90} $a_{10,0}$ $a_{11,0}$	a_{81} a_{91} $a_{10,1}$ $a_{11,1}$	a_{82} a_{92} $a_{10,2}$ $a_{11,2}$	y_8	x_5 a_{83} a_{93} $a_{10,3}$ $a_{11,3}$	a_{84} a_{94} $a_{10,4}$ $a_{11,4}$	a_{85} a_{95} $a_{10,5}$ $a_{11,5}$	y_9	x_8 a_{86} a_{96} $a_{10,6}$ $a_{11,7}$	a_{87} a_{97} $a_{10,7}$ $a_{11,7}$	a_{88} a_{98} $a_{10,8}$ $a_{11,8}$	y_{10}	x_{11} a_{89} a_{99} $a_{10,9}$ $a_{11,9}$	$a_{8,10}$ $a_{9,10}$ $a_{10,10}$ $a_{11,10}$	$a_{8,11}$ $a_{9,11}$ $a_{10,11}$ $a_{11,11}$	y_{11}

Two-dimensional partitioning

x_0 a_{00} a_{01} a_{02} y_0 a_{10} a_{11} a_{12} a_{20} a_{21} a_{22} a_{30} a_{31} a_{32}	x_3 y_1	x_6 y_2	x_9 y_3
$x_1 \uparrow$ y_4	x_4 y_5	x_7 y_6	x_{10} y_7
$x_2 \uparrow$ y_8	x_5 y_9	x_8 y_{10}	x_{11} y_{11}

Key to the algorithm

- Consider block (i, j)
- it needs to multiply by the x s in column j
- it produces part of the result of row i

Algorithm

- Collecting x_j on each processor p_{ij} by an *allgather* inside the processor columns.
- Each processor p_{ij} then computes $y_{ij} = A_{ij}x_j$.
- Gathering together the pieces y_{ij} in each processor row to form y_i , distribute this over the processor row: combine to form a *reduce-scatter*.
- Setup for the next A or A^t product

Analysis 1.

Step	Cost (lower bound)
Allgather x_i 's within columns	$\lceil \log_2(r) \rceil \alpha + \frac{r-1}{p} n \beta$
Perform local matrix-vector multiply	$\approx \log_2(r) \alpha + \frac{n}{c} \beta$
Reduce-scatter y_i 's within rows	$\approx 2 \frac{n^2}{p} \gamma$

Reduce-scatter

Time:

$$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p} n(\beta + \gamma).$$

Step	Cost (lower bound)
Allgather x_i 's within columns	$\lceil \log_2(r) \rceil \alpha + \frac{r-1}{p} n \beta$ $\approx \log_2(r) \alpha + \frac{n}{c} \beta$
Perform local matrix-vector multiply	$\approx 2 \frac{n^2}{p} \gamma$
Reduce-scatter y_i 's within rows	$\lceil \log_2(c) \rceil \alpha + \frac{c-1}{p} n \beta + \frac{c-1}{p} m \gamma$ $\approx \log_2(r) \alpha + \frac{n}{c} \beta + \frac{n}{c} \gamma$

Efficiency

Let $r = c = \sqrt{n}$, then

$$E_p^{\sqrt{p} \times \sqrt{p}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n} \frac{(2\beta + \gamma)}{\gamma}}$$

Strong scaling

Same story as before for $p \rightarrow \infty$:

$$E_p^{\sqrt{p} \times \sqrt{p}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n} \frac{(2\beta + \gamma)}{\gamma}} \sim p^{-1}$$

No strong scaling

Weak scaling

Constant memory $M = n^2/p$:

$$E_p^{\sqrt{p} \times \sqrt{p}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n} \frac{(2\beta + \gamma)}{\gamma}}$$

Weak scaling

Constant memory $M = n^2/p$:

$$E_p^{\sqrt{p} \times \sqrt{p}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n} \frac{(2\beta + \gamma)}{\gamma}} = \frac{1}{1 + \frac{\log_2(p)}{2M} \frac{\alpha}{\gamma} + \frac{1}{2\sqrt{M}} \frac{(2\beta + \gamma)}{\gamma}}$$

Weak scaling

Constant memory $M = n^2/p$:

$$E_p^{\sqrt{p} \times \sqrt{p}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n} \frac{(2\beta + \gamma)}{\gamma}} = \frac{1}{1 + \frac{\log_2(p)}{2M} \frac{\alpha}{\gamma} + \frac{1}{2\sqrt{M}} \frac{(2\beta + \gamma)}{\gamma}}$$

Weak scaling:

for $p \rightarrow \infty$ this is $\approx 1/\log_2 P$:

only slowly decreasing.

LU factorizations

- Needs a cyclic distribution
- This is very hard to program, so:
- Scalapack, 1990s product, not extendible, impossible interface
- Elemental: 2010s product, extendible, nice user interface (and it is way faster)

Table of Contents

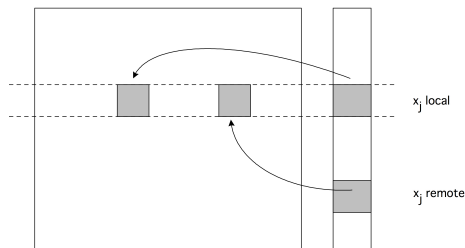
- 1 Collectives as building blocks; complexity
- 2 Scalability analysis of dense matrix-vector product
- 3 Sparse matrix-vector product**
- 4 Latency hiding / communication minimizing
- 5 Multicore block algorithms

Local and remote operations

Local and remote parts:

$$\forall_i: y_i = \sum_{j \in \text{local}} a_{ij} x_j + \sum_{j \in \text{remote}} a_{ij} x_j$$

Local part l_p can be executed right away, l_q requires communication.



Combine:

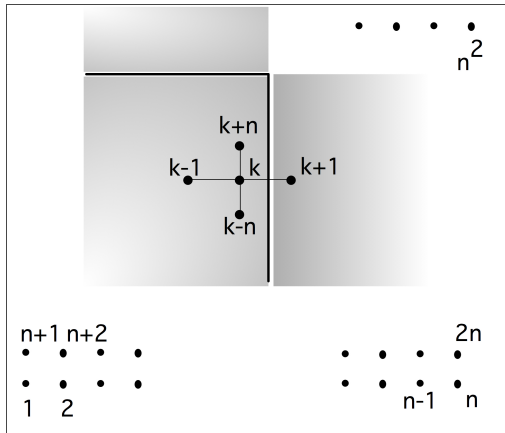
Note possible overlap communication and computation;
only used in the sparse case

Sparse matrix operations

- Traditional: PDE, discussed next
- New: graph algorithms and big data, discussed later

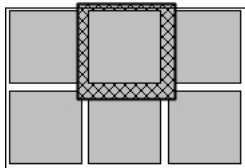
Operator view of spmvp

Difference stencil



Parallel operator view

induces ghost region:



Limited number of neighbours, limited buffer space

Matrix vs operator view

- Domain partitioning: processor 'owns' variable i
- owns all connections from i to other j s
- \Rightarrow processor owns whole matrix row
- \Rightarrow 1D partitioning of the matrix, always

$$A = \left(\begin{array}{cccc|ccc|ccc} 4 & -1 & & & 0 & -1 & & & 0 & & \\ -1 & 4 & -1 & & & & -1 & & & & \\ & & \ddots & \ddots & \ddots & & & & & & \\ & & & \ddots & \ddots & & & & & & \\ & & & & \ddots & & & & & & \\ & & & & & -1 & 4 & & & & \\ 0 & & & & -1 & 4 & 0 & & & -1 & \\ \hline -1 & & & & 0 & 4 & -1 & & & -1 & \\ & -1 & & & & -1 & 4 & -1 & & & \\ & & \uparrow & \ddots & & \uparrow & \uparrow & \uparrow & & & \\ & & k-n & & & k-1 & k & k+1 & & -1 & \\ & & & & & & & & -1 & 4 & \\ \hline & & & & & & & & & & \\ & & & & & \ddots & & & & & \\ & & & & & & & & & \ddots & \end{array} \right) \quad (1)$$

Scaling

- Same phenomenon as with dense matrix:
- n^2 variables, memory needed is cn^2/p
- 1D partitioning *of domain* does not weakly scale
 - Message size is one line: n
 - is $\sqrt{p}\sqrt{M}$, goes up with processors
- 2D partitioning *of domain* scales weakly.
 - message size $n/\sqrt{p} = \sqrt{M}$
 - constant in M

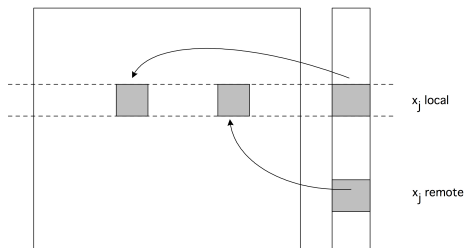
MPI implementation

- Assume general communication structure:
neighbour processors can not statically be determined
- Assume no structural symmetry
- For matrix-vector product:
each processor issues send and receive requests
- Problem: receives are easy, sends are hard
- *Inspector-executor*: one-time discovery of structure,
followed by many executions

Asymmetry in reasoning

Say

- Processor owns row i , $a_{ij} \neq 0$, processor does not own j



- Needed: message from j to i
- Processor i can discover this
- Processor j not in general

Reduce-scatter

Make $p \times p$ matrix C :

$$C_{ij} = \begin{cases} 1 & i \text{ receives from } j \\ 0 & \text{otherwise} \end{cases}$$

Then

$$s_j = \sum_i C_{ij}$$

number of messages sent by j

Reduce-scatter, proc i has C_{i*}

Reason for more cleverness

- The above is collective, implies synchronization
- temp space $O(P)$
- can we get this down to $O(\#neighbours)$?
- *can we detect that we have received all requests without knowing how many to expect?*

MPI 3 non-blocking barrier

- Barrier: test that every process has reached this point
blocking
- Ibarrier: non-blocking test
- Ibarrier calls does not block, yields `MPI_Request` pointer
- Use Wait or Test on the request

```
int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)
```

DTD algorithms

‘Distributed Termination Detection’

used to be (extremely) tricky, now easy with MPI 3

Establish sparse neighbours:

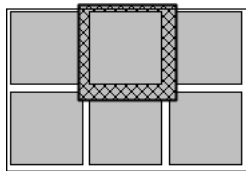
- Send all your own requests (Isend)
- Loop:
 - Test on send requests; if all done, enter non-blocking barrier
 - Probe for request messages, receive if there is something
 - If you're in the barrier, also test for the barrier to complete
- \Rightarrow if the barrier completes, you have received all your requests

(For safety, use `MPI_Issend`)

Table of Contents

- 1 Collectives as building blocks; complexity
- 2 Scalability analysis of dense matrix-vector product
- 3 Sparse matrix-vector product
- 4 Latency hiding / communication minimizing
- 5 Multicore block algorithms

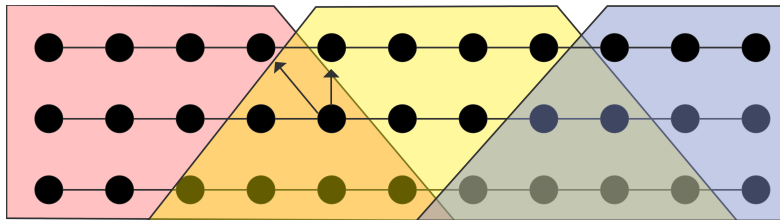
Sparse matrix vector product induces ghost region;



Is this important?

- No: surface/volume argument
- Yes: communication is much slower than computation
- Case of multiple products is considerably more interesting

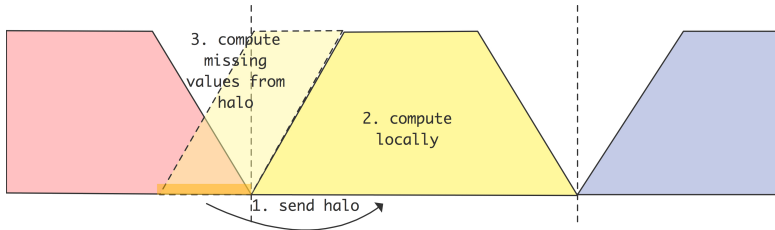
Optimization of multiple products



Recursive closure of the halo:

- Only one latency
- On-node code can be optimized (caching or cache-oblivious)

Latency hiding



Overlap halo transfer with local computation:
programming complication

Communication minimizing

Optimal solution ('communication avoiding'):

- Half the communication
- half the redundant work
- Complication: local work done in two stages

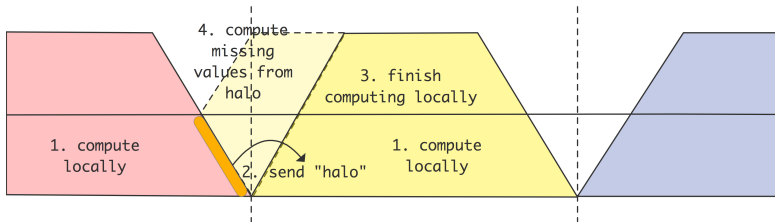


Table of Contents

- 1 Collectives as building blocks; complexity
- 2 Scalability analysis of dense matrix-vector product
- 3 Sparse matrix-vector product
- 4 Latency hiding / communication minimizing
- 5 Multicore block algorithms

Cholesky algorithm

$$\text{Chol} \begin{pmatrix} A_{11} & A_{21}^t \\ A_{21} & A_{22} \end{pmatrix} = LL^t \quad \text{where} \quad L = \begin{pmatrix} L_{11} & 0 \\ \tilde{A}_{21} & \text{Chol}(A_{22} - \tilde{A}_{21}\tilde{A}_{21}^t) \end{pmatrix}$$

and where $\tilde{A}_{21} = A_{21}L_{11}^{-t}$, $A_{11} = L_{11}L_{11}^t$.

Implementation

for $k = 1, \text{nblocks}$:

Chol: factor $L_k L_k^t \leftarrow A_{kk}$

Trsm: solve $\tilde{A}_{>k,k} \leftarrow A_{>k,k} L_k^{-t}$

Gemm: form the product $\tilde{A}_{>k,k} \tilde{A}_{>k,k}^t$

Syrk: symmetric rank- k update $A_{>k,>k} \leftarrow A_{>k,>k} - \tilde{A}_{>k,k} \tilde{A}_{>k,k}^t$

Blocked implementation

finished		
A_{kk}	$A_{k,k+1}$	$A_{k,k+2} \dots$
$A_{k+1,k}$	$A_{k+1,k+1}$	$A_{k+1,k+2} \dots$
$A_{k+2,k}$	$A_{k+2,k+2}$	
\vdots	\vdots	

Extra level of inner loops:

for $k = 1, \text{nblocks}$:

Chol: factor $L_k L_k^t \leftarrow A_{kk}$

for $\ell > k$:

Trsm: solve $\tilde{A}_{\ell,k} \leftarrow A_{\ell,k} L_k^{-t}$

for $\ell_1, \ell_2 > k$:

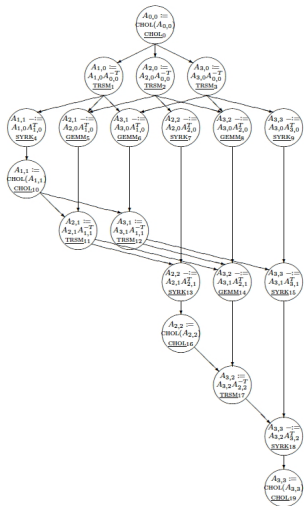
Gemm: form the product $\tilde{A}_{\ell_1,k} \tilde{A}_{\ell_2,k}^t$

for $\ell_1, \ell_2 > k, \ell_1 \leq \ell_2$:

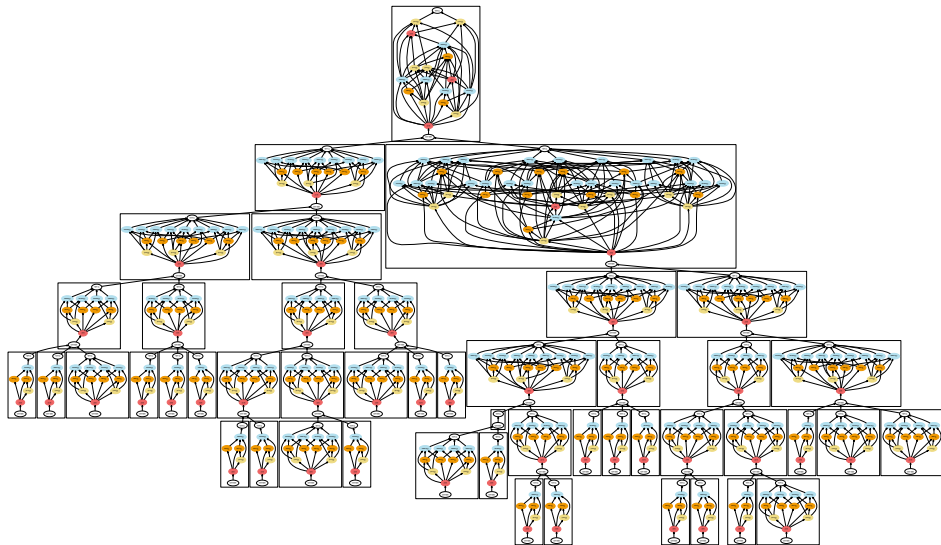
Syrk: symmetric rank- k update

$A_{\ell_1,\ell_2} \leftarrow A_{\ell_1,\ell_2} - \tilde{A}_{\ell_1,k} \tilde{A}_{\ell_2,k}^t$

You can graph this



Sometimes...



DAG schedulers

- Directed Acyclic Graph (dataflow)
- Each node has dependence on other nodes, can execute when dependencies available
- Quark/DaGue (TN): dependence on memory area written pretty much limited to dense linear algebra
- OpenMP has a pretty good scheduler
- Distributed memory scheduling is pretty hard