Parallel Sparse Matrix–Vector Multiplication

(PSC §4.3)
Option 1: represent first, distribute later

First build a global data structure to represent the sparse matrix, then distribute it over the processors:

- A parallelising compiler would do this.
- Requires **global collaboration** between the processors.
- Simple operations become complicated: in a linked list, 3 processors must work together and communicate to insert a new nonzero.
Option 2: distribute first, represent later

Distribute the matrix first, and then let each processor represent the local nonzeros:

- This assigns subsets of nonzeros to processors.
- The subsets form a partitioning of the nonzero set: subsets are disjoint and together they contain all nonzeros.
- Sequential sparse data structures can be used.
- Simple operations remain simple: insertion and deletion are local operations without communication.
- This is the preferred approach.
The most general scheme maps nonzeros to processors,

\[ a_{ij} \mapsto P(\phi(i,j)), \text{ for } 0 \leq i, j < n \text{ and } a_{ij} \neq 0, \]

where \( 0 \leq \phi(i,j) < p \).

- **Zeros** are not assigned to processors. For convenience, we define \( \phi(i,j) = -1 \) if \( a_{ij} = 0 \).

- Here, we use a 1D processor numbering.
Parallel sparse matrix–vector multiplication

Distribution of matrix and vectors

Global view

Local view

\( p = 2, \ n = 5, \ nz = 13. \) \( P(0) \) grey cells, \( P(1) \) black cells. Matrix distribution is non-Cartesian.
Where to compute $a_{ij} \cdot v_j$?

- Usually, there are many more nonzeros than vector components, $nz(A) \gg n$, so move the vector components $v_j$ to the nonzeros $a_{ij}$, not vice versa.
- Add local products $a_{ij}v_j$ belonging to the same row $i$. Result on $P(s)$ is the local contribution $u_{is}$ to $u_i$.
- Result $u_{is}$ is sent to the owner of $u_i$.
- Thus, we do not communicate elements of $A$, but only components of $v$ and contributions to components of $u$. 


How to distribute the vectors?

- In many iterative solvers, the same vector is \textit{repeatedly multiplied} by a matrix $A$.
- Usually, a few vector operations are interspersed, such as DAXPYs $y := \alpha x + y$ or inner products $\alpha := x^T y$.
- All vectors should then be distributed in the same way: $\text{distr}(u) = \text{distr}(v)$ for the operation $u := Av$.
- Sometimes, however, we compute $A^T A v$. The \textit{output vector} for $A$ is then taken as the \textit{input vector} for $A^T$.
- Now, we do not need to revert immediately to the input distribution. We allow $\text{distr}(u) \neq \text{distr}(v)$.
- We map vector components to processors by

\[ u_i \mapsto P(\phi_u(i)), \text{ for } 0 \leq i < n. \]
Deriving a parallel algorithm

- Once we have chosen the data distribution and decided to compute the products $a_{ij}v_j$ on the processor that owns $a_{ij}$, the parallel algorithm follows naturally.

- The main computation for processor $P(s)$ is multiplying each local nonzero element $a_{ij}$ by $v_j$ and adding the result into a local partial sum,

$$u_{is} = \sum_{0 \leq j < n, \phi(i,j) = s} a_{ij}v_j.$$

- Sparsity is exploited:
  - only terms with $a_{ij} \neq 0$ are summed;
  - only local partial sums $u_{is}$ are computed for which
    $$\{j : 0 \leq j < n \land \phi(i,j) = s\} \neq \emptyset.$$
Row index set

(a) On \( P(0) \), row 4 is empty. On \( P(1) \), row 1 is empty.

(b) Index set \( I_s \) of rows that are locally nonempty in \( P(s) \) is

\[
I_s = \{ i : 0 \leq i < n \land (\exists j : 0 \leq j < n \land \phi(i,j) = s) \}
\]

- \( I_0 = \{0, 1, 2, 3\} \) and \( I_1 = \{0, 2, 3, 4\} \).
Local sparse matrix–vector multiplication

\[ l_s = \{ i : 0 \leq i < n \wedge (\exists j : 0 \leq j < n \wedge \phi(i,j) = s) \} \]

(1) \{ Local sparse matrix–vector multiplication \}

\textbf{for all} \ i \in l_s \ \textbf{do}

\[ u_{is} := 0; \]

\textbf{for all} \ j : 0 \leq j < n \wedge \phi(i,j) = s \ \textbf{do}

\[ u_{is} := u_{is} + a_{ij}v_j; \]
Data structure for local sparse matrix

- Compressed row storage (CRS) suits row-oriented local matrix–vector multiplication.
- CRS must be adapted to avoid overhead of many empty rows, which typically occurs if \( c \ll p \).
- We number the nonempty local rows from 0 to \( |l_s| - 1 \). The corresponding indices \( i \) are the local indices.
- The original global indices from the set \( l_s \) are stored in increasing order in an array \( \text{rowindex} \) of length \( |l_s| \):
  \[
  i = \text{rowindex}[i].
  \]
- Address of first local nonzero of row \( i \) is \( \text{start}[i] \).
Index set $J_s$ of columns that are locally nonempty in $P(s)$ is

$$J_s = \{j : 0 \leq j < n \land (\exists i : 0 \leq i < n \land \phi(i, j) = s)\}$$

$J_0 = \{0, 1, 2\}$ and $J_1 = \{2, 3, 4\}$. 
Fanout

\[ J_s = \{ j : 0 \leq j < n \land (\exists i : 0 \leq i < n \land \phi(i,j) = s) \} \]

(0) \{ Fanout \}

\begin{verbatim}
for all \( j \in J_s \) do
  get \( v_j \) from \( P(\phi_v(j)) \);
\end{verbatim}

- The receiver knows (from its index set \( J_s \)) that it needs the vector component \( v_j \). The sender is unaware of this. Thus, the receiver initiates the communication by using a ‘get’.
- In dense algorithms, communication patterns are predictable and thus known to every processor, so that we only need ‘put’ primitives.
- In sparse algorithms, we also have to use ‘get’ primitives.
Fanin and final summation

(2) \{Fanin\} 
\textbf{for all} \ i \in I_s \ \textbf{do} 
\quad \text{put } u_{is} \ \text{in } P(\phi_u(i));

(3) \{Summation of nonzero partial sums\} 
\textbf{for all} \ i : 0 \leq i < n \ \& \ \phi_u(i) = s \ \textbf{do} 
\quad u_i := 0;
\quad \textbf{for all} \ t : 0 \leq t < p \ \& \ u_{it} \neq 0 \ \textbf{do} 
\quad u_i := u_i + u_{it};
Communication

- **Vertical arrows**: communication of components $v_j$.
  - $v_0$ is sent from $P(1)$ to $P(0)$, because of the nonzeros $a_{10} = 4$ and $a_{30} = 6$ owned by $P(0)$.
  - $v_1, v_3, v_4$ need not be sent.

- **Horizontal arrows**: communication of partial sums $u_{is}$.
Cost analysis

Wat kost het? *(Dutch for: How much does it cost?)*

- Answer depends on matrix $A$ and distributions $\phi, \phi_v, \phi_u$.
- We can obtain an upper bound on the BSP cost, assuming:
  - the matrix nonzeros are evenly spread over the processors, each processor having $\frac{cn}{p}$ nonzeros;
  - the vector components are also evenly spread, each processor having $\frac{n}{p}$ components.
- Bound may be far too pessimistic for particular distributions that are well-tailored to the matrix.
Cost of separate supersteps

(0): $P(s)$ must receive at most all $n$ components $v_j$, but not the $\frac{n}{p}$ local components, so that $h_r = n - \frac{n}{p}$. $h_s = \frac{n}{p}(p - 1)$, because the $\frac{n}{p}$ local components must be sent to all the other $p - 1$ processors.

\[ T_{(0)} = (1 - \frac{1}{p})ng + l. \]

(1): 2 flops are needed for each local nonzero.

\[ T_{(1)} = \frac{2cn}{p} + l. \]
Cost of separate supersteps (cont’d)

(2): Similar to (0).

\[ T_{(2)} = (1 - \frac{1}{p})ng + l. \]

(3): Each of the \( \frac{n}{p} \) local vector components is computed by adding at most \( p \) partial sums.

\[ T_{(3)} = n + l. \]

Total BSP cost is bounded by

\[ T_{\text{MV}} \leq \frac{2cn}{p} + n + 2(1 - \frac{1}{p})ng + 4l. \]
Efficient computation

\[ T_{MV} \leq \frac{2cn}{p} + n + 2\left(1 - \frac{1}{p}\right)ng + 4l. \]

- Computation is **efficient** if \( \frac{2cn}{p} > 2ng \), i.e., \( c > pg \). But this happens rarely: **only for very dense matrices**.
- The number of nonzeros per row \( c \), and not the density \( d \), determines the efficiency directly.
- To make the computation efficient for smaller \( c \), we can:
  - use a Cartesian distribution and exploit its 2D nature;
  - refine the general distribution using an automatic procedure to detect the **underlying matrix structure**;
  - exploit properties of **specific matrix classes**, such as random sparse matrices and Laplacian matrices.
Communication volume

- Communication volume $V$ of an algorithm is the total number of data words sent. It depends on $\phi$, $\phi_u$, $\phi_v$.
- For a given $\phi$, obtain a lower bound $V_\phi$ on $V$ by counting:
  - the number of processors $p_i$ that have a nonzero $a_{ij}$ in matrix row $i$; at least $p_i - 1$ processors must send a contribution $u_{is}$.
  - the number of processors $q_j$ that has a nonzero $a_{ij}$ in matrix column $j$.

$$V_\phi = \sum_{0 \leq i < n, \ p_i \geq 1} (p_i - 1) + \sum_{0 \leq j < n, \ q_j \geq 1} (q_j - 1).$$

- An upper bound is $V_\phi + 2n$, because in the worst case all $n$ components $u_i$ are owned by processors without a nonzero in row $i$, and similar for the components $v_j$. 
Summary

- **Distribute first**, represent later.
- Most general mapping of nonzeros and vector components to processors:
  \[
  a_{ij} \mapsto P(\phi(i,j)), \text{ for } 0 \leq i, j < n \text{ and } a_{ij} \neq 0,
  \]
  \[
  u_i \mapsto P(\phi_u(i)), \text{ for } 0 \leq i < n.
  \]
- We have derived a parallel algorithm with 4 supersteps: **fanout**, local matrix–vector multiplication, **fanin**, summation of partial sums.
- The row index set $I_s$ and column index set $J_s$ are used for exploiting the sparsity in the algorithm.
- We encountered the first **absolutely necessary** use of a ‘get’ primitive.