Solutions to the final questions of the videos of Parallel Scientific Computation, second edition

Rob H. Bisseling

August 29, 2022

Below we denote the book by [PSC2].

- **Video: BSP model.** Each processor adds 100 numbers, costing 100 flop time units (assuming we initialize the sum as 0). Then each processor sends its own sum to the responsible processor in a 9-relation. This processor then adds 10 numbers. There are 3 supersteps: computation, communication, computation. The total cost is thus

$$110 + 9g + 3l.$$

- **Video: Data distributions.** We choose the cyclic distribution, because it has a better load balance than the block distribution. Since the amount of work for a vector component x_i grows quadratically with i, the block distribution leads to an overloaded last processor $P(p-1)$. We can analyse this more precisely: assume that p divides n. By using the formula

$$\sum_{i=0}^{k} i^2 = \frac{k(k+1)(2k+1)}{6},$$

we see that the total amount of work is about $\frac{n^3}{3}$ and the average amount per processor $\frac{n^3}{3p}$. In the block distribution, the last processor has about $\frac{n^3}{p}$ work, so that we lose a factor of 3 due to load imbalance. In the cyclic distribution, the last processor has an amount of work of at most

$$T = \frac{n^3}{3p} + n^2 + np.$$

Thus the maximum amount is close to the average, and the load imbalance is of order $\mathcal{O}(n^2)$.

- **Video: Simple parallel algorithm.** The algorithm for finding the maximum value is:

Input: $\mathbf{x}$: vector of length n, distr($\mathbf{x}$) = block, $n \bmod p = 0$.

Output: $\alpha = \max\ \{x_i\ :\ 0 \leq i < n\}$

```
αs := −∞;                                   ▷ Superstep (0)
b := n/p;
for i := sb to (s + 1)b − 1 do
    if xi > αs then
        αs := xi;

for t := 0 to p − 1 do                      ▷ Superstep (1)
    put αs in P(t);

α := −∞;                                    ▷ Superstep (2)
for t := 0 to p − 1 do
    if αt > α then
        α := αt;
```

The BSP cost of the algorithm is

$$T = \frac{n}{p} + p + (p-1)g + 3l,$$

where we count comparisons as the basic operation. Instead of initializing α_s to $-\infty$ we could have initialized to the first value of the block, i.e. $\alpha_s := x_{sb}$.

- **Video: Parallel sorting.** The imbalance of the output distribution of the parallel regular sample sort can be reduced by *oversampling*, for example by taking $2p$ local samples per processor instead of p. We still generate p global samples from them to be used in splitting the local data for p processors. This reduces the maximum output block size from $\frac{2n}{p}$ to $\frac{3n}{2p}$.

  This reduction can be shown by a similar reasoning as the proof that $b_s \leq \frac{2n}{p}$ in [PSC2, section 1.8]. Assume that $n \bmod 2p^2 = 0$, so that after the local sort every processor has $2p$ subblocks of length $\frac{n}{2p^2}$ starting with a local sample.

  By construction, every output block contains exactly $2p$ local samples, and hence $2p$ complete or incomplete associated subblocks, contributing at most $2p \cdot \frac{n}{2p^2} = \frac{n}{p}$ items. The output block also contains parts of subblocks without a sample. Every processor $P(t)$ can contribute at most one such subblock to $P(s)$, because the local sample must be smaller than the splitting value of $P(s)$ and the part must also have a larger value. The contribution of those subblocks is at most $p \cdot \frac{n}{2p^2} = \frac{n}{2p}$. Therefore

$$b_s \leq \frac{n}{p} + \frac{n}{2p} = \frac{3n}{2p}.$$

- **Video: Experimental results for parallel sorting.** The BSP parameters of the computer used are $p = 512$, $r = 6$ Gflop/s, $g = 1000$, $l = 4\,000\,000$ and the problem size is $n = 10^9$.

  Therefore, the computation time (in s, not flops) is $\frac{n \log_2 n}{rp} = 0.0097$ s; the communication time is $\frac{2ng}{rp} = 0.651$ s; and the synchronization time is $\frac{5l}{r} = 0.0033$ s. The predicted total parallel time is 0.664 s. The predicted sequential time is $\frac{n \log_2 n}{r} = 4.982$ s, so that the predicted speedup is

$$S_{512}(10^9) = \frac{T_{\text{seq}}(n)}{T_p(n)} = \frac{4.982}{0.664} \approx 7.5.$$

  Not impressive. Communication dominates!

- **Video: LU decomposition.** The steps of the algorithm are:

$$A = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 4 & 4 \\ 2 & 4 & 6 \end{bmatrix} \xrightarrow{k=0} \begin{bmatrix} 2 & 2 & 2 \\ 1 & 2 & 2 \\ 1 & 2 & 4 \end{bmatrix} \xrightarrow{k=1} \begin{bmatrix} 2 & 2 & 2 \\ 1 & 2 & 2 \\ 1 & 1 & 2 \end{bmatrix}.$$

  Hence,

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 2 & 2 & 2 \\ 0 & 2 & 2 \\ 0 & 0 & 2 \end{bmatrix}.$$

- **Video: Parallel LU decomposition.** The number of flops in the matrix update of stage k is $2R_{k+1}C_{k+1}$. For the 2×2 cyclic distribution of a 8×8 matrix, we have

$$R_{k+1} = C_{k+1} = \left\lceil \frac{n-k-1}{M} \right\rceil = \left\lceil \frac{7-k}{2} \right\rceil.$$

  For $k = 0, \ldots, 7$, we get $R_1 = 4$, $R_2 = 3$, $R_3 = 3$, $R_4 = 2$, $R_5 = 2$, $R_6 = 1$, $R_7 = 1$, $R_8 = 0$, respectively, so the total number of flops is

$$T_{\text{update}} = 2 \cdot \sum_{k=0}^{7} R_{k+1}^2 = 88.$$

- **Video: Two-phase broadcasting.** We use one-phase broadcasting if $T_1 \leq T_2$, i.e.,

$$\begin{aligned} (p-1)ng + l \leq 2\left(1 - \frac{1}{p}\right) ng + 2l \quad &\Leftrightarrow \\ (p-1)ng\left(1 - \frac{2}{p}\right) \leq l \quad &\Leftrightarrow \\ n \leq \frac{p}{(p-1)(p-2)} \cdot \frac{l}{g}. \end{aligned}$$

Substituting the values $p = 10$, $g = 100$, and $l = 10\,000$ gives $n \le \frac{1000}{72} \approx 13.88$, so we use a one-phase broadcast for $n \le 13$ and a two-phase broadcast for $n \ge 14$.

- **Video: High-Performance LU Decomposition.** We cut up all three matrices into $b \times b$ blocks. Let A_i be block i of A, so that

$$A = \begin{bmatrix} A_0 \\ \vdots \\ A_{n/b-1} \end{bmatrix}.$$

Similarly, let B_j be block j of B, so that

$$B = [B_0, \ldots, B_{n/b-1}].$$

We then compute the blocks C_{ij} of the output matrix

$$C = \begin{bmatrix} C_{00} & \cdots & C_{0,n/b-1} \\ \vdots & & \vdots \\ C_{n/b-1,0} & \cdots & C_{n/b-1,n/b-1} \end{bmatrix}$$

by

for $i := 0$ **to** $\frac{n}{b} - 1$ **do**
  for $j := 0$ **to** $\frac{n}{b} - 1$ **do**
    $C_{ij} := A_i B_j$;

In each iteration of the inner loop, we perform $2b^3$ flops involving $3b^2$ data, so we have good data reuse. Since we can keep A_i in cache throughout the inner loop, this requires only the movement of $2b^2$ data vs. performing $2b^3$ flops.

- **Video: Discrete Fourier Transform.** Let $\omega_n = \mathrm{e}^{-2\pi i/n}$. If we first perform a DFT and then an IDFT, we can express the result z_k for $0 \le k < n$ as

$$\begin{aligned} z_k &= \frac{1}{n} \sum_{j=0}^{n-1} y_j \omega_n^{-jk} \\ &= \frac{1}{n} \sum_{j=0}^{n-1} \left(\sum_{l=0}^{n-1} x_l \omega_n^{jl} \right) \omega_n^{-jk} \\ &= \frac{1}{n} \sum_{l=0}^{n-1} x_l \left(\sum_{j=0}^{n-1} \omega_n^{jl} \omega_n^{-jk} \right) \\ &= \frac{1}{n} \sum_{l=0}^{n-1} x_l \left(\sum_{j=0}^{n-1} \omega_n^{j(l-k)} \right). \end{aligned}$$

For $l = k$, the expression between brackets becomes n, so term k contributes $\frac{1}{n} \cdot x_k \cdot n = x_k$. For $l \neq k$, we write $\alpha = \omega_n^{(l-k)}$, so that the expression between brackets becomes

$$\sum_{j=0}^{n-1} \alpha^j = \frac{1-\alpha^n}{1-\alpha} = 0.$$

This is because $\alpha^n = 1$, and $1 - \alpha \neq 0$. We thus have shown that $z_k = x_k$, so we retrieve the original vector.

If we reverse the order, and perform an IDFT and then a DFT, we also execute the identity operation. The proof would be similar, but it is easier to use linear algebra which states that for square matrices if $BA = I_n$, we also have $AB = I_n$ and to apply this with $A = F_n$ and $B = \frac{1}{n}\bar{F}_n$, where the bar denotes conjugation.

- **Video: Recursive Fast Fourier Transform.** This one is not for the faint-hearted. For the radix-4 algorithm, we first split the sum of the DFT into 4 partial sums, based on taking indices modulo 4. For $0 \leq k < n$, we have

$$\begin{aligned} y_k &= \sum_{j=0}^{n-1} x_j \omega_n^{jk} \\ &= \sum_{j=0}^{n/4-1} x_{4j}\omega_n^{4jk} + \sum_{j=0}^{n/4-1} x_{4j+1}\omega_n^{(4j+1)k} + \sum_{j=0}^{n/4-1} x_{4j+2}\omega_n^{(4j+2)k} + \sum_{j=0}^{n/4-1} x_{4j+3}\omega_n^{(4j+3)k} \\ &= \sum_{j=0}^{n/4-1} x_{4j}\omega_{n/4}^{jk} + \omega_n^k \sum_{j=0}^{n/4-1} x_{4j+1}\omega_{n/4}^{jk} + \omega_n^{2k} \sum_{j=0}^{n/4-1} x_{4j+2}\omega_{n/4}^{jk} + \omega_n^{3k} \sum_{j=0}^{n/4-1} x_{4j+3}\omega_{n/4}^{jk}. \end{aligned} \tag{1}$$

For $0 \leq k < \frac{n}{4}$, we write this as

$$y_k = y_k^{(0)} + \omega_n^k y_k^{(1)} + \omega_n^{2k} y_k^{(2)} + \omega_n^{3k} y_k^{(3)},$$

where the sums $y_k^{(r)}$ are the result of an FFT of length $\frac{n}{4}$. For $k \geq \frac{n}{4}$, we can reuse these sums.

In the case $\frac{n}{4} \leq k < \frac{n}{2}$, we substitute $k = k' + \frac{n}{4}$ into Eqn (1), giving

$$\begin{aligned} y_k = & \sum_{j=0}^{n/4-1} x_{4j}\omega_{n/4}^{j(k'+n/4)} + \omega_n^{(k'+n/4)} \sum_{j=0}^{n/4-1} x_{4j+1}\omega_{n/4}^{j(k'+n/4)} + \\ & \omega_n^{2(k'+n/4)} \sum_{j=0}^{n/4-1} x_{4j+2}\omega_{n/4}^{j(k'+n/4)} + \omega_n^{3(k'+n/4)} \sum_{j=0}^{n/4-1} x_{4j+3}\omega_{n/4}^{j(k'+n/4)}. \end{aligned} \tag{2}$$

Using $\omega_{n/4}^{n/4} = 1$, $\omega_n^{n/4} = -i$, $\omega_n^{2n/4} = -1$, and $\omega_n^{3n/4} = i$, and dropping the primes, we obtain

$$y_k = y_k^{(0)} - i\omega_n^k y_k^{(1)} - \omega_n^{2k} y_k^{(2)} + i\omega_n^{3k} y_k^{(3)}.$$

Similarly, in the case $\frac{n}{2} \le k < \frac{3n}{4}$, we substitute $k = k' + \frac{n}{2}$, and obtain

$$y_k = y_k^{(0)} - \omega_n^k y_k^{(1)} + \omega_n^{2k} y_k^{(2)} - \omega_n^{3k} y_k^{(3)}.$$

Finally, in the case $\frac{3n}{4} \le k < n$, we substitute $k = k' + \frac{3n}{4}$, and obtain

$$y_k = y_k^{(0)} + i\omega_n^k y_k^{(1)} - \omega_n^{2k} y_k^{(2)} - i\omega_n^{3k} y_k^{(3)}.$$

This leads to the following radix-4 algorithm:

Input: $\mathbf{x}$: vector of length n.
Output: $\mathbf{y}$: vector of length n, $\mathbf{y} = F_n\mathbf{x}$.

function FFT($\mathbf{x}, n$)

  if $n \bmod 4 = 0$ **then**
    $\mathbf{x}^{(0)} := x(0\colon 4\colon n-1)$;
    $\mathbf{x}^{(1)} := x(1\colon 4\colon n-1)$;
    $\mathbf{x}^{(2)} := x(2\colon 4\colon n-1)$;
    $\mathbf{x}^{(3)} := x(3\colon 4\colon n-1)$;

    $\mathbf{y}^{(0)} := \mathrm{FFT}(\mathbf{x}^{(0)}, n/4)$;
    $\mathbf{y}^{(1)} := \mathrm{FFT}(\mathbf{x}^{(1)}, n/4)$;
    $\mathbf{y}^{(2)} := \mathrm{FFT}(\mathbf{x}^{(2)}, n/4)$;
    $\mathbf{y}^{(3)} := \mathrm{FFT}(\mathbf{x}^{(3)}, n/4)$;

    for $k := 0$ **to** $n/4 - 1$ **do**
      $\tau_0 := y_k^{(0)}$;
      $\tau_1 := \omega_n^k y_k^{(1)}$;
      $\tau_2 := \omega_n^{2k} y_k^{(2)}$;
      $\tau_3 := \omega_n^{3k} y_k^{(3)}$;
      $y_k := \tau_0 + \tau_1 + \tau_2 + \tau_3$;
      $y_{k+n/4} := \tau_0 - i\tau_1 - \tau_2 + i\tau_3$;
      $y_{k+n/2} := \tau_0 - \tau_1 + \tau_2 - \tau_3$;
      $y_{k+3n/4} := \tau_0 + i\tau_1 - \tau_2 - i\tau_3$;
  else
    $\mathbf{y} := \mathrm{DFT}(\mathbf{x}, n)$;

Each statement in the inner loop except the first costs 6 flops, since it represents either a complex multiplication, or 3 complex adddditions/subtractions.

The total number of flops of an iteration of the inner loop is thus 42, a number we have met before;) Note that the expression $\tau_0+\tau_2$ occurs twice, so that we can save 2 flops by computing this as an intermediate result. The same holds for the expressions $\tau_0 - \tau_2$, $\tau_1 + \tau_3$, $\tau_1 - \tau_3$. Thus, an iteration of the inner loop costs 34 flops. Therefore, the total number of flops for a radix-4 FFT is

$$T(n) = 4T(\frac{n}{4}) + 34\frac{n}{4} = 4T(\frac{n}{4}) + \frac{17n}{2}.$$

Repeatedly using this formula we arrive at

$$T(n) = (\log_4 n) \cdot \frac{17n}{2} = \frac{17}{4} \cdot n \log_2 n = 4.25n \log_2 n.$$

In conclusion, a radix-4 FFT has slightly fewer flops than the $5n \log_2 n$ flops of a radix-2 FFT.

- **Video: Nonrecursive Fast Fourier Transform.** Because $S_2 = I_2$, the leftmost factor of R_8 equals $I_4 \otimes S_2 = I_4 \otimes I_2 = I_8$, so that

$$R_8 = (I_4 \otimes S_2)(I_2 \otimes S_4)(I_1 \otimes S_8) = (I_2 \otimes S_4)S_8$$

$$= \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix}.$$

Then $\mathbf{y} = R_8(0,1,2,3,4,5,6,7)^{\mathrm{T}} = (0,4,2,6,1,5,3,7)^{\mathrm{T}}$. If we write the values $x_j = j$ and y_j in binary, we obtain the following table

x_j	$(x_j)_2$	$(y_j)_2$	y_j
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Comparing the binary values, we see that $(y_j)_2$ is the bit-reverse of $(x_j)_2$.

- **Video: Parallel Fast Fourier Transform.** The redistribution from block distribution with reverse processor numbering to cyclic distribution can be done by the following algorithm.

---

function REDISTR($\mathbf{x}, n, p$, block_reverse, cyclic)

$s' := \rho_p(s)$;

for $j := s'\frac{n}{p}$ **to** $(s'+1)\frac{n}{p} - 1$ **do**
put x_j in $P(j \bmod p)$;

If we also want to specify where to put x_j in the destination processor, we do this by increasing global index, which means that the local index will be $\mathtt{j} = j$ div p.

- **Video: Sequential sparse matrix–vector multiplication.** For the matrix

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 5 \\ 4 & 0 & 0 & 0 & 4 \\ 0 & 3 & 0 & 3 & 0 \\ 0 & 0 & 2 & 0 & 0 \end{bmatrix}$$

with $n = 5$ and $nz = 7$, the CRS data structure is:

$a[k] =$	5	5	4	4	3	3	2
$j[k] =$	0	4	0	4	1	3	2
$k =$	0	1	2	3	4	5	6

$start[i] =$	0	0	2	4	6	7
$i =$	0	1	2	3	4	5

The number of data words needed for this matrix is 20, and for a general $n \times n$ matrix with nz nonzeros it is $2nz + n + 1$.

- **Video: Parallel sparse matrix–vector multiplication.** An example of the matrix for $n = 12$ is

$$A = \left[\begin{array}{cccccccccccc}
1 & 1 & & & & & & & & & & \\
1 & 1 & 1 & & & & & & & & & \\
 & 1 & 1 & 1 & & & & & & & & \\
\hline
 & & 1 & 1 & 1 & & & & & & & \\
 & & & 1 & 1 & 1 & & & & & & \\
 & & & & 1 & 1 & 1 & & & & & \\
\hline
 & & & & & 1 & 1 & 1 & & & & \\
 & & & & & & 1 & 1 & 1 & & & \\
 & & & & & & & 1 & 1 & 1 & & \\
\hline
 & & & & & & & & 1 & 1 & 1 & \\
 & & & & & & & & & 1 & 1 & 1 \\
 & & & & & & & & & & 1 & 1
\end{array}\right].$$

Since the matrix distribution is 1D, the parallel SpMV algorithm reduces to two supersteps, namely the fanout (0) and the local SpMV (1). The total synchronization time is therefore $2l$.

In the fanout, the processors $P(s)$ with $0 < s < p-1$ have to send two data words: v_{sb} to $P(s-1)$ and $v_{(s+1)b-1}$ to $P(s+1)$. $P(0)$ and $P(p-1)$ have to send only one data word. The communication cost is therefore $2g$. The total communication volume is $V = 2(p-2) + 2 = 2p - 2$.

In the local SpMV, the processors $P(s)$ with $0 < s < p-1$ possess $3b$ nonzeros, and the other two processors one nonzero less, so the computation cost is $2 \cdot 3b = \frac{6n}{p}$. The total BSP cost is

$$T = \frac{6n}{p} + 2g + 2l.$$

Note that the communication cost is much less than the value $2(1 - \frac{1}{p})ng$ given in the pessimistic upper bound.

- **Video: Cartesian matrix distribution.** We can translate the convolutional neural network to an SpMV problem involving the matrix

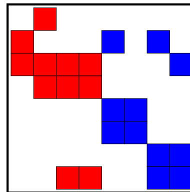
$$A = \begin{bmatrix}
a_{00} & a_{01} & & & & & & \\
a_{10} & a_{11} & a_{12} & & & & & \\
 & a_{21} & a_{22} & a_{23} & & & & \\
 & & a_{32} & a_{33} & a_{34} & & & \\
 & & & a_{43} & a_{44} & a_{45} & & \\
 & & & & a_{54} & a_{55} & a_{56} & \\
 & & & & & a_{65} & a_{66} & a_{67} \\
 & & & & & & a_{76} & a_{77}
\end{bmatrix}.$$

Thus we see that the corresponding matrix is tridiagonal. A simple solution is then to use the block row distribution, used in the final question

of the video on parallel SpMV. This distribution has a near-perfect load balance, only two supersteps and a minimal communication cost $2g$, which is hard to improve.

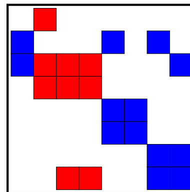
If the number of connections per neuron grows, other distributions may become attractive, such as the square Cartesian matrix distribution based on a block distribution of the matrix diagonal, which is optimal in the fully connected case.

- **Video: Mondriaan sparse matrix distribution.** The distribution before the refinement is



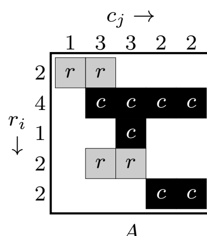
This distribution has the advantage of a perfect load balance, since both processors have 11 nonzeros. However, the communication volume is $V = 3$, since rows 1, 2, 7 are split.

The distribution after the refinement is



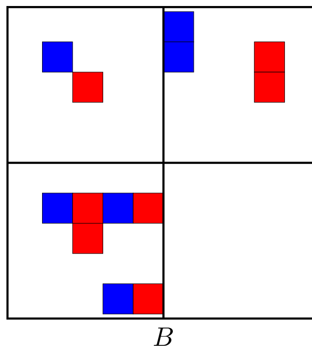
The load balance is worse, since the blue processor has 13 nonzeros and the red one only 9. Still, this is allowed since the allowed maximum is $(1+\epsilon)nz(A)/2 = 1.2 \cdot 22/2 = 13.2$. The communication volume is smaller, $V = 2$, since only rows 2 and 7 are split. Thus the refinement is beneficial.

- **Video: Comparing sparse matrix distributions.** First, we count the number of nonzeros in the rows and columns of the matrix. The counts are shown to the left and above the following matrix.



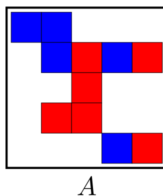
Then we assign the singleton row 2 to the column matrix (the nonzero is shown in black and marked by a 'c'). We also assign the singleton column 0 to the row matrix (the nonzero is shown in grey and marked by an 'r'). After that, we assign nonzeros a_{01}, a_{31}, a_{32} to the row matrix since these a_{ij} satisfy $r_i < c_j$. The remaining nonzeros are then assigned to the column matrix. These include the tied nonzeros from row 4.

We then form the matrix B and distribute its columns cyclically, starting with the red processor $P(0)$:



The left upper block of B contains two diagonal nonzeros, caused by columns 1 and 2 of A that are split between grey and black. The lower right block of B is empty because no rows are split between grey and black. Note that the communication volume for B is 3, because rows 1, 6, 9 are cut.

Finally, we fold B back into A, giving



Note that the communication volume for A is 3, because rows 1, 4, and column 1 are cut. Thus, the communication volumes for A and B are indeed the same.

- **Video: Random sparse matrices.** Assume for simplicity that $n \bmod p = 0$ and p is a square. Processor $P(s)$ has parts of the matrix columns $j = sb, \ldots, (s+1)b - 1$, where $b = \frac{n}{\sqrt{p}}$.

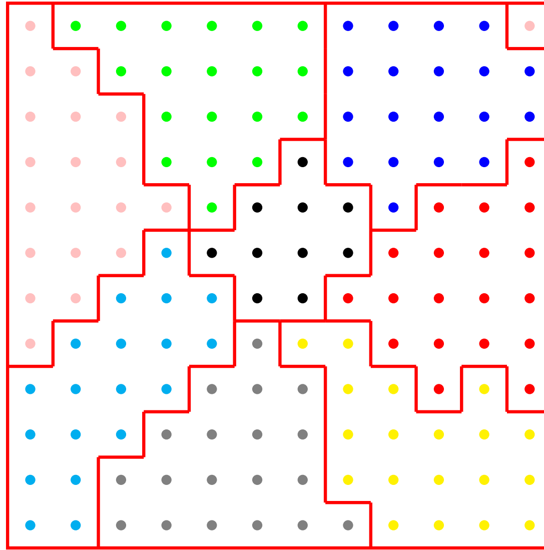
  Let $sb \leq j < (s+1)b$. The probability that $P(s)$ does not need vector component v_j is $(1-d)^{\frac{n}{\sqrt{p}}}$, because it has $\frac{n}{\sqrt{p}}$ matrix elements from column j, and this is the probability that they are all zero. The expected number of vector components v_j that $P(s)$ has to obtain through communication is then

$$h_{\mathrm{recv}} = \left(\frac{n}{\sqrt{p}} - \frac{n}{p} \right) (1-d)^{\frac{n}{\sqrt{p}}},$$

  because it may need $\frac{n}{\sqrt{p}}$ vector components of which $\frac{n}{p}$ are already locally available. Furthermore, $P(s)$ has to send its $\frac{n}{p}$ vector components to $\sqrt{p} - 1$ other processors in its processor column, each with a probability $(1-d)^{\frac{n}{\sqrt{p}}}$. This gives the same result, $h_{\mathrm{send}} = h_{\mathrm{recv}}$. Therefore, the cost of the fanout is

$$T_{(0)} = \left(\frac{n}{\sqrt{p}} - \frac{n}{p} \right) (1-d)^{\frac{n}{\sqrt{p}}} g + l.$$

- **Video: Laplacian matrices.** The best solution so far was found by Bas den Heijer in 2006; very likely it is optimal, but this has not yet been proven. His solution is:



All processors except black and yellow perform 89 flops (counting 5 for interior points, 4 for boundary points that are not corner points, and 3 for corner points). The yellow processor performs 88 flops, and the landlocked black processor which has 10 interior points performs 50 flops. Note that the pink processor has a non-contiguous area, with an isolated single grid point in the north-eastern corner of the grid. Taking the maximum of the number of flops, we find that the computation cost is 89.

All processors have 11 neighbours, except dark blue, which has 10. This means that $h_{\mathrm{recv}} = 11$, and the total communication volume is $V = 87$. All processors send 11 data words, except the pink processor, which sends 10. This means that $h_{\mathrm{send}} = 11$ and it confirms that $V = 87$. As a result, $h = 11$ and the communication cost is $11g$.

The corresponding SpMV has only two supersteps, the fanout and the local SpMV, since we first get the data from neighbouring points and then use them. Hence, the synchronization cost is $2l$. In summary, the total BSP cost is

$$T = 89 + 11g + 2l = 89 + 110 + 100 = 299.$$

- **Video: Parallel algorithm for the hybrid-BSP model.** The total number of processors is $p_1 p_2 = 1000$, so every processor has to sum 1000 numbers. This happens in superstep (0), and it costs (in flops)

$$T_{(0)} = \frac{n}{p} + l_1 = 1000 + 100 = 1100.$$

In superstep (1), every processor $P(s,t)$ sends its partial sum to $P(s,0)$, which costs

$$T_{(1)} = (p_1 - 1)g_1 + l_1 = 90 + 100 = 190.$$

In superstep (2), every processor $P(s,0)$ sums its p_1 partial sums, which costs

$$T_{(2)} = p_1 + l_1 = 10 + 100 = 110.$$

In superstep (3), every processor $P(s,0)$ sends its partial sum to $P(0,0)$, which is an expensive superstep that costs

$$T_{(3)} = (p_2 - 1)g_2 + l_2 = 9900 + 1000 = 10900.$$

In superstep (4), processor $P(0,0)$ sums its p_2 partial sums, which can most favourably be seen as a local superstep, and costs

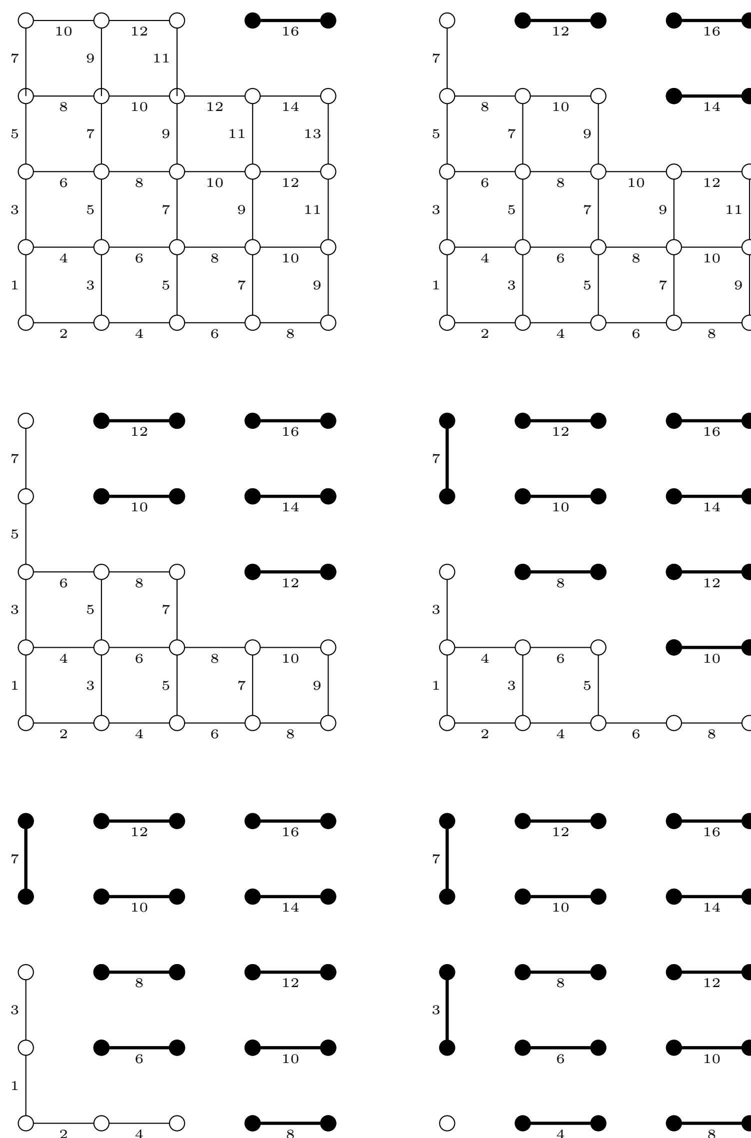
$$T_2 = p_2 + l_1 = 100 + 100 = 200.$$

Adding these up, the total cost in flops is then

$$T = 12500.$$

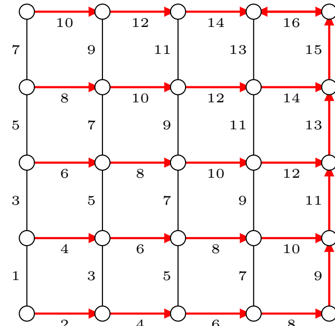
Thus, we achieve a speedup of $10^6/12\,500 = 80$ using 1000 processors. Very modest indeed, and mainly due to the expensive superstep (3).

- **Video: Sequential graph matching.** The algorithm performs the following stages:

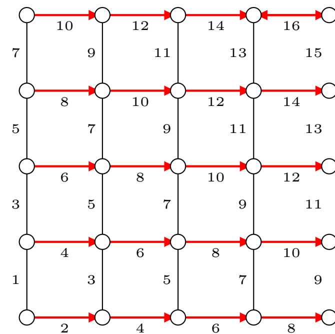


The pictures show that the computation evolves as a wavefront moving from the upper right corner of the graph to the lower left corner. The total weight of the matching produced is 110. The number of dominant edges discovered in the 6 stages is: 1, 2, 2, 3, 2, 2. This sequence is a measure of the amount of parallelism available for this graph. It means that up to 3 processors can usefully be employed.

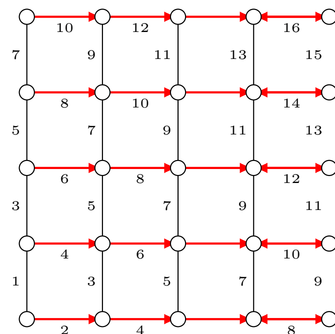
- **Video: Suitors and sorting in graph matching.** The preferences of the vertices are shown by red arrows in the following picture. Note that one preference is mutual, namely in the upper right corner, with weight 16.



Removing preferences that are not the best yields the suitors, shown by red arrows in the following picture. The edges corresponding to rejected preferences can be removed from the problem.



If we visit the vertices in a certain order when setting preferences, and we check for immediate rejection, we can reset the preference immediately, thus speeding up the whole process. The following picture shows the result of one such round in the order from top to bottom (and left to right within each row).



Note that all the vertices on the right, except the top vertex, will get rejected immediately upon their first attempt in the upwards direction, so that they have to settle for a lower weight to their left; this leads to more mutual preferences. Other actions are also carried out: matches are processed and preferences are reset for vertices that lost their preference.

- **Video: Parallel graph matching.** The complete algorithm for processing a received proposal is:

---

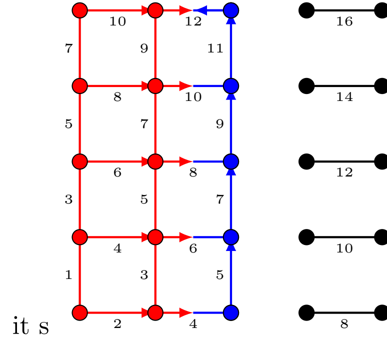
if $msg = \text{propose}(u, v)$ **then**
    { Register a match }
    if $u = pref(v)$ **then**
        $\mathcal{M}_s := \mathcal{M}_s \cup \{(u, v)\}$;
        $d_v := 0$;

    { Assign new suitor }
    $x := suitor(v)$;
    if $\omega(u, v) > \omega(x, v)$ **then**
        $suitor(v) := u$;
        RejectSuitor$(v, x, Q_s, \mathcal{V}_s, alive, pref)$
    else
        put reject(v, u) in $P(\phi(u))$;

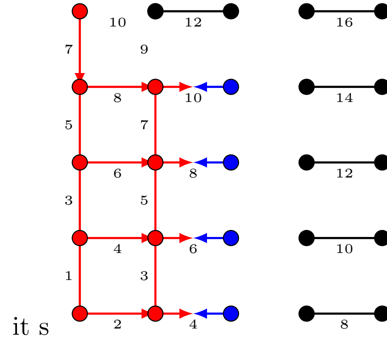
If the proposal is successful, we call the RejectSuitor function which returns a local suitor x to the queue Q_s, or sends a reject message to a nonlocal suitor x. If the proposal fails, we send a reject message to the proposer u.

- **Video: Tie-breaking and load balancing.** The graph has 5 cut edges, so the edge cut is $EC_\phi = 5$.

  In the (mixed) superstep (0), the blue processor finds 5 matches and sets the preferences of the vertices at its left boundary. Since one preference is nonlocal (with weight 12), it sends a proposal. In the same superstep, the red processor sets all preferences to the right, and since the preferences at the right boundary are all nonlocal, it sends 5 proposals. The situation at the end of superstep (0) is :



In superstep (1), the blue processor tacitly accepts the match at the top (with weight 12), and accepts all proposals from the red processor by sending an accept message. It locally registers the matches. Blue is then done. In the same superstep, the red processor also tacitly accepts the match at the top (with weight 12), removes two adjacent edges, and waits for the answer to its 4 proposals. The situation at the end of superstep (1) is :



In superstep 2, the red processor receives the 4 accept messages and then finishes its own computation, so that both processors are done. Thus there are 3 supersteps. The situation at the end of superstep (2) is the same as after the sequential computation, see the solution above. The total weight achieved is 110. The total communication volume is $V = 10$. This attains the upper bound of $2EC_\phi$

- **Video: Reducing communication in parallel graph matching.** The dense graph has $\frac{n(n-1)}{2}$ edges, since each of the n vertices is connected to all $(n-1)$ others, where we count every edge twice. Similarly, each processor has $\frac{n}{p}$ vertices, and hence $\frac{n}{2p}\left(\frac{n}{p}-1\right)$ internal edges. The total number of internal edges of all p processors is thus $\frac{n}{2}\left(\frac{n}{p}-1\right)$. The edge cut is the number of external edges, which equals

$$EC_\phi = \frac{n(n-1)}{2} - \frac{n}{2}\left(\frac{n}{p}-1\right) = \frac{n}{2}\left(n-\frac{n}{p}\right) = \frac{n^2}{2}\left(1-\frac{1}{p}\right).$$

For the broadcast of a match of vertex v, a data word has to be sent to $p-1$ processors. For all n vertices, this amounts to a communication volume of

$$V_\phi = n(p-1).$$

This means that the communication volume is a factor $\frac{n}{2p}$ smaller than the edge cut. Therefore, the extra cost of broadcasting matches is very small. But its gains may be large: many proposals may be prevented, thereby reducing communication and also computation (through shrinking adjacency lists).