2.5 Quantifying parallelism

Given:

- A p-processor parallel computer
- A problem (e.g., linear system) and an algorithm to solve it (e.g., Gaussian elimination)

---

13 $\Diamond$ **Definition:** Let $T(p)$ denote the time taken by the parallel algorithm on p processors.

$$S(p) := \frac{T(1)}{T(p)} \quad \text{is the } \textbf{speed-up} \text{ and}$$

$$E(p) := \frac{S(p)}{p} \quad \text{is the } \textbf{efficiency}$$

of the algorithm **w.r.t. itself**.

Let T' denote the time taken by the **fastest** known (serial) algorithm to solve the problem.

$$S'(p) := \frac{T'}{T(p)} \quad \text{is the } \textbf{true speed-up} \text{ and}$$

$$E'(p) := \frac{S'(p)}{p} \quad \text{is the } \textbf{true efficiency}$$

of the parallel algorithm.

Let #ops(p) denote the number of operations ($+$, $-$, $*$, often also $/$, $\sqrt{\ }$).

$$R(p) = \frac{\#\text{ops}(p)}{T(p)} \quad \text{is the } \textbf{(compute) performance}$$

of the algorithm on p processors (in **Flop/s**: **fl**oating-point **op**erations per **s**econd).

Note: By Definition 13,

$$\begin{aligned} S(p) &= \frac{\#\text{ops}(1)}{R(1)} \cdot \frac{R(p)}{\#\text{ops}(p)} \\ &\approx \frac{R(p)}{R(1)} \quad \text{if } \#\text{ops}(p) \approx \#\text{ops}(1)\ . \end{aligned}$$

More detailed analysis: Let n denote the "characteristic size" of the problem (e.g., matrix dimension for linear systems). Then

$$T(p) \rightsquigarrow T(p,n), \quad S(p) \rightsquigarrow S(p,n), \quad \text{etc.}$$

---

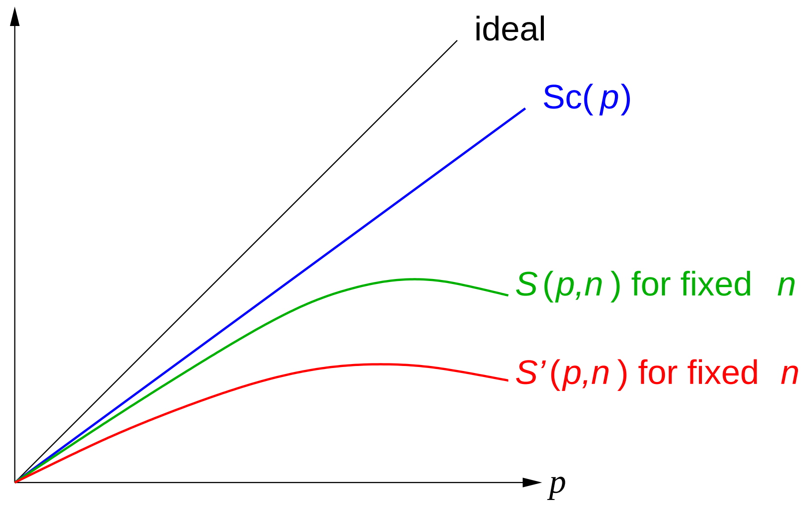
14 ◊ Definition: Given a p-processor parallel computer, the **scale-up** for a variable-size problem is given by

$$\text{Sc}(p) := \sup_{n \in \mathbb{N}} S(p,n)\ .$$

(In practice, n can be limited to problems that fit into the available memory.)

Note: $\mathrm{Sc}(p) \geq S(p,n) \geq S'(p,n)$ for any n.

Typical behavior:



---

15 ◊ Definition: A parallel algorithm for solving a problem **scales** iff

$$\liminf_{p\to\infty} \frac{\mathrm{Sc}(p)}{p} > 0$$

(i.e., the quotient $\frac{\mathrm{Sc}(p)}{p}$ is bounded from below by some $\varepsilon > 0$).

---

16 $\Diamond$ **Theorem:** (**Amdahl's law**) Let $q \in [0,1]$ denote the fraction of those operations in an algorithm, which cannot be parallelized. Then

$$S(p) \leq \frac{1}{q}\,, \quad \textbf{no matter how large } p \textbf{ is}.$$

Proof: $T(p) \geq q \cdot T(1)$. ■

---

17 $\Diamond$ **Example:** If $q = 0.01$ (i.e., 99% of all operations can be parallelized) then

$$S(p) \leq 100.$$

2.6 Communication instructions

send(D **) to** P_i : send the data D to processor P_i

receive(D **) from** P_j : receive the data D from processor P_j

broadcast(D **)** : send the data D to all processors (perhaps only to those of a specified group)

reduce : see Section 2.7.1

---

18 ◊ **Remark:** Sending data from one processor to another one typically involves

- setting up a connection between them (e.g., determining a suitable path through the network)
- transmitting the data

$\Rightarrow$ communication time for sending a length-ℓ message is

$$T_{\mathrm{comm}}(\ell) = \alpha + \beta \cdot \ell,$$

where

- α is the **latency** or start-up time,
- β is the transmission time per byte
(reciprocal of the **communication bandwidth**), and
- ℓ is the length of the message in bytes.

Realistic values: $\alpha \sim 2\mu$s, $\beta \sim 0.1$ns/B ($\widehat{=}$ 10GB/s).

Check your understanding. You should know . . .

- Speed-up $S(p)$, efficiency $E(p)$, scale-up $Sc(p)$
- What does Amdahl's law tell?
- What are the four basic communication routines?

2.7 Basic parallelization schemes

2.7.1 The fan-in principle for reduction operations

Problem: Given $a_0, \ldots, a_{n-1} \in \mathbb{R}$, compute

$$s := \sum_{j=0}^{n-1} a_j \;.$$

---

19 ◊ Algorithm: Serial sum

$s^{(-1)} := 0$

for $i = 0 : n-1$

  $\{$ assume that $s^{(i-1)} = \sum_{j=0}^{i-1} a_j \}$

  $s^{(i)} := s^{(i-1)} + a_i$

$\{ s^{(n-1)} = s \}$

Dependence graph:

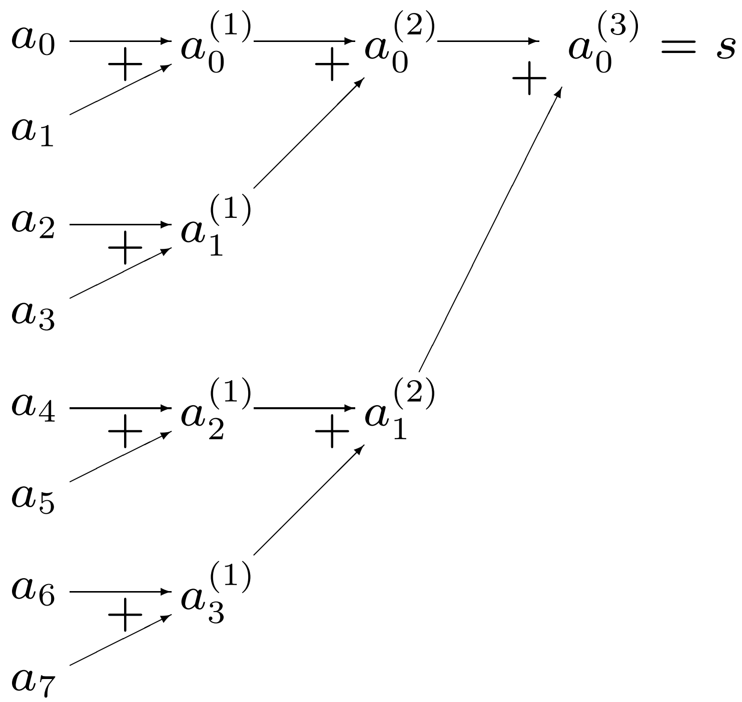
$$\begin{tikzcd} s^{(-1)} \arrow[r, "+"'] & s^{(0)} \arrow[r, "+"'] & s^{(1)} \arrow[r, "+"'] & s^{(2)} \arrow[r] & \cdots \arrow[r, "+"'] & s^{(n-1)} \\ a^{(0)} \arrow[ur] & a^{(1)} \arrow[ur] & a^{(2)} \arrow[ur] & \cdots & a^{(n-1)} \arrow[ur] & \end{tikzcd}$$

Nodes represent intermediate quantities,

(directed) edges represent dependences (e.g., $s^{(0)}$ and $a^{(1)}$ must be available to compute $s^{(1)}$),

the "+" are optional, specifying the computations.

Different approach (here for $n = 8 = 2^3$):



{ to fan out: to spread out in the shape of a fan }

---

20 ◊ **Algorithm:** Serial **fan-in summation**, assumes $n = 2^N$

{ let $a_j^{(0)} \equiv a_j$, $j = 0 : n-1$ }

for $k = 1 : N$

{ assume that for $i = 0 : 2^{N-(k-1)} - 1$, $a_i^{(k-1)}$ is the sum of the ith length-2^{k-1} subsequence of a, i.e.,

$$a_i^{(k-1)} = \sum_{j=i2^{k-1}}^{(i+1)2^{k-1}-1} a_j \;\}$$

for $i = 0 : 2^{N-k} - 1$

$$a_i^{(k)} := a_{2i}^{(k-1)} + a_{2i+1}^{(k-1)}$$

{ $a_0^{(N)} = s$ }

Potential for parallelism: The i loop can be done in parallel.

Let $p = 2^N$, processors $P_0, \ldots, P_{p-1}$.

For each "stage index" k, choose 2^{N-k} of the processors (labelled $P_i^{(k)}$, $i = 0 : 2^{N-k} - 1$), such that

- $P_i^{(k)}$ computes the partial sum $a_i^{(k)} := a_{2i}^{(k-1)} + a_{2i+1}^{(k-1)}$
  { $a_{2i}^{(k-1)}$ is available in $P_{2i}^{(k-1)}$, $a_{2i+1}^{(k-1)}$ is in $P_{2i+1}^{(k-1)}$ }

---

21 ◊ **Algorithm:** Fan-in summation with $p = 2^N$ processors; all processors run the same pseudocode

{ each of the 2^N processors $P_i^{(0)}$ holds $a_i =: a_{\text{local}}$ }

for $k = 1 : N$
    { assume that each processor $P_i^{(k-1)}$,
    $i = 0 : 2^{N-(k-1)} - 1$, holds $a_{\text{local}} \equiv a_i^{(k-1)}$ }
    for $i = 0 : 2^{N-k} - 1$
        if me $= P_{2i}^{(k-1)}$ **or me** $= P_{2i+1}^{(k-1)}$
            { I have one summand needed for $a_i^{(k)}$ }
            send(a_{local}) **to** $P_i^{(k)}$
        if me $= P_i^{(k)}$
            { I am computing $a_i^{(k)}$ }
            receive(summand_1) **from** $P_{2i}^{(k-1)}$
            receive(summand_2) **from** $P_{2i+1}^{(k-1)}$
            $a_{\text{local}} := \text{summand}_1 + \text{summand}_2$
{ $s = a_0^{(N)}$ is the result a_{local} in $P_0^{(N)}$ }

---

22 $\Diamond$ **Remark:** To add $n = q \cdot 2^N$ numbers with $p = 2^N$ processors ($q, N \in \mathbb{N}$), start with the partial sums

$$a_{\text{local}} := \sum_{j=iq}^{(i+1)q-1} a_j \quad \text{in } P_i^{(0)}, \; i = 0 : 2^N - 1.$$

Computation of a_{local}: E.g., with Algorithm 19

---

23 $\Diamond$ **Definition:** A **deadlock** in a parallel algorithm occurs when a processor wants to receive data that are never sent.

---

24 $\Diamond$ **Remark:** Depending on the implementation, even a **send** operation may terminate only when the corresponding **receive** of the partner is completed (**blocking send**).

This does not happen if outgoing data are buffered

$\Rightarrow$ **send** completes when data are in the buffer

(unless buffer overflows).

Problem: Algorithm 21 is deadlock-free (for blocking sends) only if all $P_i^{(k-1)}$, $i = 0 : 2^{N-(k-1)} - 1$, are different from all $P_i^{(k)}$, $i = 0 : 2^{N-k} - 1$; otherwise deadlocks may occur.

This condition cannot be fulfilled for $k = 1$!

Solution: Careful selection of the processors eliminates a (critical) part of the communication.

More precisely, let

$$P_i^{(k)} := P_{i \cdot 2^k}, \quad k = 0 : N, \; i = 0 : 2^{N-k} - 1 \; . \quad (2.1)$$

Then

$$P_{2i}^{(k-1)} = P_{2i \cdot 2^{k-1}} = P_{i \cdot 2^k} = P_i^{(k)} \; ,$$

i.e., the **receive**(summand_1) and the first case for **send** can be skipped because the processor would send to itself.

25 ◊ Algorithm: Improved fan-in summation

> **for** $k = 1 : N$
>   **for** $i = 0 : 2^{N-k} - 1$
>   **if me** $= P_{2i+1}^{(k-1)}$ $\{ = P_{i2^k+2^{k-1}} \}$
>   { I have the summand $a_{2i+1}^{(k-1)}$ that is needed in $P_i^{(k)}$ to compute $a_i^{(k)}$ }
>   **send**(a_{local}) **to** $P_i^{(k)}$ $\{ = P_{i2^k} \}$
>   **if** me $= P_i^{(k)}$
>   { I am computing $a_i^{(k)}$ }
>   **receive**($\mathrm{summand}_2$) **from** $P_{2i+1}^{(k-1)}$
> $a_{\mathrm{local}} := a_{\mathrm{local}} + \mathrm{summand}_2$

---

26 ◊ Remark: With the logical to physical mapping $P_i^{(k)} = P_{i \cdot 2^k}$, the indices of the communication partners in Algorithms 25 differ by 2^{k-1}, i.e., only in bit $k-1$

$\Rightarrow$ in a N-dimensional hypercube all communication is between directly connected processors.

27 ♢ Remark: In Algorithm 25, each processor performs

$$\sum_{k=1}^{N} \sum_{i=0}^{2^{N-k}-1} \mathcal{O}(1) = \mathcal{O}(n)$$

operations, mostly for finding out, which rôle the processor plays in stage k (in the majority of cases, none).

In practice, this must be done without a loop.

28 ♢ Algorithm: Fan-in summation

for $k = 1 : N$
  if (**me** is a multiple of 2^{k-1})
      { only these procs were active in stage $k-1$ }
      and (**me** is not a multiple of 2^k)
      { only these procs must send }
    send(a_{local}) **to** $P_{\textbf{me}-2^{k-1}}$
  if (**me** is a multiple of 2^k)
    { only these procs are active in stage k }
    receive(summand_2) **from** $P_{\textbf{me}+2^{k-1}}$
    $a_{\text{local}} := a_{\text{local}} + \text{summand}_2$

⇒ only $\mathcal{O}(N) = \mathcal{O}(\log n)$ operations per processor.

29 $\diamondsuit$ **Theorem:** Let $a_i \in \mathbb{R}$, $i = 0 : n-1$ ($n = q \cdot 2^N$) and $p = 2^N$.

Neglecting communication, running Algorithm 28 to compute $s := \sum_{i=0}^{n-1} a_i$ yields

$$
\begin{aligned}
S(p) &= \frac{q \cdot 2^N - 1}{(q-1) + N} , \\
E(p) &= \frac{q - \frac{1}{2^N}}{(q-1) + N} .
\end{aligned}
$$

Proof: Let T_+ denote the time for a (serial) addition

$$
\begin{aligned}
\Rightarrow T(1) &= (n-1) \cdot T_+ , \\
&\quad \{ \; N = 0, \, q = n \text{ means serial summation } \} \\
T(p) &= (q-1) \cdot T_+ \\
&\quad \{ \text{ serial addition } a_i^{(0)} := \sum_{j=iq}^{(i+1)q-1} a_j \text{ of } q \\
&\quad \text{numbers in each processor } P_i^{(0)} \; \} \\
&\quad + N \cdot T_+ \\
&\quad \{ \text{ at most one add per processor in each} \\
&\quad \text{pass of the } k \text{ loop } \}
\end{aligned}
$$

$$
\begin{aligned}
\Rightarrow S(p) &= \frac{T(1)}{T(p)} = \frac{q \cdot 2^N - 1}{(q-1) + N} , \\
E(p) &= \frac{1}{2^N} \cdot S(p) . \qquad \blacksquare
\end{aligned}
$$

30 ◊ Example:

$$q = 1 \;\Rightarrow\; E(p) = \frac{1 - \frac{1}{2^N}}{N} \approx \frac{1}{N} = \frac{1}{\log_2 p} \quad \{\ p = 2^N\ \}$$

$$q \to \infty \;\Rightarrow\; E(p) \to 1 \quad \{\ \text{unrealistic !}\ \}$$

Realistic: $q \leq C$ (**limited memory** per processor):

$$\begin{aligned} E(p) &= \frac{q - \frac{1}{2^N}}{(q-1) + N} \\ &\leq \frac{C}{N} \\ &\to 0 \quad \text{as } p \to \infty \text{ (i.e., } N \to \infty) \end{aligned}$$

$\Rightarrow$ the fan-in algorithm **does not scale**.

31 ◊ Remark: Since communication is expensive, the real speed-up is much lower.

With the communication model from Remark 18, the overall time for 8-byte (IEEE double precision) computations is

$$T(p) \;=\; (q-1)\cdot T_+ + N\cdot(\alpha + 8\mathrm{B}\cdot\beta + T_+)\ ,$$

where T_+ is the time for one addition, α is the start-up time for communication, and β is the transmission time per byte.

Using $q = n/2^N$ and the { JUQUEEN } values $T_+ = 1/(4 \cdot 1.6\text{GHz}) \sim 0.156\text{ns}$, $\alpha = 2.5\mu\text{s}$, and $\beta = 0.025\text{ns/B}$, one obtains

$$\begin{aligned} T(p) &= \left(\frac{n}{2^N} - 1\right) \cdot 0.156\text{ns} \\ &\quad + N \cdot (2.5\mu\text{s} + 8\text{B} \cdot 0.025\text{ns/B} + 0.156\text{ns}) \\ &= \left(\frac{n}{2^N} - 1\right) \cdot 0.156\text{ns} + N \cdot 2500.356\text{ns} \ . \end{aligned}$$

For $n = 2^{20}$,

$$T(p = 1) \approx 163.6\mu\text{s}, \quad T(p = n) \approx 50.0\mu\text{s},$$

and the time is minimized for $N = 6$, i.e., adding $1\,048\,576$ numbers is fastest on $2^6 = 64$ processors, yielding

$$S_{\max} = S(64) = \frac{T(1)}{T(64)} \approx \frac{163.6\mu\text{s}}{17.56\mu\text{s}} \approx 9.3 \ .$$

Adding more processors slows down the computation.

32 ◊ Remark: The fan-in principle applies to any expression $a_0 \circ a_1 \circ \ldots \circ a_{n-1}$ ($n = 2^N$), where $\circ$ is an **associative** operation, e.g.,

- $\max_{i=0}^{n-1} a_i \qquad (a \circ b = \max\{a, b\})$
- $\min_{i=0}^{n-1} a_i$
- $\prod_{i=0}^{n-1} a_i$
- gcd($m_0, \ldots, m_{n-1}$) { greatest common divisor }
- lcm($m_0, \ldots, m_{n-1}$) { least common multiple }
- $\sum_{i=0}^{n-1} A_i$ { matrices }
- $\prod_{i=0}^{n-1} A_i$

33 ◊ Definition: In a reduction operation,

reduce$(a_0, \ldots, a_{n-1}$; $\circ$; res ; $P_i, i \in I$; $P_k)$,

the processors P_i, $i \in I$, work together to compute the result res $= a_0 \circ \ldots \circ a_{n-1}$, where $\circ$ is an associative operation. The result is made available in P_k.

34 ◊ Theorem: (Backward stability of summation)

If the sum $s := \sum_{j=0}^{n-1} a_j$ of n floating-point numbers a_j is computed with a floating-point arithmetic satisfying

$$\underbrace{\overline{x+y}}_{\text{computed result}} = \underbrace{(x+y)}_{\text{exact result}} \cdot (1+\delta) \quad \text{with } |\delta| \leq \varepsilon \qquad (2.2)$$

for all floating-point numbers x, y and some fixed $0 < \varepsilon \ll 1$ (**machine precision**) then

$$\underbrace{\bar{s}}_{\text{computed sum}} = \sum_{j=0}^{n-1} \underbrace{\widetilde{a}_j}_{\text{perturbed summands}}$$

with

$$\widetilde{a}_j = a_j \prod_{\ell=j}^{n-1} (1+\delta_\ell) \qquad \text{if Algorithm 19 is used} \qquad (2.3)$$

or

$$\widetilde{a}_j = a_j \cdot \prod_{\nu=1}^{N} (1+\delta_{j\nu}) \quad \text{for } n = 2^N \text{ and Alg. 20,} \quad (2.4)$$

where $|\delta_\ell|, |\delta_{j\nu}| \leq \varepsilon$.

35 ◊ Remarks:

a) IEEE arithmetic quarantees (2.2), unless an under-/overflow occurs.

b) To first order,

$$|\widetilde{a}_j - a_j| \lessapprox (n-j)\cdot\varepsilon\cdot|a_j| \quad \text{for Alg. 19,} \quad \text{and}$$

$$|\widetilde{a}_j - a_j| \lessapprox \log_2 n\cdot\varepsilon\cdot|a_j| \quad \text{for Alg. 20 .}$$

Therefore, if no information on the magnitude of the a_j is available, Algorithm 20 gives smaller **bounds** for the errors.

Proof for Theorem 34:

For (2.3), rewrite Algorithm 19 in floating-point arithmetic:

$$\underbrace{\overline{s^{(-1)}}}_{\text{computed quantity}} := 0 \qquad \{\text{ no rounding error }\}$$

for $i = 0 : n-1$

$$\{\text{ assume } \underbrace{\overline{s^{(i-1)}}}_{\text{computed}} = \sum_{j=0}^{i-1} \underbrace{a_j}_{\text{exact inputs}} \cdot \prod_{\ell=j}^{i-1}(1+\delta_\ell),$$

where $|\delta_\ell| \leq \varepsilon$ }

$$\overline{s^{(i)}} := \overline{s^{(i-1)}} \underbrace{\oplus}_{\text{floating-point addition}} a_i$$

{ Then

$$\begin{aligned}
\overline{s^{(i)}} &= (\overline{s^{(i-1)}} + a_i)\cdot(1+\delta_i) \quad \text{for some } |\delta_i| \le \varepsilon \quad \text{by (2.2)}\\
&= \left(\sum_{j=0}^{i-1} a_j \prod_{\ell=j}^{i-1}(1+\delta_\ell) + a_i\right)\cdot(1+\delta_i)\\
&= \sum_{j=0}^{i-1} a_j \prod_{\ell=j}^{i}(1+\delta_\ell) + a_i(1+\delta_i)\\
&= \sum_{j=0}^{i} a_j \prod_{\ell=j}^{i}(1+\delta_\ell) \qquad \}
\end{aligned}$$

{ Therefore $\overline{s} = \overline{s^{(n-1)}} = \sum_{j=0}^{n-1} a_j \prod_{\ell=j}^{n-1}(1+\delta_\ell)$ }

This proves (2.3).

For (2.4), rewrite Algorithm 20 in floating-point arithmetic:

{ $\overline{a_j^{(0)}} \equiv a_j,\ j = 0 : 2^N - 1$; thus $\overline{a_j^{(0)}} = a_j \underbrace{\prod_{\nu=1}^{0}(1+\delta_{j\nu})}_{\equiv 1}$ }

for $k = 1 : N$

{ assume that $\overline{a_i^{(k-1)}} = \sum_{j=i2^{k-1}}^{(i+1)\cdot 2^{k-1}-1} a_j \prod_{\nu=1}^{k-1}(1+\delta_{j\nu})$

holds for $i = 0 : 2^{N-(k-1)} - 1$ }

for $i = 0 : 2^{N-k} - 1$

$$\overline{a_i^{(k)}} := \overline{a_{2i}^{(k-1)}} \oplus \overline{a_{2i+1}^{(k-1)}}$$

{ Thus

$$
\begin{aligned}
\overline{a_i^{(k)}} \;&=\; \left[\sum_{j=i2^k}^{i2^k+2^{k-1}-1} a_j \prod_{\nu=1}^{k-1} (1+\delta_{j\nu}) \right. \\
&\qquad \left. + \sum_{j=i2^k+2^{k-1}}^{(i+1)\cdot 2^k-1} a_j \prod_{\nu=1}^{k-1} (1+\delta_{j\nu}) \right] \cdot (1+\delta_{ik}) \\
&\qquad \text{for some } |\delta_{ik}| \le \varepsilon \quad \text{by (2.2)} \\
&= \sum_{j=i2^k}^{(i+1)\cdot 2^k-1} a_j \prod_{\nu=1}^{k} (1+\delta_{j\nu}) \\
&\qquad \text{with } \delta_{jk} = \delta_{ik} \text{ for all } j = i2^k : (i+1)2^k - 1 \}
\end{aligned}
$$

{ Therefore $\overline{a_0^{(N)}} = \sum_{j=0}^{2^N-1} a_j \prod_{\nu=1}^{N} (1+\delta_{j\nu})$ }

This proves (2.4). ■

Check your understanding. You should know . . .

- How does the fan-in principle work?
- How fan-in works when $p < n$.
- Whether fan-in scales and why not.
- What a "global reduce" is.
- There is an issue regarding numerical round-off errors. What is it, exactly?
- What is a deadlock, and how can we avoid these?
- That we distinguish between blocking and non-blocking communications
- Assume that you have to compute the inner product of two vectors x and y, which are distributed over the processors. (This is a very common task!) How can we perform this using fan-in?

2.7.2 Parallel matrix multiplication

Given $A = (a_{ik}) \in \mathbb{R}^{n\times m}$ and $B = (b_{kj}) \in \mathbb{R}^{m\times q}$,

compute $C = (c_{ij}) = A \cdot B \in \mathbb{R}^{n\times q}$, where

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

Motivation: Stochastic models are often described by a **state change matrix** A, where a_{ij} is the probability that an object moves from state j to state i.

For a given initial distribution $s^{(0)}$ of the objects to the states, a new distribution $s^{(1)}$ is obtained by $s^{(1)} = A \cdot s^{(0)}$.

Analogously, for the nth distribution we have $s^{(n)} = A \cdot s^{(n-1)} = \ldots = A^n \cdot s^{(0)}$, and thus the state change matrix from $s^{(0)}$ to $s^{(n)}$ is $A^n = A \cdot \ldots \cdot A$.

Such models are also used in telecommunication and for search engines.

36 ◊ Definition: A **partitioning** of a set I is a system of sets, $\{ I_r : r = 1 : R \}$, such that

(i) $I_r \neq \emptyset$ for all r,

(ii) $I_r \cap I_s = \emptyset$ for all $r \neq s$, and

(iii) $\displaystyle\bigcup_{r=1}^{R} I_r = I$.

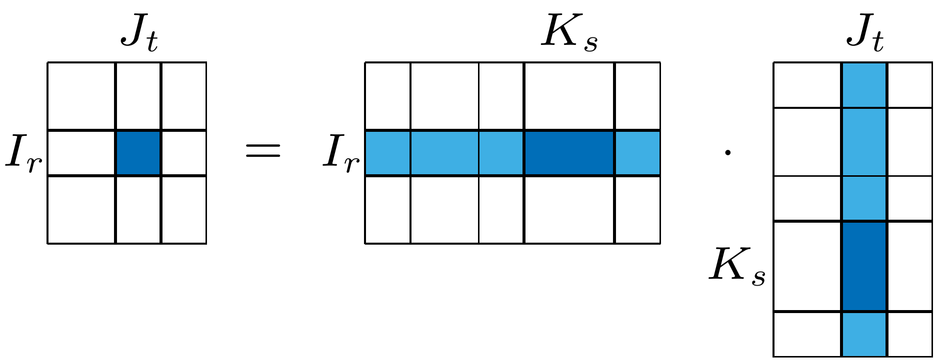
Let the parallel computer contain $p = R \cdot S \cdot T$ processors $P_{r,s,t}$, $r = 1 : R$, $s = 1 : S$, $t = 1 : T$, and let

$$\begin{array}{ll} \{\, I_r\,,\ r = 1 : R \,\} & \text{be a partitioning of } \{1, \ldots, n\}, \\ \{\, K_s\,,\ s = 1 : S \,\} & \text{be a partitioning of } \{1, \ldots, m\}, \qquad \text{and} \\ \{\, J_t\,,\ t = 1 : T \,\} & \text{be a partitioning of } \{1, \ldots, q\}. \end{array}$$

Each processor $P_{r,s,t}$ is assumed to hold the submatrices (“**blocks**”)

$$\begin{aligned} A_{rs} &:= (a_{ik})_{i \in I_r, k \in K_s} \qquad \text{and} \\ B_{st} &:= (b_{kj})_{k \in K_s, j \in J_t} \,. \end{aligned}$$

Then each block $C_{rt} := (c_{ij})_{i \in I_r, j \in J_t}$ of C is given by



$$C_{rt} = \sum_{s=1}^{S} A_{rs} \cdot B_{st}$$

37 ◊ Algorithm: Parallel matrix–matrix multiplication; code for all $P_{r,s,t}$

{ each processor $P_{r,s,t}$ computes the product of the "local" subblocks }

$\widetilde{C}_{rt} := A_{rs} \cdot B_{st}$

{ the sum of these products gives C_{rt} }

$\textbf{reduce}(\ \widetilde{C}_{rt}\ ;\ +\ ;\ C_{rt}\ ;\ P_{r,\sigma,t} : \sigma = 1 : S\ ;\ \underbrace{P_{r,?,t}}_{\text{any of the processors participating in the reduction}}\ )$

38 ◊ Remarks:

a) Each processor participates in only one reduction
$\Rightarrow$ no deadlock.

b) The workload is balanced iff

$$\underbrace{|I_r| \cdot |K_s| \cdot |J_t|}_{\text{work for local multiplikation}} \quad \text{is roughly equal for all } P_{r,s,t}\ .$$

(This implies that $\underbrace{|I_r| \cdot |J_t|}_{\text{local work for reduction}}$ is balanced, too.)

In general, one can achieve

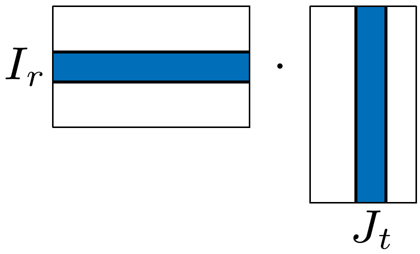
$$|\ |I_r| - |I_{r'}|\ | \ \leq\ 1\ , \qquad r, r' = 1 : R,$$

$$
\begin{aligned}
|\,|K_s| - |K_{s'}|\,| &\leq 1\,, \qquad s, s' = 1 : S, \\
|\,|J_t| - |J_{t'}|\,| &\leq 1\,, \qquad t, t' = 1 : T.
\end{aligned}
$$

---

39 ♢ Example: (Special cases)

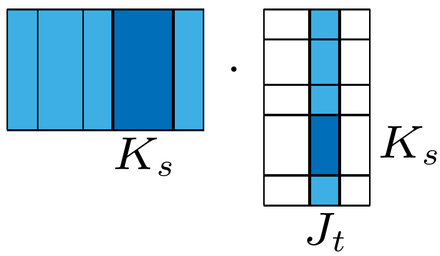
1. $S = 1$, i.e., $K_1 = \{1, \ldots, m\}$



   Each processor holds complete rows of A and complete columns of B

   $\Rightarrow P_{rt}$ computes C_{rt} locally, **no communication at all**

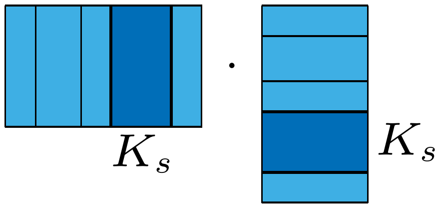
2. $R = 1$, i.e., $I_1 = \{1, \ldots, n\}$



   Processors hold complete columns of A (and, perhaps, of C)

   $\Rightarrow T$ reduction operations with n-by-$|J_t|$ matrices, each involving S processors

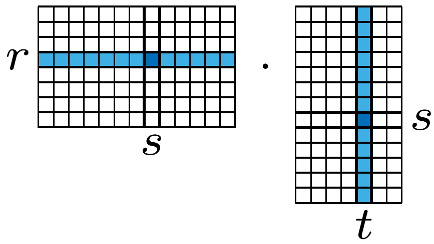
3. $R = T = 1$



Processors hold complete columns of A and complete rows of B

$\Rightarrow$ only one reduction operation with n-by-q matrices involving all S processors

4. $R = n, S = m, T = q$



Each of the $n \cdot m \cdot q$ processors holds a singly entry of A and a single entry of B

$\Rightarrow$ $n \cdot q$ reduction operations with real numbers, each involving m processors

$\Rightarrow$ total time

$$T(p = n \cdot m \cdot q) = \mathcal{O}(\log m) \ ,$$

but not practical due to huge p.

40 ◊ Remark: Algorithm 37 is “wasting” memory.

On a serial machine, we need $nm + mq + nq$ elements to hold A, B, and C.

In Algorithm 37, processor $P_{r,s,t}$ holds A_{rs}, B_{st}, and $\widetilde{C}_{rt}$:

$$|I_r| \cdot |K_s| + |K_s| \cdot |J_t| + |I_r| \cdot |J_t|$$

$\Rightarrow$ total memory required:

$$\sum_{r=1}^{R} \sum_{s=1}^{S} \sum_{t=1}^{T} (|I_r| \cdot |K_s| + |K_s| \cdot |J_t| + |I_r| \cdot |J_t|)$$
$$= T \cdot nm + R \cdot mq + S \cdot nq$$

41 ◊ Remark: Matrix–vector multiplication is a special case with $q = 1$, i.e., $T = 1$.

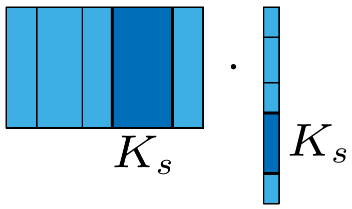
Special cases:

1. $S = 1$

inner product

$\Rightarrow$ no communication

2. $R = 1$



"**column sweep**", **outer product**

$\Rightarrow$ one reduction with length-n vectors involving all S processors

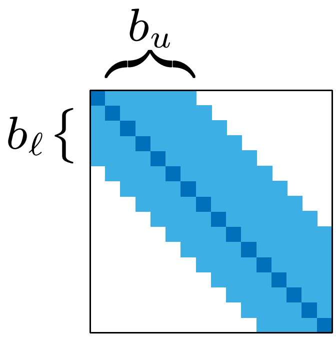
For both variants, the memory requirements are

$$\sum_{r=1}^{R} \sum_{s=1}^{S} (|I_r| \cdot |K_s| + |K_s| + |I_r|) = nm + R \cdot m + S \cdot n$$

$\Rightarrow$ not much higher than in the serial algorithm $(nm + m + n)$, if $R \ll n$, $S \ll m$.

42 ◊ Definition: $A = (a_{ij}) \in \mathbb{R}^{n \times n}$ is **banded** with **lower bandwidth** b_ℓ and **upper bandwidth** b_u if

$$a_{ij} = 0 \quad \text{whenever } i > j + b_\ell \text{ or } j > i + b_u \ .$$



43 ◊ Remarks:

a) Banded matrices are often stored by diagonals:
In an n-by-$(b_\ell + 1 + b_u)$ array, each column holds a non-zero diagonal of A.

b) The product of two banded matrices is again banded.

c) Using the "storage by diagonals" scheme, a parallel (banded) matrix–matrix multiplication is possible analogously to Algorithm 37,
and the matrix–vector multiplication carries over as well.
In contrast to the dense case, the blocks A_{rs}, etc., stored in different processors do **overlap**.

Check your understanding. You should know . . .

- What is a partitioning?
- Arithmetic work for matrix-matrix multiplication
- The general parallel matrix-matrix multiplication algorithm using three partitionings for $\{1, \ldots, n\}$, $\{1, \ldots, m\}$, and $\{1, \ldots, q\}$
- Work per processor?
- How many reduction operations do we perform?
- Special cases when one of the partitionings is just one set.
- Special case when we have nmq processors
- The inner and outer product form of parallel matrix–vector multiplication
- Storage requirements
- Why banded matrices are special