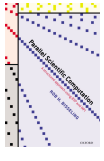
Parallel Inner Product Computation
(PSC §1.3)

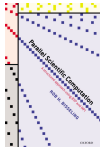


Inner product of two vectors

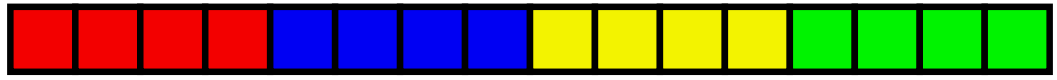
The inner product of two vectors $\mathbf{x} = (x_0, \ldots, x_{n-1})^{\mathrm{T}}$ and $\mathbf{y} = (y_0, \ldots, y_{n-1})^{\mathrm{T}}$ is defined by

$$\alpha = \mathbf{x}^{\mathrm{T}}\mathbf{y} = \sum_{i=0}^{n-1} x_i y_i.$$

Here, 'T' denotes transposition. All vectors are column vectors.



Data distributions for vector



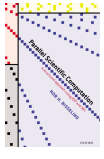
block

P(0) *P(1)* *P(2)* *P(3)*

cyclic

p = # processors = 4

n = vector length = 16



Block distribution

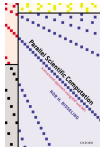
- The block distribution is defined by

$$x_i \longmapsto P(i \operatorname{div} b), \text{ for } 0 \leq i < n.$$

  Here the div operator stands for dividing and rounding down: $i \operatorname{div} b = \lfloor i/b \rfloor$.

- The block size is $b = \lceil \frac{n}{p} \rceil$ = $\frac{n}{p}$ rounded up.

- For $n = 9$ and $p = 4$, this assigns 3, 3, 3, 0 vector components to the processors, respectively. You may blink at an empty processor, but this distribution is just as good as 3, 2, 2, 2. Really!



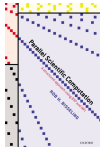
Cyclic distribution

The cyclic distribution is defined by

$$x_i \longmapsto P(i \bmod p), \text{ for } 0 \leq i < n.$$

This distribution is easiest to compute. Note the advantage of starting to count at zero: the formula becomes very simple.

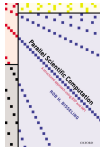
Some kids have been raised to start counting at zero.



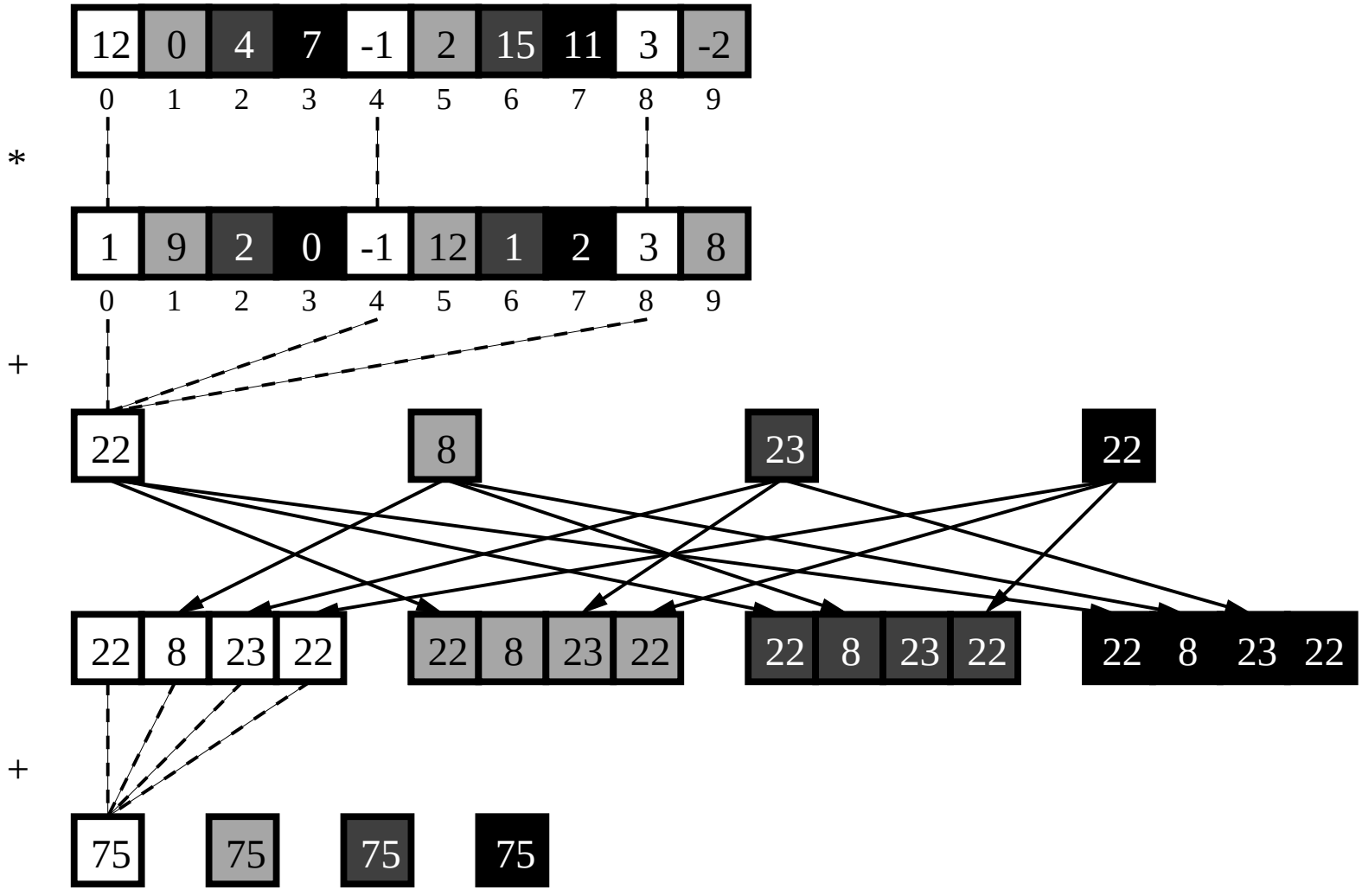
Parallel inner product computation

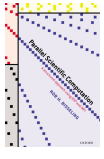
Design decisions:

- Assign x_i and y_i to the same processor, for all i.
  This makes computing $x_i \cdot y_i$ a local operation.
  Thus $\mathrm{distr}(\mathbf{x}) = \mathrm{distr}(\mathbf{y})$.
- Choose a distribution with an even spread of vector components. Both block and cyclic distributions are fine. We choose cyclic, following the way card players deal their cards.
- The data distribution naturally leads to a work distribution and a parallel algorithm.



Example for $n = 10$ and $p = 4$



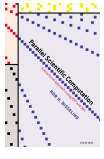


Parallel inner product algorithm for $P(s)$

input: $\mathbf{x}, \mathbf{y}$: vector of length n,
$\mathrm{distr}(\mathbf{x}) = \mathrm{distr}(\mathbf{y}) = \phi$,
with $\phi(i) = i \bmod p$, for $0 \leq i < n$.

output: $\alpha = \mathbf{x}^T \mathbf{y}$.

(0) $\alpha_s := 0;$
for $i := s$ **to** $n - 1$ **step** p **do**
$\alpha_s := \alpha_s + x_i y_i;$



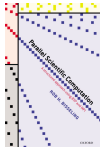
Parallel inner product algorithm for $P(s)$

input: $\mathbf{x}, \mathbf{y}$: vector of length n,
$\mathrm{distr}(\mathbf{x}) = \mathrm{distr}(\mathbf{y}) = \phi$,
with $\phi(i) = i \bmod p$, for $0 \leq i < n$.

output: $\alpha = \mathbf{x}^T \mathbf{y}$.

(0) $\alpha_s := 0;$
for $i := s$ **to** $n - 1$ **step** p **do**
$\alpha_s := \alpha_s + x_i y_i;$

(1) **for** $t := 0$ **to** $p - 1$ **do**
put α_s in $P(t)$;



Parallel inner product algorithm for $P(s)$

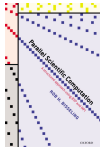
input: $\mathbf{x}, \mathbf{y}$: vector of length n,
$\mathrm{distr}(\mathbf{x}) = \mathrm{distr}(\mathbf{y}) = \phi$,
with $\phi(i) = i \bmod p$, for $0 \leq i < n$.

output: $\alpha = \mathbf{x}^T \mathbf{y}$.

(0) $\alpha_s := 0$;
for $i := s$ **to** $n-1$ **step** p **do**
$\alpha_s := \alpha_s + x_i y_i$;

(1) **for** $t := 0$ **to** $p-1$ **do**
put α_s in $P(t)$;

(2) $\alpha := 0$;
for $t := 0$ **to** $p-1$ **do**
$\alpha := \alpha + \alpha_t$;



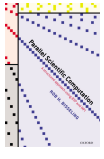
Single Program, Multiple Data (SPMD)

- Only one program text needs to be written. All processors run the same program, but on their own data.
- The program text is parametrised in the processor number s, $0 \leq s < p$, also called processor identity. The actual execution of the program depends on s.
- Processor $P(s)$ computes a partial inner product

$$\alpha_s = \sum_{0 \leq i < n,\ i \bmod p = s} x_i y_i.$$

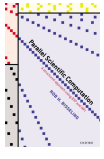
  This computation is completely local.
- The corresponding computation superstep (0) costs

$$2 \left\lceil \frac{n}{p} \right\rceil + l.$$



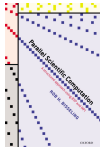
Result needed on all processors

- The partial inner products must be added. This could have been done by $P(0)$, i.e. processor 0.
- Sending the α_s to $P(0)$ is a $(p-1)$-relation. Sending them to $P(*)$, i.e., to all the processors, costs the same. The cost is $(p-1)g + l$.
- Computing α on $P(0)$ costs the same as computing it on all the processors redundantly, i.e. in a replicated fashion. The cost is $p + l$.
- Often, the result is needed on all processors. An example is iterative linear system solvers. The algorithm does just this.



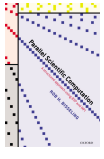
Total BSP cost of inner product

$$T_{\text{inprod}} = 2\left\lceil \frac{n}{p} \right\rceil + p + (p-1)g + 3l.$$



One-sided communication

- The ‘put’ operation involves an active sender and a passive receiver. We assume all puts are accepted. Thus we can define each data transfer by giving only the action of one side.
- No clutter in programs: shorter and simpler texts.
- No danger of the dreaded deadlock. What happens if both processors want to receive first? Deadlock can easily occur in message passing, with an active sender and an active receiver that must shake hands, or kiss. This may cause lots of problems.
- Another one-sided communication is the ‘get’. The name says it all.
- One-sided communications are more efficient.



Summary

- We design algorithms in Single Program, Multiple Data style. Each processor runs its own copy of the same program, on its own data.
- The block and cyclic distributions are commonly used in parallel computing. Both are suitable for an inner product computation.
- The BSP style encourages balancing the communication among the processors. Sending all data to one processor is discouraged. Better: all to all.
- One-sided communications such as puts and gets are easy to use and efficient.