Bulk Synchronous Parallel

Albert-Jan Yzelman

28 October 2011

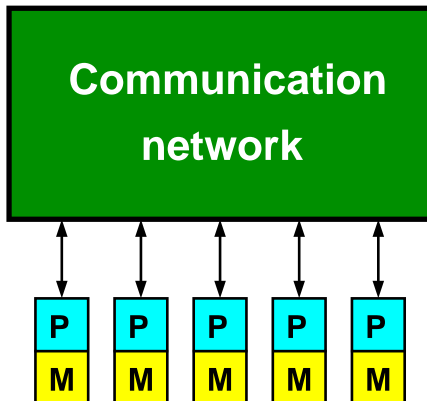
The model

1. The model
2. Distributed-memory programming
3. Shared-memory programming
4. Sparse matrix–vector multiplication

How to program parallel machines? By using bridging models:

- Message Passing Interface (MPI)
- **Bulk Synchronous Parallel** (BSP)

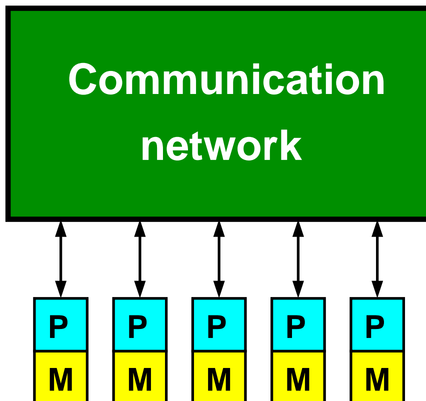
Leslie G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, Volume 33 (1990), pp. 103–111



Assumptions:

- Homogeneous processing speeds

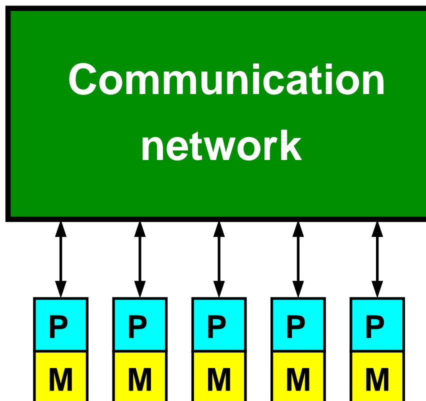
A BSP computer has p processors each running at r flops per second



Assumptions:

- Local processor memory

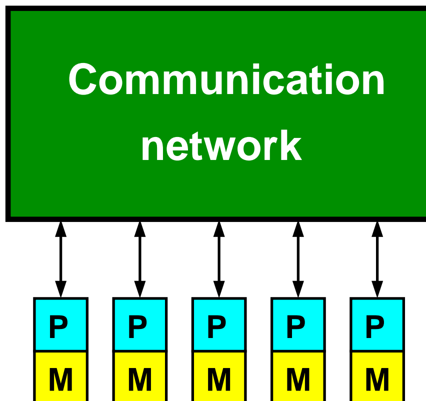
Inter-process communication is only allowed through the network



Assumptions:

- All-to-all network

Utilising the network, processes can synchronise in l time, and send data (full duplex) at a bandwidth rate of g^{-1}

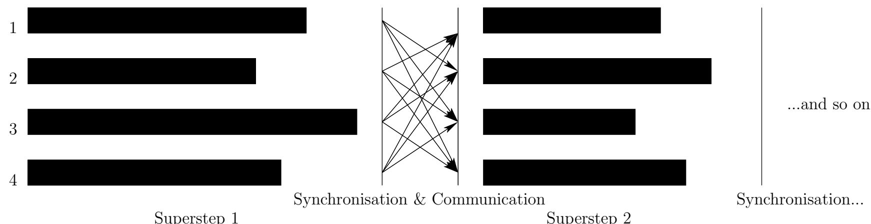


Assumptions:

- Homogeneous processing speeds (p, r)
- Local processor memory
- All-to-all network (l, g)

A Bulk Synchronous Parallel *algorithm*:

- Computations are grouped into supersteps
- An algorithm does not communicate during computation
- Communication only occurs in-between supersteps



The Bulk Synchronous Parallel *cost model*:

- suppose we have a BSP computer with (p, r, l, g),
- suppose the algorithm we run has t supersteps,

The Bulk Synchronous Parallel *cost model*:

- suppose we have a BSP computer with (p, r, l, g),
- suppose the algorithm we run has t supersteps,
- the amount of work in superstep i by process s is $w_i^{(s)}$,

The Bulk Synchronous Parallel *cost model*:

- suppose we have a BSP computer with (p, r, l, g),
- suppose the algorithm we run has t supersteps,
- the amount of work in superstep i by process s is $w_i^{(s)}$,
- the amount of data sent after superstep i by process s is $t_i^{(s)}$,

The Bulk Synchronous Parallel *cost model*:

- suppose we have a BSP computer with (p, r, l, g),
- suppose the algorithm we run has t supersteps,
- the amount of work in superstep i by process s is $w_i^{(s)}$,
- the amount of data sent after superstep i by process s is $t_i^{(s)}$,
- the amount received after superstep i by process s is $c_i^{(s)}$.

The Bulk Synchronous Parallel *cost model*:

- suppose we have a BSP computer with (p, r, l, g),
- suppose the algorithm we run has t supersteps,
- the amount of work in superstep i by process s is $w_i^{(s)}$,
- the amount of data sent after superstep i by process s is $t_i^{(s)}$,
- the amount received after superstep i by process s is $c_i^{(s)}$.

Define $h_i^{(s)} = \max\{t_i^{(s)},\ c_i^{(s)}\}$ (full-duplex).

The Bulk Synchronous Parallel *cost model*:

- suppose we have a BSP computer with (p, r, l, g),
- suppose the algorithm we run has t supersteps,
- the amount of work in superstep i by process s is $w_i^{(s)}$,
- the amount of data sent after superstep i by process s is $t_i^{(s)}$,
- the amount received after superstep i by process s is $c_i^{(s)}$.

Define $h_i^{(s)} = \max\{t_i^{(s)},\ c_i^{(s)}\}$ (full-duplex).

Then the algorithm run time T on this BSP computer is:

$$T = \frac{1}{r} \cdot \sum_{i=1}^{t} \max_s w_i^{(s)}$$

The Bulk Synchronous Parallel *cost model*:

- suppose we have a BSP computer with (p, r, l, g),
- suppose the algorithm we run has t supersteps,
- the amount of work in superstep i by process s is $w_i^{(s)}$,
- the amount of data sent after superstep i by process s is $t_i^{(s)}$,
- the amount received after superstep i by process s is $c_i^{(s)}$.

Define $h_i^{(s)} = \max\{t_i^{(s)},\ c_i^{(s)}\}$ (full-duplex).

Then the algorithm run time T on this BSP computer is:

$$T = \frac{1}{r} \cdot \sum_{i=1}^{t} \max_s w_i^{(s)} + \sum_{i=1}^{t-1} \Big(l$$

The Bulk Synchronous Parallel *cost model*:

- suppose we have a BSP computer with (p, r, l, g),
- suppose the algorithm we run has t supersteps,
- the amount of work in superstep i by process s is $w_i^{(s)}$,
- the amount of data sent after superstep i by process s is $t_i^{(s)}$,
- the amount received after superstep i by process s is $c_i^{(s)}$.

Define $h_i^{(s)} = \max\{t_i^{(s)},\ c_i^{(s)}\}$ (full-duplex).

Then the algorithm run time T on this BSP computer is:

$$T = \frac{1}{r} \cdot \sum_{i=1}^{t} \max_s w_i^{(s)} + \sum_{i=1}^{t-1} \left(l + g \cdot \max_s h_i^{(s)} \right)$$

Distributed-memory programming

1 The model

2 Distributed-memory programming

3 Shared-memory programming

4 Sparse matrix–vector multiplication

BSP primitives:

- `bsp_init(...)`
  `bsp_begin(P)`
  `bsp_end()`
  `bsp_abort()`

BSP primitives:

- `bsp_init(...)`
  `bsp_begin(P)`
  `bsp_end()`
  `bsp_abort()`
- `bsp_nprocs()`
  `bsp_pid()`

BSP primitives:

- `bsp_init(...)`
  `bsp_begin(P)`
  `bsp_end()`
  `bsp_abort()`
- `bsp_nprocs()`
  `bsp_pid()`
- `bsp_sync()`

BSP primitives:

- `bsp_init(...)`
  `bsp_begin(P)`
  `bsp_end()`
  `bsp_abort()`
- `bsp_nprocs()`
  `bsp_pid()`
- `bsp_sync()`
- `bsp_put(source, dest, dest_PID)`
  `bsp_get(source, source_PID, dest)`

BSP primitives:

- `bsp_init(...)`
  `bsp_begin(P)`
  `bsp_end()`
  `bsp_abort()`
- `bsp_nprocs()`
  `bsp_pid()`
- `bsp_sync()`
- `bsp_put(source, dest, dest_PID)`
  `bsp_get(source, source_PID, dest)`
- `bsp_send(data, dest_PID)`
  `bsp_qsize()`
  `bsp_move()`

Example: inner product kernel

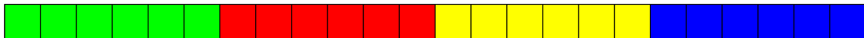
Goal: calculate $\alpha = x^T \cdot y$ with $x, y \in \mathbb{R}^n$, in parallel.

Definition (Vector distribution)

Let $x \in \mathbb{R}^n, p \in \mathbb{N}$. A distribution ϕ of x over p processors is a function:

$$\phi : [0, n-1] \rightarrow [0, p-1]$$

A vector distribution gives us a way to decide which vector elements are stored at which of the p processors.



Definition (Block distribution)

Let x, p as before. Then the distribution with

$$\phi_{\text{block}}(i) = i \text{ div } \lceil n/p \rceil$$

defines a block distribution on x.

Definition (Cyclic distribution)

Let x, p as before. Then the distribution with

$$\phi_{\text{cyclic}}(i) = i \bmod p$$

defines a cyclic distribution on x.

We have to choose the distributions ϕ_x and ϕ_y of the vectors x, y.

The inner product kernel continuously multiplies two entries $x_i y_i$,

We have to choose the distributions ϕ_x and ϕ_y of the vectors x, y.

The inner product kernel continuously multiplies two entries $x_i y_i$,

so ϕ_x should be equal ϕ_y.

We have to choose the distributions ϕ_x and ϕ_y of the vectors x, y.

The inner product kernel continuously multiplies two entries $x_i y_i$,

so ϕ_x should be equal ϕ_y.

This means the exact distribution we choose is irrelevant!

Of course, 'irrelevant' up to the demand for load-balance:

the amount of work each processor has to do corresponds exactly with the number of nonzeroes distributed to it, so for all processors s:

$$\#\{i \in \{0, \ldots, n-1\} | \phi(i) = s\} \approx n/p.$$

When n is not a multiple of p, then using the block distribution might give unbalanced results.

So let us assume the cyclic distribution ($\phi_x = \phi_y : i \rightarrow i \bmod p$).

2-step inner product calculation:

1. $P = bsp_nprocs()$
   $t =$ new double$[P]$
   $\alpha = 0$
   for $i = 0$ to x.length do
       add $x_i \cdot y_i$ to α
   for $s = 0$ to P
       $bsp_put(s, \alpha, t, bsp_pid())$
   $bsp_sync()$
2. $\alpha = 0$
   for $s = 0$ to P
       add t_s to α
   return α

2-step inner product calculation:

1. $P = bsp_nprocs()$
   $t =$ new double$[P]$
   $\alpha = 0$
   for $i = 0$ to x.length do
       add $x_i \cdot y_i$ to α
   for $s = 0$ to P
       $bsp_put(s, \alpha, t, bsp_pid())$
   $bsp_sync()$
2. $\alpha = 0$
   for $s = 0$ to P
       add t_s to α
   return α

$\forall s : w_0^{(s)} = p + n/p$

2-step inner product calculation:

1. $P = bsp_nprocs()$
   $t =$ new double$[P]$
   $\alpha = 0$
   for $i = 0$ to x.length do
      add $x_i \cdot y_i$ to α
   for $s = 0$ to P
      $bsp_put(s, \alpha, t, bsp_pid())$
   $bsp_sync()$
2. $\alpha = 0$
   for $s = 0$ to P
      add t_s to α
   return α

$\forall s : w_0^{(s)} = p + n/p,\ t_0^{(s)} = p$

2-step inner product calculation:

1. $P = bsp_nprocs()$
   $t =$ new double$[P]$
   $\alpha = 0$
   for $i = 0$ to x.length do
       add $x_i \cdot y_i$ to α
   for $s = 0$ to P
       $bsp_put(s, \alpha, t, bsp_pid())$
   $bsp_sync()$
2. $\alpha = 0$
   for $s = 0$ to P
       add t_s to α
   return α

$\forall s : w_0^{(s)} = p + n/p,\ t_0^{(s)} = p,\ c_0^{(s)} = p$

2-step inner product calculation:

1. $P = bsp_nprocs()$
   $t =$ new double$[P]$
   $\alpha = 0$
   for $i = 0$ to x.length do
       add $x_i \cdot y_i$ to α
   for $s = 0$ to P
       $bsp_put(s, \alpha, t, bsp_pid())$
   $bsp_sync()$
2. $\alpha = 0$
   for $s = 0$ to P
       add t_s to α
   return α

$$\forall s : w_0^{(s)} = p + n/p,\ t_0^{(s)} = p,\ c_0^{(s)} = p,\ h_0^{(s)} = p$$

2-step inner product calculation:

1. $P = bsp_nprocs()$
   $t =$ new double$[P]$
   $\alpha = 0$
   for $i = 0$ to x.length do
      add $x_i \cdot y_i$ to α
   for $s = 0$ to P
      $bsp_put(s, \alpha, t, bsp_pid())$
   $bsp_sync()$
2. $\alpha = 0$
   for $s = 0$ to P
      add t_s to α
   return α

$\forall s : w_0^{(s)} = p + n/p,\ t_0^{(s)} = p,\ c_0^{(s)} = p,\ h_0^{(s)} = p\ ,\ w_1^{(s)} = p;$

2-step inner product calculation:

1. $P = bsp_nprocs()$
   $t =$ new double$[P]$
   $\alpha = 0$
   for $i = 0$ to x.length do
       add $x_i \cdot y_i$ to α
   for $s = 0$ to P
       $bsp_put(s, \alpha, t, bsp_pid())$
   $bsp_sync()$
2. $\alpha = 0$
   for $s = 0$ to P
       add t_s to α
   return α

$$\forall s : w_0^{(s)} = p + n/p,\ t_0^{(s)} = p,\ c_0^{(s)} = p,\ h_0^{(s)} = p\ ,\ w_1^{(s)} = p;$$

$$T_{\text{inprod}} = 1/r(2p + n/p) + l + gp.$$

The cost of many other BSP algorithms can be expressed:

$$T_{\mathrm{inprod}} = 1/r(2p + n/p) + l + gp$$

$$T_{\mathrm{LU}} \approx 1/r(\frac{2n^3}{3p} + \frac{3n^2}{2\sqrt{p}}) + 8ln + g\frac{3n^2}{\sqrt{p}}$$

$$T_{\mathrm{FFT},\ 1<p\leq\sqrt{n}} = 1/r\frac{5n\log_2 n}{p} + 3l + g\frac{2n}{p}$$

$$T_{\mathrm{SpMV}} = 1/r\left(2\max_s \mathrm{nz}_s + \max_s w_2^{(s)}\right) + 2l + g(\max_s h_0^{(s)} + \max_s h_1^{(s)})$$

$$T_{\mathrm{SpMV_sh}} = 1/r\left(2\max_s \mathrm{nz}_s + \max_s w_2^{(s)}\right) + l + g\max_s\left(h_0^{(s)} + h_1^{(s)}\right)$$

See:

- Rob H. Bisseling, *Parallel Scientific Computation: A Structured Approach using BSP and MPI*, Oxford University Press, 2004
- Yzelman and Bisseling, *An Object-Oriented BSP Library for Multicore Programming*, Concurrency and Computation: Practice and Experience, 2011 (in press)

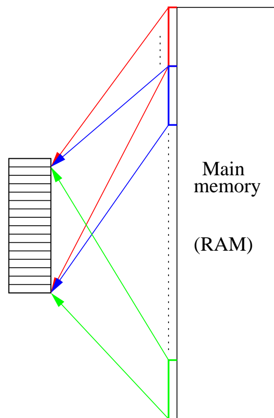
Shared-memory programming

1. The model
2. Distributed-memory programming
3. Shared-memory programming
4. Sparse matrix–vector multiplication

The modulo mapped, or naive, cache ($k = 1$):

Divide the main memory (RAM) in stripes of size L_S.

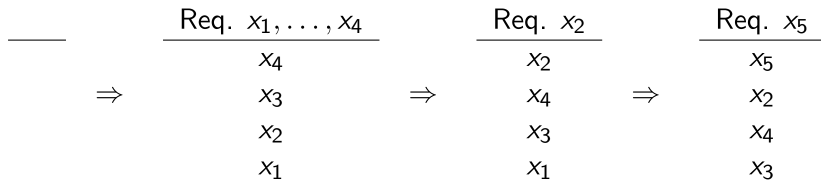
The ith line in RAM is mapped to the cache line $i \bmod L$:



The 'ideal' cache ($k = L$):

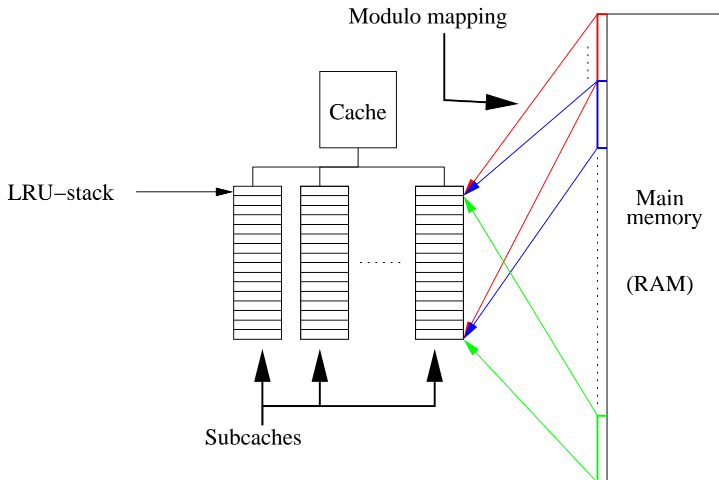
Instead of a naive modulo mapping, new lines are assigned according to pre-defined policy.

For instance, the 'Least Recently Used (LRU)' policy:

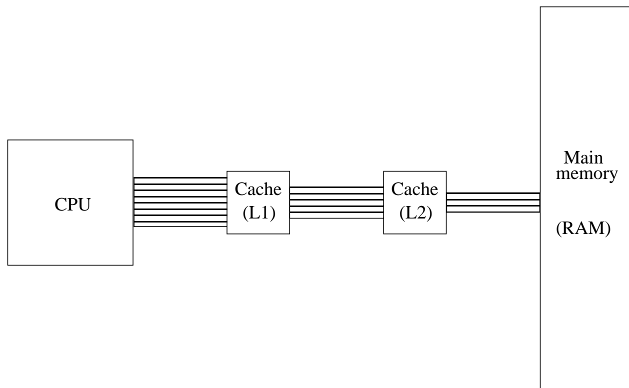


Realistic caches ($1 < k < L$):

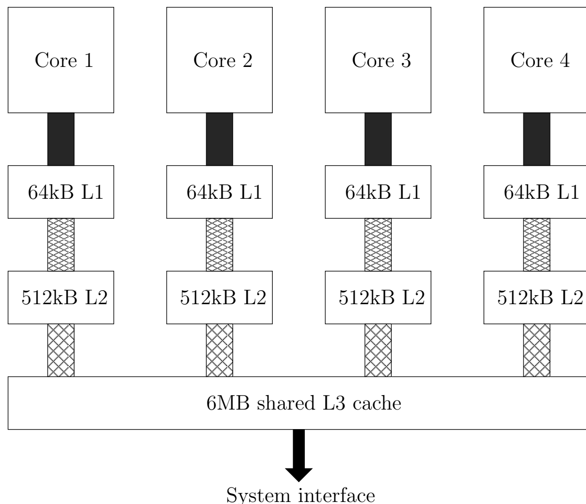
$1 < k < L$, combining modulo-mapping and the LRU policy



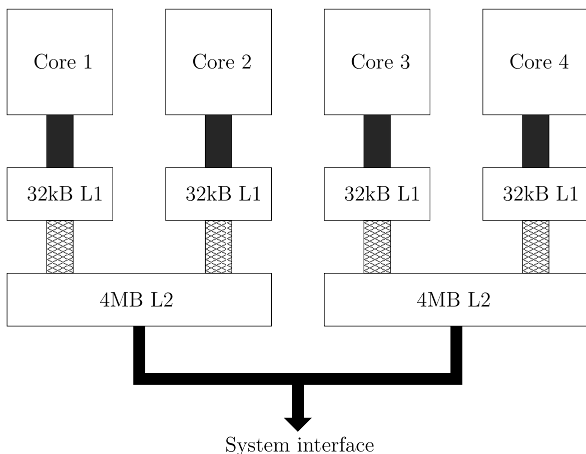
Realistic cache architectures employ multi-level caching:



	Intel Core2 (Q6600)		AMD Phenom II (945e)	
L1:	32kB	$k = 8$	$S = 64$kB	$k = 2$
L2:	4MB	$k = 16$	$S = 512$kB	$k = 8$
L3:	-	-	$S = 6$MB	$k = 48$



BSP: $(4, 3\text{GHz}, l, g)$



BSP: $(4, 2.4\text{GHz}, l, g)$; but Non-Uniform Memory Access (NUMA)!

Some g, l values for different architectures (in ms.):

Processor (p)	l	g
Intel Q6600 (2)	0.013	0.0003
AMD 945e (2)	0.036	0.0004
Intel Q6600 (4)	0.048	0.0005
AMD 945e (4)	0.050	0.0014
Cray T3E (64)	0.052	0.0022

See:

- Rob H. Bisseling, *Parallel Scientific Computation: A Structured Approach using BSP and MPI*, Oxford University Press, 2004
- Yzelman and Bisseling, *An Object-Oriented BSP Library for Multicore Programming*, Concurrency and Computation: Practice and Experience, 2011 (in press)

BSP-style programming

MulticoreBSP is a BSP programming library for shared-memory parallel computing. It defines a new function:

`bsp_direct_get`, a blocking variant of the normal *bsp_get*.

This primitive can *potentially* save a superstep, as no explicit synchronisation is necessary after a BSP get.

In the case of the inner-product kernel:

1. $P = bsp_nprocs()$
   $t =$ new double$[P]$
   $\alpha = 0$
   for $i = 0$ to x.length do
       add $x_i \cdot y_i$ to α
   for $s = 0$ to P
       $bsp_put(s, \alpha, t, bsp_pid())$
   $bsp_sync()$
2. $\alpha = 0$
   for $s = 0$ to P
       add t_s to α
   return α

The shared-memory direct-get variant becomes:

1. $P = bsp_nprocs()$
   $\tilde{\alpha} = 0$
   for $i = 0$ to x.length do
   add $x_i \cdot y_i$ to $\tilde{\alpha}$
   $bsp_sync()$
2. $\alpha = 0$
   for $s = 0$ to P
   add $bsp_direct_get(s, \tilde{\alpha})$ to α
   return α

Not that much effect:

$$T_{\text{inprod}} = 1/r(2p + n/p) + l + gp$$
$$T_{\text{inprod_sh}} = 1/r(p + n/p) + l + gp$$

Alternative programming libraries

There are other dedicated programming models for shared-memory computing, such as, for instance, POSIX threads (PThreads).

One common difference of BSP (and MPI) with dedicated shared-memory libraries,

Alternative programming libraries

There are other dedicated programming models for shared-memory computing, such as, for instance, POSIX threads (PThreads).

One common difference of BSP (and MPI) with dedicated shared-memory libraries, is the existance of a shared memory.

Alternative programming libraries

There are other dedicated programming models for shared-memory computing, such as, for instance, POSIX threads (PThreads).

One common difference of BSP (and MPI) with dedicated shared-memory libraries, is the existance of a shared memory.

BSP ignores this (except for the direct-get), other systems may explicitly model a shared memory (PThreads, UPC, BSPRAM);

However, this opens up the way for some pitfalls.

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

s	the current processor ID,
p	the total number of processors (threads),
n	the size of the input vectors.

Output: $x^T y$

- double $\alpha = 0.0$
- for $i = s$ to n step p
- $\quad \alpha \mathrel{+}= x_i y_i$
- return α

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

- s the current processor ID,
- p the total number of processors (threads),
- n the size of the input vectors.

Output: $x^T y$

- double $\alpha = 0.0$
- for $i = s$ to n step p
- $\alpha \mathrel{+}= x_i y_i$
- return α

Data race!

(For $n = p = 2$, output can be $x_0 y_0$, $x_1 y_1$, **or** $x_0 y_0 + x_1 y_1$)

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

- s the current processor ID,
- p the total number of processors (threads),
- n the size of the input vectors.

Output: $x^T y$

- double $\alpha[p]$
- for $i = s$ to n step p
- $\quad \alpha_s \mathrel{+}= x_i y_i$
- return $\sum_{i=0}^{p-1} \alpha_i$

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

- s the current processor ID,
- p the total number of processors (threads),
- n the size of the input vectors.

Output: $x^T y$

- double $\alpha[p]$
- for $i = s$ to n step p
- $\quad \alpha_s \mathrel{+}= x_i y_i$
- return $\sum_{i=0}^{p-1} \alpha_i$

False sharing!

(Various processors access and update the same cache lines)

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

s	the current processor ID,
p	the total number of processors (threads),
n	the size of the input vectors.

Output: $x^T y$

- double $\alpha[8p]$ (for architectures with $L_s \leq 64$ bytes (in which 8 doubles fit)
- for $i = s$ to n step p
- $\qquad \alpha_{8s} \mathrel{+}= x_i y_i$
- return $\sum_{i=0}^{p} \alpha_{8i}$

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

s	the current processor ID,
p	the total number of processors (threads),
n	the size of the input vectors.

Output: $x^T y$

- double $\alpha[8p]$ (for architectures with $L_s \leq 64$ bytes (in which 8 doubles fit)
- for $i = s$ to n step p
- $\alpha_{8s} \mathrel{+}= x_i y_i$
- return $\sum_{i=0}^{p} \alpha_{8i}$

Inefficient cache use!

(All threads access virtually all cache lines; $\Theta(pn)$ data movement)

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

- s the current processor ID,
- p the total number of processors (threads),
- n the size of the input vectors.

Output: $x^T y$

- double $\alpha[8p]$ (for architectures with $L_s \leq 64$ bytes, in which 8 doubles fit)
- for $i = s \cdot \lceil n/p \rceil$ to $(s+1) \cdot \lceil n/p \rceil$
- α_{8s} += $x_i y_i$
- return $\sum_{i=0}^{p} \alpha_{8i}$

Solution: block distribution

(Now inefficiency only at boundaries; $\mathcal{O}(n + p - 1)$ data movement)

Sparse matrix–vector multiplication

1. The model
2. Distributed-memory programming
3. Shared-memory programming
4. Sparse matrix–vector multiplication

Another example: Sparse matrix–vector multiplication

Definition (Matrix distribution)

Let A be an $m \times n$ matrix, and $I = \{0, \ldots, m-1\} \times \{0, \ldots, n-1\}$. Then

$$\phi : I \to \{0, \ldots, p-1\}$$

defines a matrix distribution of A over p processors. Or equivalently,

$$\phi : I \to \{0, \ldots, M-1\} \times \{0, \ldots, N-1\}$$

with $p = M \cdot N$.

(Equivalent since we can take the processor map $f : (i, j) \to sN + t$.)

Definition (2D matrix distribution)

Let A, m, n, l be as previously. Then a 2D matrix distribution is:

$$\phi(i,j) = (\phi_0(i,j), \phi_1(i,j)).$$

Definition (Cartesian matrix distribution)

Let A, m, n, l be as previously. Then a matrix distribution with:

$$\phi(i,j) = (\phi_0(i), \phi_1(j))$$

is called Cartesian.

Definition (1D row distribution)

$$\phi(i,j) = (\phi_0(i), 0), \text{ with } M = p \text{ and } N = 1.$$

Definition (1D column distribution)

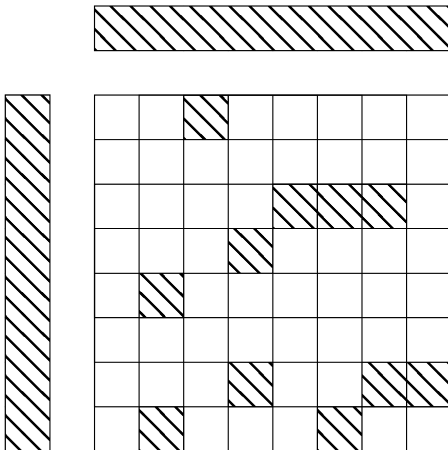
$$\phi(i,j) = (0, \phi_1(j)), \text{ with } M = 1 \text{ and } N = p.$$

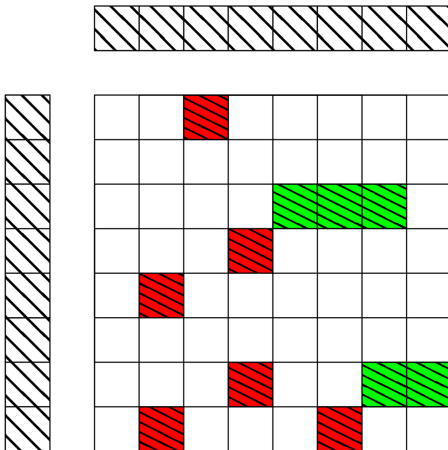
Definition (Block matrix distribution)

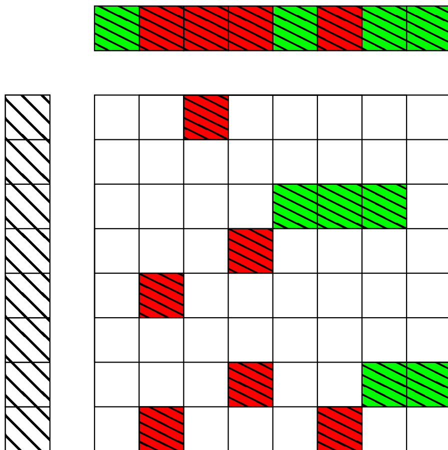
$$\phi(i,j) = (\phi_{\text{block}}(i), \phi_{\text{block}}(j)) = (i \text{ div } \lceil \text{m/M} \rceil, j \text{ div } \lceil \text{n/N} \rceil).$$

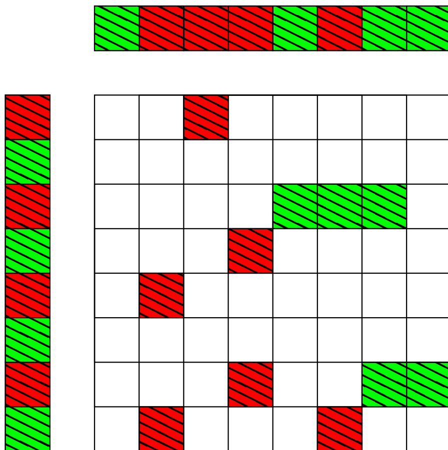
Definition (Cyclic matrix distribution)

$$\phi(i,j) = (\phi_{\text{cyclic}}(i), \phi_{\text{cyclic}}(j)) = (i \bmod M, j \bmod N).$$

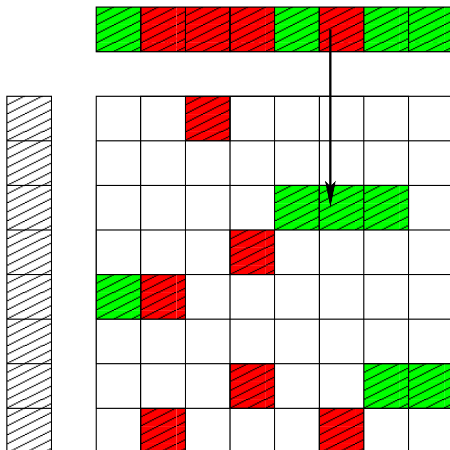






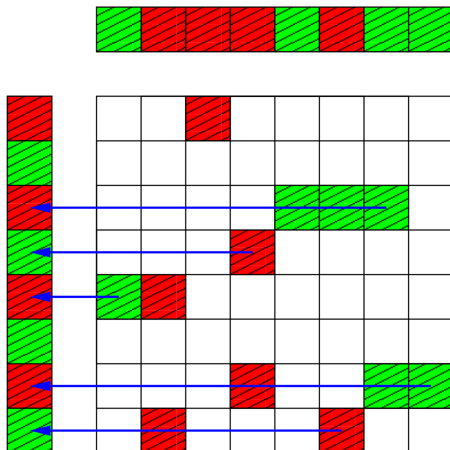


Step 1 (*fan-out*): not all processors have the elements from x they need; processors need to get the missing items. Only one communication is needed, x is distributed well.



Step 2 (*mv*): use received elements from x for multiplication.

Step 3 (*fan-in*): send local results to the correct processors; here, y is distributed cyclically, obviously a bad choice.



The original BSP algorithm:

1. **for all** nonzeroes k **from** A
   - **if** column of k is not local
     - **request** element from x from the appropiate processor

   synchronise
2. **for all** nonzeroes k **from** A
   - do the SpMV for k

   send all non-local row sums to the correct processor
   synchronise
3. **add** all incoming row sums to the corresponding $y[i]$

Alternative (2-step) SpMV algorithm in MulticoreBSP:

1. **for all** nonzeroes k **from** A
   - **if** both row and column of k are local
     - **add** do the SpMV for k
   - **if** column of k is not local
     - **direct get** element from x, and do SpMV for k

   send all non-local row sums to the correct processor
   synchronise
2. **add** all incoming row sums to the corresponding $y[i]$

If interested in writing parallel programs on your computer:

http://www.multicorebsp.com

These slides & further practice material can be found at:

http://people.cs.kuleuven.be/~albert-jan.yzelman/education.php