Shared-memory computing

Albert-Jan Yzelman

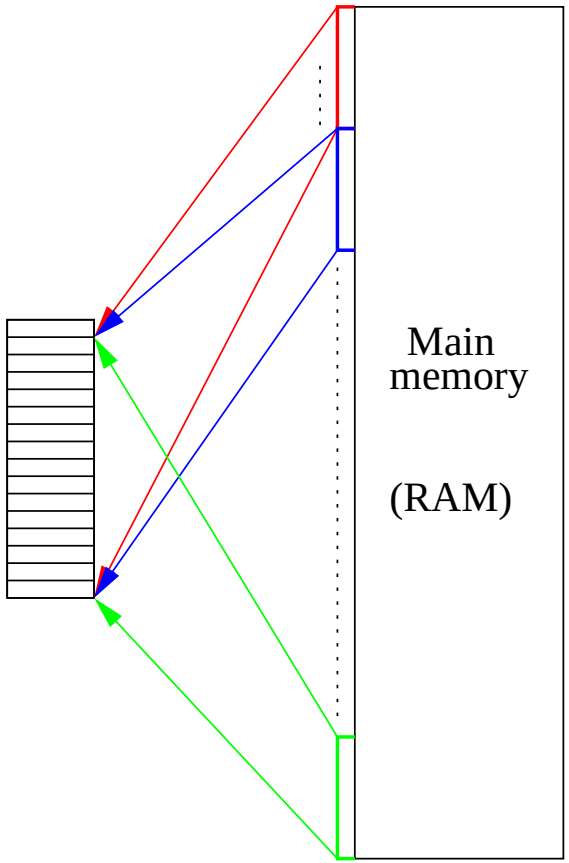
19 September 2012

Caching

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 mod 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:

		Req. $x_1, \ldots, x_4$		Req. x_2		Req. x_5
		x_4		x_2		x_5
	$\Rightarrow$	x_3	$\Rightarrow$	x_4	$\Rightarrow$	x_2
		x_2		x_3		x_4
		x_1		x_1		x_3

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with $k = L = 4$:

x_0

$\Longrightarrow$

———

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with $k = L = 4$:

x_0 a_{00}

x_0

$\Longrightarrow$ $\Longrightarrow$

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with $k = L = 4$:

$$\begin{array}{ccccc} x_0 & & a_{00} & & y_0 \\ & & x_0 & & a_{00} \\ & \Longrightarrow & & \Longrightarrow & x_0 \Longrightarrow \\ \hline & & \hline & & \hline \end{array}$$

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with $k = L = 4$:

$$\begin{array}{ccccccc} x_0 & & a_{00} & & y_0 & & x_1 \\ & & x_0 & & a_{00} & & y_0 \\ & \Longrightarrow & & \Longrightarrow & x_0 & \Longrightarrow & a_{00} & \Longrightarrow \\ \hline & & \hline & & \hline & & x_0 \\ & & & & & & \hline \end{array}$$

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with $k = L = 4$:

$$\begin{array}{ccccccccc} x_0 & & a_{00} & & y_0 & & x_1 & & a_{01} \\ & & x_0 & & a_{00} & & y_0 & & x_1 \\ & \Longrightarrow & & \Longrightarrow & x_0 & \Longrightarrow & a_{00} & \Longrightarrow & y_0 \\ \hline & & \hline & & \hline & & x_0 & & a_{00} \\ & & & & & & \hline & & \hline \\ & & & & & & & & x_0 \end{array} \Longrightarrow$$

Dense matrix–vector multiplication

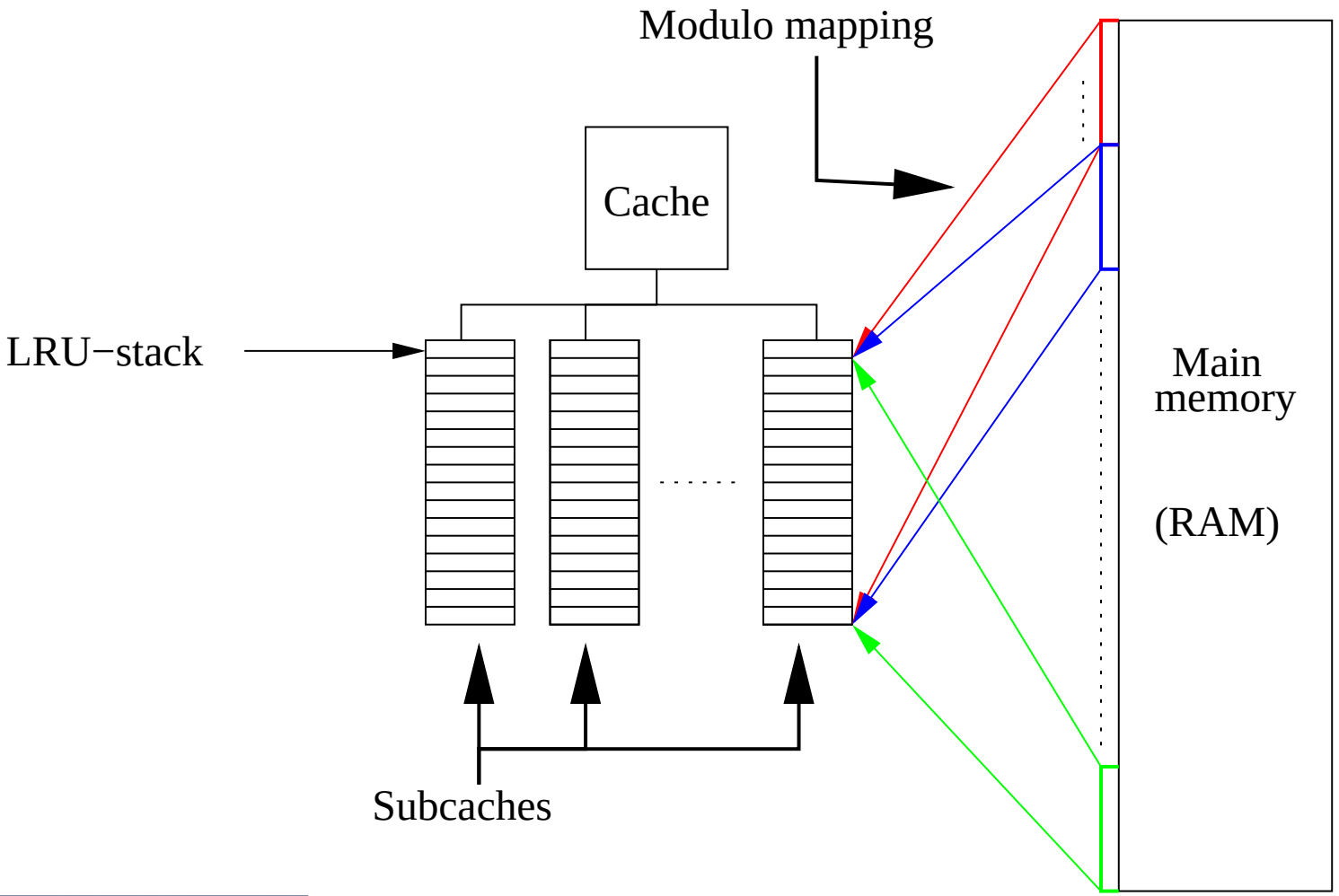
$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with $k = L = 4$:

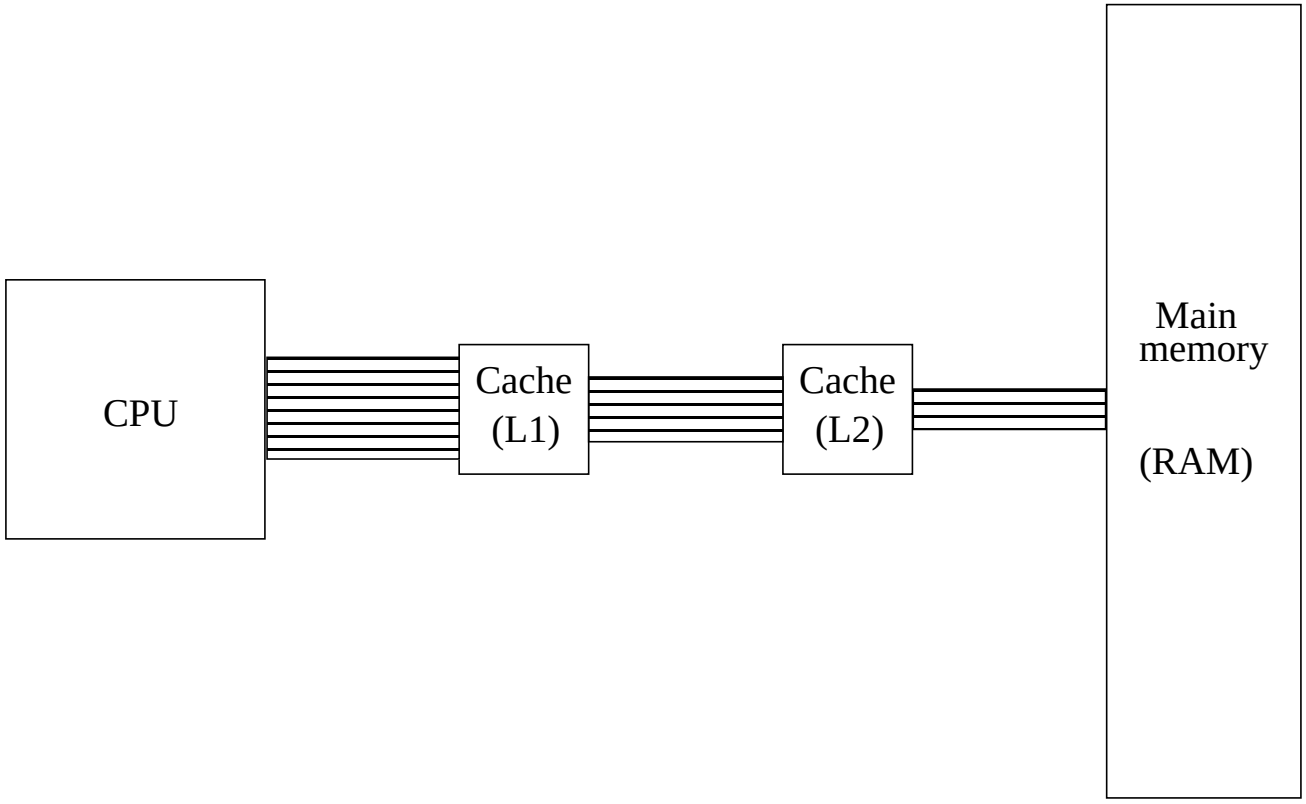
$$\begin{array}{ccccccccccc} x_0 & & a_{00} & & y_0 & & x_1 & & a_{01} & & y_0 \\ & & x_0 & & a_{00} & & y_0 & & x_1 & & a_{01} \\ & \Longrightarrow & & \Longrightarrow & x_0 & \Longrightarrow & a_{00} & \Longrightarrow & y_0 & \Longrightarrow & x_1 \\ \hline & & \hline & & \hline & & x_0 & & a_{00} & & a_{00} \\ & & & & & & \hline & & \hline & & \hline \\ & & & & & & & & x_0 & & x_0 \end{array}$$

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

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

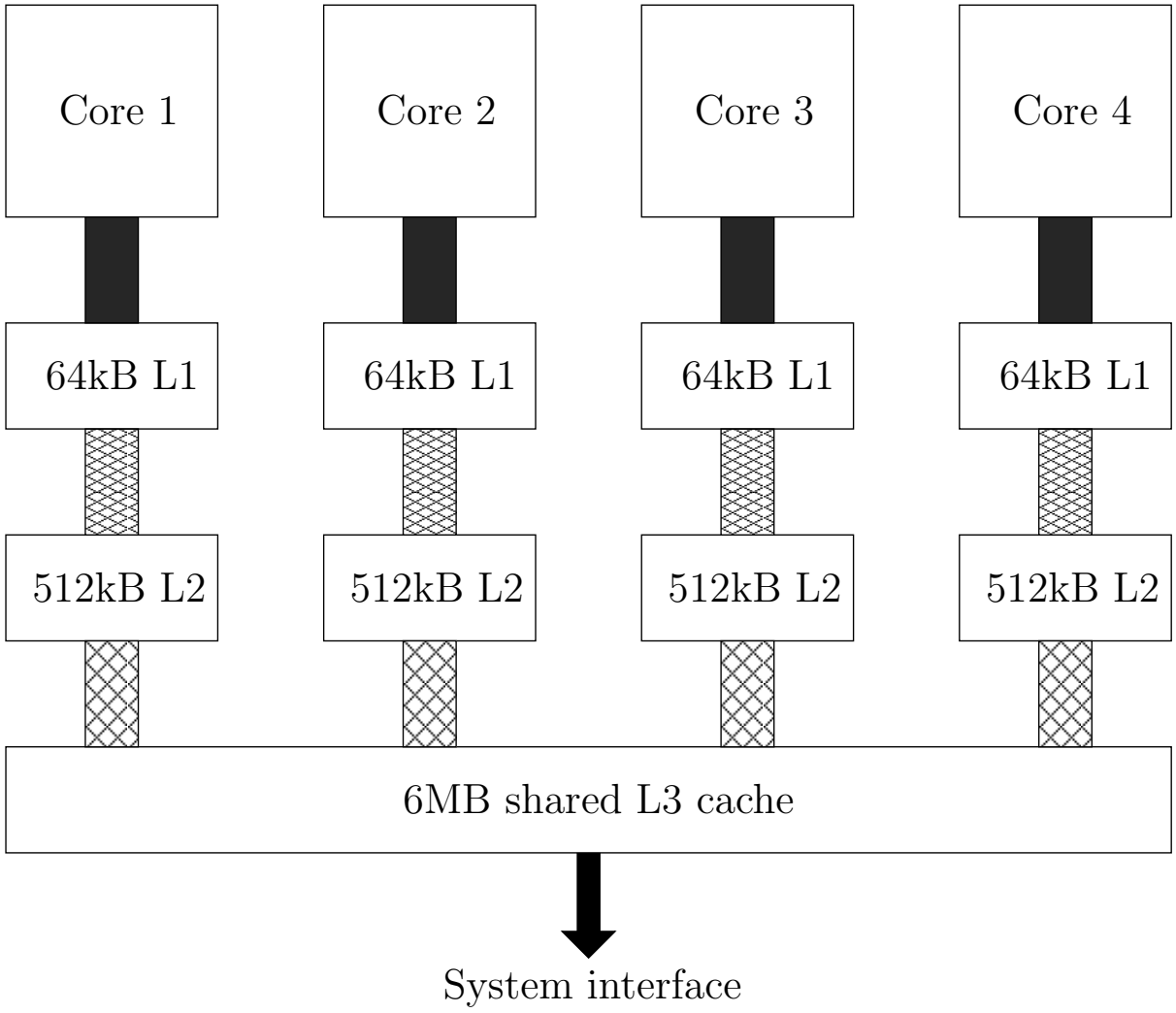


Realistic cache architectures employ multi-level caching:



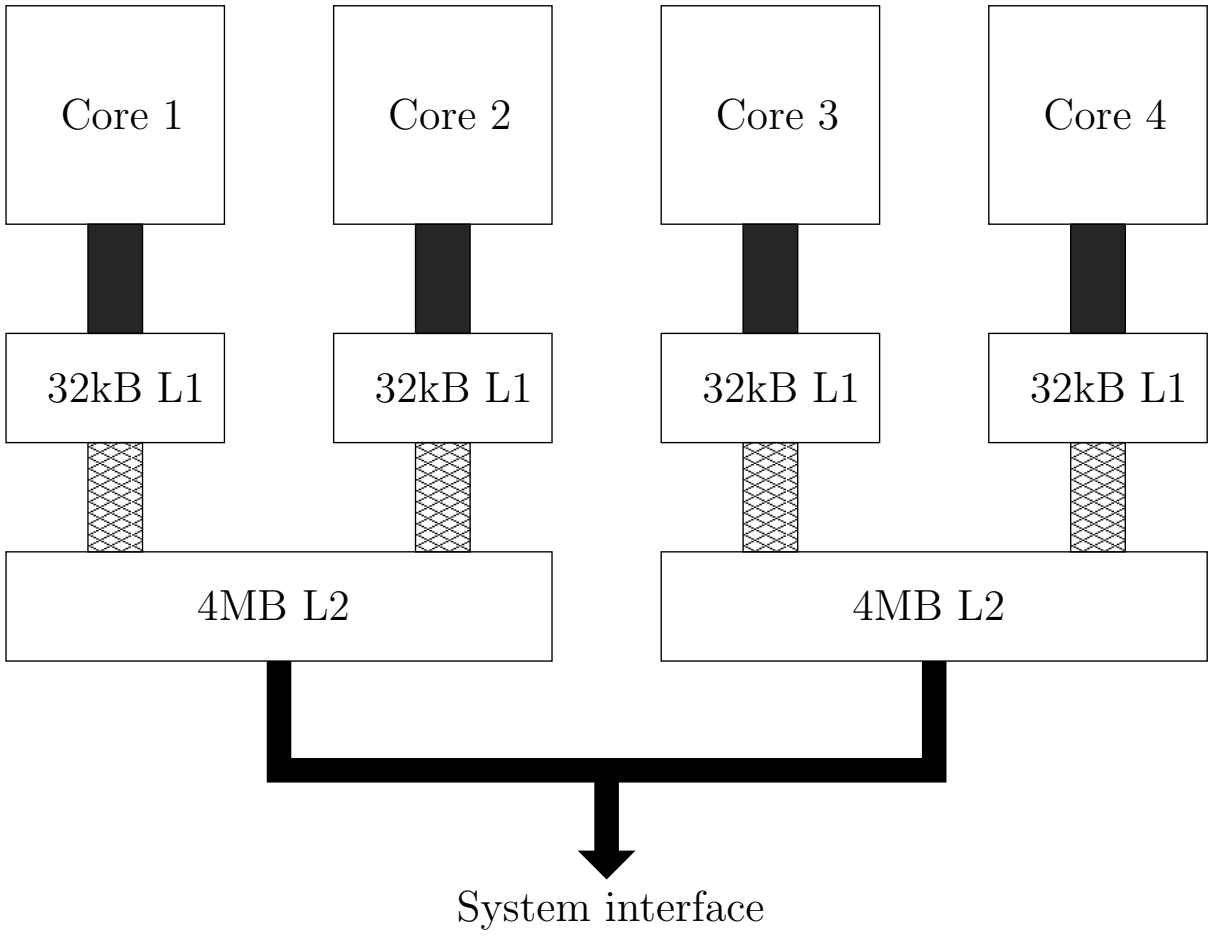
	Intel Core2 (Q6600)		AMD Phenom II (945e)	
L1:	32kB	$k = 8$	$S = 64$kB	$k = 2$
L2:	4MB	$k = 16$	$S = 512$kB	$k = 8$
L3:	-	-	$S = 6$MB	$k = 48$

Most architectures employ shared caches:



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

Most architectures employ shared caches:



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

Bulk Synchronous Parallel

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

Processor (p)	l	g
AMD 945e (2)	0.036	0.0004
Intel Q6600 (2)	0.013	0.0003
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 24(5), pp. 533–553 (2012)

MulticoreBSP is a programming language which explicitly uses the BSP model, but applied to shared-memory computers.

It is based on the distributed-memory Oxford BSP library (BSPlib).

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(dest_t, source, dest, dest_offset, length)`
  `bsp_get(src_t, source, source_offset, dest, length)`

BSP primitives:

- `bsp_init(...)`
  `bsp_begin(P)`
  `bsp_end()`
  `bsp_abort()`
- `bsp_nprocs()`
  `bsp_pid()`
- `bsp_sync()`
- `bsp_put(dest_t, source, dest, dest_offset, length)`
  `bsp_get(src_t, source, source_offset, dest, length)`
- `bsp_send(data, dest_PID)`
  `bsp_qsize()`
  `bsp_move()`

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

MulticoreBSP 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.

The shared-memory direct-get variant of the inner-product 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), OpenMP, or Cilk.

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

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

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

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

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

BSP ignores this (except for the direct-get);

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

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

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

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
- $\alpha \mathrel{+}= x_i y_i$
- 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
- α += $x_i y_i$
- 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
- $\alpha_s \mathrel{+}= x_i y_i$
- 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
- α_s += $x_i y_i$
- 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
- $\alpha_{8s} \mathrel{+}= x_i y_i$
- 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
- α_{8s} += $x_i y_i$
- 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.

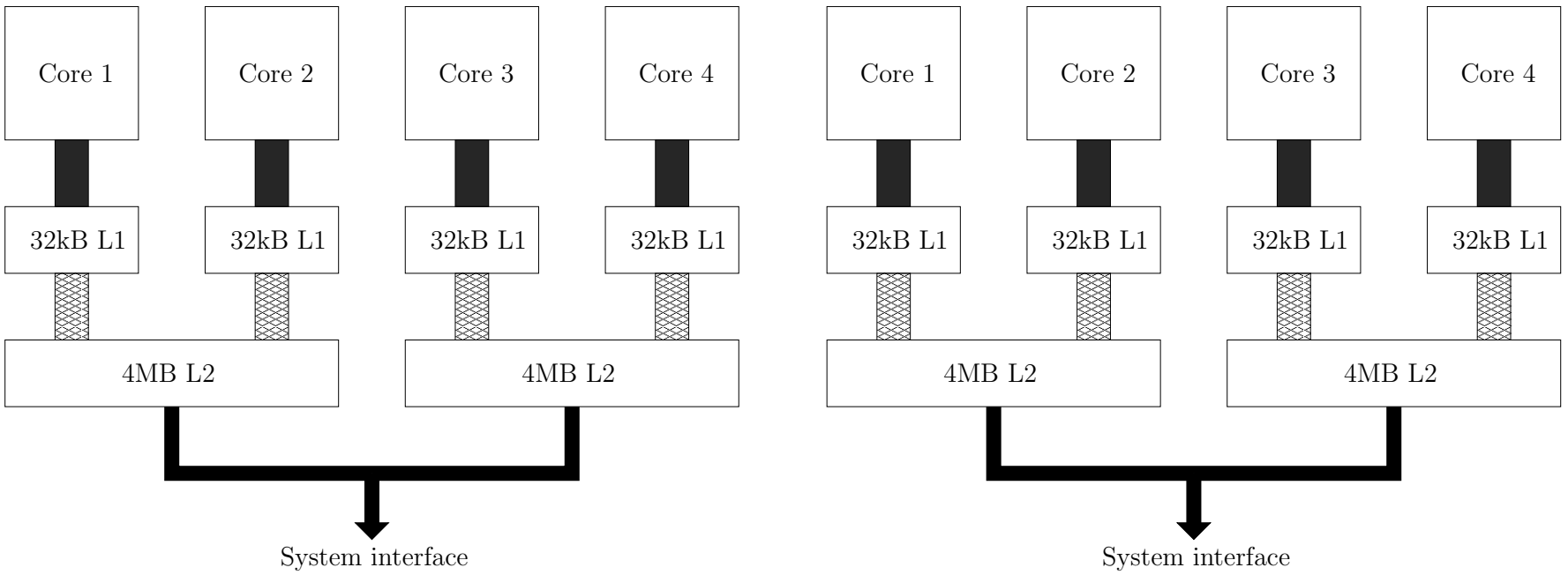
Output: $x^T y$

- 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 $\sum_{i=0}^{p} \alpha_{8i}$

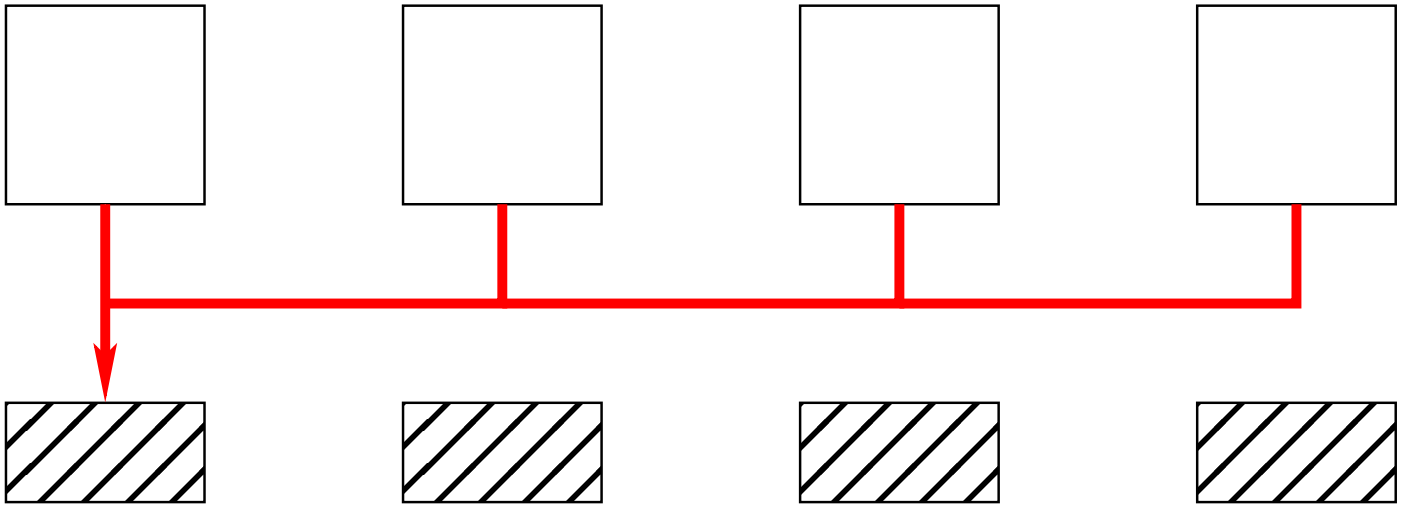
Solution: block distribution

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

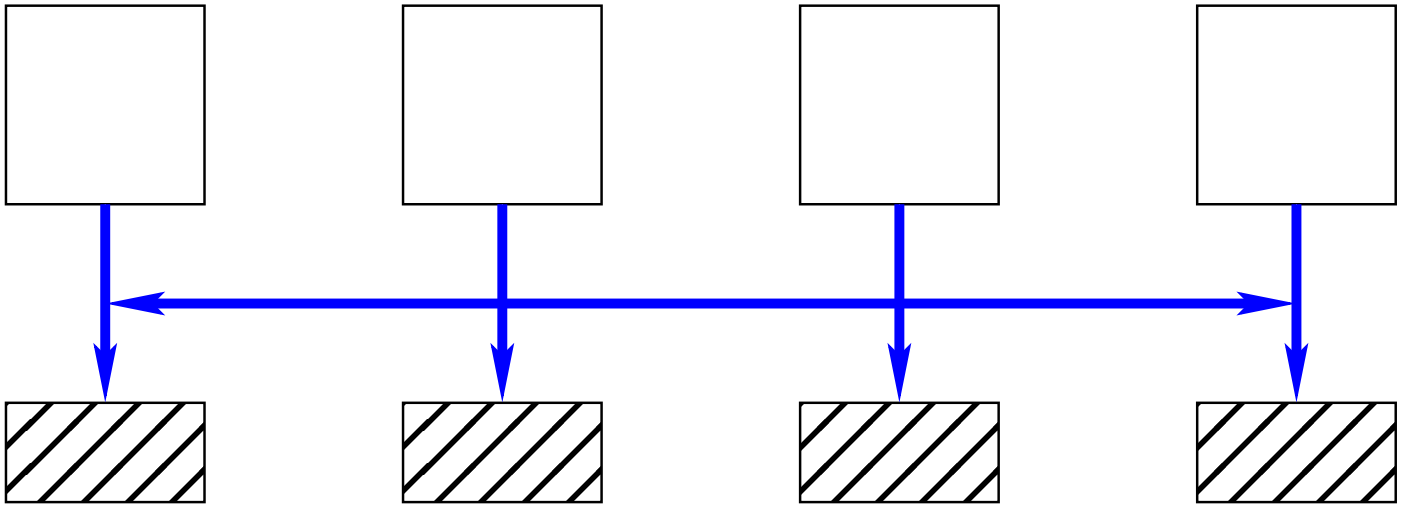
Another problem is non-uniform memory access (NUMA). Consider for instance a multiple-socket machine:



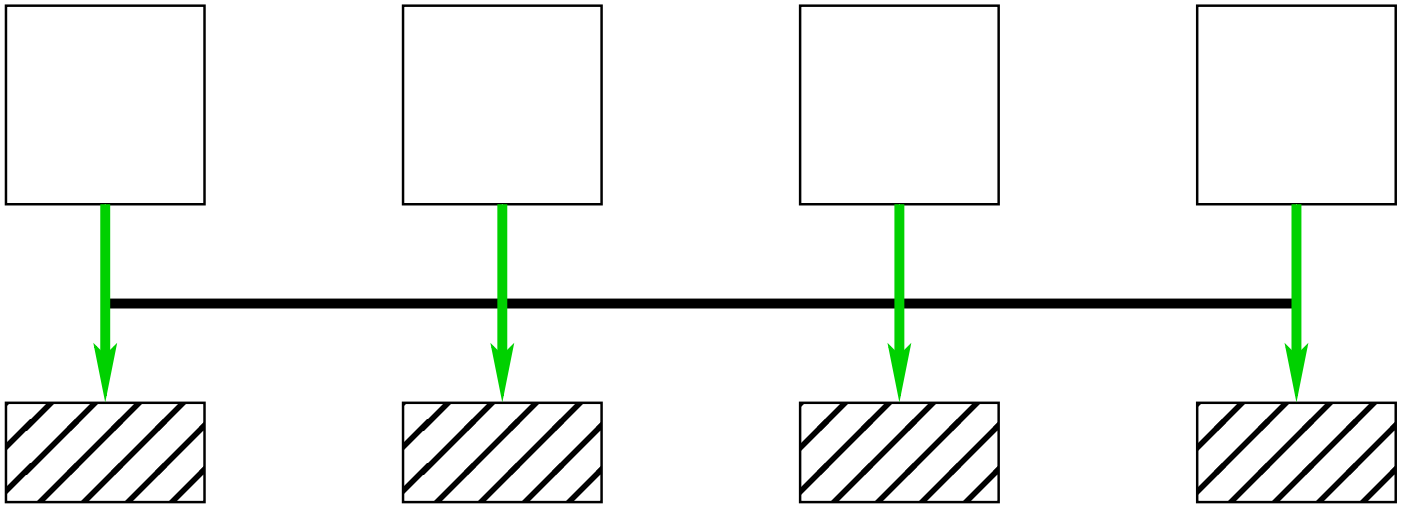
If each processor moves data from and to the same memory element, the effective bandwidth is shared.



Solution (?): interleaved allocation



Scalable solution: local allocation with thread pinning



If each processor moves data from and to its own unique memory element, *the bandwidth multiplies with the number of available elements*.

- Data races,
- false sharing,
- inefficient caching.
- inefficiencies due to NUMA;

using BSP you almost automatically avoid these issues.

MulticoreBSP for C is a library and depends only on the POSIX threads extensions and the POSIX real-time extensions, which are available on most modern systems. Hence only the MulticoreBSP header has to be included to enable BSP programming:

```
#include "mcbsp.h"
int main( int argc, char **argv ) {
    bsp_begin( bsp_nprocs() );
    printf( "Hello world from thread %ld!\n", bsp_pid() );
    bsp_end();
}
```

Linking should include the library and dependencies:

```
gcc hello_world.c libmcbsp1.0.0.a -pthread -lrt
```

(This is different from BSPlib or BSPonMPI)

Conclusion

MulticoreBSP for C (and Java) are freely available:

`http://www.multicorebsp.com`