Shared-memory parallel computing

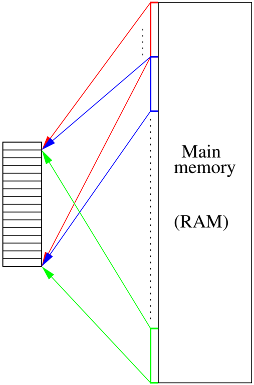
Albert-Jan Yzelman

14th of November, 2014

Shared-memory architectures and paradigms

Caches

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



The ith line in RAM is mapped to the cache line $i \bmod L$, where L is the number of available cache lines.

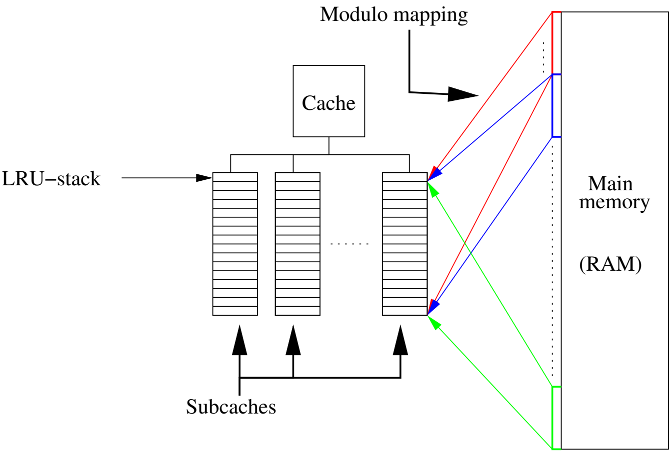
Caches

A smarter cache follows a pre-defined policy instead; 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

Caches

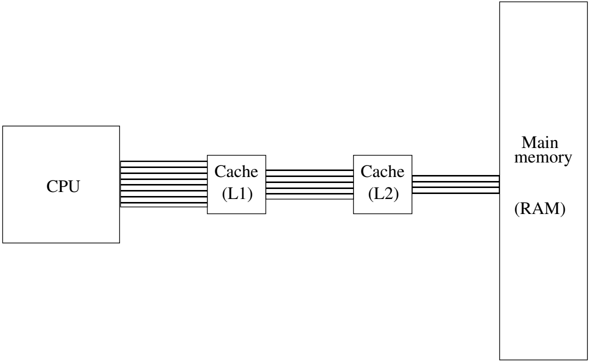
Realistic caches combine modulo-mapping and the LRU policy:



k is the number of subcaches; there are L/k LRU stacks.

Caches

Realistic caches are used within multi-level memory hierarchies:



	Intel Core2 (Q6600)		AMD Phenom II (945e)		Intel Westmere (E7-2830)	
L1:	32kB	$k = 8$	$S = 64$kB	$k = 2$	$S = 256$kB	$k = 8$
L2:	4MB	$k = 16$	$S = 512$kB	$k = 8$	$S = 2$MB	$k = 8$
L3:	-	-	$S = 6$MB	$k = 48$	$S = 24$MB	$k = 24$

Caches and multiplication

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 LRU caching:

x_0

$\Longrightarrow$

Caches and multiplication

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 LRU caching:

$$\begin{array}{c} x_0 \\ \\ \\ \\ \hline \end{array} \Longrightarrow \begin{array}{c} a_{00} \\ x_0 \\ \\ \\ \hline \end{array} \Longrightarrow$$

Caches and multiplication

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 LRU caching:

$$\begin{matrix} x_0 \\ \\ \\ \hline \end{matrix} \Longrightarrow \begin{matrix} a_{00} \\ x_0 \\ \\ \hline \end{matrix} \Longrightarrow \begin{matrix} y_0 \\ a_{00} \\ x_0 \\ \hline \end{matrix} \Longrightarrow$$

Caches and multiplication

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 LRU caching:

x_0		a_{00}		y_0		x_1	
		x_0		a_{00}		y_0	
	$\Longrightarrow$		$\Longrightarrow$	x_0	$\Longrightarrow$	a_{00}	$\Longrightarrow$
						x_0	

Caches and multiplication

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 LRU caching:

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

Caches and multiplication

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 LRU caching:

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
						x_0		a_{00}		a_{00}
___		___		___		___		___		___
								x_0		x_0

Caches and multiplication

When k, L are larger, we can predict:

- lower elements from x are evicted while processing the first row; this causes $\mathcal{O}(n)$ cache misses on $m - 1$ rows.

Caches and multiplication

When k, L are larger, we can predict:

- lower elements from x are evicted while processing the first row; this causes $\mathcal{O}(n)$ cache misses on $m - 1$ rows.

Fix:

- stop processing a row before an element from x would be evicted; first continue with the next rows.

This results in column-wise 'stripes' of the dense A.

Caches and multiplication

When k, L are larger, we can predict:

- lower elements from x are evicted while processing the first row; this causes $\mathcal{O}(n)$ cache misses on $m - 1$ rows.

Fix:

- stop processing a row before an element from x would be evicted; first continue with the next rows.

This results in column-wise 'stripes' of the dense A. But now:

- elements from the vector y can be prematurely evicted; $\mathcal{O}(m)$ cache misses on each block of columns.

Caches and multiplication

When k, L are larger, we can predict:

- lower elements from x are evicted while processing the first row; this causes $\mathcal{O}(n)$ cache misses on $m - 1$ rows.

Fix:

- stop processing a row before an element from x would be evicted; first continue with the next rows.

This results in column-wise 'stripes' of the dense A. But now:

- elements from the vector y can be prematurely evicted; $\mathcal{O}(m)$ cache misses on each block of columns.

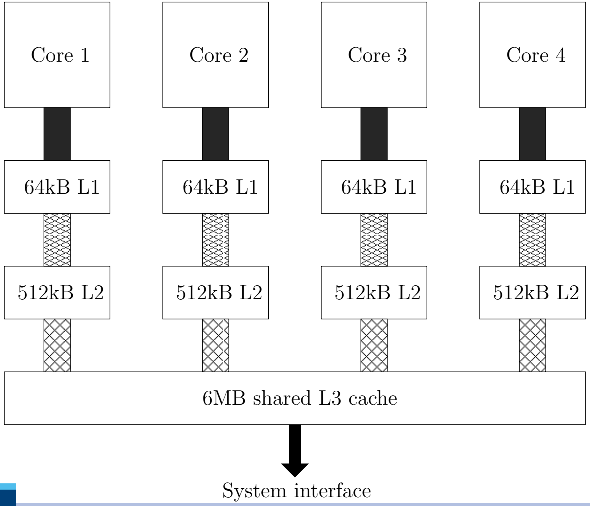
Fix:

- stop processing before an element from y is evicted; first do the remaining column blocks.

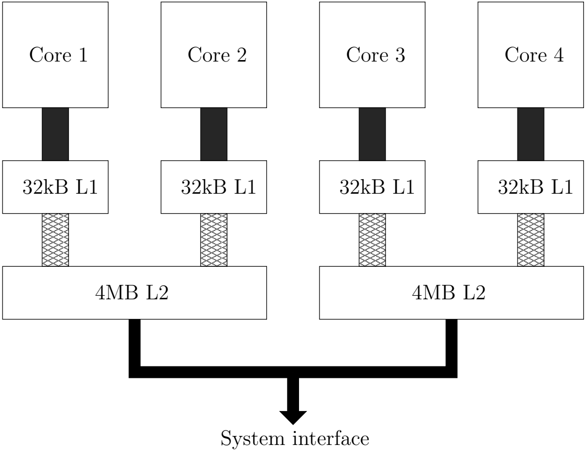
Consecutive processing of $p \times q$ submatrices (cache-aware blocking).

Caches and multicore

Most architectures employ shared caches; $(p, r, l, g) = (4, 3\text{GHz}, l, g)$:



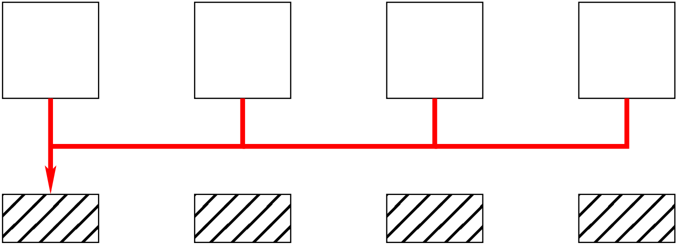
Caches and multicore: NUMA



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

Dealing with NUMA: distribution types

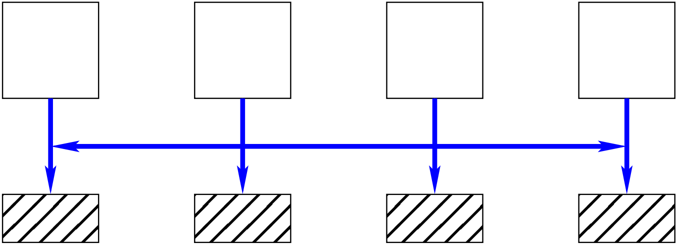
Implicit distribution, centralised local allocation:



If each processor moves data to the same single memory element, the **bandwidth is limited** by that of a single memory controller.

Dealing with NUMA: distribution types

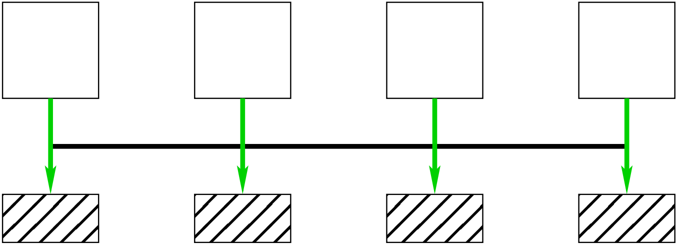
Implicit distribution, centralised interleaved allocation:



If each processor moves data from all memory elements, the bandwidth multiplies **if accesses are uniformly random**.

Dealing with NUMA: distribution types

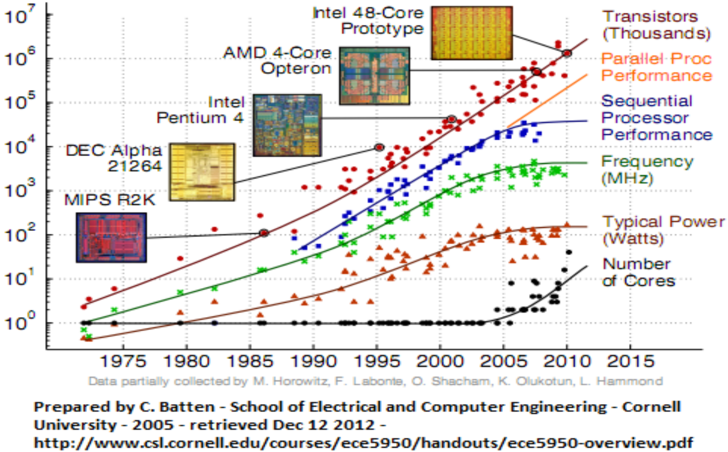
Explicit distribution, distributed local allocation:



If each processor moves data from and to its own unique memory element, the **bandwidth multiplies**.

Bandwidth

CPU speeds stall, but Moore's Law is still alive:



(Illustration by C. Batten, from https://scs.senecac.on.ca/~gpu610/pages/content/intro.html)

Bandwidth

CPU speeds stall, but Moore's Law now translates to an increasing amount of cores per die, i.e., **the effective flop rate of processors still rises** as it always has.

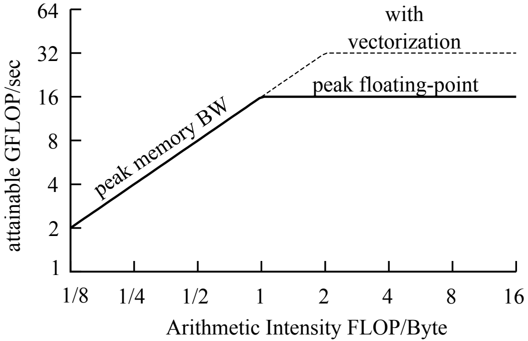
- But what about **bandwidth**?

Technology	Year	Speed
EDO	1970s	27 Mbyte/s
SDRAM	early 1990s	53 Mbyte/s
RDRAM	mid 1990s	1.2 Gbyte/s
DDR	2000	1.6 Gbyte/s
DDR2	2003	3.2 Gbyte/s
DDR3	2007	6.4 Gbyte/s
DDR3	2013	11 Gbyte/s

Will the **effective bandwidth per core** keep decreasing?

Bandwidth

Arithmetic intensity:



- If your computation has enough work per data element, it is **compute bound**. Otherwise it is **bandwidth bound**.
- If you are bandwidth bound, reducing your memory footprint, i.e., **compression**, directly results in faster execution.

(Image courtesy of Prof. Wim Vanroose, UA)

Applications

Parallel inner product

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: $x^T y$

Shared-memory SPMD program with 'double α;' globally allocated:

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

Parallel inner product

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: $x^T y$

Shared-memory SPMD program with ‘double α;’ globally allocated:

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

Data race! (for $n = p = 2$, output can be $x_0 y_0$, $x_1 y_1$, **or** $x_0 y_0 + x_1 y_1$)

Parallel inner product

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: $x^T y$

Shared-memory SPMD program with 'double $\alpha[p]$;' globally allocated:

- for $i = s$ to n step p
- $\qquad \alpha_s \mathrel{+}= x_i y_i$
- synchronise
- return $\sum_{i=0}^{p-1} \alpha_i$

Parallel inner product

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: $x^T y$

Shared-memory SPMD program with '**double** $\alpha[p]$;' globally allocated:

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

False sharing! (processors access and update the same cache lines)

Parallel inner product

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: $x^T y$

Shared-memory SPMD program with '**double $\alpha[8p]$;**' globally allocated:

- for $i = s$ to n step p
- $\quad\quad \alpha_{8s} \mathrel{+}= x_i y_i$
- synchronise
- return $\sum_{i=0}^{p-1} \alpha_{8i}$

Parallel inner product

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: $x^T y$

Shared-memory SPMD program with 'double $\alpha[8p]$;' globally allocated:

- for $i = s$ to n step p
- $\quad\alpha_{8s} \mathrel{+}= x_i y_i$
- synchronise
- return $\sum_{i=0}^{p-1} \alpha_{8i}$

Inefficient cache use: $\Theta(pn)$ data movement.

(All threads access all cache lines)

Parallel inner product

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: $x^T y$

Shared-memory SPMD program with 'double $\alpha[8p]$;' globally allocated:

- for $i = s \cdot \lceil n/p \rceil$ to $(s+1) \cdot \lceil n/p \rceil$
- $\quad \alpha_{8s} \mathrel{+}= x_i y_i$
- synchronise
- return $\sum_{i=0}^{p-1} \alpha_{8i}$

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

Central obstacles for SpMV multiplication

The second example application is the sparse matrix–vector multiplication

$$y = Ax.$$

Three obstacles for an efficient shared-memory parallel sparse matrix–vector (SpMV) multiplication kernel:

- inefficient cache use,
- limited memory bandwidth, and
- non-uniform memory access (NUMA).

Inefficient cache use

SpMV multiplication using CRS, LRU cache perspective:

$x_?$

$\Longrightarrow$

Inefficient cache use

SpMV multiplication using CRS, LRU cache perspective:

$x_?$ $\Longrightarrow$ $a_{0?}$ $x_?$ $\Longrightarrow$

Inefficient cache use

SpMV multiplication using CRS, LRU cache perspective:

$x_?$		$a_{0?}$		y_0		
		$x_?$		$a_{0?}$		
	$\Longrightarrow$		$\Longrightarrow$	$x_?$	$\Longrightarrow$	

Inefficient cache use

SpMV multiplication using CRS, LRU cache perspective:

$x_?$		$a_{0?}$		y_0		$x_?$		
		$x_?$		$a_{0?}$		y_0		
	$\Longrightarrow$		$\Longrightarrow$	$x_?$	$\Longrightarrow$	$a_{0?}$	$\Longrightarrow$	
						$x_?$		

Inefficient cache use

SpMV multiplication using CRS, LRU cache perspective:

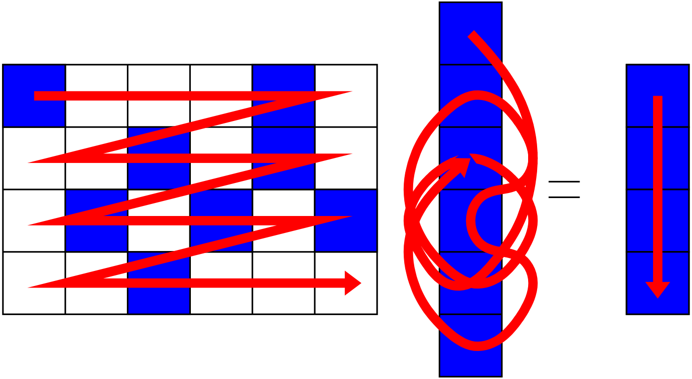
$$\begin{array}{c} x_? \end{array} \Longrightarrow \begin{array}{c} a_{0?} \\ x_? \end{array} \Longrightarrow \begin{array}{c} y_0 \\ a_{0?} \\ x_? \end{array} \Longrightarrow \begin{array}{c} x_? \\ y_0 \\ a_{0?} \\ x_? \end{array} \Longrightarrow \begin{array}{c} a_{??} \\ x_? \\ y_0 \\ a_{0?} \\ x_? \end{array} \Longrightarrow \begin{array}{c} y_? \\ a_{??} \\ x_? \\ y_? \\ a_{0?} \\ x_? \end{array}$$

We cannot predict memory accesses in the sparse case:

- simple blocking is not possible.

Inefficient cache use

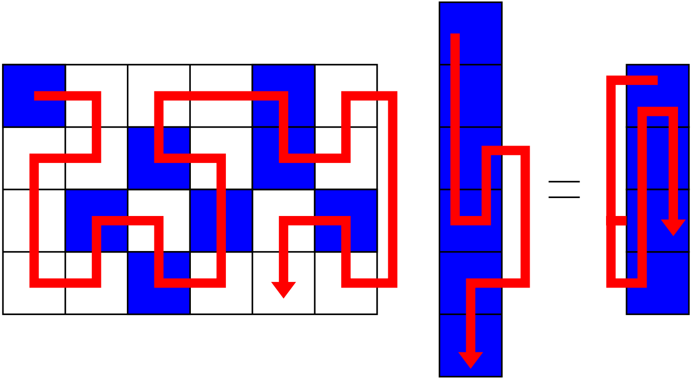
Visualisation of the SpMV multiplication $Ax = y$ with nonzeroes processed in row-major order:



Accesses on the input vector are completely unpredictable.

Inefficient cache use

Visualisation of the SpMV multiplication $Ax = y$ with nonzeroes processed in an order defined by the **Hilbert curve**:



Accesses on both vectors have more **temporal locality**.

Bandwidth issues

The arithmetic intensity of an SpMV multiply lies between

$$\frac{2}{3} \text{ and } \frac{2}{5} \text{ flop per byte.}$$

On an 8-core 2.13 GHz (with AVX), and 10.67 GB/s DDR3:

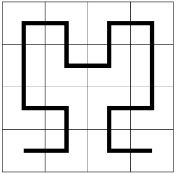
	CPU speed	Memory speed
1 core	$8.5 \cdot 10^9$ nz/s	$4.3 \cdot 10^9$ nz/s
8 cores	$68 \cdot 10^9$ nz/s	$4.3 \cdot 10^9$ nz/s

The SpMV multiplication is clearly bandwidth-bound on modern CPUs.

Sparse matrix storage

The coordinate format stores nonzeroes in arbitrary order:

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$



COO:

$$A = \begin{cases} V & [7\ 1\ 4\ 1\ 2\ 3\ 3\ 2\ 1\ 1] \\ J & [0\ 0\ 0\ 1\ 2\ 2\ 3\ 3\ 3\ 2] \\ I & [3\ 2\ 0\ 0\ 1\ 0\ 1\ 2\ 3\ 3] \end{cases}$$

Storage requirements:

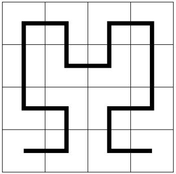
$$\Theta(3nz),$$

where nz is the number of nonzeroes in A.

SpMV multiplication

Multiplication using COO:

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$



$$A = \begin{cases} V & [7\ 1\ 4\ 1\ 2\ 3\ 3\ 2\ 1\ 1] \\ J & [0\ 0\ 0\ 1\ 2\ 2\ 3\ 3\ 3\ 2] \\ I & [3\ 2\ 0\ 0\ 1\ 0\ 1\ 2\ 3\ 3] \end{cases}$$

Sequential algorithm:

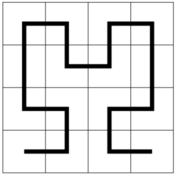
for $k = 0$ **to** $nz - 1$ **do**

  add $V_k \cdot x_{J_k}$ to y_{I_k}

SpMV multiplication

Multiplication using COO:

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$



$$A = \begin{cases} V & [7\ 1\ 4\ 1\ 2\ 3\ 3\ 2\ 1\ 1] \\ J & [0\ 0\ 0\ 1\ 2\ 2\ 3\ 3\ 3\ 2] \\ I & [3\ 2\ 0\ 0\ 1\ 0\ 1\ 2\ 3\ 3] \end{cases}$$

#omp parallel for private(k) schedule(dynamic, 8)

for $k = 0$ **to** $nz - 1$ **do**

  add $V_k \cdot x_{J_k}$ to y_{I_k}

Is this OK?

Sparse matrix storage

Assuming a row-major order of nonzeroes enables **compression**:

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$

CRS:

$$A = \begin{cases} V & [4\ 1\ 3\ 2\ 3\ 1\ 2\ 7\ 1\ 1] \\ J & [0\ 1\ 2\ 2\ 3\ 0\ 3\ 0\ 2\ 3] \\ \hat{I} & [0\ 3\ 5\ 7\ 10] \end{cases}$$

Storage requirements:

$$\Theta(2nz + m + 1).$$

SpMV multiplication

Multiplication using CRS:

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix},$$

$$A = \begin{cases} V & [4\ 1\ 3\ 2\ 3\ 1\ 2\ 7\ 1\ 1] \\ J & [0\ 1\ 2\ 2\ 3\ 0\ 3\ 0\ 2\ 3] \\ \hat{I} & [0\ 3\ 5\ 7\ 10] \end{cases}$$

Sequential kernel:

for $i = 0$ **to** $m - 1$ **do**
  for $k = \hat{I}_i$ **to** $\hat{I}_{i+1} - 1$ **do**
    add $V_k \cdot x_{J_k}$ to y_i

SpMV multiplication

Multiplication using CRS:

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix},$$

$$A = \begin{cases} V & [4\ 1\ 3\ 2\ 3\ 1\ 2\ 7\ 1\ 1] \\ J & [0\ 1\ 2\ 2\ 3\ 0\ 3\ 0\ 2\ 3] \\ \hat{I} & [0\ 3\ 5\ 7\ 10] \end{cases}$$

#omp parallel for private(i, k) schedule(dynamic, 8)

for $i = 0$ **to** $m - 1$ **do**

  for $k = \hat{I}_i$ **to** $\hat{I}_{i+1} - 1$ **do**

    add $V_k \cdot x_{J_k}$ to y_i

Fine-grained parallelisation

The OpenMP SpMV multiplication algorithm was **fine-grained**.

- typically there are more rows than processes $m \gg p$, thus
- there are more tasks than processes.

Fine-grained parallelisation

The OpenMP SpMV multiplication algorithm was **fine-grained**.

- typically there are more rows than processes $m \gg p$, thus
- there are more tasks than processes.

The idea is that load-balancing, and scalability, are automatically attained by **run-time scheduling**.

- scalability is limited only by the amount of parallelism (i.e., the algorithmic span, or the critical path length).

Requires implicit (interleaved) allocation of all data.

Fine-grained parallelisation

The OpenMP SpMV multiplication algorithm was **fine-grained**.

- typically there are more rows than processes $m \gg p$, thus
- there are more tasks than processes.

The idea is that load-balancing, and scalability, are automatically attained by **run-time scheduling**.

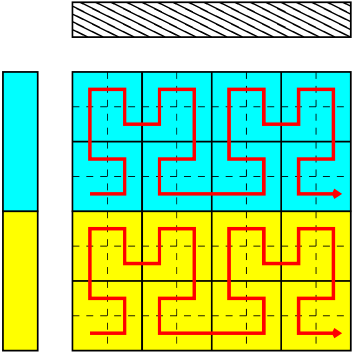
- scalability is limited only by the amount of parallelism (i.e., the algorithmic span, or the critical path length).

Requires implicit (interleaved) allocation of all data. But this does not play well with NUMA. Alternatives:

- 1D SpMV: distribute A and y rowwise.
- 2D SpMV: distribute A, x, and y.

NUMA: Parallel 1D SpMV

Distribute rows to processes, do local blocking and Hilbert ordering:



Allows for explicit (local) allocation of the sparse matrix A and the output vector y; x is implicitly distributed and interleaved.

Ref.: Yzelman and Roose, "High-Level Strategies for Parallel Shared-Memory Sparse Matrix–Vector Multiplication", IEEE Trans. Parallel and Distributed Systems, doi: 10.1109/TPDS.2013.31 (2013).

NUMA: Parallel 1D SpMV

The SPMD code is still very simple. Initialisation:

- find which rows $I \subset \{0, \ldots, m-1\}$ are ours;
- order nonzeroes blockwise;
- impose a Hilbert-curve ordering on these blocks;
- allocate and store the local matrix $A^{(s)}$ (in the above order) using a compressed data structure;
- allocate a local $y^{(s)}$ (intialise to 0).

The input vector x is kept in global memory.

NUMA: Parallel 1D SpMV

The SPMD code is still very simple. Initialisation:

- find which rows $I \subset \{0, \ldots, m-1\}$ are ours;
- order nonzeroes blockwise;
- impose a Hilbert-curve ordering on these blocks;
- allocate and store the local matrix $A^{(s)}$ (in the above order) using a compressed data structure;
- allocate a local $y^{(s)}$ (intialise to 0).

The input vector x is kept in global memory. Multiplication:

- Execute $y^{(s)} = A^{(s)}x$.

Implemented in POSIX Threads.

NUMA: Parallel 2D SpMV

2D SpMV

Input vector communication:

- retrieving values from x is called **fan-out**, and
- is implemented by using **bsp_get**.
- Elements from x are communicated in a one-to-many fashion.

Output vector communication:

- sending contributions to non-local y is **fan-in**.
- Implementation happens through **Bulk Synchronous Message Passing** (BSMP).
- Elements from y are communicated in a many-to-one fashion.

NUMA: Parallel 2D SpMV

Do sparse matrix partitioning as a **pre-processing** step. Then, in BSP:

for each a_{ij} that is local to s **do**
    if x_j is not local **then**
        bsp_get x_j from remote process
bsp_sync()

NUMA: Parallel 2D SpMV

Do sparse matrix partitioning as a **pre-processing** step. Then, in BSP:

for each a_{ij} that is local to s **do**
    if x_j is not local **then**
        bsp_get x_j from remote process
bsp_sync()
for each a_{ij} that is local to s **do**
    add $a_{ij} \cdot x_j$ to y_i

NUMA: Parallel 2D SpMV

Do sparse matrix partitioning as a **pre-processing** step. Then, in BSP:

for each a_{ij} that is local to s **do**
    if x_j is not local **then**
        bsp_get x_j from remote process
bsp_sync()
for each a_{ij} that is local to s **do**
    add $a_{ij} \cdot x_j$ to y_i
    if y_i is not local **then**
        bsp_send (y_i, i) to the owner of y_i
bsp_sync()

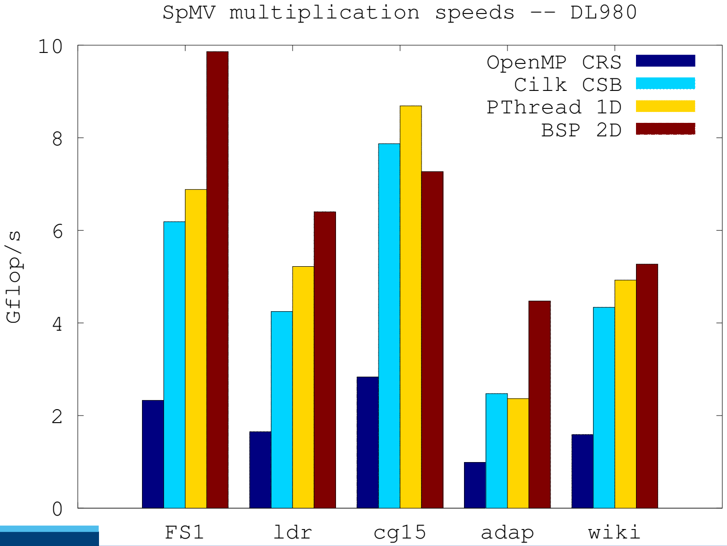
NUMA: Parallel 2D SpMV

Do sparse matrix partitioning as a **pre-processing** step. Then, in BSP:

for each a_{ij} that is local to s **do**
    if x_j is not local **then**
        bsp_get x_j from remote process
bsp_sync()
for each a_{ij} that is local to s **do**
    add $a_{ij} \cdot x_j$ to y_i
    if y_i is not local **then**
        bsp_send (y_i, i) to the owner of y_i
bsp_sync()
while bsp_qsize() > 0 **do**
    $(\alpha, i) =$ **bsp_move()**
    add α to y_i

explicit allocation of thread-local data!

Results


SpMV multiplication speeds -- DL980
Gflop/s
0
2
4
6
8
10
FS1
ldr
cg15
adap
wiki
OpenMP CRS
Cilk CSB
PThread 1D
BSP 2D

BSP 'direct get'

The 'direct get' is a **blocking** one-sided get instruction.

- bypasses the BSP model, but is consistent with `bsp_hpget`.

BSP ‘direct get’

The ‘direct get’ is a **blocking** one-sided get instruction.

- bypasses the BSP model, but is consistent with `bsp_hpget`.

Its intended case is within supersteps that

- contain only BSP ‘get’ primitives,
- guarantee source data remains unchanged.

BSP 'direct get'

The 'direct get' is a **blocking** one-sided get instruction.

- bypasses the BSP model, but is consistent with `bsp_hpget`.

Its intended case is within supersteps that

- contain only BSP 'get' primitives,
- guarantee source data remains unchanged.

Replacing those primitives with calls to `bsp_direct_get` allows **merging** this superstep with its following one, thus

saving a synchronisation step.

Ref.: Yzelman and Bisseling, "An Object-Oriented Bulk Synchronous Parallel Library for Multicore Programming", Concurrency and Computation: Practice and Experience 24(5), pp. 533-553 (2012).

BSP 'hp send'

BSP programming is transparent and safe because of

1. buffering on destination,
2. buffering on source.

This costs memory.

BSP 'hp send'

BSP programming is transparent and safe because of

1. buffering on destination,
2. buffering on source.

This costs memory. Alternative: high-performance (`hp`) variants.

- `bsp_move`; **copies** a message from its incoming communications queue into local memory.

BSP 'hp send'

BSP programming is transparent and safe because of

1. buffering on destination,
2. buffering on source.

This costs memory. Alternative: high-performance (`hp`) variants.

- `bsp_move`; **copies** a message from its incoming communications queue into local memory.
- `bsp_hpmove`; evades this by returning the user a pointer into the queue.

BSP ‘hp send’

BSP programming is transparent and safe because of

1. buffering on destination,
2. buffering on source.

This costs memory. Alternative: high-performance (`hp`) variants.

- `bsp_move`; **copies** a message from its incoming communications queue into local memory.
- `bsp_hpmove`; evades this by returning the user a pointer into the queue.
- `bsp_hpsend`; delays reading source data until the message is sent. Local source data should remain unchanged!

(`bsp_hpput` and `bsp_hpget` also exist.)

BSP 2D SpMV

Step 1: fan-out. Request **contiguous** ranges of x.

```
typedef std::vector< fanQuadlet >::const_iterator IT;
for( IT it = fanIn.begin(); it != fanIn.end(); ++it ) {
    const size_t src_P    = it->remoteP;
    const size_t src_ind  = it->remoteStart;
    const size_t dest_ind = it->localStart;
    const size_t length   = it->length;
    bsp_direct_get( src_P,
                    x,
                    src_ind * sizeof( double ),
                    x + dest_ind,
                    length  * sizeof( double )
                  );
}
```

BSP 2D SpMV

Step 2: local SpMV multiplication:

```
if( A != NULL )
    A->zax( x, y ); //('zax' stands for z=Ax)
```

We use Compressed BICRS storage with the nonzeroes in row-major order. *A* is a pointer to an instance of a C++ sparse matrix class.

Yzelman and Roose, "High-level strategies for parallel shared-memory sparse matrix–vector multiplication", IEEE TPDS, 2013 (in press); paper: http://dx.doi.org/10.1109/TPDS.2013.31, software: http://albert-jan.yzelman.net/software/#SL

BSP 2D SpMV

Step 3: fan-in (I). Send **chunks** of row contributions.

```
//the tagsize is initialised to 2*sizeof( size_t )
//fanOut[ i ] has the following layout:
//{ size_t remoteP, localStart, remoteStart, length; }
typedef unsigned long int size_t;

for( size_t i = 0; i < fanOut.size(); ++i ) {
    const size_t dest_P  = fanOut[ i ].remoteP;
    const size_t src_ind = fanOut[ i ].localStart;
    const size_t length  = fanOut[ i ].length;
    bsp_hpsend( dest_P,
                &( fanOut[ i ].remoteStart ),
                y + src_ind, length * sizeof( double ) );
}
bsp_sync();
```

BSP 2D SpMV

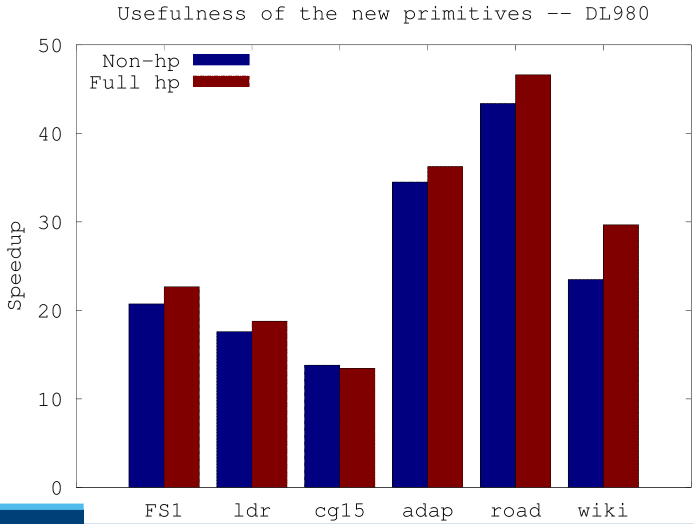
Step 4: fan-in (II). Handle incoming contributions.

```
size_t *msg_tag;
double *msg_payload;
while( bsp_hpmove( (void**)&msg_tag,
                   (void**)&msg_payload
                 ) != SIZE_MAX ) {
    const size_t y_dest = msg_tag[ 0 ];
    const size_t length = msg_tag[ 1 ];
    for(  size_t i = 0; i < length; ++i )
        y[ y_dest + i ] += msg_payload[ i ];
}
```

This finishes our implementation of the 2D SpMV multiply.

Results – new primitives

We test the new primitives using the BSP 2D SpMV multiply:



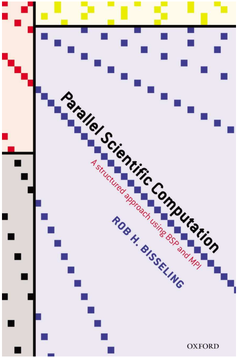
Summary

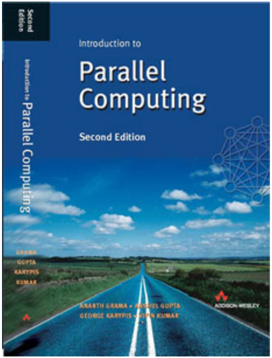
We have seen

- hardware properties of modern shared-memory architectures,
- how this affects shared-memory programming and data locality,
- common pitfalls of non-BSP shared-memory programming like data races and false sharing (in OpenMP, Cilk, and PThreads),
- how shared-memory BSP programming avoids these issues, and
- how to attain high performance algorithms using BSP.

Metrics for parallel efficiency

Sources





- Rob H. Bisseling; *Parallel Scientific Computing*, Oxford Press.
- Grama, Gupta, Karypis, Kumar; *Parallel Computing*, Addison Wesley.

Measuring performance

Definition (Parallel overhead)

- let T_{seq} be the time taken by a sequential algorithm;
- let T_p be the time taken by a parallelisation of that algorithm, using p processes.

Then, the parallel overhead T_{o} is given by

$$T_{\text{o}} = pT_p - T_s.$$

(Effort is proportional to the number of workers multiplied with the duration of their work, that is, equal to pT_p.)

Best case: $T_o = 0$, such that $T_p = T_{\text{seq}}/p$.

Measuring performance

Definition (Speedup)

Let T_{seq}, p, and T_p be as before. Then, the speedup S is given by

$$S(p) = T_{\text{seq}} / T_p.$$

Measuring performance

Definition (Speedup)

Let T_{seq}, p, and T_p be as before. Then, the speedup S is given by

$$S(p) = T_{\text{seq}} / T_p.$$

- Target: $S = p$ (no overhead; $T_o = 0$).
- Best case: $S > p$ (**superlinear speedup**).
- Worst case: $S < 1$ (slowdown).

Measuring performance

What is T_{seq}?

- Many sequential algorithms solving the same problem.

Measuring performance

What is T_{seq}?

- Many sequential algorithms solving the same problem.
- When determining the speedup S,
  compare against the best sequential algorithm
  (that is available on your architecture).

Measuring performance

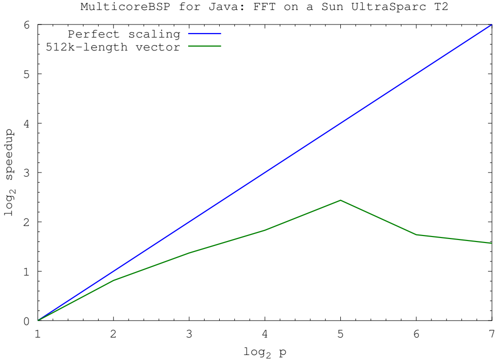
What is T_{seq}?

- Many sequential algorithms solving the same problem.
- When determining the speedup S,
  compare against the best sequential algorithm
  (that is available on your architecture).
- When determining the overhead T_o,
  compare against the most similar algorithm
  (maybe even take $T_{\text{seq}} = T_1$).

Measuring performance

Definition (strong scaling)

$$S(p) = T_{\text{seq}}/T_p = \Omega(p) \quad (\text{i.e., } \mathit{lim\ sup}_{p\to\infty}|S(p)/p| > 0)$$

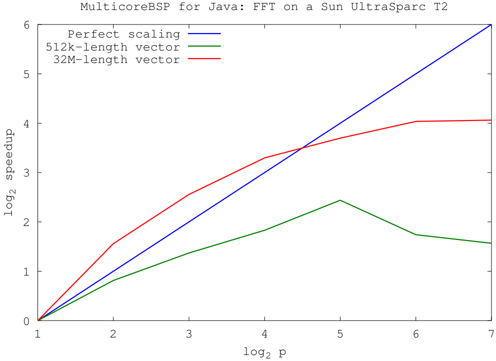


Question: is it reasonable to expect strong scalability for solving a problem using (good) parallel algorithms?

Measuring performance

Definition (strong scaling)

$$S(p) = T_{seq}/T_p = \Omega(p) \quad (\text{i.e., } \mathit{lim\ sup}_{p\to\infty}|S(p)/p| > 0)$$

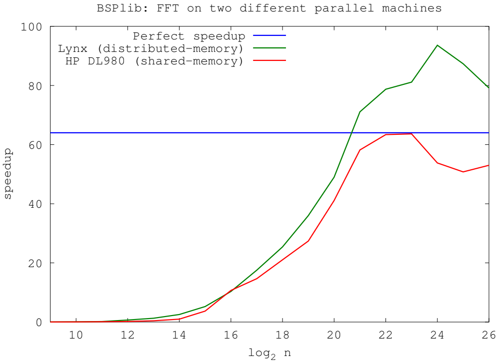


Answer: not as $p \to \infty$. You cannot efficiently clean a table with 50 people, or paint a single wall with 500 painters.

Measuring performance

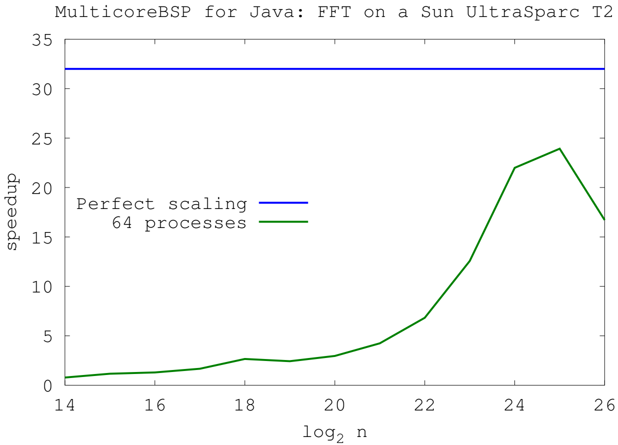
Definition (weak scaling)

$$S(n) = T_{seq}(n)/T_p(n) = \Omega(1),\ n \to \infty,\ \text{with } p \textbf{ fixed.}$$



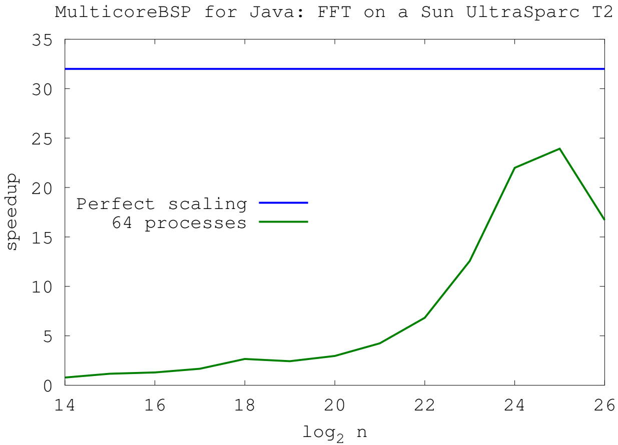
For large enough problems, we do expect to make maximum use of our parallel computer.

Measuring performance: example



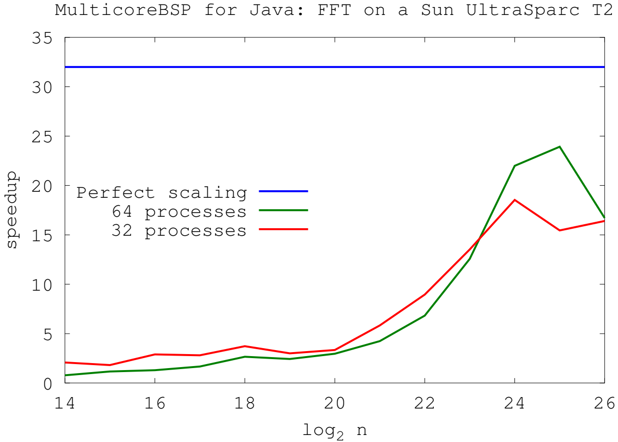
- This processor advertises 64 processors,
-

Measuring performance: example



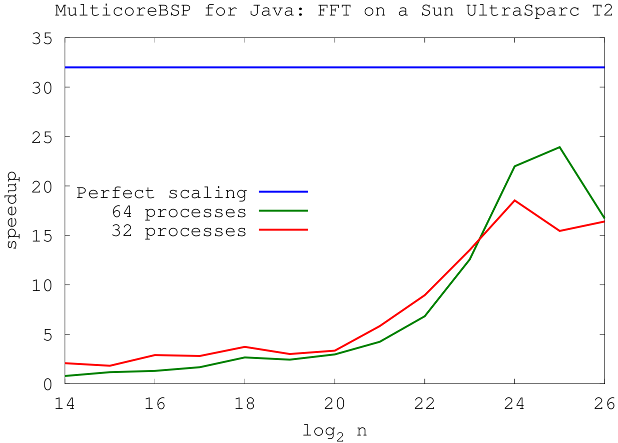
- This processor advertises 64 processors, but only has 32 FPUs.
- We oversubscribed!

Measuring performance



- This processor advertises 64 processors, but only has 32 FPUs.
- Be careful with oversubscription! (**Including hyperthreading**!)

Measuring performance



- This processor advertises 64 processors, but only has 32 FPUs.
- **Q**: would you say this algorithm scales on the Sun Ultrasparc T2?

Measuring performance

A: if the speedup stabilises around 16x, then

yes,

since the relative efficiency is stable.

Definition (Parallel efficiency)

Let T_{seq}, p, T_p, and S as before. The parallel efficiency E equals

$$E = \frac{T_{\text{seq}}}{p} / T_p = T_{\text{seq}} / pT_p = S/p.$$

What is parallelism?

Defining T_{seq} and T_p enables a precise definition of how 'parallel' certain algorithms are:

Definition (Parallelism)

Consider a parallel algorithm that runs in T_p time. Let T_{seq} the time taken by the best sequential algorithm that solves the same problem. Then the **parallelism** is given by

$$\frac{T_{\text{seq}}}{T_\infty} = \lim_{p\to\infty} \frac{T_{\text{seq}}}{T_p}.$$

This kind of analysis is fundamental for fine-grained parallelisation schemes.

- Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: an efficient multithreaded runtime system. SIGPLAN Not. 30, 8 (August 1995), pp. 207-216.

Measuring performance

- If there is no overhead ($T_o = pT_p - T_{\text{seq}} = 0$), the efficiency $E = 1$; decreasing the overhead increases the efficiency.

Weak scalability asks what happens if the problem size increases...

Measuring performance

- If there is no overhead ($T_o = pT_p - T_{\text{seq}} = 0$), the efficiency $E = 1$; decreasing the overhead increases the efficiency.

Weak scalability asks what happens if the problem size increases...

...but what is a sensible definition of the 'problem size'?

Consider the following applications:

- inner-product calculation;
- binary search;
- sorting (quicksort).

Measuring performance

Problem	Size	Run-time
Inner-product	$\Theta(n)$ bytes	$\Theta(n)$ flops
Binary search	$\Theta(n)$ bytes	$\Theta(\log_2 n)$ comparisons
Sorting	$\Theta(n)$ bytes	$\Theta(n \log_2 n)$ swaps
FFT	$\Theta(n)$ bytes	$\Theta(n \log_2 n)$ flops

Hence the problem size is best identified by T_{seq}.

Question:

- How should the ratio T_o / T_{seq} behave as $T_{\text{seq}} \to \infty$, for the algorithm to scale in a weak sense?

Measuring performance

If $T_o / T_{\text{seq}} = c$, with $c \in \mathbb{R}_{\geq 0}$ constant, then

$$\frac{pT_p - T_{\text{seq}}}{T_{\text{seq}}} = pS^{-1} - 1 = c, \text{ so}$$

$$S = \frac{p}{c+1}, \text{ which is constant when } p \text{ is fixed.}$$

Note that here, $E = S/p = \frac{1}{c+1}$

Question:

- How should the ratio T_o / T_{seq} behave as $p \to \infty$, for the algorithm to scale in a strong sense?

Measuring performance

If $T_o / T_{\text{seq}} = c$, with $c \in \mathbb{R}_{\geq 0}$ constant, then

$$\frac{pT_p - T_{\text{seq}}}{T_{\text{seq}}} = pS^{-1} - 1 = c, \text{ so}$$

$$S = \frac{p}{c+1}.$$

Note that here, $E = S/p = \frac{1}{c+1}$ which is still constant!

Answer:

- Exactly the same! Both strong and weak scalability are **iso-efficiency** constraints (E remains constant).

Measuring performance

Definition (iso-efficiency)

Let E be as before. Suppose $T_o = f(T_{\text{seq}}, p)$ is a known function. Then the iso-efficiency relation is given by

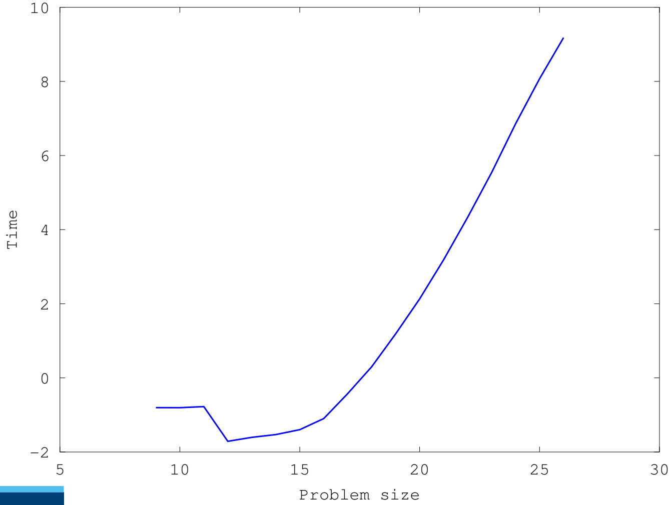
$$T_{\text{seq}} = \frac{1}{\frac{1}{E} - 1} f(T_{\text{seq}}, p).$$

This follows from the definition of E:

$$\begin{aligned} E^{-1} &= pT_p/T_{\text{seq}} + 1 - \frac{T_{\text{seq}}}{T_{\text{seq}}} \\ &= 1 + \frac{pT_p - T_{\text{seq}}}{T_{\text{seq}}} \\ &= 1 + \frac{T_o}{T_{\text{seq}}}, \qquad \text{so } T_{\text{seq}}(E^{-1} - 1) = T_o. \end{aligned}$$

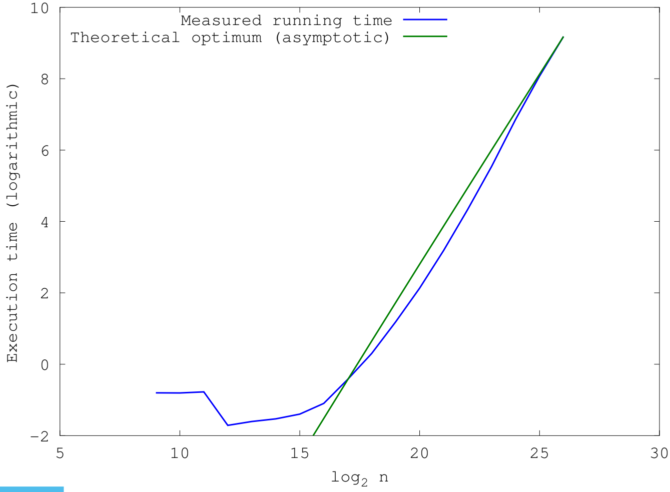
Questions

- Does this scale?



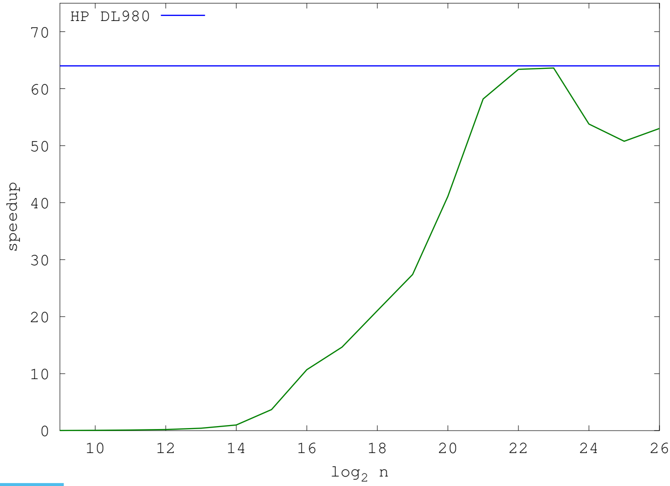
Questions

- Does this scale? **Yes**! (It's again an FFT)



Questions

- Better use speedups when investigating scalability.



In summary...

A BSP algorithm is **scalable** when

$$T = \mathcal{O}(T_{\text{seq}}/p + p).$$

This considers scalability of the speedup and includes parallel overhead.

In summary...

A BSP algorithm is **scalable** when

$$T = \mathcal{O}(T_{\text{seq}}/p + p).$$

This considers scalability of the speedup and includes parallel overhead. It does not include **memory scalability**:

$$M = \mathcal{O}(M_{\text{seq}}/p + p),$$

where M is the memory taken by one BSP process and M_{seq} the memory requirement of the best sequential algorithm.