Parallel Partition Revisited

Leonor Frias and Jordi Petit

Dep. de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya

WEA 2008

Overview

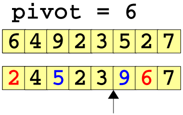
Partitioning an array with respect to a pivot is a basic building block of key algorithms such as as *quicksort* and *quickselect*.

Overview

Partitioning an array with respect to a pivot is a basic building block of key algorithms such as as *quicksort* and *quickselect*.

Given a pivot, rearrangement s.t for some splitting position s,

- elements at the left of s are $\leq$ pivot
- elements at the right of s are $\geq$ pivot



`pivot = 6`

6	4	9	2	3	5	2	7

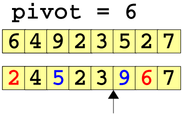
2	4	5	2	3	9	6	7

Overview

Partitioning an array with respect to a pivot is a basic building block of key algorithms such as as *quicksort* and *quickselect*.

Given a pivot, rearrangement s.t for some splitting position s,

- elements at the left of s are $\leq$ pivot
- elements at the right of s are $\geq$ pivot



`pivot = 6`

6	4	9	2	3	5	2	7
2	4	5	2	3	9	6	7

Sequential cost:

- n comparisons
- m swaps

Overview

Partitioning an array with respect to a pivot is a basic building block of key algorithms such as as *quicksort* and *quickselect*.

Overview

Partitioning an array with respect to a pivot is a basic building block of key algorithms such as as *quicksort* and *quickselect*.

Nowadays, multi-core computers are ubiquitous.

Overview

Partitioning an array with respect to a pivot is a basic building block of key algorithms such as as *quicksort* and *quickselect*.

Nowadays, multi-core computers are ubiquitous.

Several suitable parallel partitioning algorithms for these architectures exists.

- Algorithms by Francis and Pannan, Tsigas and Zang and MCSTL.

Overview

Partitioning an array with respect to a pivot is a basic building block of key algorithms such as as *quicksort* and *quickselect*.

Nowadays, multi-core computers are ubiquitous.

Several suitable parallel partitioning algorithms for these architectures exists.

HOWEVER, they perform more operations than the sequential algorithm.

Overview

Partitioning an array with respect to a pivot is a basic building block of key algorithms such as as *quicksort* and *quickselect*.

Nowadays, multi-core computers are ubiquitous.

Several suitable parallel partitioning algorithms for these architectures exists.

HOWEVER, they perform more operations than the sequential algorithm.

IN THIS PAPER:

- Show how to modify these algorithms so that they achieve a minimal number of comparisons.
- Provide implementations and a detailed experimental comparison.

Outline

1. Previous work
2. Algorithm
3. Experiments
4. Conclusions
5. References

Partitioning in parallel: overview

General pattern

1. Sequential setup of each processor's work
2. Parallel main phase in which most of the partitioning is done
3. Cleanup phase

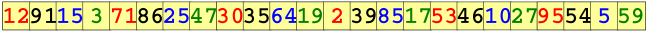
p processors used to partition an array of n elements ($p \ll n$).

Partitioning in parallel: STRIDED (1)

STRIDED algorithm by Francis and Pannan.

1. Setup: Division into *p* pieces, elements in a piece with stride *p*

`pivot = 40, p = 4`



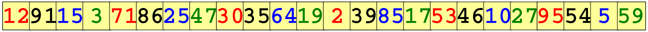
12	91	15	3	71	86	25	47	30	35	64	19	2	39	85	17	53	46	10	27	95	54	5	59

Partitioning in parallel: STRIDED (1)

STRIDED algorithm by Francis and Pannan.

1. Setup: Division into *p* pieces, elements in a piece with stride *p*

`pivot = 40, p = 4`



12	91	15	3	71	86	25	47	30	35	64	19	2	39	85	17	53	46	10	27	95	54	5	59

2. Main phase: Sequential partitioning in each piece

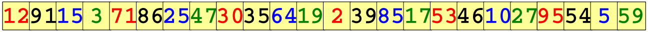
12	39	15	3	2	35	25	27	30	86	5	19	71	91	10	17	53	46	85	47	95	54	64	59

Partitioning in parallel: STRIDED (1)

STRIDED algorithm by Francis and Pannan.

1. Setup: Division into p pieces, elements in a piece with stride p

`pivot = 40, p = 4`

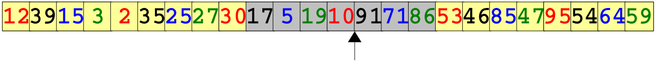


12	91	15	3	71	86	25	47	30	35	64	19	2	39	85	17	53	46	10	27	95	54	5	59

2. Main phase: Sequential partitioning in each piece

12	39	15	3	2	35	25	27	30	86	5	19	71	91	10	17	53	46	85	47	95	54	64	59

3. Cleanup: Sequential partitioning in the not correctly partitioned range

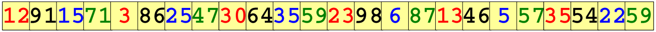


12	39	15	3	2	35	25	27	30	17	5	19	10	91	71	86	53	46	85	47	95	54	64	59

Partitioning in parallel: STRIDED (2)

STRIDED Analysis:

- Main phase: $\Theta(n/p)$ parallel time
- Cleanup phase: $O(1)$ expected but can be $\Theta(n)$



12	91	15	71	3	86	25	47	30	64	35	59	23	98	6	87	13	46	5	57	35	54	22	59

Partitioning in parallel: STRIDED (2)

STRIDED Analysis:

- Main phase: $\Theta(n/p)$ parallel time
- Cleanup phase: $O(1)$ expected but can be $\Theta(n)$

12	91	15	71	3	86	25	47	30	64	35	59	23	98	6	87	13	46	5	57	35	54	22	59

Partitioning in parallel: STRIDED (2)

STRIDED Analysis:

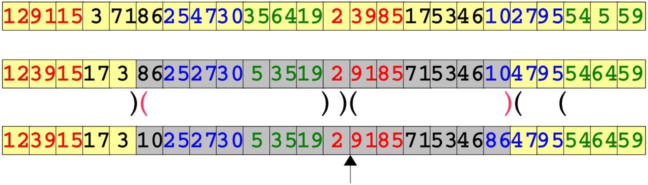
- Main phase: $\Theta(n/p)$ parallel time
- Cleanup phase: $O(1)$ expected but can be $\Theta(n)$

BESIDES, it has poor cache locality.

Partitioning in parallel: BLOCKED

We can generalize STRIDED to blocks to improve cache locality.

`pivot = 40, p = 4, b = 3`



12	9 1	15	3	71	86	25	47	30	35	64	19	2	39	85	17	53	46	10	27	95	54	5	59
12	39	15	17	3	86	25	27	30	5	35	19	2	91	85	71	53	46	10	47	95	54	64	59
12	39	15	17	3	10	25	27	30	5	35	19	2	91	85	71	53	46	86	47	95	54	64	59

If $b = 1$, BLOCKED is equal to STRIDED.

Partitioning in parallel: F&A (1)

Processors take elements from both ends of the array as they are needed.

Fetch-and-add instructions are used to *acquire* the elements.

Blocks of elements are used to avoid too much synchronization.

References:

- PRAM model: Heidelberger *et al.*
- *real* machines: Tsigas and Zhang and MCSTL library

Partitioning in parallel: F&A (2)

1. Setup: Each processor takes one left block and one right block

12	91	15	3	71	86	25	47	30	35	64	19	2	39	85	17	53	46	10	27	95	54	5	59

Partitioning in parallel: F&A (2)

1. Setup: Each processor takes one left block and one right block

12	91	15	3	71	86	25	47	30	35	64	19	2	39	85	17	53	46	10	27	95	54	5	59

2. Main phase: Sequential partitioning in sequence made by left block + right block. When one block border is reached and so *neutralized*, another block is acquired.

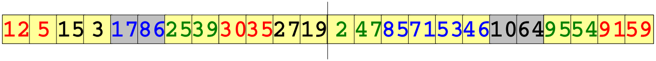
12	5	15	3	71	86	25	47	30	35	64	19	2	39	85	17	53	46	10	27	95	54	91	59

Partitioning in parallel: F&A (2)

1. Setup: Each processor takes one left block and one right block

12	91	15	3	71	86	25	47	30	35	64	19	2	39	85	17	53	46	10	27	95	54	5	59

2. Main phase: Sequential partitioning in sequence made by left block + right block. When one block border is reached and so *neutralized*, another block is acquired.



12	5	15	3	17	86	25	39	30	35	27	19	2	47	85	71	53	46	10	64	95	54	91	59

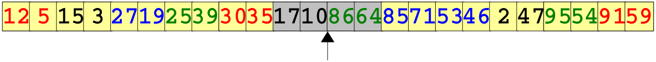
Partitioning in parallel: F&A (2)

1. Setup: Each processor takes one left block and one right block

12	91	15	3	71	86	25	47	30	35	64	19	2	39	85	17	53	46	10	27	95	54	5	59

2. Main phase: Sequential partitioning in sequence made by left block + right block. When one block border is reached and so *neutralized*, another block is acquired.

3. Cleanup: At most *p* blocks remain not completely partitioned (unneutralized). The unpartitioned elements must be moved to the *middle*
   - Tsigas and Zhang do it sequentially.
   - MCSTL moves the blocks in parallel and applies recursively parallel partition to this range.



12	5	15	3	27	19	25	39	30	35	17	10	86	64	85	71	53	46	2	47	95	54	91	59

Partitioning in parallel: F&A (3)

F&A Analysis:

- Main phase: $\Theta(n/p)$ parallel time
- Cleanup phase:
  - Tsigas and Zhang: $O(bp)$
  - MCSTL: $\Theta(b \log p)$

New Parallel Cleanup Phase

Existing algorithms disregard part of the work done in the main parallel phase when cleaning up.

New Parallel Cleanup Phase

Existing algorithms disregard part of the work done in the main parallel phase when cleaning up.

We present a new cleanup algorithm.

New Parallel Cleanup Phase

Existing algorithms disregard part of the work done in the main parallel phase when cleaning up.

We present a new cleanup algorithm.

- It avoids redundant comparisons.
- The elements are swapped fully in parallel.

We apply it on the top of STRIDED, BLOCKED and F&A algorithms.

Terminology (1)

Our algorithm is described in terms of

- Subarray
- Frontier: Defines two parts (left and right) in a subarray
- Misplaced element

Their realization depends on the algorithm used in the main parallel phase.

Terminology (1)

Our algorithm is described in terms of

- Subarray
- Frontier: Defines two parts (left and right) in a subarray
- Misplaced element

Their realization depends on the algorithm used in the main parallel phase.

m: total number of misplaced elements
M: total number of subarrays that may have misplaced elements.

Terminology (1)

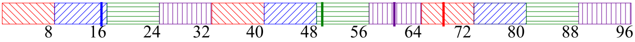
Our algorithm is described in terms of

- Subarray
- Frontier: Defines two parts (left and right) in a subarray
- Misplaced element

Their realization depends on the algorithm used in the main parallel phase.

m: total number of misplaced elements
M: total number of subarrays that may have misplaced elements.

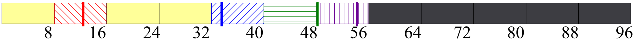
Terminology for BLOCKED



- Subarray: each of the p pieces.
- Frontier: position that would occupy the pivot after partitioning the array.
- Misplaced elements: as in the sequential algorithm.
- $M \leq p$

Terminology for F&A

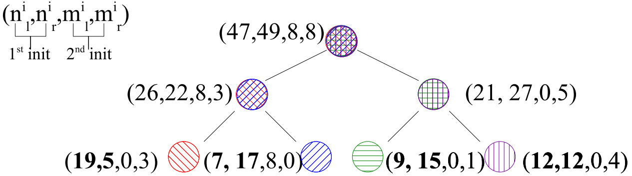
We deal separately and analogously with left and right blocks.



- Subarray: one block.
- Frontier: separates the processed part in a block from the unprocessed part.
- Misplaced elements: unprocessed elements not in the *middle* and processed elements that are in the *middle*.
- $M \leq 2p$ (p unneutralized blocks which could be all misplaced and almost full)

Data Structure

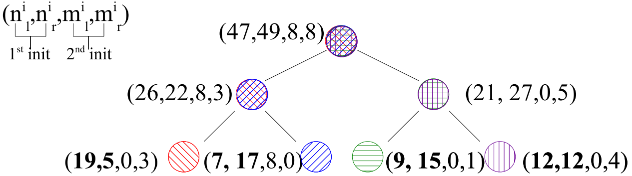
Shared arrayed binary tree with M leaves.



- Leaves: information on the subarrays
- Internal nodes: accumulate children information

Data Structure

Shared arrayed binary tree with M leaves.



- Leaves: information on the subarrays
- Internal nodes: accumulate children information

Use: deciding pairs of elements to be swapped without doing new comparisons

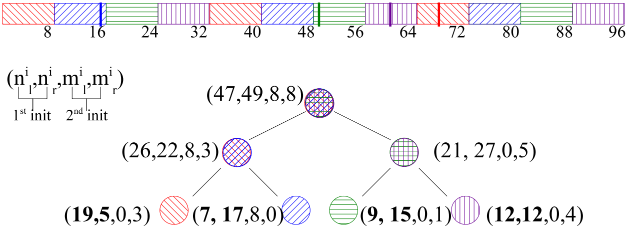
Algorithm (1)

Tree initialization

1. First initialization of the leaves: Computation of $n^i_{l,r}$.
2. First initialization of the non-leaves: Computation of $n^j_{l,r}$, v.
3. Second initialization of the leaves: Computation of $m^i_{l,r}$.
4. Second initialization of the non-leaves: Computation of $m^j_{l,r}$.

Tree initialization for BLOCKED

Computation of $n^i_{l,r}$: trivially
(the layout is deterministic, b and i are known)

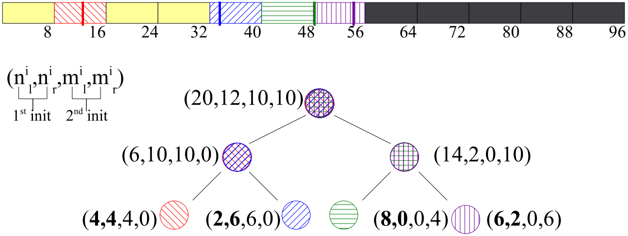


Tree initialization for F&A

Computation of $n^i_{l,r}$: Trivially once the subarrays are known.

Determination of the subarrays:

- The unneutralized blocks are known after the parallel phase.
- To locate the misplaced neutralized blocks, the unneutralized blocks are sorted by address and then, traversed.

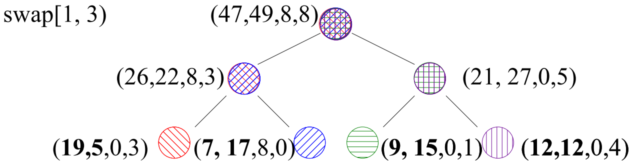


Algorithm (2)

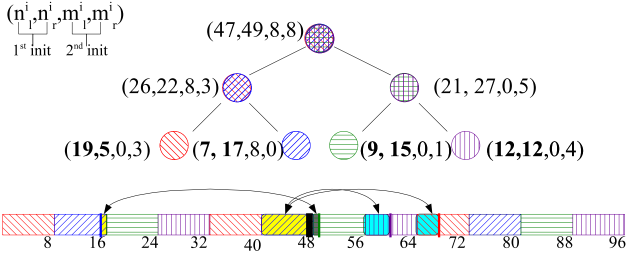
Parallel swapping

Independent of the algorithm in the main parallel phase.

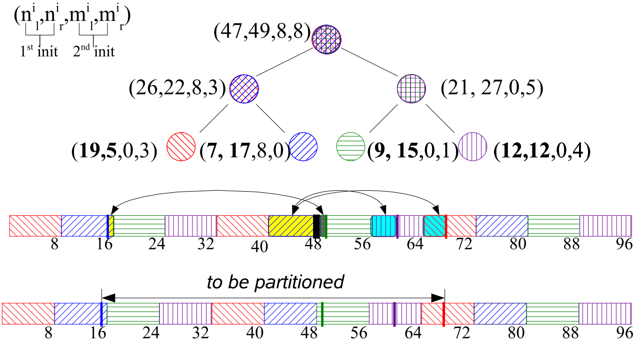
The misplaced elements to swap are divided equally among the processors.



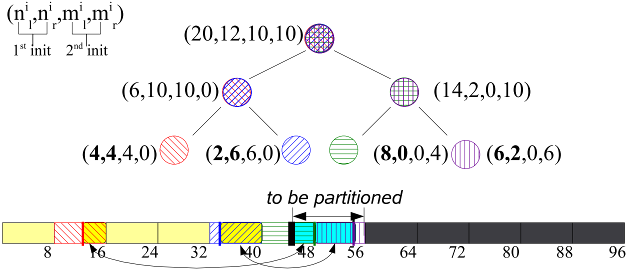
Parallel swapping for BLOCKED



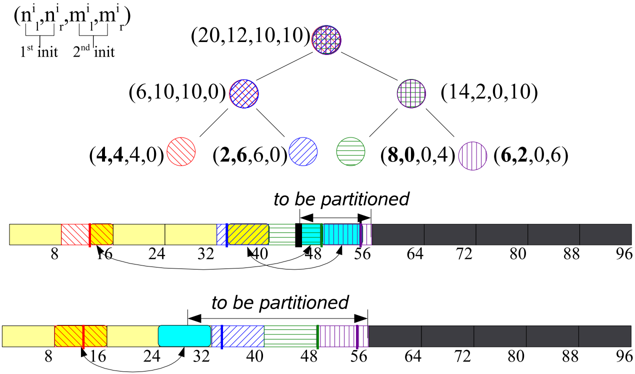
Parallel swapping for BLOCKED



Parallel swapping for F&A



Parallel swapping for F&A



Algorithm (3)

Completion

BLOCKED : The array is already partitioned.

F&A : The array is partitioned except for the elements in the *middle* (not yet processed).

- Apply recursively parallel partitioning in the *middle*.

  We provide a better cost bound making recursion on b ($b \leftarrow b/2$ for $\log p$ times) instead of p.

Analysis: comparisons & swaps

Blocked

	comparisons		swaps	
	original	tree	original	tree
main	n		$\leq n/2$	
cleanup	$v_{max} - v_{min}$	0	$m/2$	$m/2$
total	$n + v_{max} - v_{min}$	n	$\leq \frac{n+m}{2}$	$\leq \frac{n+m}{2}$

F&A

	comparisons		swaps	
	original	tree	original	tree
main	$n - \lvert V \rvert$		$\leq \frac{n-\lvert V \rvert}{2}$	
cleanup	$\leq 2bp$	$\lvert V \rvert$	$\leq 2bp$	$\leq m/2 + \lvert V \rvert$
total	$\leq n + 2bp$	n	$\leq \frac{n-\lvert V \rvert}{2} + 2bp$	$\leq \frac{n+m}{2} + \lvert V \rvert$

Analysis: worst-case time

BLOCKED		
	parallel time	
	original	tree
main	$\Theta(n/p)$	
cleanup	$\Theta(v_{max} - v_{min})$	$\Theta(m/p + \log p)$
total	$\Theta(n)$	$\Theta(n/p + \log p)$

F&A		
	parallel time	
	original	tree
main	$\Theta(n/p)$	
cleanup	$\Theta(b \log p)$	$\Theta(\log^2 p + b)$[1]
total	$\Theta(n/p + b \log p)$	$\Theta(n/p + \log^2 p)$

---

[1] better provided that $\log p \leq b$

Implementation

Algorithms: STRIDED, BLOCKED, F&A (MCSTL & own)

- With original cleanup
- With our cleanup

Languages: C++, OpenMP

STL `partition` interface.

Setup

Machine

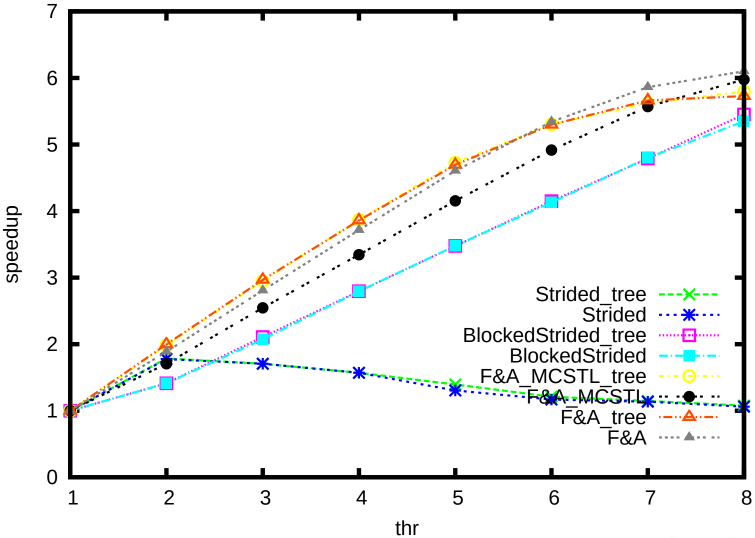
- 4 GB of main memory
- 2 sockets x Intel Xeon quad-core processor at 1.66 GHz with a shared L2 cache of 4 MB shared among two cores

Compiler: GCC 4.2.0, `-O3` optimization flag.

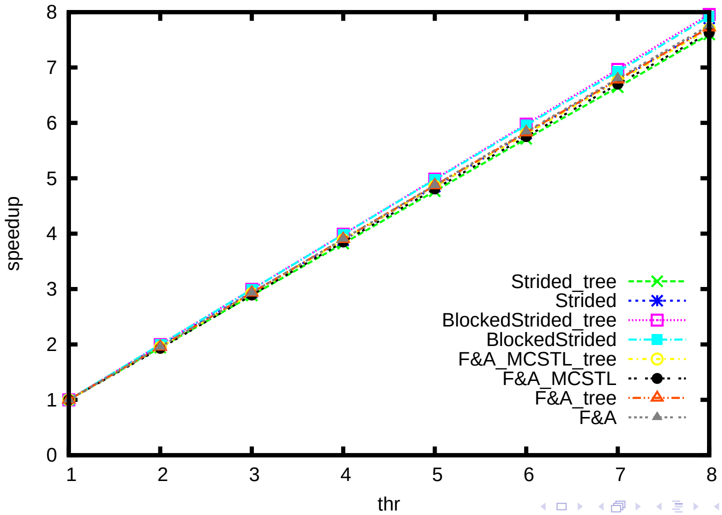
Measurements:

- 100 repetitions
- Speedups with respect to the sequential algorithm in the STL

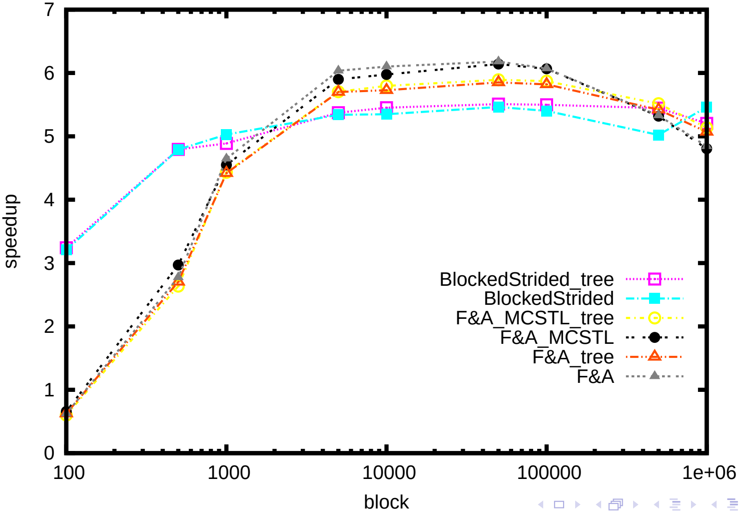
Parallel partition speedup, $n = 10^8$ and $b = 10^4$



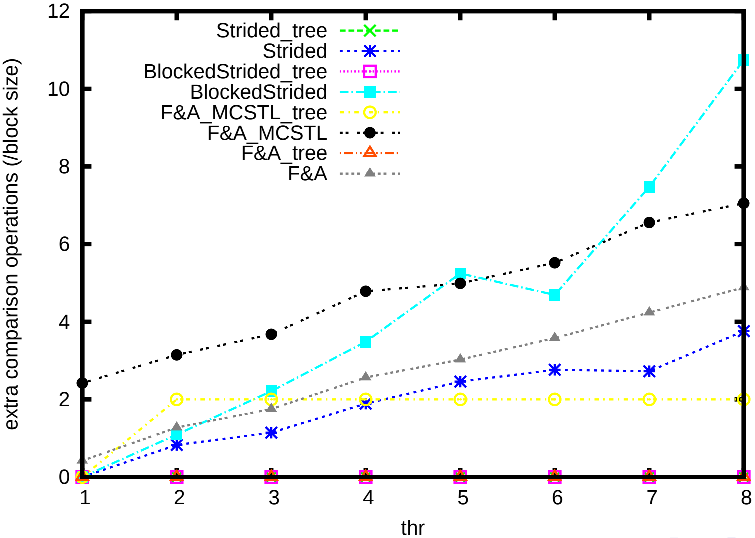
Parallel partition speedup for costly $<$, $n = 10^8$ and $b = 10^4$



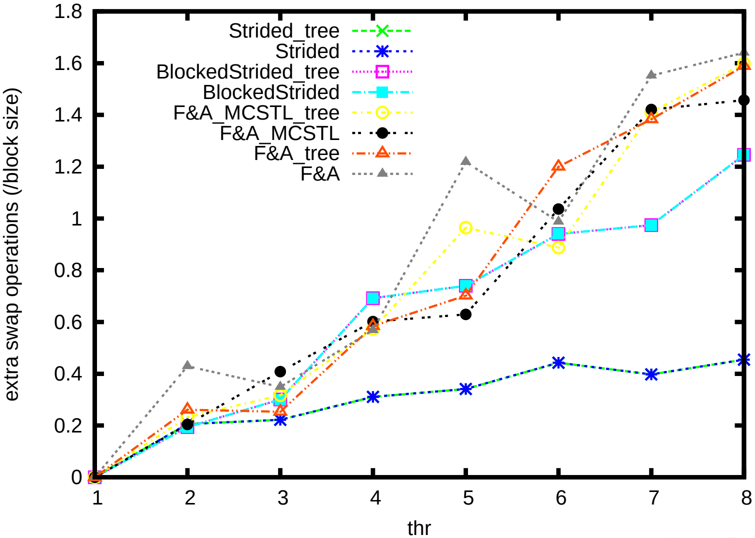
Parallel partition with varying block size, $n = 10^8$ and num_threads $= 8$



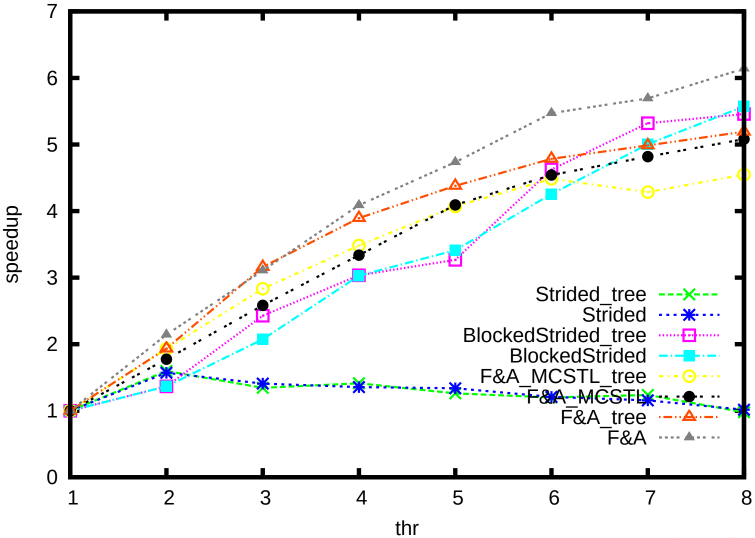
Number of extra comparisons, $n = 10^8$ and $b = 10^4$



Number of extra swaps, $n = 10^8$ and $b = 10^4$



Parallel quickselect speedup, $n = 10^8$ and $b = 10^4$



Conclusions (1)

We have presented, implemented and evaluated several parallel partitioning algorithms suitable for multi-core architectures.

Conclusions (1)

We have presented, implemented and evaluated several parallel partitioning algorithms suitable for multi-core architectures.

Algorithmic contributions:

- Novel cleanup algorithm NOT disregarding any comparisons made in the parallel phase.
- Applied to STRIDED, BLOCKED and F&A partitioning algorithms.
    - STRIDED and BLOCKED : worst-case parallel time from $\Theta(n)$ to $\Theta(n/p + \log p)$.
- Better cost bound for F&A changing recursion parameters.

Conclusions (2)

Implementation contributions: carefully designed implementations following STL `partition` specifications.

Detailed experimental comparison. Conclusions:

- Algorithm of choice: F&A (ours was best).
- Benefits in practice of the cleanup algorithm very limited.
- I/O limits performance as the number of threads increases.

Thank you for your attention

More information:
`http://www.lsi.upc.edu/~lfrias`.

References

R. S. Francis and L. J. H. Pannan.
A parallel partition for enhanced parallel quicksort.
Parallel Computing, 18(5):543–550, 1992.

P. Heidelberger, A. Norton, and John T. Robinson.
Parallel quicksort using fetch-and-add.
IEEE Trans. Comput., 39(1):133–138, 1990.

J. Singler, P. Sanders, and F. Putze.
The Multi-Core Standard Template Library.
In *Euro-Par 2007: Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 682–694, Rennes, France, 2007. Springer Verlag.

P. Tsigas and Y. Zhang.
A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000.
In *11th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2003)*, pages 372–381, 2003.