Parallel Partition Revisited[*]

Leonor Frias [†]   Jordi Petit

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
lfrias@lsi.upc.edu

Abstract

In this paper we consider parallel algorithms to partition an array with respect to a pivot. We focus on implementations for current widely available multi-core architectures. After reviewing existing algorithms, we propose a modification to obtain the minimal number of comparisons. We have implemented these algorithms and drawn an experimental comparison.

1 Introduction

The partitioning of an array is a basic building block of many key algorithms, as quicksort and quickselect. Partitioning an array with respect to a pivot x consists of rearranging its elements such that, for some splitting position s, all elements at the left of s are smaller than x, and all other elements are greater or equal than x. It is well known that an array of n elements can be partitioned sequentially and in-place using exactly n comparisons and m swaps, where m is the number of elements in the array that are greater than x and whose original position is smaller than s.

In this paper we consider the problem of partitioning an array in parallel, focusing on current widely available multi-core architectures.

Several algorithms have already been proposed to efficiently perform a partition in parallel [4, 2, 10, 8, 9]. In this paper we consider a simple algorithm by Francis and Pannan [2], a fetch-and-add based algorithm by Tsigas and Zhang [10] and a variation of it implemented in the MCSTL library [9]. These algorithms, which we survey in Section 2, seem suitable for a practical multi-core implementation. However, in order to avoid too much synchronization, they perform more than n comparisons and m swaps. Though very different in nature, they can be divided into three main phases: a) A sequential setup of the work of each processor, b) a parallel main phase in which most of the partitioning is done, and c) a cleanup phase, which is usually done sequentially.

In this paper we show that these parallel partitioning algorithms disregard part of the work done during the parallel main phase when doing the cleanup phase. In order to overcome this drawback, we propose an alternative parallel cleanup phase that uses the whole comparison information of the parallel phase. A small static order-statistics tree helps us to efficiently locate the elements to be swapped and to swap them in parallel. With this new method, we obtain scalable parallel partitioning algorithms that achieve an optimal number of comparisons. We provide a detailed analysis.

We have implemented all these algorithms, both with their original cleanup and with our cleanup. C++ and OpenMP were used, aiming at a suitable multi-core implementation of the current sequential implementation of the `partition()` function at the Standard Template Library (STL) of the C++ programming language [5]. The only previously available parallel implementation was the one in the MCSTL library [9]. We have evaluated all the algorithms presented in this paper either using or not our cleanup algorithm. Our goal is to get a comparison of their behavior when executed on a currently inexpensive widely available parallel machine, namely a machine with two quad-core processors.

The paper is organized as follows. In Section 2, we present the considered algorithms. Then, in Section 3, we present our cleanup algorithm. In Section 4, we briefly present our implementation of

[*]Supported by ALINEX project (TIN2005-05446) and FET proactive integrated project 15964 (AEOLUS)

[†]Supported by grant number 2005FI 00856 of the *Agència de Gestió d'Ajuts Universitaris i de Recerca* with funds of the European Social Fund.

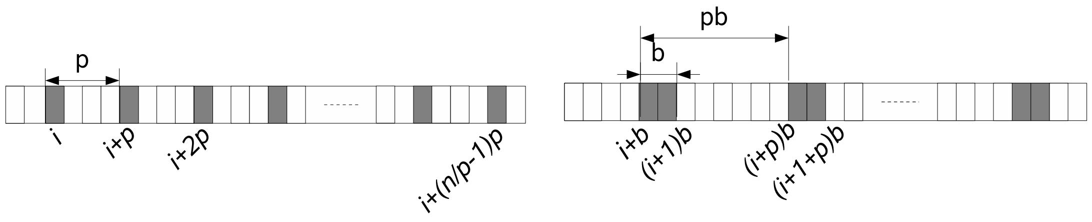


Figure 1: Comparison of STRIDED and BLOCKED algorithms.

the previous algorithms. Finally, in Section 5, we present our experimental results. We sum up the conclusions of this work in Section 6.

2 Previous work and a variant

In this section, we present an overview of the partitioning algorithms we consider in this paper: STRIDED by Francis and Pannan [2], F&A by Heidelberger *et al.* [4] (for a PRAM) and by Tsigas and Zhang [10] (for a multiprocessor). We also present BLOCKED, our cache-aware variation of STRIDED.

In all cases, the algorithms use p processors to partition in parallel, and the input is given by an array of n elements and a pivot. We assume $p \ll n$. Our description of the algorithms disregards some details as rounding issues and extreme cases; we refer the reader to the original papers for complete details.

2.1 STRIDED algorithm

The STRIDED algorithm [2] works as follows:

1. **Setup:** The setup is done sequentially: The input is (conceptually) divided into p pieces of size n/p. The pieces are not made of consecutive elements, but one of every p elements instead. That is, the i-th piece is made up of elements $i, i+p, i+2p, \ldots$; see Figure 1 left.
2. **Main phase:** The main phase is done in parallel by p processors. The i-th processor gets a piece, applies sequential partitioning on it, and returns its splitting position v_i.
3. **Cleanup:** Let $v_{min} = \min\{v_i : 1 \le i \le p\}$ and $v_{max} = \max\{v_i : 1 \le i \le p\}$. It holds that all the elements at the left of v_{min} and at the right of v_{max} are already well placed with respect to the pivot. In order to complete the partition, sequential partition is applied to the range (v_{min}, v_{max}).

The main phase takes $\Theta(n/p)$ parallel time. For random inputs, the cleanup phase is expected to take constant time. However, in [2] it not stated that this cleanup requires $\Theta(n)$ time in the worst-case: Half of the pieces may only contain elements smaller than the pivot and the rest of pieces may contain only elements greater than the pivot. Then, $v_{min} \le p$ and $v_{max} \ge n-p$, and $|(v_{min}, v_{max})| = \Theta(n)$. Therefore, in the worst case, the parallel efficiency of STRIDED tends to 0.

2.2 BLOCKED algorithm

Accessing elements with stride p in the previous algorithm may incur in a high cache miss ratio in the memory hierarchy. To overcome this problem, we propose our BLOCKED algorithm that uses blocks of b consecutive elements instead of individual elements. Each block in the piece is separated by stride p blocks; see Figure 1 right.

As STRIDED is a particular case of BLOCKED with $b = 1$, in the following we will often just consider BLOCKED.

2.3 F&A algorithms

We now consider the algorithms from [4], which originated in the PRAM model. Their basic idea is that elements from both ends of the array are taken using fetch-and-add instructions. Fetch-and-add instructions (atomically increment a variable and return its original value) were introduced in [3] and are useful, for instance, to implement synchronization and mutual exclusion.

In a first approach, exactly one element is taken at a time and so, at the end of the parallel phase, the array is already partitioned. In this case, n fetch-and-add operations are used. In a second approach, the algorithm is generalized to blocks: a block of b elements is acquired at each fetch-and-add instruction. So, the number of fetch-and-add instructions is n/b. However, in this case, some sequential cleanup remains to be applied after the parallel phase.

Later, [10] presented a version of this second approach for conventional parallel machines obtaining quite good speedups for quicksort. More recently, a further variant of this blocked algorithm has been included in the MCSTL library [9]. In the latter, the cleanup phase is partially done in parallel.

Let us now briefly describe these F&A algorithms:

1. **Setup:** Each processor takes two blocks, one from each end of the array. Namely, one left block and one right block.
2. **Main phase:** While there are blocks, each processor applies the so-called *neutralize* method to its two blocks. The neutralize method consists on applying the sequential partitioning algorithm to the array made by (conceptually) concatenating the right block to the left. However, the left and right pointers to the current elements cannot cross the borders of a block. When a left (right) block is completely processed (i.e. neutralized), a fresh left (right) block is acquired and processed.
3. **Cleanup:** At this point, at most p blocks remain unneutralized. Each author presents a different cleanup algorithm:
   — In [10], while unneutralized blocks remain, one block is taken from each end and neutralization is applied to them. Then, the unneutralized blocks are placed between the neutralized blocks. At most p blocks need to be swapped and this is done sequentially. Finally, sequential partition is applied to the range of blocks with unprocessed elements.
   — In [9], in contrast, all neutralized blocks are placed between the neutralized blocks. Then, the parallel partitioning algorithm is applied recursively to this range. The number of processors is divided by two in each call until there is only one processor or one block. Finally, the remaining array is partitioned sequentially.

The main parallel phase takes $\Theta(n/p)$ parallel time. In [10] the cleanup phase takes $\Theta(bp)$ time and is done sequentially. Rather, in [9], the cleanup takes $\Theta(b\log p)$ parallel time.

3 The new parallel cleanup phase

We now present a new cleanup phase that uses all the information from the parallel phase so that no element is compared twice with the pivot and the elements are fully swapped in parallel. We have successfully applied our cleanup algorithm on the top of STRIDED, BLOCKED and F&A. This section is organized as follows: first, we introduce some terminology. Then, we present the data structure on which our cleanup algorithm relies. Then, we present the cleanup algorithm itself. Finally, we analyze the resulting STRIDED, BLOCKED and F&A algorithms.

3.1 Terminology

In the following, we shall use the following terms to describe our algorithm. A *subarray* is the basic unit of our algorithm and data structure. The *splitting position* v of an array is the position that would occupy the pivot after partitioning. A *frontier* separates a subarray in two consecutive parts that have different properties. A *misplaced element* is an element that must be moved by our algorithm. We denote by m the total number of misplaced elements and by M the total number of subarrays that may have misplaced elements.

The case of BLOCKED. In the case of BLOCKED, subarrays correspond with exactly one of the p pieces. Moreover, the BLOCKED algorithm can easily know v after the parallel phase. The frontier of a subarray corresponds with the position that would occupy the pivot after partitioning this subarray. Thus, a frontier defines a left and a right part. A misplaced element corresponds either to an element smaller than the pivot that is on the right of v (misplaced on the right) or to an element greater than the pivot that is on the left of v (misplaced on the left). The total number of subarrays that may have misplaced elements (M) is at most p.

The case of F&A. In the case of F&A, a subarray corresponds to one block. The frontier separates a processed part from an unprocessed part. The processed part of left blocks is the left part and the processed part of right blocks is the right part. F&A cannot know v after the parallel phase, but can ensure that v is in some range $V = [v_{\text{beg}}, v_{\text{end}}]$. A misplaced element corresponds either with a processed element that is in V or with an unprocessed element that is not in V.

We consider now the array of elements made by left blocks and the array of elements made by right blocks. We denote by β the global border that separates the left and right blocks (i.e. the point where no more blocks could be obtained).

In the left blocks, a misplaced element on the left is an unprocessed element in a position smaller than v_{beg} and a misplaced element on the right is a smaller element than the pivot in a position greater or equal than v_{beg}. Let μ_l be the total number of misplaced elements in left blocks and rank those misplaced elements starting to count from the leftmost towards the right. In the right blocks, a misplaced element on the left is an element greater than the pivot in a position smaller or equal than v_{end} and a misplaced element on the right is an unprocessed element in a position greater than v_{beg}. Let μ_r be the total number of misplaced elements in right blocks and rank those misplaced elements starting to count from the rightmost towards the left. Note that $m = \mu_l + \mu_r$ and that the total number of subarrays that may have misplaced elements M is at most $2p$, because there at most p blocks that may contain unprocessed elements and there at most p blocks that may contain misplaced processed elements.

3.2 The data structure

We use a complete binary tree with M leaves (or the next power of two if M is not a power of two) to know which pairs of elements must be swapped. This tree is shared by all processors and is stored in an array (like a heap, which provides easy and efficient access to the nodes).

Each leaf of the tree stores information of the i-th subarray. Specifically, how many elements are misplaced to the left and to the right of its frontier (m_l^i and m_r^i) and how many elements are in the left and in the right to its frontier (n_l^i and n_r^i). The internal nodes accumulate the information of their children but do not add any new information. In particular, the root stores the information of the array made of all the subarrays in the leaves.

So, our tree data structure can be considered as a special kind of order-statistics tree in which the internal nodes have no information by themselves. An order-statistics tree (see e.g. [1, Section 14]) perform rank operations efficiently using the information of the size of the subtrees.

Figure 2 shows two instances of our tree data structure.

3.3 The algorithm

Tree initialization phase. We first show how to efficiently compute our data structure. The tree is initialized using two bottom-up phases. The first phase depends on the partition algorithm used in the main parallel phase, whilst the rest of phases do not.

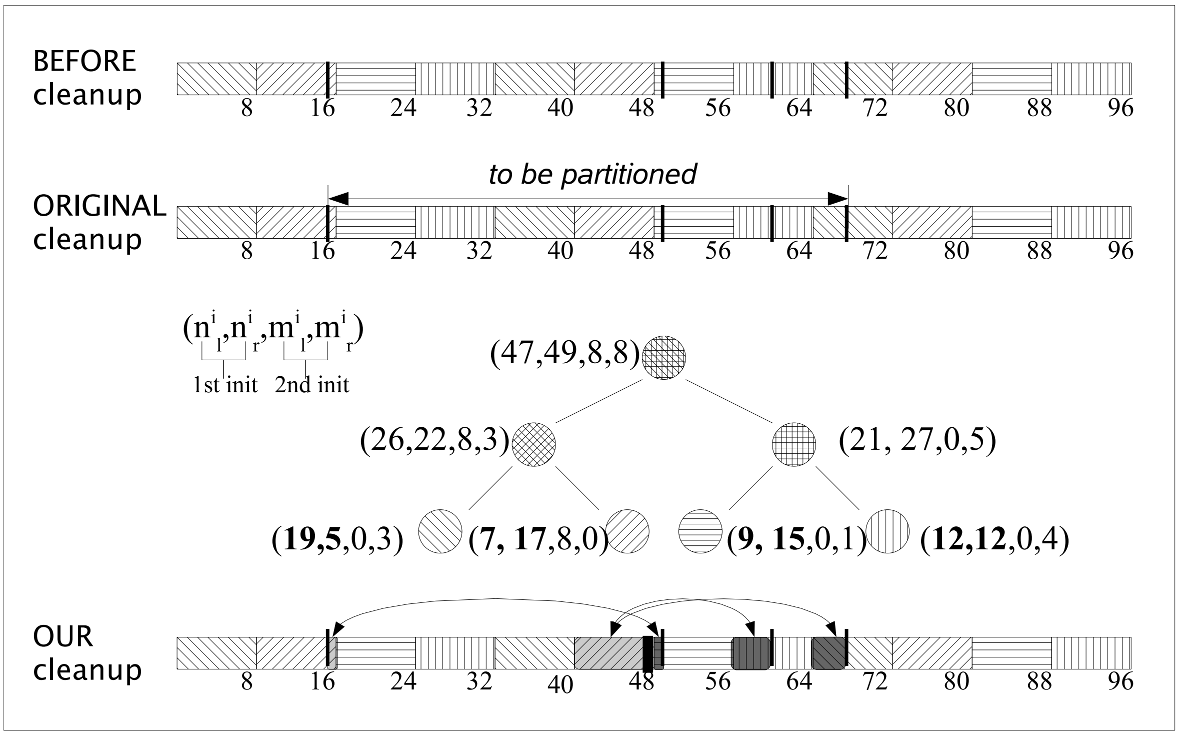
First initialization of the leaves. In the case of BLOCKED, the leaves get the values n_l^i and n_r^i that have easily been computed for each subarray i during the parallel phase.

In the case of F&A, the left (right) blocks that contain unprocessed elements can easily known after the parallel phase. The left (right) blocks that contain misplaced elements but have already been processed can only be located between the left (right) unprocessed blocks. In order to locate the latter blocks efficiently, we sort the unneutralized blocks with respect to the initial position of the block in the array. Then, we iterate on left (right) blocks (sequentially) to the left (right) of the border β until p neutralized blocks have been found or the leftmost (rightmost) unneutralized block has been reached.

First initialization of the non-leaves: Using a parallel reduce operation, each internal node computes its n_l^j and n_r^j values from its children. As a result, the root stores the number of left and right elements in the whole array. Thus, the splitting point v can be directly deduced.

Second initialization of the leaves: The leaves get the values m_l^i and m_r^i using n_l^i, n_r^i and v. At this point, it may turn out that some subarrays have no misplaced elements. This does not disturb the correctness of our algorithm.

Second initialization of the non-leaves: The number of misplaced elements for the internal nodes are computed using a second parallel reduction operation on m_l^j and m_r^j fields.



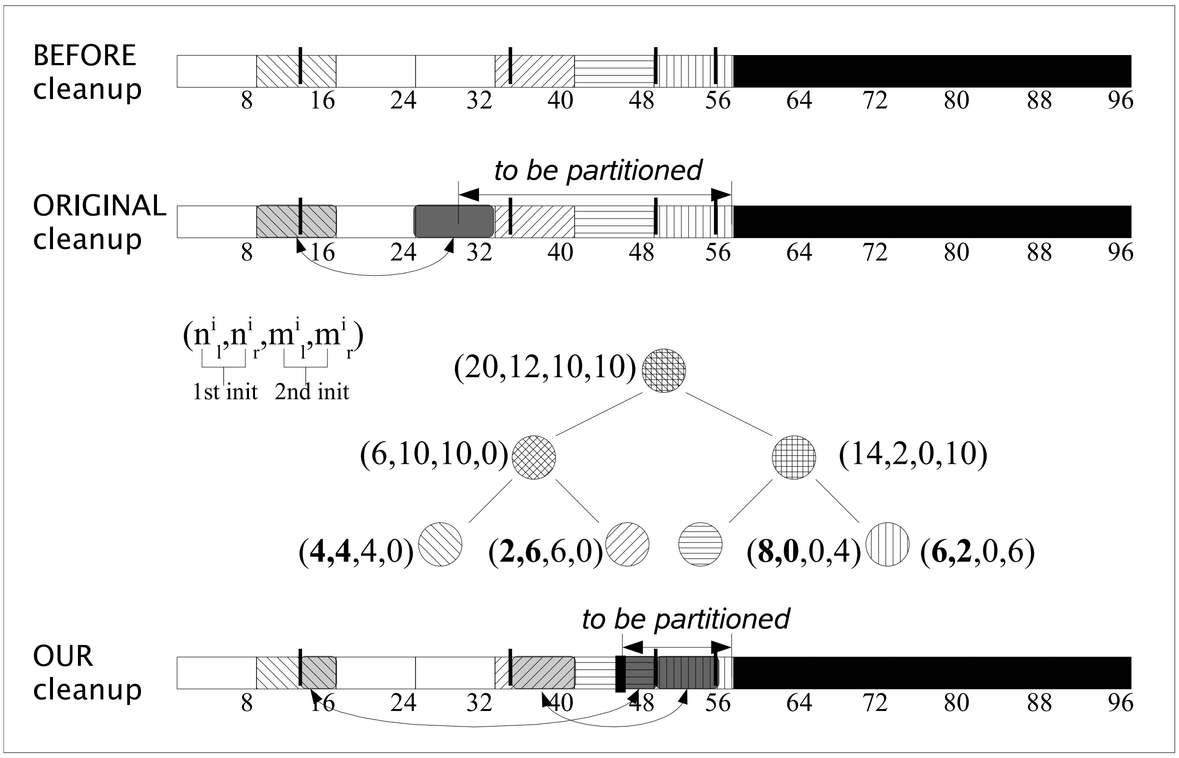


Figure 2: Example of our data structure.

Parallel swapping phase. We now show how to use the information in our data structure in order to swap the misplaced elements in parallel and without using any comparison operation. This phase is independent of the specific partitioning algorithm.

The total number of misplaced elements is used to distribute the work equally among the processors. A range of ranks $[r_i, s_i)$ of misplaced elements to swap is assigned to each processor.

The elements are swapped in ascending rank. Specifically, the j-th misplaced left element is swapped with the j-th misplaced right element. To locate the first pair of elements to swap, respective rank queries are made to the tree. That is, a query is made for the r_i left misplaced element and another for the r_i right misplaced element. Misplaced elements are swapped as long as the rank s_i is not reached. If the rank s_i has not yet been reached but the current subarray has no more misplaced elements, the next subarray is fetched. Let c_i be the position in the tree corresponding to the current subarray. Then, the next subarray of left misplaced elements is in $c_i + 1$ and the next subarray of right misplaced elements is in $c_i - 1$.

Completion phase. This phase depends on the specific partitioning algorithm. In the case BLOCKED, the whole array has already been partitioned and we are done. In the case of F&A, some unprocessed elements may remain. When this happens, V is not empty and includes exclusively all the unprocessed elements. In order to obtain a valid partition, we apply $\log p$ times our parallel partitioning algorithm recursively in V using blocks of half their original size until b elements or less remain. Sequential partition is applied to those remaining elements. Note that in each recursive call, the size of the problem is at most half the previous call because p blocks are guaranteed to have been fully processed.

3.4 Cost analysis

The following theorem shows that our cleanup phase achieves an optimal number of comparisons:

Theorem 3.1. *The* BLOCKED *and* F&A *algorithms perform exactly n comparisons when using our cleanup algorithm.*

Proof. The tree initialization and the parallel swapping phases perform no comparisons for both BLOCKED and F&A.

In the case of BLOCKED, the completion phase is empty. Therefore, no comparisons are performed during cleanup for BLOCKED and thus, comparisons are only performed during the main parallel phase, which are exactly n.

In the case of F&A, after the first main parallel phase exactly $n - |V|$ elements have been compared and $|V|$ elements have remained unprocessed. In the next recursive step, V is the input. Besides, elements can only be compared during a certain parallel phase and at most once. All the elements must be eventually compared because our algorithm produces a valid partition, therefore, our cleanup algorithm makes exactly $|V|$ comparisons, and therefore, n comparisons are needed as a whole. □

The following results bound the parallel time of the BLOCKED and F&A algorithms:

Lemma 3.2. *The tree initialization phase takes $\Theta(\log p)$ parallel time for* BLOCKED *and* F&A *algorithms.*

Proof. The algorithm-independent part takes $\Theta(\log p)$ parallel time because all the work is done in parallel, and is dominated by the two parallel reductions, which can be performed in logarithmic parallel time [7].

In the case of BLOCKED, the algorithm-dependent part takes constant parallel time because the initial value of each leaf can be computed trivially and in parallel.

In the case of F&A, the algorithm-dependent part takes $\Theta(\log p)$ parallel time, because we first sort the $2p$ elements in $\Theta(\log p)$ parallel time using the p processors [6].

Thus, in both cases, the cost is $\Theta(\log p)$ parallel time. □

Lemma 3.3. *The parallel swapping phase performs exactly $m/2$ swap operations and requires $\Theta(m/p)$ parallel time for any partition algorithm. In the case of* F&A*, this parallel time is $O(b)$.*

BLOCKED						
	total comparisons		total swaps		parallel time	
	original	with tree	original	with tree	original	with tree
main	n		$\leq n/2$		$\Theta(n/p)$	
cleanup	$v_{max} - v_{min}$	0	$m/2$	$m/2$	$\Theta(v_{max} - v_{min})$	$\Theta(m/p + \log p)$
total	$n + v_{max} - v_{min}$	n	$\leq (n+m)/2$	$\leq (n+m)/2$	$O(n)$	$\Theta(n/p + \log p)$

F&A						
	comparisons		swaps		parallel time	
	original	with tree	original	with tree	original	with tree
main	$n - \lvert V \rvert$		$\leq (n - \lvert V \rvert)/2$		$\Theta(n/p)$	
cleanup	$\leq 2bp$	$\lvert V \rvert$	$\leq 2bp$	$\leq m/2 + \lvert V \rvert$	$\Theta(b \log p)$	$\Theta(\log^2 p + b)$
total	$\leq n + 2bp$	n	$\leq (n - \lvert V \rvert)/2 + 2bp$	$\leq (n+m)/2 + \lvert V \rvert$	$\Theta(n/p + b \log p)$	$\Theta(n/p + \log^2 p)$

Table 1: Summary of costs for BLOCKED and F&A algorithms

Proof. There are m misplaced elements after the parallel phase. In the parallel swapping phase, pairs of misplaced elements are swapped so that their final position is not misplaced. Therefore, $m/2$ swap operations are needed. Besides, the pairs to swap are evenly divided among the p processors, thus swapping all the pairs takes $\Theta(m/p)$ time.

In the case of F&A, as $m \leq 2bp$, swapping all the pairs takes $O(b)$ parallel time. □

Theorem 3.4. *The cleanup phase takes $\Theta(m/p + \log p)$ parallel time for the* BLOCKED *algorithm and the whole partition takes $\Theta(n/p + \log p)$ parallel time.*

Proof. From Lemmas 3.2 and 3.3 it follows that the cleanup phase takes $\Theta(m/p + \log p)$ parallel time for the BLOCKED algorithm. Given that $m = O(n)$, the whole BLOCKED algorithm takes $\Theta(n/p + \log p)$ parallel time in the average and in the worst-case. □

Theorem 3.5. *Consider $p \leq b$. The cleanup phase takes $\Theta(\log^2 p + b)$ parallel time for the* F&A *algorithm and the whole partition takes $\Theta(n/p + \log^2 p + b)$ parallel time.*

Proof. The F&A algorithm takes $T(n,p) = \Theta(n/p) + C(b,p)$, where $C(b,p)$ is the cost of our cleanup algorithm. Our cleanup algorithm takes $C(b,p) = b + \log p + T'(b/2, \log p)$ parallel time, where T' is defined by the following recurrence:

$$T'(b,i) = \begin{cases} O(3b + \log p) + T'(b/2, i-1) & \text{if } i > 1, \\ O(2b/p) & \text{otherwise.} \end{cases}$$

Note that there are $2\beta p$ blocks at the beginning of each recursive step and $\log p - 1$ recursive steps are needed. It follows that $C(b,p) = O(\log^2 p + b)$ parallel time. □

The new bound $\Theta(n/p + \log p)$ for F&A improves previous bounds for this algorithm (provided $\log p \leq b$, which is of practical relevance).

Table 1 summarizes worst-case results for BLOCKED and F&A algorithms.

4 Implementation

Since implementations of STRIDED were not available, we have resorted to implement it ourselves. We have also implemented its BLOCKED variant, which we have previously introduced to improve its cache performance. As for the F&A algorithm, we have taken its implementation from MCSTL 0.7.3-beta and we have implemented it independently by ourselves.

Our implementation of F&A and the one in MCSTL differ in the following: a) our implementation statically assigns the initial work and, so, avoids mutual exclusion here; b) our implementation does not use as many `volatile` variables and critical regions are slightly simpler; and c) our implementation performs less comparisons using a better book-keeping.

On the other hand, we have implemented our cleanup phase on the top of the previous four algorithms.

All the algorithms are coded in C++ and OpenMP. Their functionality follows the specification of the `partition()` function of the STL, so that they could replace the current sequential implementation.

All our code is available at http://www.lsi.upc.edu/∼lfrias/parpar.

5 Experimental analysis

Experimental setting. The tests were run on a machine with 4 GB of main memory that has two sockets, with an an Intel Xeon quad-core processor each one at 1.66 GHz with a shared L2 cache of 4 MB shared among two cores. We have used the GCC 4.2.0 compiler with the `-O3` optimization flag.

Basic evaluation. We have first analyzed the speedup of the three considered algorithms (STRIDED, BLOCKED, and F&A) with or without the application of our cleanup phase when partitioning a large number of random integers. The speedup is always measured with respect to the sequential partition algorithm of the STL and use two to eight parallel threads. The block size *b* has been set to 10^4 (see the reason below). All tests have been repeated 100 times; figures show averages.

Figure 3(a) shows the results. In this figure, Strided refers to our implementation of the original STRIDED algorithm, BlockedStrided refers to our implementation of the original BLOCKED algorithm, F&A_MCSTL refers to the MCSTL implementation of the F&A algorithm, F&A refers to our own implementation of F&A. We add the suffix _tree to the previous labels to refer to the algorithm with our modified parallel cleanup phase.

According to the measured speedups, we can see that F&A is better than BLOCKED, which is better than STRIDED. Whereas the speedup of STRIDED is nonexistent for more than two threads, BLOCKED performs reasonably well and our F&A implementation achieves some better results than the MCSTL F&A. We can also observe that the use of our cleanup phase maintains the same speedups for STRIDED and BLOCKED and improves slightly the speedup of F&A, making it almost perfect for up to 4 threads.

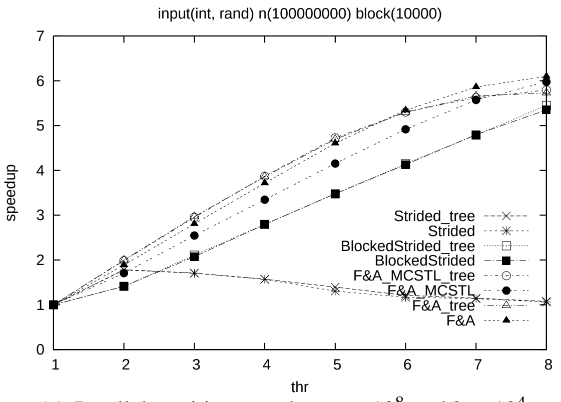
The awful performance of STRIDED is due to its high cache miss ratio; its behavior clearly contrasts with BLOCKED (which uses blocks of elements rather than individual elements).

In order to better understand the reason of the sudden decay of efficiency when using more than 5 threads we have devised two new experiments. The first one reproduces the same experiment as before but uses a slower comparison function. The result is shown in Figure 3(c), were we can observe that all algorithms show similar behavior and excellent speedups with up to 8 threads. Specifically, there is not much of a difference whether our cleanup phase is used or nor. On the other hand, our second experiment has consisted in measuring the speedup of a trivial parallel program to compute the sum of two arrays. The resulting speedups (not shown) are also not optimal for the biggest number of threads. So, we can conclude that memory bandwidth is limiting the efficiency of the partitioning algorithms, which are demanding with regard to I/O.

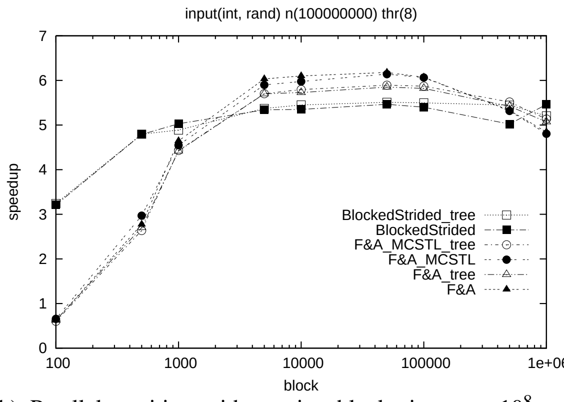
Influence of block size. As said before, several algorithms rely on a block size parameter *b*. In order to determine its optimal value, we have executed various tests, with eight threads and different values of *b*. The results in Figure 3(b) show that, except for very small block sizes, the performance is not much affected. Furthermore, given that for smaller input sizes, big block sizes are not convenient, our selection has been $b = 10^4$.

Operations count. We now analyze in more detail the behavior of the cleanup phase of each of the implementations counting operations. Figures 3(e) and 3(f) show respectively the number of extra comparisons and swaps with respect to the sequential implementation. Note that these results are depicted divided by the block size.

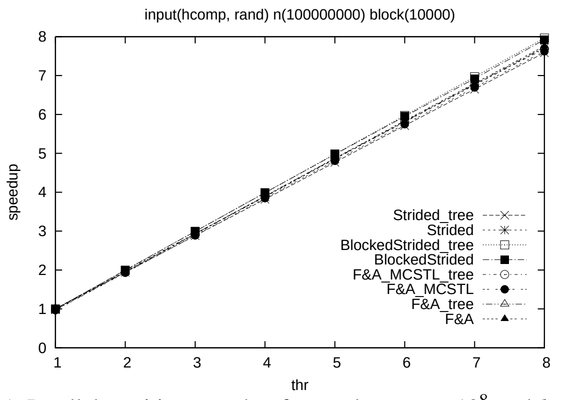
First of all, Figure 3(e) shows that our cleanup algorithm leads to a parallel partition implementation making no extra comparisons. Combining our cleanup algorithm with the original MCSTL algorithm does not achieve so, because in their main parallel phase, extra comparisons are performed each time a new block is fetched. Specifically, our experiments show that two comparisons are repeated per block in the average.



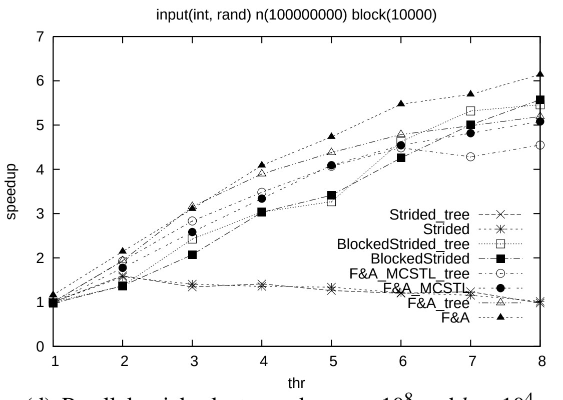
(a) Parallel partition speedup, $n = 10^8$ and $b = 10^4$



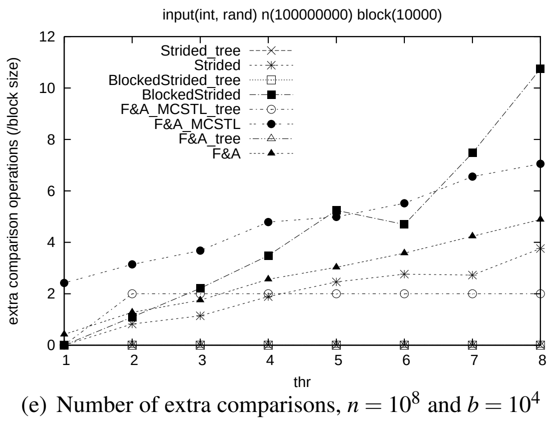
(b) Parallel partition with varying block size, $n = 10^8$ and num_threads $= 8$



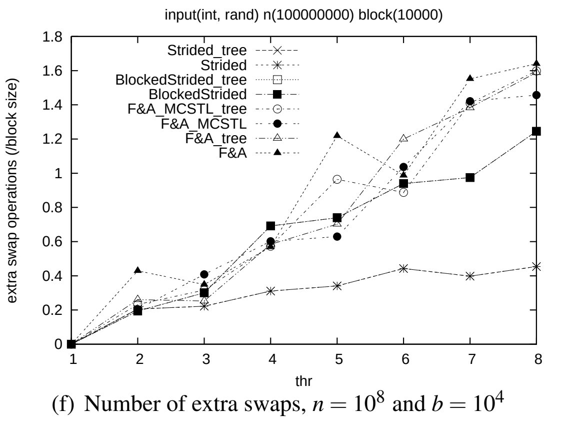
(c) Parallel partition speedup for costly $<$, $n = 10^8$ and $b = 10^4$



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



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



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

Figure 3: Experimental results

Secondly, Figure 3(f) shows that our cleanup algorithm does not need more swaps than the original cleanup algorithms. Essentially, the same number of extra swaps are needed. In the case of F&A, we could not give such an equality analytically.

As a by product, the counter results show that the number of misplaced elements resulting from the parallel phase of partitioning an array of random elements is really small, no matter which is the algorithm. In particular, note that STRIDED is the algorithm that performs less extra operations but whose performance is worst because of its bad cache usage.

Application: quickselect. Quicksort and quickselect are the typical applications of partitioning. As quicksort offers two (not exclusive) ways to be parallelized —parallelizing the partition step and parallelizing the independent work by dive and conquer, we found more interesting to focus on quickselect. For this test, we have just used the `nth_element()` function in the STL, calling any of the parallel partitioning algorithms in this paper (which just use one thread if the array is small).

The results are shown in Figure 3(d). These are coherent with those of partition but different given that the relative behavior between the algorithms changes slightly with the size of the input. First, our F&A implementation advantage increases. Second, our cleanup algorithm harms a little F&A based quickselect. Indeed, in our experiments we have observed that for a big number of threads and as input gets smaller, using our cleanup algorithm with F&A is counterproductive. Finally, Figure 3(d) shows that the simple BLOCKED algorithm performs quite well.

6 Conclusions

In this paper we have presented, implemented and evaluated several parallel partitioning algorithms suitable for multi-core architectures.

From an algorithmic point of view, we have described a novel cleanup parallel algorithm that does not disregard comparisons made during the parallel phase. This cleanup has successfully been applied to three partitioning algorithms: STRIDED, BLOCKED (a cache-aware implementation of the former) and F&A. In the case of STRIDED and BLOCKED, a benefit of our cleanup is reducing its parallel time in the worst case from $\Theta(n)$ to $\Theta(n/p+\log p)$. In the case of F&A, we have shown how to modify it to reduce its parallel time from $\Theta(n/p+b\log p)$ to $\Theta(n/p+\log^2 p)$. Unlike their original versions, these algorithms perform the minimal number of comparisons when using our cleanup phase.

As automatic parallelization is still limited, and as parallel programming is hard and expensive, the use of parallel libraries is a simple way to benefit from the increasing power of the new generation of multi-core processors. From this engineering perspective, we have contributed carefully designed implementations of the afore mentioned algorithms that are compliant with the `partition()` function of the STL.

Finally, and from an experimental point of view, we have conducted an evaluation to compare those algorithms and implementations. According to our experiments, the partitioning algorithm of choice is F&A, because it scales nicely. Moreover, our implementation performs slightly better than the one in MCSTL. However, the results also show that, in practice, the benefits of our cleanup algorithm are limited. This happens because the number of misplaced elements after the parallel phase are very small.

Our experiments also show that I/O between the memory and the processor limits the performance achieved by parallel implementations as the number of threads increases. It remains to be further investigated how these results change for a bigger number of available cores or/and memory bandwidth.

References

[1] T. H. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, Cambridge, 2nd edition, 2001.

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

[3] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer - designing a MIMD, shared-memory parallel machine. In *ISCA '98: 25 years of*

the international symposia on Computer architecture (selected papers), pages 239–254, New York, NY, USA, 1998. ACM Press.

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

[5] International Standard ISO/IEC 14882. *Programming languages — C++*. American National Standard Institute, 1st edition, 1998.

[6] J. JáJá. *An introduction to parallel algorithms*. Addison-Wesley, Redwood City, CA, USA, 1992.

[7] V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, Boston, MA, USA, 2002.

[8] J. Liu, C. Knowles, and A. Davis. A cost optimal parallel quicksorting and its implementation on a shared memory parallel computer. In *Proceeding of Parallel and Distributed Processing and Applications, Third International Symposium, ISPA 2005*, volume 3758 of *Lecture Notes in Computer Science*, pages 491–502. Springer, 2005.

[9] 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. Springer Verlag.

[10] 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.