File: C:/bsd/rigel/sort/Papers/QP2.docx
Date: 2020 June

Experiments with Parallel Single & Multi Pivot Quicksort

Dennis de Champeaux
ddc2 AT ontooo DOT com
2020

Abstract
Quicksort has evolved beyond single pivot algorithms. We describe the results of their performance when they are parallelized. Three algorithms using respectively 1, 2 and 3 pivots, different array sizes and up to four threads provide a 'cube' of timing data. Quicksort's performance degrades on large arrays - due to cache misses - and is compared when the thread count increases. The 3-pivot algorithm degrades less than the 1-pivot version because of less array operations. In spite of these headwinds we still can report speed improvements over the current state of the art quicksort versions we found.

Introduction
Single pivot Quicksort is a fast, in place sorting algorithm, [5]. Its quadratic worst case has been successfully tamed by the Musser-defense [8]: keep track of the recursion depth and if too high, switch to heapsort thereby guaranteeing NlogN behavior. Multi pivot algorithms are finally showing up [2], [10], [6]. Parallelizing them and using current multi-core, shared memory machines is a next opportunity. This paper focusses exclusively on empirical findings and ignores the developments during the previous decade on the theoretical side.

Preliminaries
The three algorithms used in this study are respectively 1-pivot FourSort, 2-pivot FiveSort and 3-pivot SixSort, see [3]; the naming of them comes from the number of sorting members in these hybrids. The selection of these algorithms is motivated in the appendix *Our 1, 2 and 3-pivot sequential quicksort algorithms*. The performance of the sequential versions of these algorithms compares favorably against two qsort versions in the standard libraries, against DPQ that is part of the java.util package, and against MPQ [6].

These three algorithms have novel *design* features including more hybridization. Hybridization was introduced a long time ago by using insertion sort on small segment to avoid the overhead of sampling on these segments for pivot selection. Sampling is typically considered overhead to be minimized. Instead, we use many array elements, sort them and extract one or more pivots from the result. Insisting that segments must be properly initialized before partitioning starts has helped to reduce the code footprint and the comparison counts while the timings have improved as well. A 'special' member of the hybrids takes over when segment initialization fails, with as

an example a segment with constant elements. We employ the single-element-moving technique in some members of the hybrids as an alternative to the usual pairwise swapping to move elements around. We refer to their codes for the details, since alternative design choices are not the topic of this paper.

Implementation of a new algorithm uses typically first an integer array to avoid having to specify a comparison function since the coding language has comparison operators. Testing performance is better done with array elements that require a comparison function because these can be arbitrary expensive. Such a function can be instrumented also to obtain comparison counts, which is a better metric than swap counts. The timing data that we use for this paper has been obtained by sorting 'objects' with an integer field.

We have used uniform distributions throughout that are generated by the random number generator in C. Noteworthy is that while the allowed range was 0-32M and the maximum array size was 16M, distributions contained significantly many identical values. Algorithms that cannot exploit segments with constant data are at a serious disadvantage - as shown in the appendix *Our 1, 2 and 3-pivot sequential quicksort algorithms*.

Parallelization of quicksort type algorithms is 'absurdly' easy from a design perspective since the partitioning procedure generates disjoint array segments that can be handled in parallel. Creating a fixed number of threads initially avoids the repeated overhead of creating them. The appendix *How to parallelize quicksort* has the code how, among others, an idle thread grabs a task from a stack.

Overall performance improves somewhat by doing the first partitioning also in parallel with two threads. The recipe is to obtain a good pivot, partition in parallel the left half side and the right half side and swap the two resulting middle segments.

The raw data that we will discuss was obtained on an AMD machine: AMD fx-8350, 4Ghz, Win8.1+Cygwin, L1 cache 4x64Kb instruction + 8x16Kb data, L2 cache 4x2Mb, L3 cache 8Mb.

Timings were obtained with the usual warm ups and taking the average of repetitions. An array four times smaller was sorted with four times more repetitions.

The single thread timings were obtained with the sequential versions of the algorithms to avoid the overhead of putting tasks on and off a stack.

Observations

The raw data that we will discuss is available in the appendix *The raw data*.

Sorting small arrays – the 4K ones – is slower with 2, 3 and 4 threads than by using simply the sequential version; this is the case for each of the three algorithms considered. The overhead of thread creation is a likely cause.

One thread

The algorithms we timed are supposed to behave with an NlogN complexity. Taking for N1=4K and N2=16M we obtain for N2logN2/N1logN1 = 8192. Multiplying this number with the timing for the 4K array we obtain the expected timing for the 16M array. The actual timing for the sequential FourSort algorithm is 2.15 times higher, for FiveSort 2.21 and for SixSort 2.17. {We obtained on a Pentium machine the ratios 3.28, 2.94 and 3.19.} Cache misses is the candidate explanation as elaborated below.

Two threads

The situation improves for larger arrays. The 2-thread/1-thread timing ratio for FourSort converges to 0.61when the array size increases, see figure 1.

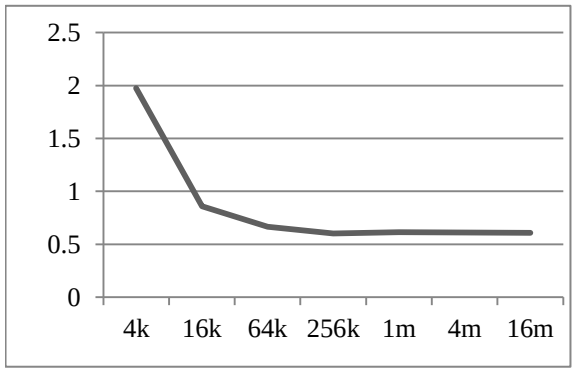

Fig 1 The 2-thread/1-thread timing ratio for FourSort

Similarly the 2-thread/1-thread timing ratio for FiveSort converges to 0.58 when the array size increases to 16M. And again for SixSort this ratio converges to 0.60.

Certainly it would have been better if these ratios had converged closer to 0.5, especially because the first partitioning is done also with two threads. Since all three converge to around 0.6 – while these algorithms are quite different – we can assume that there is a common candidate for the discrepancy; see below for an elaboration.

Three threads

The 3-thread/1-thread timing ratio for FourSort converges to 0.49 when the array size increases, see figure 2.

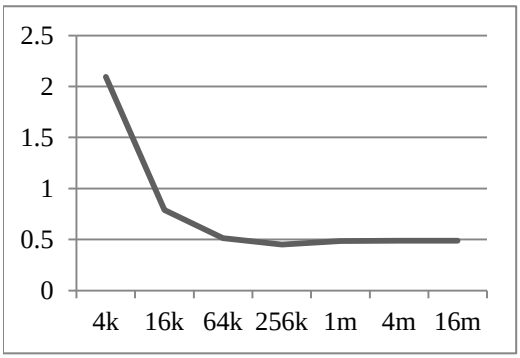

Fig 2 The 3-thread/1-thread timing ratio for FourSort

Similarly the 3-thread/1-thread timing ratio for FiveSort converges to 0.46 when the array size increases to 16M. And again for SixSort this ratio converges to 0.48.

Most significant of these three data points is that they differ even more from the target 0.33 than what we found with the 2-thread/1-thread timing ratio.

Four threads

The 4-thread/1-thread timing ratio for FourSort converges to 0.43 when the array size increases, see figure 3.

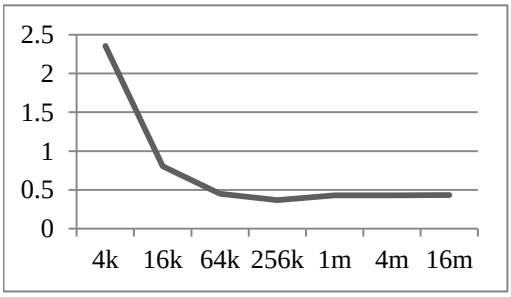

Fig 3 The 4-thread/1-thread timing ratio for FourSort

Similarly the 4-thread/1-thread timing ratio for FiveSort converges to 0.40, and the SixSort ratio converges to 0.43.

Again these ratios are far away from the target 0.25 and yet again FiveSort is the most close.

Another way of looking at the data is the timing ratios that compares FiveSort and SixSort against FourSort on arrays of size 16M only, see Table 1.

Table 1

# threads	1	2	3	4
FourSort	1	1	1	1
FiveSort	1.02	0.97	0.96	0.93
SixSort	0.90	0.89	0.88	0.89

Our 2-pivot FiveSort while marginally better than FourSort loses out against SixSort. Our best case complexity analysis shows – given the assumptions – that a 2-pivot algorithm uses more comparisons than 1- and 3-pivot algorithms. Hence we ignore it further.

Discussion

We attribute the deviation from NlogN performance when the array size increases, as reported above for single threaded FourSort, to cache misses. Our data allows identifying the impact of additional cache misses when more than one thread operates on the input array. We can calculate the ratio of the best plausible performance when multi threads are used against what we actually measured. More specifically we calculate the excess ratio with:

- 8192 times the timing of single thread FourSort on the 4K array, against
- the timings obtained on the 16M arrays, where the ratios for the threads 2-4 are adjusted because their timings are supposed to be two-to-four times less.

For FourSort we get:

# threads	1	2	3	4
Excess ratio	2.15	2.62	3.16	3.72

Clearly the threads have more trouble getting access to array elements due to congestion.

Similarly for SixSort we obtain:

# threads	1	2	3	4
Excess ratio	1.93	2.32	2.78	3.30

The relatively better ratios for SixSort over FourSort are consistent with our best case complexity analysis (given our assumptions): 3-pivot quicksort incurs less array operations than 1-pivot quicksort.

While these ratios are sobering, we can observe the benefit from using more threads – assuming that more cores are available. The timings for FourSort on 16M arrays:

# threads	1	2	3	4
Timing	6.80	4.14	3.33	2.94

Similarly for SixSort we obtain:

# threads	1	2	3	4
Timing	6.10	3.67	2.93	2.61

The advantage that 3-pivot SixSort has over 1-pivot FourSort is preserved when multiple pivots are used.

The 'best' timing ratio: 4-thread SixSort/ 1-thread FourSort is 0.383. An appendix gives the timing ratio of FourSort/B&M [B&M] at 0.966; hence with this ratio we get 4-thread SixSort/ 1-thread B&M at 0.370.

It should be clear by now that quicksort is a 'victim' of the memory hierarchy with multiple caches. We found two fixes. One used different hardware; Tsigas & Zhang (2003) [9]. The other one surrenders the in-place feature of quicksort; DistributionSort [4].

Conclusions and beyond

When we compare single threaded 3-pivot SixSort in the appendix *Our 1, 2 and 3-pivot quicksort algorithms* we can claim a 16% performance improvement against the state of the art as represented by the 1-pivot qsort sorters in the libraries. However, when we compare SixSort against our own 1-pivot FourSort similarly the advantage shrinks to 10% and this applies also for the threads in the range 2-4.

The timing of quicksort as a function of the array size does not scale to NlogN due to cache misses. Sobering is that each of our 1, 2 and 3-pivot parallelized algorithms also falls short similarly against expected performance: 2-threads improve with only 0.6 instead of 0.5, 3-threads improve 0.48 instead of 0.33, and 4-threads 0.43 improve instead of 0.25. The discrepancy ratio increases from 1.2 via 1.45 to 1.72.

Combining the slowdown for single thread FourSort when increasing from 4K to 16M arrays at 2.15 and the 0.43/0.25 = 1.72 slowdown when using 4 threads, the combined slowdown factor is 3.7.

Current machine architectures appear to be optimized for fine grained parallelism. Hence, different hardware is likely required for better performance of the single & parallelized multi-pivot versions.

In the meantime, the sorting community can still safely use the better performances of these novel quicksort versions that are duly protected against quadratic deterioration. Compensating for cache misses remains an excellent challenge.

References

[1] Jon Bentley and M. Douglas McIlroy, *Software - Practice and Experience*, vol. 23 (11), 1249-1265, 1993.

[2] https://github.com/ddccc/designchoices

[3] www.github.com/ddccc/C7

[4] [DistributionSort] https://en.wikipedia.org/wiki/Cache-oblivious_distribution_sort

[5] Hoare, C.A.R., "Quicksort", *Computer Journal*, vol 5, no 1, pp 10-16, 1962.

[6] Kushagra, S., A. Lopez-Ortiz & J. I. Munro, “Multi-Pivot Quicksort: Theory and Experiments”, 2013 November, http://epubs.siam.org/doi/pdf/10.1137/1.9781611973198.6

[7] Muqaddas, A, "Triple State Quicksort", https://arxiv.org/abs/1505.00558

[8] Musser, D., "Introspective Sorting and Selection Algorithms", *Software: Practice and Experience* (Wiley) **27** (8): 983–993, 1997.

[9] Tsigas, P. & Y. Zhang, "A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000", 2003, available at: http://www.cse.chalmers.se/~tsigas/papers/Pquick.pdf

[10] Yaroslavskiy, V., “Dual-Pivot Quicksort”, 2009.
The original link to this paper is dead.

Appendix Our 1, 2 and 3-pivot sequential quicksort algorithms

We compare in this appendix our quicksort versions against what is available in the libraries and what has recently become available.

The Gnu C Library has a quicksort version written by Douglas C. Schmidt around 1991. Its versatility remains amazing. One can use it for sorting native types (like integers, floats, etc.), for referenced items (like objects) and also for value items (like fixed length strings). The interface requires in addition to the array and its size, a comparison function and also a description of the length of the array elements in terms of the number of bytes.

A similar version, with the same interface, was written by Bentley & McIlroy (1993) and described in [1].

Our comparison of these two versions use referenced items that have an integer field. We obtained on an I3 laptop with an array size of 16M and uniform distribution the following results (LQ is the Schmidt version, B&M is the Bentley & McIlroy version):

Version	# comparisons	timing
LQ	4.70e8	15840
B&M	4.15e8	13107

LQ is the version available on Linux. There is no need here to descend into details, but there are indeed numerous opportunities to improve the design choices in LQ.

An array with constant data is a potential source for a worse case, quadratic performance. Both versions avoid a quadratic disaster, but differently:

Version	# comparisons	timing
LQ	3.86e8	4174
B&M	1.68e7	149

The B&M version is clearly superior, which is explained by its design choice to check an array element also for being equal to the pivot, and, if so, handle it differently.

Still, not all is well for the B&M version. Its code, available in the standard C library and in the Cygwin library is different from the description in [1]. Someone 'improved' it by switching to insertion sort under certain conditions. Our testing found distributions causing quadratic timings. In addition, the B&M design can be improved by ordering its two recursive calls, processing the smallest segment first and replacing the second call by tail iteration.

We are pleased to report that the B&M performance can be improved by our unpublished version FourSort, [3]. It deals only with referenced items and has an improved way to obtain pivots. For uniform distributions:

Version	# comparisons	timing
B&M	4.15e8	13107
FourSort	4.01e8	12664

For constant data:

Version	# comparisons	timing
B&M	1.68e7	149
FourSort	1.68e7	84

The versions discussed thus far use only a single pivot. Multi-pivot quicksorts have arrived in the previous decade. A dual-pivot (DPQ) version was added to the Java util package around 2011 for the Java native types, but not for objects. We ported that version to C and tested it against our FourSort and our 2-pivot FiveSort, [3]. We obtained for uniform distributions:

Version	# comparisons	timing
FourSort	4.01e8	12664
DPQ	5.63e8	12642
FiveSort	4.14e8	12339

The high comparison count of DPQ is partly explained by the higher comparison count of FiveSort against FourSort, which is predicted by a best-case complexity analysis for uniform distributions by assuming an oracle for optimal pivots. There are also opportunities to improve the DPQ design. We conjecture that the high comparison count was early on not noticed because DPQ was faster when tested on integer arrays since there is evidence that a 2-pivot quicksort incurs less array operations while the higher comparison count is masked by the comparisons done by the hardware. We are not the first one reporting problems with DPQ. Muqaddas

observed also that DPQ can be surpassed easily in [6]: "Dual Pivot Quicksort algorithm definitely results in significantly high number of swaps."

We obtained similar observations by comparing a 3-pivot version MPQ by [5] against our 3-pivot SixSort, [3], and against an experimental 3-pivot version C7 [3].

Version	# comparisons	timing
FourSort	4.01e8	12664
MPQ	4.55e8	11154
SixSort	4.05e8	11029
C7	4.05e8	10464

Appendix The raw data

FourSort timing data

# threads	1	2	3	4
4k	0.000386	0.000762	0.000808	0.000908
16k	0.00184	0.00158	0.00145	0.00148
64k	0.00937	0.00624	0.0048	0.0042
256k	0.0462	0.0278	0.0208	0.0169
1m	0.281	0.172	0.136	0.121
4m	1.46	0.891	0.711	0.626
16m	6.8	4.14	3.33	2.94

FiveSort timing data

4k	0.000384	0.000757	0.000807	0.000906
16k	0.00183	0.00159	0.00143	0.00147
64k	0.0092	0.00618	0.00478	0.00423
256k	0.046	0.0274	0.0208	0.0171
1m	0.286	0.169	0.137	0.118
4m	1.49	0.875	0.695	0.604
16m	6.97	4.04	3.19	2.74

SixSort timing data

4k	0.000344	0.000731	0.000783	0.000911
16k	0.00164	0.00155	0.00149	0.00169
64k	0.00824	0.00579	0.00485	0.0051
256k	0.0411	0.0255	0.0206	0.0193
1m	0.254	0.159	0.131	0.12
4m	1.33	0.812	0.658	0.592
16m	6.1	3.67	2.93	2.61

Appendix How to parallelize quicksort

We describe here how to transform a sequential Quicksort version *qs*, into a parallel version *qsp* using Posix threads.

Assume a sequential code fragment of qs with two recursive calls. (We *do* recommend ordering them to minimize stack space by executing the smallest segment first, but we leave that out here).

```
i = partitionIndex(A, n, m);
// A[i]=T, k<i -> A[k]<=T, i<k -> T<A[k]
qs(A, n, i-1);
qs(A, i+1, m);
```

Assuming that the first one is the smallest, we put for qsp the largest call on a task stack and proceed with the smallest one (we leave out replacing the recursive call by iteration) and get:

```
i = partitionIndex(A, n, m);
// A[i]=T, k<i -> A[k]<=T, i<k -> T<A[k]
addTaskSynchronized(ll, newTask(i+1, m)); // ll is the stack
qsp(A, n, i-1); // replace by iteration as desired
// qsp(A, i+1, m); this call is on the stack
// We are done because another thread will do the other sub segment
```

The addition to the stack function is:

```
void addTaskSynchronized(struct stack *ll, struct task *t) {
  pthread_mutex lock( &condition_mutex2 );
  push(ll, t);
  pthread_cond_signal( &condition_cond2 );
  pthread_mutex_unlock( &condition_mutex2 );
} // end of addTaskSynchronized
```

This requires having defined already the critical region protectors:

```
pthread_mutex_t condition_mutex2 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_cond2  = PTHREAD_COND_INITIALIZER;
```

The threads keep executing *sortThread.* It has a loop with first checking whether a task is available on the stack. If so, it extracts the array arguments and invokes quicksort. If there is no task the thread will suspend provided there is still at least one active thread. The thread will terminate when no task was found and there is no other thread still active.

```
// threads execute sortThread
void *sortThread(void *AA) { // AA-argument is NOT used
  struct task *t = NULL;
  for (;;) {
    pthread_mutex_lock( &condition_mutex2 );
      sleepingThreads++;
      while ( NULL == ( t = pop(ll) ) &&
              sleepingThreads < NUMTHREADS ) {
        pthread_cond_wait( &condition_cond2, &condition_mutex2 );
      }
      if ( NULL != t ) sleepingThreads--;
```

```
    pthread_mutex_unlock( &condition_mutex2 );
    if ( NULL == t ) { // Finished! Notify other threads if any
      pthread_mutex_lock( &condition_mutex2 );
      pthread_cond_signal( &condition_cond2 );
      pthread_mutex_unlock( &condition_mutex2 );
      break;
    }
    int n = getN(t);
    int m = getM(t);
    free(t);
    qsp(A, n, m); // A is global
  } // end of for-loop
  return NULL;
} // end of sortThread
```

Remains how this machinery gets started. Here a sketch of a procedure *callQs* that gets the ball rolling. The first partition is done sequentially in this version:

```
void callQs( << here init arguments >> ) {
// Check first whether the array size warrants parallel processing.
// If not call a sequential version and get out.
  sleepingThreads = 0;
  NUMTHREADS = numberOfThreads;
  ll = newStack();
  struct task *t = newTask(0, size-1);
  addTaskSynchronized(ll, t);
  pthread_t thread_id[NUMTHREADS];
  int i; int errcode;
  for ( i = 0; i < NUMTHREADS; i++ ) { // create threads ...
    if ( errcode=pthread_create(&thread_id[i], NULL,
                      sortThread, (void*) A) ) {
      errexit(errcode,"callQs -- pthread_create");
    }
  }
  for ( i = 0; i < NUMTHREADS; i++ ) { // ... and wait they finish
    if ( errcode=pthread_join(thread_id[i], NULL) ) {
      errexit(errcode,"callQs -- pthread_join");
    }
  }
  free(ll);
} // end of callQs
```

To perform the first partition with two threads we have to add code:

```
void callQs2( << here init arguments >> ) {
```

```
// Check first whether the array size warrants parallel processing.
// If not call a sequential version and get out.
/*
Preprocess to obtain a pivot T in the segment [N,M], arrange the
corners and the middle position e3 so that:

    |-----------------------|-----------------------|
    N                       e3                      M
    A[N] < T             A[e3] = T                T<=A[M]
{Bypass doing the first partition in parallel if this configuration
cannot be obtained.}
Use two threads to partition the left and right side to obtain:
           A            B          C             D
    |------------|-----------|------------|-----------|
    N            i1          e3           i2          M
    A[N] < T             A[e3] = T                T<=A[M]
where the elements in A and C are less than T and the elements in B
and D are greater or equal than T.
Swap the B and C segments.
Add the A+C and B+D segments to the task stack.
Create and launch the specified number of threads as above.
*/
} // end of callQs2
```

Appendix Optimistic Complexity Analysis

We give an optimistic complexity analysis for 1, 2 and 3 pivot quicksort type sorting algorithms. This is *not* a worst case analysis but a best case analysis, which yields at most conjectures.

We make the following simplifying assumptions:
- We have a zero cost oracle that provides perfect pivots, which yield during partitioning equally sized sub-segments
- We invoke the sorting algorithms all the way down (hence avoiding the improvement of using insertion sort on small segments because we have perfect pivots)
- We assume that elements to be moved do *not* require subsequent re-relocations (as is the case for 1-pivot partitioning)

An array with uniform distribution of integers with a cheap way to construct pivots approximates these assumptions.

We define:
A as the time to access an array element
S as the time to store an array element
C as the time to compare two elements
T2(N) is the optimal time to sort an array of size *N* with a single pivot
T3(N) is the optimal time to sort an array of size *N* with two pivots
T4(N) is the optimal time to sort an array of size *N* with three pivots

log2, *log3* and *log4* are respectively the logarithm functions for base 2, 3 and 4.

Partitioning an array of size *N* into two sub-segments takes the time:
*N * (A + S/2 + C)*
because we must: access each element, store half of them (assuming a perfect pivot), and compare each element against the pivot.
We must do the same for the two equal sub-segments, and since we have a uniform distribution and a perfect pivot the time for the two segments is:
*2 * (N/2) * (A + S/2 + C) =*
*N * (A + S/2 + C)*
We must repeat partitioning *log2(N)* times. Thus the total time for partitioning with one pivot is:
*T2(N) = N*log2(N)*(A + S/2 + C)*

Partitioning an array of size *N* into three sub-segments takes the time:
*N * (A + S*2/3 + C*5/3)*
because we must: access each element, store 2/3 of them (assuming a perfect pivot), and compare each element only once against one pivot with chance 1/3, and with chance 2/3 to check against both pivots; thus:
*1 * (1/3) + 2 * (2/3)* gives the coefficient 5/3.
We must do the same for the three equal sub-segments, and since we have a uniform distribution and perfect pivots the time for the three segments is:
*3 * (N/3) * (A + S*2/3 + C*5/3) =*
*N * (A + S*2/3 + C*5/3)*
We must repeat partitioning *log3(N)* times. Thus the total time for partitioning with two pivots is:
*T3(N) = N*log3(N)*(A + S*2/3 + C*5/3)*
given: *log3(N) = log2(N)*0.6310* we get:*T3(N) = N*log*
*T3(N) = N*log2(N)*(A*0.6310 + S*0.4207 + C*1.051)*

Partitioning an array of size *N* into four sub-segments takes the time:
*N * (A + S*3/4 + C*2*
because we must: access each element, store 3/4 of them (assuming a perfect pivot), compare each element twice, once against the middle pivot and once against the left or right pivot.
We must do the same for the four equal sub-segments, and since we have a uniform distribution and perfect pivots the time for the four segments is:
*4 * (N/4) * (A + S*3/4 + C*2) =*
*N * (A + S*3/4 + C*2)*
We must repeat partitioning *log4(N)* times. Thus the total time for partitioning with three pivots is:
*T4(N) = N*log4(N)*(A + S*3/4 + C*2)*
given: *log4(N) = log2(N)*0.5* we get:
*T4(N) = N*log2(N)*(A*0.5 + S*0.375 + C)*

Lining up T2, T3 and T4 gives:
*T2(N) = N*log2(N)*(A + S/2 + C)*
*T3(N) = N*log2(N)*(A*0.6310 + S*0.4207 + C*1.051)*
*T4(N) = N*log2(N)*(A*0.5 + S*0.375 + C)*

This analysis *suggests*, but does *not* prove, that T4 with 3-pivots has the sweet spot. The next one could be T8 with 7 pivots, but it is unclear how to avoid re-relocations of elements to be moved with that many pivots.