File: c:/bsd/rigel/sort/Papers/Boosting.docx
2018 February

Boosting 1, 2 & 3 Pivot Quicksort through Hybridization and Parallelization

Dennis de Champeaux / OntoOO
temp AT OntoOO DOT com
2018 February

Abstract

Quicksort's quadratic worst case is a thing of the past through hybridization with heapsort. One-pivot quicksort benefits from further hybridization by using slow *and* fast 'collapsing the walls' loops, which respectively focus on minimizing comparisons using index checks versus maximizing speed by relying on sentinels that block running into an (initialized) segment. Recognizing 'troublesome' distributions and forwarding them to a Dutch Flag algorithm member that puts elements equal to the pivot in the middle leads to further hybridization. We show a favorable 0.86 timing ratio against the best one-pivot competitor we have identified on uniform distributions and a 0.90 timing ratio on a family of 'troublesome' distributions. Our two-pivot version has an additional algorithm member, which compares favorable against the only competitor identified. The comparison count of our two-pivot hybrid, while better than the competitor, is still unfavorable against our one-pivot hybrid. This result is consistent with our optimistic, best case complexity analysis. Our three-pivot hybrid replaces the two-pivot member with one that partitions a segment in four regions. This one compares also favorable, 0.81 timing ratio, against the only competitor we have identified. A fourth topic is a design pattern using the C *pthread* package with which we parallelized our novel hybrids. Favorable speed ups are reported for the number of threads in the range 2-7. Replacing pairwise swapping by the single element moving technique, using finite state machine control logic in some of our algorithm members, and using two-gap array layouts have been key design decisions for the achieved speedups on the varying distributions. Timing tests on three generation machines with different caching architectures showed only minimal differences.

Introduction

Hoare published in-place quicksort in 1962 [Hoare]. A high level recursive description of a quicksort version is:

- If the input array segment has length zero or one then done.
- Find an array element (the pivot) that can be used to separate the array elements in two non-empty subsets.
- Find the separator index that is returned by a partitioning operation that shuffles the array elements of the segment around such that, at the left side the elements satisfy, say, property *P* and at the right side the elements satisfy the property *not P*.
- Recurs on respectively the left and the right sub segments to complete the algorithm.

Quicksort had for decades a stained reputation. While being fast, the worst case behavior could be quadratic. We had to wait until 1997 when D. Musser created a fix that guarantees O(nlogn) performance [Musser]: add a recursion depth parameter that is initialized at the top level; the depth parameter is lowered at reach recursive calls; switch to heapsort when the depth parameter reaches zero.

This is an example of hybridization. Way earlier it was recognized that it pays to find a promising pivot, which increases the chance that both partitions have equal size. Avoiding the overhead of this preprocessing led to using insertion sort on small segment, typically less than 16. This became the first hybridization of quicksort.

Switching between algorithm members is cheap in both cases: test the size of a numeric parameter. There are more opportunities to extend hybridization in the one pivot quicksort realm and even more when the number of pivots increases, which is the focus of this paper.

Partitioning with one pivot has designs that minimize the number of comparisons through index checks versus designs that use faster executing loops that use sentinels: initialized segments that 'bounce' a loop running into them. We describe hybrids that combine the two design choices. Inputs with many identical elements can be problematic. The Musser defense will prevent a quadratic blowup but heapsort is slow. We describe a Dutch Flag sorter algorithm member that can take care of 'difficult' distributions with marginal overhead; it initializes the middle region for elements equal to the pivot in the middle of the segment. Proper initialization of a segment's regions allows fast loops and simplifies substantially the design of multi pivot algorithms. The Dutch Flag sorter can be relied on when initialization of segment's regions fail.

A two-pivot algorithm requires more overhead to construct the pivots. Hence, by analogy with the use of insertion sort in the one-pivot case, a design in which a two-pivot member sits on top of a one-pivot hybrid leads to further hybridization. A three-pivot algorithm will similarly sit on top of a hybrid also, but *not* on a two-pivot hybrid as we will discuss. The Dutch Flag sorter can assist as well two- and three-pivot algorithm members.

Our one-, two- and three-pivot hybrids are compared against the best algorithms we have found in each category and against each other. We use mostly uniform distributions for the comparisons but also 'difficult' distributions from the Bentley&McIlroy test bench [Bentley & McIlroy] and some others.

Road map

Since we describe three new algorithms and their parallel versions it helps to outline the structure of the presentation. The next section deals with preliminaries. Subsequently we deal with the one, two and three pivot hybrids FourSort, FiveSort and SixSort. Each of these sections describes competitive algorithms and our hybrid. Subsequently we give the results of numerous comparison tests and their interpretation. A next section presents findings of parallel versions of our hybrids. A lessons learned sections follows the conclusions. Appendices provide disaggregated test data and the C-codes of selected algorithms. The codes of all hybrid members are available on GitHub.

Preliminaries

Algorithms visited

We will visit, among others, the following public versions:

- Qsort (LQ) in the Linux library made by D.C. Schmidt around 1991
- Qsort (B&M) in the Cygwin library (maintained by Red Hat), and other libraries, made by Bentley, J.L. & M.D. McIlroy in 1993 [Bentley & McIlroy]
- Introsort (Intro) in the C++ library described by D. Musser in 1997 [Musser]
- Dual Pivot Quicksort (DPQ) in the Java util library made by Bentley, J.L. & J. Bloch around 2011.
- Multi-Pivot Quicksort (MPQ) an experimental 3-pivot sorter made by Kushagra, S., A. Lopez-Ortiz & J. I. Munro around 2013 [Kushagra].

We contrast in passing designs and experimental comparisons against our own hybrids FourSort (1-pivot), FiveSort (2-pivot) and SixSort (3-pivot).

What to sort?
Developing a new sorting algorithm is best done by using integer arrays to get the control logic correct. The performance of such an implementation can be misleading because the comparison operators are available in the hardware. Hence we use instead 'objects' that require the specification of a comparison function. Throughout our objects contain just an integer field (and a float that is not used).

Type general versus type specific sorting
Type general sorters were developed in the 90-ies (the two qsorts) and ASPS by Chen in 2008 [Chen]. Given that sorting is typically done on records/objects it pays focusing on sorting those only (and thus we require always a comparison function). Still we developed also integer versions for our algorithms; they are available at [Integersorters].

Where to find the codes
All hybrids have been coded in C. The DPQ version was ported from Java. The MPQ version was coded using the description found in [Kushagra].

One pivot algorithms with extensive testing infrastructure are available at:
[FourSort] https://github.com/ddccc/foursort

Two pivot algorithms with similar infrastructure can be found at:
[FiveSort] https://github.com/ddccc/fivesort

Three pivot algorithms and infrastructure are at:
[SixSort] https://github.com/ddccc/sixsort

Testing procedures
Timing tests for a wide range (1K-16M) uses repetitions dependent on the size of the input segments: 32K accumulations on arrays of size 1K, 16K accumulations on arrays of size 2K, … and 2 on arrays of size 16M. Detailed comparisons of different algorithms/ hybrids/ members of hybrids is mostly done on 16M arrays where the average is taken of three repetitions that take the average of five runs with different seeds of the random number generator. Warm ups are standard practice, see the infrastructures on GitHub. The comparisons using the Bentley-McIllroy test bench [Bentley & McIlroy] are done on 1M arrays with warm ups but without

repetitions; the runs of each family in the test bench accumulates the outcome of 105 distributions.

We used multiple platforms to perform comparison tests containing processors from different generations with different caching support, among others:

- Intel Pentium M (P5), 1.73Ghz, WinXP+Cygwin, L1 cache ? L2 cache 2Mb
- Intel I3-2310M (I3), 2.1Ghz, Win7+Cygwin, L1 cache 2x32Kb instruction + 2x32Kb data, L2 cache 2x256Kb, L3 cache 3Mb
- AMD fx-8350 (AM), 4Ghz, Win8.1+Cygwin, L1 cache 4x64Kb instruction + 8x16Kb data, L2 cache 4x2Mb, L3 cache 8Mb

The P5 machine has a single cpu; the I3 machine has two cpus and four threads; the AM machine has four cpus and eight threads. The P5 machine has a 32-bit architecture; the two others are 64-bit machines. The P5 and I3 machines are laptops; the AM machine is a desktop and closer to an industrial strength machine.

One-pivot sorting

This section compares the design and performance of our FourSort against qsort/ B&M from Bentley & McIlroy [Bentley & McIlroy]. First we discuss why using that particular one pivot quicksort. Candidate competitors we found are: ASPS [Chen], Intro, LQ, and Muqaddas [Muqaddas].

ASPS
The ASPS algorithm is atypical. While quicksort works its way down to smaller sub segments until it bottoms out at using insertion sort, ASPS starts with a small sorted segment in the left corner. Iteratively that segment is increased by ‘taking a bite’ from the unsorted segment to the right. The bite is partitioned using the median element of the sorted segment. The two partitions of the bite are moved in the middle of the sorted segment. The two smaller regions, each with a sorted and an unsorted component, are handled recursively. Tests confirm that ASPS is competitive on uniform distributions. Its asymmetric design could, however, be problematic. Whether a Musser type defense is necessary remains unclear. Whether hybridization and/or parallelization is applicable could be explored but is beyond the scope of this study.

Intro
The one-pivot Introsort version is available in C++. The original paper by Musser [Musser] does *not* contain the partitioning component. He just focused on its wrapper with the counter of the recursive depth for the partitioning function. Implementations found on the web of Introsort are not competitive. Insertion sort is best invoked immediately when the input segment is small because the segment is already in the data cache. The implementations found, however, use the old design where insertion sort is calling once at the end on the top level input. Another factor slowing down these implementations is that insertion sort is called on segment sizes up to 16, while 7 has been shown to be more effective. Constructing pivots from only three elements is another problem.

LQ
The qsort version in Linux has similar problems as Intro. Insertion sort is called once at the end, only the median of three elements is used for the construction the pivot, and insertion sort is

called on segments less than 5, which is unusually low. The swap operation has overhead due to a design error. The pivot is stored at a corner in other designs, but not in LQ. Hence every swap operation must check whether the pivot was moved and if so the reference to the pivot must be updated. The Linux LQ algorithm uses a right moving loop that moves elements not less than the pivot to the right and the reverse for the left moving loop. This entails that also elements equal to the pivot are being swapped. A comparison of LQ against qsort/B&M on a 16M array with uniform data yielded 13% more comparisons and 17% more time.

Muqaddas
The array layout of his Triple State QuickSort (TSQ) version is the Dutch Flag with: elements less than the pivot, a gap, elements equal to the pivot, a gap and elements greater than the pivot. The implementation is difficult due to relying on pairwise swapping to move elements around. He reports: "We trade space for performance, at the price of n/2 temporary extra spaces in the worst case." This disqualifies TSQ because quicksort is in-place (with limited additional stack space).

Qsort/ B&M
The array layout of the B&M version addresses upfront the anomaly of inputs with numerous, if not all, identical elements, because these can lead to a quadratic disaster. While the Muqaddas design stores elements equal to the pivot in a middle partition, the B&M version stores them initially in regions at the corners of the segment. The loop moving to the right stores them at the left side and the reverse for the left moving loop. A key design decision in B&M is the minimization of the number of comparisons. This is achieved in both left and right moving loops by checking whether the gap has closed: did the indices cross? This makes these loops more expensive than loops that rely on a sentinel at the other side - through proper initialization -, which, however, come at the penalty of an additional comparison when the sentinel is hit. Moving elements equal to the pivots initially to the corners entail the penalty of secondary moves of these corner regions to the middle. {Numerous libraries contain a broken version of the qsort/B&M; see the appendix Corrected qsort/B&M Code. We used the original version.}

FourSort
The goals for FourSort have been to be the best one pivot implementation regarding timing, the number of comparisons (and avoiding quadratic disasters), and applicable to all kinds of distributions for all input array sizes. Satisfying all these goals at the same time remains challenging. Thus far avoiding quadratic disasters has been achieved with minimal overhead through the Musser defense but at the price of slow execution by heapsort. Finding different balances of minimizing timings and comparisons and relying on the Musser defense is available in FourSort (and the other hybrids) by adjusting parameters.

The FourSort hybrid contains the members:

- *cut2f*, applied to the largest segments with fast loops; it has escapes to *cut2* when a problematic distribution is encountered; it delegates small (2000) segments to *cut2*
- *cut2*, employs a pivot constructor that aims at proper initialization of the corners and has escapes to the *dflgm* member in case of failure; it delegates small (1000) segments to *quicksort0*

- *quicksort0*, a hybrid itself, which deals with smaller segments than *cut2f* and *cut2*, has less initialization overhead and has fast loops. It contains also escapes to *dflgm*; it delegates segments sized less than 120 to a qsort/B&M version, and segments sized less than 7 to *insertion sort*
- *insertion sort* for small segments
- *dflgm*, a Dutch Flag sorter that takes over when the other members encountered 'trouble'
- *heapsort* for when the recursion depth parameter becomes zero

The *quicksort0* member is internally a hybrid also. It uses slow loops like:

```
   while ( I <= J && (r = compareXY(A[I], T)) <= 0 ) { … }
```

and fast loops like:

```
   while ( compareXY(A[++I], T) <= 0 )
```

It uses a B&M like implementation with slow loops that minimizes comparisons on segments up to size120. Large segments are handled by a quicksort sort implementation with fast loops and an asymmetric two segment array layout: greater or equal than the pivot at the left side and greater than the pivot at the right side. The pivot is stored in the left corner. A fast loop runs to the left where the left corner acts like a sentinel. Most likely the fast loop will not hit the sentinel and partitioning can proceed with fast loops on large arrays; hence with minimal penalties for spurious comparisons due to using sentinels. In the unlikely case that the first loop hits the sentinel there is something weird with the input - say all elements are the same. If so, the segment is delegated to the *dflgm* member, which we discuss below. In case the first fast loop stopped immediately at the right corner a slow right moving loop is invoked that checks at each move to prevent running out of the segment. If it does, then again the segment is delegated to *dflgm*. If the right moving loop did not hit the right corner, partitioning can proceed with fast loops.

The member *quicksort0*, in combination with *insertion sort*, *dflgm* and *heapsort*, can be used as a standalone sorter that can compete against FourSort itself. The code of *quicksort0* is available in an appendix and on GitHub.

Using 16M arrays with uniform distributions on the AM machine yielded the following:

quicksort0		qsort/B&M	
comparisons	timing	comparisons	timing
4.25e8	10993	4.17e8	12755

This shows that on theses arrays with uniform distributions *quicksort0* uses 1.9% more comparisons while using 16% less clock ticks than qsort/B&M.

We tested *quicksort0* also against 420 distributions of the Bentley-McIllroy test bench [Bentley & McIlroy]. The detail results for the families sawtooth, rand2, stagger and slopes are in the appendix Test Bench. Aggregating the outcomes we get:

quicksort0		qsort/B&M		
timing	faster	timing	faster	timing ratio
131953	345	145290	75	0.910

This outcome shows nicely that a sorter can be generically faster than another one, but there are (always) distributions with exceptions.

FourSort's *cut2f* member starts with checking whether the recursion depth is too high or whether the segment (size less than 2000) must be delegated to *cut2*. It uses the median of 9 elements to obtain the pivot, like *quicksort0*. It uses, like *quicksort0*, one slow and three fast loops to do the partitioning. In case of encountering a problematic distribution, delegation goes to *cut2* instead of the *dflgm* member. The code of *cut2f* differs from *quicksort0* in that escapes to *dflgm* are replaced by delegations to *cut2*; smaller segments, less than 2000, are delegated to *cut2* instead of to qsort/B&M. The code of *cut2f* is in an appendix.

The *cut2* member starts like *cut2f* but uses a different, more costly, procedure to obtain the pivot. It picks 5 elements from the segment and sorts them. The 3rd element becomes the pivot and will become the left corner provided it is smaller than the 5th element. If so the 5th element will be swapped in the right corner and will be the sentinel for a right moving loop. Otherwise the distribution is considered 'difficult' and *dflgm* is tasked to handle it. The code of *cut2* is in an appendix and on GitHub.

Using 16M arrays, similarly as for *quicksort0,* produced for FourSort:

FourSort		qsort/B&M	
comparisons	timing	comparisons	timing
4.25e8	11039	4.17e8	12595

This entails that on theses arrays with uniform distributions FourSort uses 1.9% more comparisons while using 14% less clock ticks than qsort/B&M.

We tested FourSort as well on the same 420 distributions of the Bentley-McIllroy test bench [Bentley & McIlroy]. The details for the families sawtooth, rand2, stagger and slopes are in the appendix Test Bench. Aggregating the outcomes we get:

FourSort		qsort/B&M		
timing	faster	timing	faster	timing ratio
132274	353	146815	67	0.903

FourSort and its member *quicksort0* are comparably faster on both types of distributions than qsort/B&M, with the difference larger for 'difficult' distributions. FourSort is somewhat slower than *quicksorts0* on uniform distributions and marginally faster on the 'difficult' distributions. Both of them depend on the member *dflgm* for their performance on 'difficult' distributions, which we describe in the next section.

Finding better pivots has been proposed by Kurosawa [Kurosawa] by taking the median of *3^k* elements, with *k* being a function of the segment to be partitioned. Comparing quicksort + insertion sort against this combination where quicksort was modified with Kurosawa's proposal was not an improvement. The modified version required 13% more comparisons and 14% more time.

Dealing with arrays that have 'difficult' distributions
The *quicksort0*, *cut2f* and *cut2* members delegate 'difficult' input to the Dutch Flag *dflgm* member. The design of this member (and of *cut4* in SixSort) is atypical and is likely the most innovative contribution. The array layout of *dflgm* has three regions with initially two gaps: elements less than the pivot, a gap, elements equal to the pivot, a gap and elements larger than the pivot. This layout competes against an asymmetric layout (with an easier implementation) with only one gap: elements less than the pivot, elements equal to the pivot, a gap and elements greater than the pivot. The latter design requires always extraneous swapping of elements into the second region to accommodate elements that must be added to the left region.

Our design with initially two gaps can be achieved with a finite state machine where the states are the labels and the transitions the goto's. Labels in the code have a pictorial depiction of the status of the gaps and an annotation describing which region needs to receive the element in the 'roving' variable *x*. When, for example, two gaps are open and an element in the variable *x* must be added to the left segment we have a state with a label, here *L0*, with the annotation and the code fragment in which *p3* is the pivot:

```
L0:
   /*
      |---)-----(----)-------(----|
      N   i     lw  up       j    M
      x -> L
   */
  if ( lw < i ) { i--;
    goto L1L;
  }
  // i <= lw
  y = A[i];
  r = compareXY(y, p3);
  if ( r < 0 ) { i++;
    goto L0;
  }
  // 0 <= r
  A[i++] = x; x = y;
  if ( 0 < r ) {
    goto R0;
  }
  // r = 0
  // goto ML0; // fall through
```

{The annotation helps not only the development of the code but can be used as well for specifying the formal loop invariant that transitions to *L0* must satisfy.}

Relocating array elements is facilitated by replacing pairwise swapping with the 'single element moving technique', which was originally described already by Hoare but was reinvented by us.

The iterative process of this technique starts by copying the left corner element in the 'roving' variable *x* and replacing that corner temporarily with a copy of the pivot in the middle of the segment. The process continues until both gaps are closed. At that time, there is still something in the roving variable *x* that needs to be inserted in the array. Some swapping involving the left corner will provide the location where the last roving variable element should be inserted. The single element moving technique process iterates as follows (see also the code above):

- The element in the roving variable *x* 'knows' in which active site (*i, lw, up, j*) it must be added.
- An element in the gap adjacent to that active site that does not belong there is lifted out and put in *y*, the element in the roving variable *x* is dropped in and the roving variable *x* gets the lifted out element *y*.
- The lifted out element 'is told' where it should go and the process repeats.

When both gaps are closed, *dlgm* invokes the member that did the delegation on the left and right regions. The code of *dflgm* is available at [FourSort], [FiveSort], and [SixSort].

The finite state machine design for array layouts with two gaps is applied similarly for members in FiveSort and SixSort that employ respectively two and three pivots. Reducing the need for re-relocation of already processed array elements is each time the goal, which is taken for granted in typical, one-pivot designs.

Uniform distributions do not trigger the invocation of *dflgm*. We have shown its contributions above on the test bench with 'difficult' distributions. The next simple tests have distributions with constant data and with a small, varying amount of noise; the #-column has the number of comparisons, the T-column the number of clock ticks on the AM machine with 16M arrays:

Percentage	*quicksort0*		FourSort		qsort/B&M	
of noise:	#	T	#	T	#	T
0	1.68e7	99	1.68e7	103	1.68e7	154
1	2.28e7	308	2.28e7	311	2.27e7	353
2	2.61e7	397	2.62e7	398	2.60e7	462
4	3.31e7	628	3.34e7	626	3.27e7	690
8	4.72e7	1050	4.78e7	1066	4.70e7	1149
16	8.28e7	2002	7.86e7	1954	7.63e7	2119

This table shows again that designs that perform similarly on uniform data can differ on 'abnormal' distributions. The difference between *quicksort0* and FourSort is marginal except when the percentage of noise is 16. The difference between qsort/B&M and the two others is what we have seen before: it uses fewer comparisons, but it takes more time to get the job done. The *dflgm* member puts elements identical to the pivot immediately in the middle, while qsort/B&M has to swap them to the middle after partitioning is done.

The next test shows how *dflgm* in FourSort handles arrays where the number of different elements is more than one but still limited, with again an array size N of 16M.

# distinct entries	# comparisons	timing
2	2.51e7	520
4	4.89e7	1366
8	6.71e7	2507
16	8.77e7	3545
32	1.047e8	4363
64	1.234e8	5231
128	1.426e8	6065

The number of comparisons for an array with 2^k distinct entries in this list is approximated by the formula N*(1+k*1.06). This formula is a special case of a result reported in [Wegner]: A multi-set of N elements with n<<N distinct values can be sorted in time O(Nlogn). Hence, the *dflgm* member provides the Wegner result to FourSort (and FiveSort and SixSort) for 'free'.

The contribution of *dflgm* is shown even better by enabling and disabling *dflgm* in *quicksort0*. Testing on a 16M array with constant data produces:

quicksort0 with *dflgm*		*quicksort0* without *dflgm*	
comparisons	timing	comparisons	timing
1.67e7	307	7.21e8	12512

Quicksort0 without *dflgm* escapes a quadratic disaster by virtue of the *heapsort* defense, but the performance price for not using *dflgm* is substantial.

Timing ratios with different array sizes and on three machines
We report here timing ratios of quicksort0/ B&M and for FourSort/ B&M with uniform distributions where the array size ranges from 1K to 16M. The ratios decrease gradually and thus we report only three data points:
quicksort0/ B&M:

Machine/ array size	1K	64K	16M
P5	1.01	0.97	0.96
I3	0.91	0.86	0.87
AM	0.96	0.89	0.86

FourSort/ B&M:

Machine/ array size	1K	64K	16M
P5	1.01	0.97	0.96
I3	0.91	0.85	0.86
AM	0.93	0.88	0.86

The one-pivot case
If qsort/B&M is currently indeed the best in class of the one-pivot quicksort family, then it appears that *quicksort0* and FourSort are improvements according to our timing tests. The latter two are, unlike qsort/B&M, protected through the Musser defense. The FourSort hybrid, by using the *cut2* member inside *cut2f,* is likely better able handling ‘difficult’ distributions than *quicksort0* (and qsort/B&M). Hence we proceed by using FourSort in the comparisons of the two and three pivot tests.

Two-pivot sorting

This section compares the design and performance of our FiveSort against DPQ from Bentley & Bloch. DPQ’s ‘claim to fame’ is the addition to Java’s util package for the native types (int, long, float, short, double, …), although not for objects. We start by describing the port of DPQ to C.

DPQ

We removed, to speed up DPQ, the infrastructure aimed at recognizing whether the array is nearly sorted because we focus on sorting objects/ records. Similarly we removed infrastructure for switching to merge sort. Insertion sort is applied to short arrays up to size 47. The two pivots are obtained by picking 5 elements from the array and sorting them. The 2nd and 4th elements are the two pivots. Partitioning proceeds with an asymmetric array layout: regions of elements less than the small pivot, elements between the two pivots, a gap, and elements greater than the large pivot. This layout entails re-relocation of array elements when a gap element must be squeezed into the left region. The left and right regions are sorted recursively when the gap closes. If the center region is less than 4/7 of the segment, it is sorted recursively with one pivot where it employs the 3rd element of the 5 elements used to obtain the initial two pivots. Otherwise the center region is handled differently: internal pivot values are swapped to the two ends, which produces the layout: elements equal to the left pivot, elements between the two pivots and elements equal to the right pivot. When that is finished the new center region is sorted recursively.

FiveSort

The FiveSort hybrid's *tps* member that does partitioning into three regions can be combined with FourSort or with *quicksort0*. Testing did not yield a significant difference; hence we combined it with FourSort. The *tps* member delegates segments smaller than 3000 to the member *cut2f*. The layout in *tps* resembles a Dutch Flag, like the one of *dflgm*, with two gaps: elements less than the small pivot, a gap, elements between the two pivots, a gap, and elements greater than the large pivot. Obtaining the two pivots is done similarly as in DPQ with 5 elements. The sorted elements are used also for proper initialization of the regions; a failure causes an escape to *dflgm*. A finite state machine design is used while moving elements is done not only with pairwise swaps but also with three way swaps. The code of the start state of the state machine is shown in an appendix. The full code is available at [FiveSort].

We compare these two two-pivot algorithms only against FourSort. Using 16M arrays with uniform distributions on the AM machine yielded the following:

FourSort		FiveSort		DPQ	
comparisons	timing	comparisons	timing	comparisons	timing
4.248e8	11053	4.480e8	10512	5.504e8	12040

FiveSort is 5.1% faster than FourSort but needs 5.5% more comparisons. The data for DPQ speaks for itself. The description of DPQ above has the design choices that hamper its performance. Muqaddas observed also that DPQ can be surpassed easily in [Muqaddas]: "Dual Pivot Quicksort algorithm definitely results in significantly high number of swaps."

Testing FiveSort against FourSort using the test bench and showing more details about the different families gives:

Machine AM, array size 1M:

Version:	FiveSort		FourSort		timing ratio
Family	faster	timing	faster	timing	
Sawtooth	92	25698	13	28863	0.890

Rand2	61	28774	44	28541	1.008
Stagger	64	38032	41	37993	1.001
Slopes	61	31711	44	29994	1.057
Total	278	124215	142	125391	

The timing ratio of FiveSort/FourSort on the aggregates is just 1.0.

Timing ratios with different array sizes and on three machines
We report here timing ratios of FiveSort/FourSort with uniform distributions where the array size ranges from 1K to 16M. The ratios change little and thus we report only three data points:

Machine/ array size	1K	64K	16M
P5	1.00	0.99	0.91
I3	0.99	1.01	0.97
AM	1.00	0.99	0.95

The two-pivot case
Two-pivot research (re)started with a research note by V.Yaroslavskiy (2009). It had strong claims about superior performance that would surpass the B&M performance – without evidence. Somehow his ideas convinced J. Bentley and J. Bloch (the latter a Java expert) to implement DPQ in Java and Bloch managed to add it to the Java util package, but only for Java's native types (which allow to exploit the comparison functions in the hardware). An author (not identified here) claimed in a paper that DPQ uses fewer comparisons than single pivot quicksort, but agreed in a next paper that it needs more comparisons. Muqaddas wrote carefully [Muqaddas]: "Further research would be of interest to try reducing the high number of swaps in Dual Pivot Quicksort".

The performance of our own FiveSort casts doubts on the two-pivot venture. The appendix Optimistic Complexity Analysis uses the following strong assumptions:
- We have a zero cost oracle that provides perfect pivots, which yield during partitioning equally sized sub-regions
- 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 a uniform distribution and an algorithm with a cheap way to construct pivots approximate these assumptions.

The comparison ratios (with these assumptions) between ideal two- and one-pivot algorithms are that two-pivot algorithms need:

- 0.63 time to access array elements,
- 0.42 time to store array elements, but
- 1.051 times to compare two elements.

It looks like that our analysis is close enough to explain why FiveSort uses 5.5% more comparisons in the test that is close to the assumptions, see above, and why its advantages based on less array access operations decreases. It explains also why DPQ was not implemented for objects. It violates explicitly our assumption that array elements do not need to re-relocations.

Whether the observations of our optimistic complexity analysis can be re-derived with weaker assumptions, we must leave to others.

The implementation of *tps*'s Dutch Flag is hard. Our first 1985 version is lost. A next 2009 version is sized 19Kb. We have now the distinct benefit of using the escape to *dflgm* when initialization of the segment regions fail, which simplifies the coding and we have now a 13Kb code size. Still numerous design options are available and we tried several, among which a single gap one. An alternative employs the single element moving technique with two gaps, with a 12Kb size. Here its data for the FiveSort/ FourSort ratio using that version on three machines with uniform data on three array sizes:

Machine/ array size	1K	64K	16M
P5	1.00	1.00	0.93
I3	1.00	1.02	1.00
AM	1.00	0.99	0.95

The ratios are slightly worse than obtained with the version having a larger footprint and less 'pleasant' code. More important, however, is that all versions we tried were not a significant improvement over our FourSort.

Three-pivot sorting

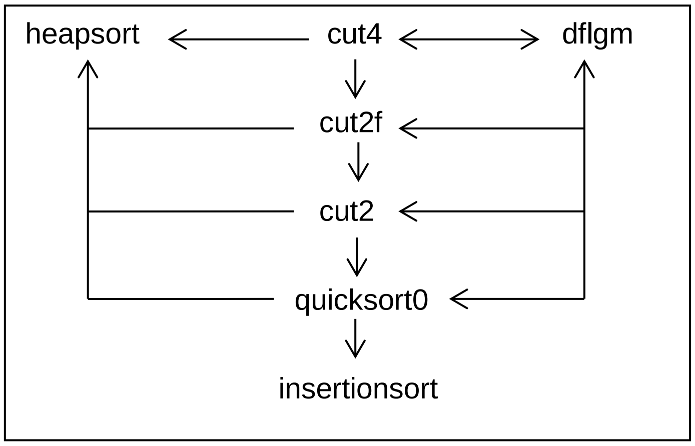
This section compares the design and performance of our SixSort against MPQ from Kushagra et al. [Kushagra] and against FourSort. MPQ is the only three-pivot algorithm that we have found. It was implemented only for integers, we believe. We start by describing the port of MPQ, as described in [Kushagra], to C.

MPQ

The publication [Kushagra] has only pseudo code for the partitioning part. We equipped the algorithm with invoking insertion sort on segments sized less than 13. Pivots were constructed by sorting seven elements and extracting the three pivots from the 2nd, 4th and 6th element. The array layout is symmetric with: elements less than the small pivot, elements greater than the small pivot and smaller than the middle pivot, a gap, with a similar arrangement at the right side for the large pivot. There are two loops that each nibble at the gap to find elements that belong at the 'own' side but may have to be squeezed in through a re-relocation or need to be transported to the other side. When both loops stop they use pairwise swaps or triple swaps to steer the elements to be exchanged into the proper region. We added ordering of the recursive calls and used tail iteration for the 4th call.

SixSort

The SixSort hybrid is a 'clone' of FiveSort. The *tps* member in FiveSort that uses two pivots is replaced by the member *cut4*, which uses three pivots to partition a segment into four regions. The control structure between the SixSort members is as follows:



SixSort starts by checking whether the recursion depth is exceeded. Subsequently it checks whether the segment size is less than 3000, which warrants delegation to *cut2f*. Obtaining the three pivots justifies using more resources. A mini array of minimal size 20 is created in the middle of the segment by swapping equally space elements from the segment. The mini array is sorted and the three pivots are extracted and stored in the corners: at left the middle pivot and the left pivot, at right the right one. Initializing the four regions and escaping to *dflgm* in case of failure has shrunk the code footprint to 57% of the original, which had to take care of all contingencies. Here the start state with the similar *x* and *y* variable usage as done for *dflgm* when deciding that the *x*-element should be added to the *L*-region:

```
TLgMLMRgRL:
       /*   L                               R
          |o----]---------+[-|-]+--------[-----|
          N     i        lw     up        j    M
           x -> L              z
        */
       i++;
       y = A[i];
       if ( compareXY(y, middle) <= 0 ) {
         if ( compareXY(y, maxl) <= 0 ) goto TLgMLMRgRL;
         // y -> ML
         if ( lw < i ) {
        /*   L                              R
          |o----][-----------|-]+--------[-----|
          N      i           z  up        j    M
           x -> L
        */
         goto TLgMLMRgRML;
       }
       // middle < y
           A[i] = x;
          x = y;
           goto TLMLMRgRL;
         }
         A[i] = x;
         x = y;
       if ( compareXY(minr, y) <= 0 ) {
         goto TLgMLMRgRR;
       }
       goto TLgMLMRgRMR;
```

Helpful has been the encoding of state semantics in the label names: the occurrence of a "g" indicates that the gap is present/not closed. The last part L, ML, MR, or R describes which region the element in *x* should go to; indicated also with the arrow annotation in the comment.

We compared the two three-pivot algorithms also against FourSort. Using 16M arrays with uniform distributions on the AM machine yielded the following:

FourSort		SixSort		MPQ	
comparisons	timing	comparisons	timing	comparisons	timing
4.248e8	11053	4.152e8	9630	4.552e8	11854

SixSort is 14.8% faster than FourSort and needs 2.3% less comparisons. The data for MPQ speaks for itself; it cannot even compete against one pivot FourSort. The single gap layout causing re-relocations, among other design choices, hampers its performance.

Testing SixSort against FourSort using the test bench and showing more details about the different families gives:

Machine AM, array size 1M:

Version:	SixSort		FourSort		timing ratio
Family	faster	timing	faster	timing	
Sawtooth	103	20875	2	29254	0.713
Rand2	103	26085	2	28889	0.903
Stagger	103	35398	2	38482	0.920
Slopes	91	28472	14	29889	0.952
Total	400	110830	20	126514	

The timing ratio of SixSort on the aggregates is 0.876

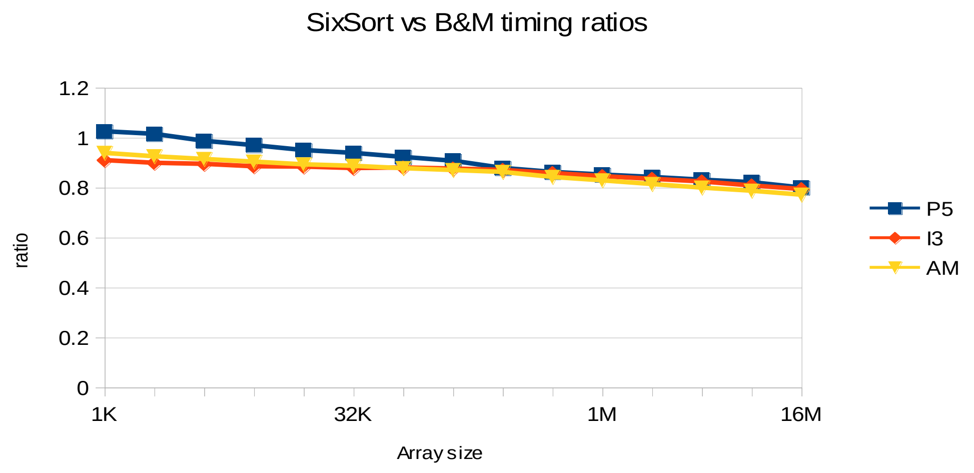
Timing ratios with different array sizes and on three machines
We report here timing ratios of SixSort/FourSort with uniform distributions where the array size ranges from 1K to 16M. The ratios change little and thus we report only three data points:

Machine/ array size	1K	64K	16M
P5	1.00	0.95	0.83
I3	1.00	1.02	0.93
AM	1.00	0.99	0.86

The three-pivot case
The 14% speed improvement of SixSort with three pivots over FourSort may still leave room for improvements; say with a design using 2-way, 3-way and 4-way swaps.

Using more than 3 pivots may not be feasible. Our optimistic analysis shows that multiple pivots reduce array operations but under the assumption that re-relocations are not necessary. This is approximated in SixSort with its two gaps. A next opportunity would take seven pivots for partitioning into eight regions with four gaps to minimize re-relocations. Gaps, however, do not close at the same time. The SixSort code must deal with two different scenarios. Code with four gaps must handle 24 different ways that gaps close. This would make the code footprint, likely, too large.

The next graph shows the SixSort/ B&M timing ratios for arrays ranging from 1K to 16M on P5, I3 and AM.



The commonality of the three graphs on machines with quite different caching architectures is noteworthy. Here the differences for three array sizes:

Array size/machine	1K	64K	16M
P5	1.024	0.922	0.800
I3	0.909	0.880	0.793
AM	0.938	0.877	0.771

{Only the *quicksort0* member is active on the 1K arrays.}

Parallel versions

Parallelization introduces yet another perspective on multi pivot quicksort. Quicksort type algorithms can be parallelized easily on shared memory machines with a limited number of processors. A simple approach, with 1-pivot FourSort, is doing the first partition sequentially and using two threads for subsequent processing of the two obtained partitions. Alternatively the two partitions are put on a stack and the two threads are tasked to partition segments popped from the stack. When a segment is partitioned the largest one is pushed on the stack and the smaller one is handled through tail recursion.

The initial segment can be partitioned in a parallel as well by splitting it up in two equal sides, partition the two sides (using a single common pivot) with two threads, interchange the two obtained inner sub segments and proceed as above with the resulting two partitions. These tactics can be applied, with adjustments, for FiveSort and SixSort.

A design pattern (exploiting the *pthread* package) has been used to implement these narratives. Key is the separation of the semantics of predefined threads and the semantics of applications, like our quicksorts. The appendix Parallel Design Pattern provides details of the design pattern.

Tests on AM with arrays ranging from 1M, 2M, … to 16M and reporting only for the corners of the array sizes the ratios against sequential FourSort yielded:

# threads	PFourSort	PFiveSort	PSixSort
2	0.56-0.55	0.57-0.54	0.54-0.50
3	0.43-0.41	0.44-0.40	0.43-0.37
4	0.37-0.34	0.37-0.34	0.36-0.31
5	0.34-0.30	0.34-0.29	0.32-0.27
6	0.31-0.27	0.31-0.27	0.30-0.25
7	0.31-0.26	0.32-0.25	0.31-0.24

Looking closer at these ratios:

- PFiveSort is virtually identical to PFourSort and can be ignored.
- PSixSort is marginally better than PFourSort on larger arrays.
- All of them underperform when the number of threads is more than 2. For example, only PSixSort obtains a ratio close to 0.33 on 16M arrays with three threads.
- Still, the ratios show useful speed ups.

We tested separately the impact of doing the first partitioning in parallel or not (only using two threads in our test). Doing the first partitioning also in parallel yielded a 5-7% timing advantage.

The parallel case
Hybridization (with substantial effort) of sequential quicksort has produced with three pivots up to 23% speedups against qsort/B&M. Parallelization, clearly, has yielded greater yields.

We addressed caching concerns for the sequential case indirectly by testing on three generations hardware having drastically varying caching architectures. We observed differences but not excessive ones. The situation is more intricate when multiple threads are active. Parallel FourSort with two threads increases the number of active array locations to four, and to eight for SixSort. This suggests that FourSort is a better candidate when the threads count increases beyond which SixSort does not improve due to data cache congestion.

An alternative approach uses many threads (32) to do partitioning in parallel *on special hardware* [Tsigas & Zhang]. Simplified, each thread partitions pairs of small blocks from different sides of the array, where each pair fit in one of the multiple data caches of that hardware. Blocks may have to be swapped and a bit of serial processing may be required to obtain a first partitioning. Recursive processing proceeds similarly.

Hardware support for massive parallel quicksort appears required.

Summary
We have described opportunities to improve quicksort type algorithms in terms of speed and quality: avoiding quadratic deterioration on 'difficult' inputs. We showed the advantage of:

- surrendering type generality
- using three pivots instead of a one or two for sequential versions

- going beyond hybrids with two members by using hybrids with up to seven member algorithms (actually eight members because one of them is a two member hybrid itself)
- the parallel version of one-pivot FourSort and of three-pivot SixSort

We showed specifically the advantages of FourSort and SixSort against qsort/B&M (in the Cygwin and other libraries). In addition, we compared FourSort and SixSort against ports into C of Java's 2-pivot DPQ and ports into C of 3-pivot MPQ [Kushagra].

Users of these sorting functions will see performance improvement due to reduced movements of array elements, with better handling of non-uniform distributions and with no more worries about quadratic performance degradation.

While SixSort is faster than FourSort on the machines we tested, its footprint is larger. Future hardware architectures may impact their relative performance.

Our parallel versions look also promising candidates to complement/ replace current sequential ones.

Lessons learned

- Getting a quicksort algorithm correct is best done by using integer arrays
- Performance comparisons on integer arrays can differ from those using referenced elements because the comparison operations are available in the hardware
- Multi-pivot's key advantage over single pivot quicksort is reduced array access – according to our optimistic complexity analysis. This entails that the advantage of 3-pivot SixSort over 1-pivot FourSort decreases when the relative cost of the comparison operation increases
- Ascertaining that partition regions are non-empty before partitioning starts, allows faster loops and reduces substantially the amount of code for multi-pivot versions. If proper initialization cannot be established the array gets delegated to a Dutch Flag type member, which can handle 'difficult' inputs – as explained in this paper
- Finite state machine designs in algorithm members of FourSort, FiveSort and SixSort rely for their effective implementation on goto-s; hence they cannot be ported as-is to Java
- The single element moving technique for moving array elements around (instead of pairwise swapping) has been key for effective implementations that realize the opportunities for speedups
- Hybridization with up to eight member algorithms, each with a specific 'mission' and with cheap ways to activate them, is part of the strength of FourSort, FiveSort and SixSort
- The performance of our two pivot FiveSort, sequential and parallel, supports the conjecture, obtained from our optimistic complexity analysis, that two pivot quicksorts are not competitive
- Parallel versions of quicksort on shared memory machines can run into data cache limits when the number of threads increase

References

[Bentley & McIlroy] Bentley, J.L. & M.D. McIlroy, "Engineering a Sort Function", *Software-Practice and Experience*, vol 23, no 11, pp 1249-1265, 1993 November; available at: http://www.cs.fit.edu/~pkc/classes/writing/samples/bentley93engineering.pdf
and at:
http://www.enseignement.polytechnique.fr/informatique/profs/Luc.Maranget/421/09/bentley93engineering.pdf

[Chen] Chen, J-C, "Symmetry Partition Sort", *Software-Practice and Experience*, vol 38, no 7, pp 761-773, 2008 June.

[DPQ] Dual Pivot Quicksort source is at:
http://www.docjar.com/html/api/java/util/DualPivotQuicksort.java.html
And older version of DPQ is described in Vladimir Yaroslavskiy, “Dual-Pivot Quicksort”, 2009.
http://iaroslavski.narod.ru/quicksort/DualPivotQuicksort.pdf

[FiveSort] https://github.com/ddccc/fivesort

[FourSort] https://github.com/ddccc/foursort

[GitHub2014] https://github.com/ddccc

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

[Integersorters] https://github.com/ddccc/Integersorters

[Kurosawa] Kurosawa, N., “Quicksort with median of medians is considered practical”, https://arxiv.org/abs/1608.04852

[Kushagra] 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

[Muqaddas] Muqaddas, A, “Tripple State Quicksort”, https://arxiv.org/abs/1505.00558

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

[SixSort] https://github.com/ddccc/sixsort

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.

Appendix Corrected qsort/B&M code
The code below has the fixed and modified partitioning component of the qsort procedure originally described in [Bentley & McIlroy]; we omitted the prefix of the code with auxiliary infrastructure. The fix is the commented out removal of the invocation of insertion sort. The modification is ordering the recursive calls and replacing the last one with tail iteration. {This code is somewhat obtuse because of its type independence; the *es* parameter indicates the size in bytes of elements in the array.}

```
void bentley(a, n, es, cmp) void *a; size_t n; size_t es; int (*cmp)();
{
  char *pa, *pb, *pc, *pd, *pl, *pm, *pn;
  int d, r, swaptype;
  // The use of swap_cnt and the 2nd invocation of insertionsort has been removed
  // int swap_cnt;
loop:  SWAPINIT(a, es);
  // swap_cnt = 0;
       if (n < 7) {
              for (pm = (char *) a + es; pm < (char *) a + n * es; pm += es)
                     for (pl = pm; pl > (char *) a && cmp(pl - es, pl) > 0;
                          pl -= es)
                            swap(pl, pl - es);
              return;
       }
       pm = (char *) a + (n / 2) * es;
       if (n > 7) {
              pl = a;
              pn = (char *) a + (n - 1) * es;
              if (n > 40) {
                     d = (n / 8) * es;
                     pl = med33(pl, pl + d, pl + 2 * d, cmp);
                     pm = med33(pm - d, pm, pm + d, cmp);
                     pn = med33(pn - 2 * d, pn - d, pn, cmp);
```

```
            }
            pm = med33(pl, pm, pn, cmp);
    }
    swap(a, pm);
    pa = pb = (char *) a + es;
    // pa = pb = (char *) a;
    pc = pd = (char *) a + (n - 1) * es;
    for (;;) {
            while (pb <= pc && (r = cmp(pb, a)) <= 0) {
                    if (r == 0) {
                             // swap_cnt = 1;
                            swap(pa, pb);
                            pa += es;
                    }
                    pb += es;
            }
            while (pb <= pc && (r = cmp(pc, a)) >= 0) {
                    if (r == 0) {
                             // swap_cnt = 1;
                            swap(pc, pd);
                            pd -= es;
                    }
                    pc -= es;
            }
            if (pb > pc) break;
            swap(pb, pc);
            // swap_cnt = 1;
            pb += es;
            pc -= es;
    }
    /*
    if (swap_cnt == 0) {  // Switch to insertion sort
            for (pm = (char *) a + es; pm < (char *) a + n * es; pm += es)
                    for (pl = pm; pl > (char *) a && cmp(pl - es, pl) > 0;
                         pl -= es)
                            swap(pl, pl - es);
            return;
    }
    */
    pn = (char *) a + n * es;
    r = min(pa - (char *)a, pb - pa);
    vecswap(a, pb - r, r);
    r = min(pd - pc, pn - pd - es);
    vecswap(pb, pn - r, r);
    /* Ordering on the recursive calls has been added to obtain at most
       log2(N) stack space
    if ((r = pb - pa) > es)
            bentley(a, r / es, es, cmp);
    if ((r = pd - pc) > es) {
              // Iterate rather than recurse to save stack space
            a = pn - r;
            n = r / es;
            goto loop;
    }
    */
  int left =  pb - pa;
  int right = pd - pc;
  if ( left <= right ) {
     if ( left > es ) bentley(a, left / es, es, cmp);
     if ( right > es ) {
        // Iterate rather than recurse to save stack space
        a = pn - right;
        n = right / es;
```

```
         goto loop;
      }
   } else {
      if ( right > es ) bentley(pn-right, right / es, es, cmp);
      if ( left > es ) {
         // Iterate rather than recurse to save stack space
         // a = pn - left;
         n = left / es;
         goto loop;
      }
   }
} // end of bentley
```

We used in our B&M tests their version described in their original paper [Bentley & McIlroy]. A web search of their implementation shows numerous versions (in multiple libraries) with all the same modification: an attempt to speed it up using insertion sort. This 'improvement' is actually detrimental as shown by running B&M with the 'improvement' on the Bentley & McIlroy Stagger test-bench:

```
  Size: 1048576 m:  524288 tweak: 0 B&M Time:   1359
  Size: 1048576 m:  524288 tweak: 1 B&M Time:   1328
  Size: 1048576 m:  524288 tweak: 2 B&M Time:   1343
  Size: 1048576 m:  524288 tweak: 3 B&M Time:   1470
  Size: 1048576 m:  524288 tweak: 5 B&M Time:   1408
  Size: 1048576 m: 1048576 tweak: 0 B&M Time: 322797
  Size: 1048576 m: 1048576 tweak: 1 B&M Time: 245797
  Size: 1048576 m: 1048576 tweak: 2 B&M Time:  40688
  Size: 1048576 m: 1048576 tweak: 3 B&M Time: 136016
  Size: 1048576 m: 1048576 tweak: 5 B&M Time: 314220
```

and without the 'improvement':

```
  Size: 1048576 m:  524288 tweak: 0 B&M Time:   1313
  Size: 1048576 m:  524288 tweak: 1 B&M Time:   1327
  Size: 1048576 m:  524288 tweak: 2 B&M Time:   1297
  Size: 1048576 m:  524288 tweak: 3 B&M Time:   1344
  Size: 1048576 m:  524288 tweak: 5 B&M Time:   1375
  Size: 1048576 m: 1048576 tweak: 0 B&M Time:   1235
  Size: 1048576 m: 1048576 tweak: 1 B&M Time:   1249
  Size: 1048576 m: 1048576 tweak: 2 B&M Time:   1265
  Size: 1048576 m: 1048576 tweak: 3 B&M Time:   1235
  Size: 1048576 m: 1048576 tweak: 5 B&M Time:   1219
```

Appendix Quicksort0

The *quicksort0* member is itself a hybrid. Small arrays (<= 120) are handled by a B&M version that minimizes comparisons, while larger arrays are processed with faster loops that rely on sentinels.

```
// calculate the median of 3
int med(void **A, int a, int b, int c,
       int (*compareXY ) (const void *, const void * ) ) {
  return
    compareXY( A[a], A[b] ) < 0 ?
    ( compareXY( A[b], A[c] ) < 0 ? b : compareXY( A[a], A[c] ) < 0 ? c : a)
    : compareXY( A[b], A[c] ) > 0 ? b : compareXY( A[a], A[c] ) > 0 ? c : a;
} // end med

void vswap(void **A, int N, int N3, int eq) {
  void *t;
  while ( 0 < eq ) { eq--; t = A[N]; A[N++] = A[N3]; A[N3++] = t; }
}
```

```
void quicksort0c();
// Quicksort function for invoking quicksort0c.
void quicksort0(void **A, int N, int M, int (*compare)()) {
  //  printf("quicksort0 N %i M %i \n", N, M);
  int L = M - N;
  if ( L <= 0 ) return;
  if ( L < 7 ) {
    insertionsort(A, N, M, compare);
    return;
  }
  int depthLimit = 2.5 * floor(log(L));
  // list(A, N, M);
  quicksort0c(A, N, M, depthLimit, compare);
} // end quicksort0

void dflgm();
void insertionsort();
// Quicksort equipped with a defense against quadratic explosion;
// calling heapsort if depthlimit exhausted
void quicksort0c(void **A, int N, int M, int depthLimit, int (*compareXY)()) {
  // printf("Enter quicksort0c N: %d M: %d %d\n", N, M, depthLimit);
  // printf(" gap %d \n", M-N);
  while ( N < M ) {
    int L = 1+ M - N;
    if ( L < 7 ) {
      insertionsort(A, N, M, compareXY);
      return;
    }
    if ( depthLimit <= 0 ) {
      heapc(A, N, M, compareXY);
      return;
    }
    depthLimit--;

    // 7 <= L
    int p0 = N + (L>>1); // N + L/2;
    if ( 7 < L ) {
      int pn = N;
      int pm = M;
      if ( 40 < L ) {
       int d = (L-2)>>3; // L/8;
       pn = med(A, pn, pn + d, pn + 2 * d, compareXY);
       p0 = med(A, p0 - d, p0, p0 + d, compareXY);
       pm = med(A, pm - 2 * d, pm - d, pm, compareXY);
      }
      p0 = med(A, pn, p0, pm, compareXY);
    }

    /* optional check when inputs have many equal elements
    if ( compareXY(A[N], A[M]) == 0 ) {
      dflgm(A, N, M, p0, quicksort0c, depthLimit, compareXY);
      return;
    } */

    // p0 is index to 'best' pivot ...
    iswap(N, p0, A); // ... and is put in first position

    register void *T = A[N]; // pivot
    register int I, J; // indices
    register void *AI, *AJ; // array values
    // int k;
    int small = 120; //
    if ( L <= small ) {

      I = N+1; J = M;
      int N2 = I, M2 = J, l, r, eql, eqr;
    Left2:
      while ( I <= J && (r = compareXY(A[I], T)) <= 0 ) {
         if ( 0 == r ) { iswap(N2, I, A); N2++; }
         I++;
      }
      while ( I <= J && (r = compareXY(A[J], T)) >= 0 ) {
       if ( 0 == r ) { iswap(M2, J, A); M2--; }
       J--;
      }
      if ( I > J ) goto Skip2;
      iswap(I, J, A);
      I++; J--;
      goto Left2;

  Skip2:
      l = N2-N; r = I-N2;
      eql = ( l < r ? l : r );
      vswap(A, N, I-eql, eql);
      l = J+N-N2;
      if ( 0 < l-N )  {
       // printf("Left recursion N %i l %i\n", N, l);
       quicksort0c(A, N, l, depthLimit, compareXY);
      }
      l = M2-J; r = M-M2;
      eqr = ( l < r ? l : r );
      vswap(A, I, M-eqr+1, eqr);
      // right 'recursion' tail
      N = I + (M-M2);
      if ( N < M ) { continue; }
      return;
    }

    // 1st round of partitioning
       // The left segment has elements <= T
       // The right segment has elements > T
    /*
         |----------]-------------[-----------|
         N   <=T    I             J   >T      M
    */
    J = M+1;
    while ( compareXY(T, A[--J]) < 0 );
    if ( N == J ) { // poor pivot  N < x -> T < A[x], suspect bad input
      int px =  N + (L>>1); // N + L/2;
      iswap(p0, px, A);
      dflgm(A, N, M, px, quicksort0c, depthLimit, compareXY);
      return;
    }
    AJ = A[J]; // A[J] <= T

    // N < J <= M
    I = N+1;
    if (J < M ) {
      while ( compareXY(A[I], T) <= 0 ) I++;
    }
    else { // J = M
      if ( compareXY(T, A[M]) == 0 ) { // bail out
       int px =  N + (L>>1); // N + L/2;
       iswap(p0, px, A);
       dflgm(A, N, M, px, quicksort0c, depthLimit, compareXY);
```

```
       return;
      }
      // define M2?
      while ( I < J && compareXY(A[I], T) <= 0 ) { I++; }
      if ( M == I ) { // all elements are <= T, suspect bad input
       int px =  N + (L>>1); // N + L/2;
       iswap(p0, px, A);
       dflgm(A, N, M, px, quicksort0c, depthLimit, compareXY);
       return;
      }
    }

    if ( I == J ) {
      I++;
      goto Skip;
    }
    if ( I < J ) { // swap
      A[J] = A[I]; A[I] = AJ;
      if ( I+1 == J ) {
       J--; I++;
       goto Skip;
      }
    } else { // J+1 = I }
      goto Skip;
    }
    // fall through

  Left:
    /*
         |----------]-------------[-----------|
         N   <=T    I             J   >T      M
    */
    while ( compareXY(A[++I],  T) <= 0 );
    if ( J < I ) goto Skip;
    AI = A[I];
    while ( compareXY(T, A[--J]) < 0 );
    AJ = A[J];
    if ( I < J ) { // swap
      A[I] = AJ; A[J] = AI;
      goto Left;
    }
 Skip:
    // Tail iteration
    if ( (I - N) < (M - J) ) { // smallest one first
      quicksort0c(A, N, J, depthLimit, compareXY);
      N = I;
    } else {
      quicksort0c(A, I, M, depthLimit, compareXY);
      M = J;
    }
  } // end of while loop
} // end of quicksort0c
```

Appendix Test Bench Details

Machine AM, array size 1M:

Version:	*quicksort0*		qsort/B&M		timing ratio
Family	faster	timing	faster	timing	
Sawtooth	70	31064	35	32970	0.942
Rand2	85	30075	20	32997	0.911

Stagger	98	39827	7	45036	0.884
Slopes	92	30987	13	34287	0.904

Machine AM, array size 1M:

Version:	FourSort		qsort/B&M		timing ratio
Family	faster	timing	faster	timing	
Sawtooth	83	29971	22	32797	0.914
Rand2	79	30222	26	33215	0.913
Stagger	99	40784	6	46255	0.882
Slopes	92	31197	13	34548	0.903

Appendix Cut2f

The *cut2f* code 'tests the water' with the first two loops. When no delegation is required fast loops take over that depend on sentinels.

```
// Cut2fc does 2-partitioning with one pivot.
// Cut2fc invokes cut2c when trouble is encountered.
void cut2fc(void **A, int N, int M, int depthLimit, int (*compareXY)()) {
  int L;
  Start:
  // printf("cut2fc %d %d %d \n", N, M, M-N);
  if ( depthLimit <= 0 ) {
    heapc(A, N, M, compareXY);
    return;
  }
  L = M - N +1;
  if ( L < cut2fLimit ) {
    cut2c(A, N, M, depthLimit, compareXY);
    return;
  }
  depthLimit--;

    int p0 = N + (L>>1); // N + L/2;
    int pn = N;
    int pm = M;
    int d = (L-2)>>3; // L/8;
    pn = med(A, pn, pn + d, pn + 2 * d, compareXY);
    p0 = med(A, p0 - d, p0, p0 + d, compareXY);
    pm = med(A, pm - 2 * d, pm - d, pm, compareXY);
    p0 = med(A, pn, p0, pm, compareXY);
    iswap(N, p0, A); // ... and is put in first position

  register void *T = A[N];  // pivot
  register int I, J; // indices
  register void *AI, *AJ; // array values
  // int k; // tracing
    // 1st round of partitioning
       // The left segment has elements <= T
       // The right segment has elements > T
    /*
         |----------]-------------[-----------|
         N   <=T    I             J   >T      M
    */

    J = M+1;
    while ( compareXY(T, A[--J]) < 0 );
    if ( N == J ) {
      cut2c(A, N, M, depthLimit, compareXY);
      return;
    }
    AJ = A[J]; // A[J] <= T
    I = N+1;
    if (J < M ) {
      while ( compareXY(A[I], T) <= 0 ) I++;
    }
    else { // J = M
      if ( compareXY(T, A[M]) == 0 ) { // bail out
       cut2c(A, N, M, depthLimit, compareXY);
       return;
      }
      while ( I < J && compareXY(A[I], T) <= 0 ) { I++; }
      if ( M == I ) { // all elements are <= T, suspect bad input
       cut2c(A, N, M, depthLimit, compareXY);
       return;
      }
    }
    if ( I < J ) { // swap
      A[J] = A[I]; A[I] = AJ;
      if ( I+1 == J ) { J--; I++; goto Skip; }
      goto Left;
    }
    if (I == J+1 ) goto Skip;
    // I = J
     I++;
     goto Skip;

     // The left segment has elements < T
     // The right segment has elements >= T
     // Proceed with fast loops
  Left:
       while ( compareXY(A[++I], T) <= 0 );
       if ( J < I ) goto Skip;
       AI = A[I];
       while ( compareXY(T, A[--J]) < 0 );
       AJ = A[J];
       if ( I < J ) { // swap
         A[I] = AJ; A[J] = AI;
         goto Left;
       }
 Skip:
       // Tail iteration
       if ( (I - N) < (M - J) ) { // smallest one first
         cut2fc(A, N, J, depthLimit, compareXY);
         N = I;
         goto Start;
       }
       cut2fc(A, I, M, depthLimit, compareXY);
       M = J;
       goto Start;

} // end of cut2fc
// (*  OF cut2; *) the brackets remind that this was once Pascal code
```

Appendix Cut2

The *cut2* code's main difference is using 5 sorted segment elements to obtain the pivot and to decide whether to delegate to *dflgm* or not.

```
// Cut2c does 2-partitioning with one pivot.
// Cut2c invokes dflgm when trouble is encountered.
void cut2c(void **A, int N, int M, int depthLimit, int (*compareXY)()) {
  int L;
```

```
Start:
if ( depthLimit <= 0 ) {
  heapc(A, N, M, compareXY);
  return;
}
L = M - N +1;
if ( L < cut2Limit ) {
  quicksort0c(A, N, M, depthLimit, compareXY);
  return;
}
depthLimit--;
register void *T; // pivot
register int I, J; // indices
register void *AI, *AJ; // array values
// int k;
  int sixth = (L + 1) / 6;
  int e1 = N  + sixth;
  int e5 = M - sixth;
  int e3 = N + (L>>1); // N + L/2;  // the midpoint
  int e4 = e3 + sixth;
  int e2 = e3 - sixth;
  // Sort these elements using a 5-element sorting network
  void *ae1 = A[e1], *ae2 = A[e2], *ae3 = A[e3], *ae4 = A[e4], *ae5 = A[e5];
  void *t;
  // if (ae1 > ae2) { `t = ae1; ae1 = ae2; ae2 = t; }
  if ( 0 < compareXY(ae1, ae2) ) { t = ae1; ae1 = ae2; ae2 = t; } // 1-2
  if ( 0 < compareXY(ae4, ae5) ) { t = ae4; ae4 = ae5; ae5 = t; } // 4-5
  if ( 0 < compareXY(ae1, ae3) ) { t = ae1; ae1 = ae3; ae3 = t; } // 1-3
  if ( 0 < compareXY(ae2, ae3) ) { t = ae2; ae2 = ae3; ae3 = t; } // 2-3
  if ( 0 < compareXY(ae1, ae4) ) { t = ae1; ae1 = ae4; ae4 = t; } // 1-4
  if ( 0 < compareXY(ae3, ae4) ) { t = ae3; ae3 = ae4; ae4 = t; } // 3-4
  if ( 0 < compareXY(ae2, ae5) ) { t = ae2; ae2 = ae5; ae5 = t; } // 2-5
  if ( 0 < compareXY(ae2, ae3) ) { t = ae2; ae2 = ae3; ae3 = t; } // 2-3
  if ( 0 < compareXY(ae4, ae5) ) { t = ae4; ae4 = ae5; ae5 = t; } // 4-5
  // ... and reassign
  A[e1] = ae1; A[e2] = ae2; A[e3] = ae3; A[e4] = ae4; A[e5] = ae5;

  // if ( compareXY(A[M], ae5) < 0 ) iswap(M, e5, A);
  if ( compareXY(ae5, ae3) <= 0) {
    // give up because cannot find a good pivot
    // dflgm is a dutch flag type of algorithm
    void cut2c();
    dflgm(A, N, M, e3, cut2c, depthLimit, compareXY);
    return;
  }
  iswap(e5, M, A); // right corner OK now
  // put pivot in the left corner
  iswap(N, e3, A);
  T = A[N];
  // initialize running indices
  I= N;
  J= M;

     // The left segment has elements < T
     // The right segment has elements >= T
Left:
     while ( compareXY(A[++I], T) <= 0 );
     AI = A[I];
     while ( compareXY(T, A[--J]) < 0 ); AJ = A[J];
     if ( I < J ) { // swap
       A[I] = AJ; A[J] = AI;
       goto Left;
     }
```

```
       // Tail iteration
       if ( (I - N) < (M - J) ) { // smallest one first
         cut2c(A, N, J, depthLimit, compareXY);
         N = I;
         goto Start;
       }
       cut2c(A, I, M, depthLimit, compareXY);
       M = J;
       goto Start;
} // (*  OF cut2; *) the brackets remind that this was once Pascal code
```

Appendix Tps

Here a state of the *tps* state machine:

```
       /*
         |)----------(--)-------------(|
        N i         lw  up           j M
         N < x < i -> A[x] < pl
         lw < x < lup -> pl <= A[x] <= pr
         j < x < M -> pr < A[x]
       */
 again:
       while ( i <= lw ) {
         ai = A[i];
         if ( compareXY(ai, pl) < 0 ) { i++; continue; }
         if ( compareXY(pr, ai) < 0 ) {
           while( compareXY(pr, A[j]) < 0 ) j--;
           aj = A[j]; // aj <= pr
           if ( compareXY(aj, pl) < 0 ) {
             A[i++] = aj; A[j--] = ai;
             if ( j < up ) { j++;
              if ( lw < i ) { i--; goto done; }
              goto rightClosed;
             }
             continue;
           }
           // aj -> M
           if ( j < up ) { // right gap closed
             j++; goto rightClosedAIR;
           } // up <= j
           goto AIRAJM;
         }
         // ai -> M
         goto AIM;
       }
       // left gap closed
       i--;
       goto leftClosed;
```

Appendix Parallel Design Pattern

Our parallel design pattern can be applied to sequential applications that decompose into independent sub tasks. The top level members in FourSort, FiveSort and SixSort that generate partitions to be processed recursively (or with tail iteration) are prime examples. Separating the semantics of the threads and the application is achieved by a protected stack containing tasks. Applications add tasks to the stack and threads take them from there. The code for the stack with its push and pop operations is on GitHub. The member *cut2fp* in parallel FourSort adds tasks to the stack *ll* with:

```
       // Tail iteration
```

```
      if ( (I - N) < (M - J) ) { // smallest one first
        // cut2fc(A, N, J, depthLimit, compareXY);
        // N = I;
        addTaskSynchronized(ll, newTask(A, I, M, depthLimit, compareXY));
        M = J;
        goto Start;
      }
      // cut2fc(A, I, M, depthLimit, compareXY);
      // M = J;
      addTaskSynchronized(ll, newTask(A, N, J, depthLimit, compareXY));
      N = I;
      goto Start;
```

Each thread takes tasks from the stack, or waits, by executing:

```
void *sortThread(void *AAA) { // AAA is not used
  // int taskCnt = 0;
  //  printf("Thread number: %ld #sleepers %d\n",
  //         pthread_self(), sleepingThreads);
  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 ) {
      pthread_mutex_lock( &condition_mutex2 );
      pthread_cond_signal( &condition_cond2 );
      pthread_mutex_unlock( &condition_mutex2 );
      break;
    }
    void **A = getA(t);
    int n = getN(t);
    int m = getM(t);
    int depthLimit = getDL(t);
    int (*compare)() = getXY(t);
    free(t);
    // taskCnt++;
    cut2fpc(A, n, m, depthLimit, compare);
  }
  //  printf("Exit of Thread number: %ld taskCnt: %d\n", pthread_self(), taskCnt);
  return NULL;
} // end sortThread
```

Both parties, the application and the threads, are ignorant about each other's affairs. C's *pthread* package is an acquired taste indeed, but it gets the job done. The termination condition for a thread is not only that the stack is empty; in addition no other threads should be active any more. A terminating thread notifies another waiting one, if any.