File: c:/bsd/rigel/sort/Papers/DesignChoices.odt
2014 June
(C) Dennis de Champeaux/ OntoOO

Design choices in the Quicksort family with empirical comparisons

Dennis de Champeaux / OntoOO
ddcc AT OntoOO DOT com
2014 June

Dedicated to Tulane's CS-Department where this work started in 1984 (and which remains closed after the Katrina hurricane).

Abstract

Many variants of quicksort have been developed during the last 50 years. We describe design choices that allow classifying a few well known members as well new comers of the quicksort family. The pros and cons of the design choices are illustrated with extensive run-time comparisons of the selected variants. Tests include sorting uniform distributions, weird distributions generated by the Bentley-McIlroy test-bench, distributions of equal values with a varying amount of noise and distributions of sorted data with a variant amount of noise. This collection of test shows at least that there is no overall strict ordering when considering only single threaded algorithms. We do find evidence that single pivot algorithms can be surpassed statistically by two pivot algorithms, which in turn can be surpassed statistically by three pivot algorithms. Our new algorithms FourSort and SixSort outperform qsort on records with respectively up to 13% and 23%. Multi-threaded variants outperform single threaded variants. As expected, performance does not scale linearly with the number of threads available to an algorithm. Parallel SixSort with four threads is up to three times faster than single threaded FourSort. Performance variances due to machine cache differences are shown abundantly.

Introduction

Tony 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 ready.
- Find an array element (the pivot) that can be used to hopefully separate the array elements in two equal sized subsets, for example those less or equal than the pivot and those greater or equal than the pivot.
- Find the separator index that is returned by a partitioning operation that shuffles the array elements of the segment around such that elements to the left of the index are less or equal than the pivot value, elements to the right are equal or greater than the pivot value and the pivot is in the separator index.
- Recursion on respectively the left and the right sub segments concludes the algorithm.

An alternative, asymmetric separation condition assumes that two segments will not be empty and the partitioning operation generates a layout, for example, like:

```
|-------------------------][--------------------|
                 <                  >=
```

Either way, one runs the risk that an array of size *N* is partitioned into two non-empty segments respectively of size *1* and of size *N-1*. If this happens at all recursion levels, say because all input elements are equal, then, without taking precautions, the algorithm will run in quadratic time instead of the intended *NlogN* time. Preventing quadratic explosions is now a half century endeavor and this paper is, among others, devoted to this topic.

After developing new variants in the quicksort family we faced the challenge of comparing them against the status quo. It turns out that the existing algorithms have each different strengths and weaknesses. Hence we describe design features for classifying the existing ones. A side benefit is that new algorithms can be characterized with these design choices as well. Unique salient aspects of the selected algorithms are outlined also.

It turns out that the algorithms considered cannot be strictly ordered. Still we provide performance comparisons where warranted. These comparisons show the impact of some of the design decisions in these algorithms. Comparisons using arrays with integers differ substantially from arrays with references to records. We analyze the causes of these differences.

Hardware concerns

Current computer memory comes in different 'flavors': regular memory, second level memory cache, and first level instruction and data level caches. Cache sizes vary, which may cause performance differences on different platforms. We have observed time ratio differences on different platforms. Hence we show results for different machines.

Compilation concerns

Coding choices may impact results as well. One would think that:

```
          if ( condition ) then A else B
```

executes like:

```
          if ( not condition ) then B else A
```

We noticed a case in our code where a 2% timing difference occurred. A comment in the code from another party warned against using the `i++` construct in certain settings. An appendix shows substantial differences between the *gcc* compiler and the Intel *icc* compiler, which produces way faster code. All the results in this paper have been obtained with *gcc*.

Design Choices

This section is devoted to distinct design choices that will be employed to classify selected algorithms in the quicksort family.

Type of array elements

We limit ourselves here to algorithms coded in C. The quicksort version given in [Bentley & McIlroy] pioneered a multi-type version. This requires the quicksort procedure to have an argument that specifies the size of elements in the array in terms of the number of bytes. This allows, for example, to sort integers, doubles, strings of a certain length, records accessed by pointers, etc. Since this design choice comes with a price, an alternative is to have distinct algorithms for different native types and one for data accessed through pointers. The latter choice has been made for the native types in Java; see the DPQ algorithm below.

Algorithm layer(s)

The original quicksort as sketched in the introduction has just a single layer (although with recursive invocations). Typically, quicksort algorithms have at least two layers. The bottom layer consists of insertionsort that takes care of small array segments, while the next layer does some preprocessing before embarking on partitioning and issuing recursive invocations. The overhead of preprocessing warrants the delegation to insertionsort. More than two layers are possible as well when exploiting algorithm components that do different types of partitioning and/or have more elaborate preprocessing actions. Preprocessing a segment can yield the recognition of a special case that can or even should be processed with a dedicated algorithm, which, for example, can be made aware of the originating algorithm so that subsequent recursive invocations can be handled by the originating one again.

Number of pivots

A pivot is an element in an array segment used for partitioning the segment. The quicksort in the introduction employs a single pivot. Recently algorithms have emerged that use two and even three pivots.

Pivot construction

A simple way to construct a single pivot is to take the first element of an array segment. The chance to obtain a pivot that partitions a segment in two equal sizes increases when taking the median of three elements (say left element, middle element, and right element). Larger segments are typically partitioned with a pivot obtained from a median of nine elements. When two pivots are needed, a common technique is to sort five equally spaced elements and use the second and fourth element. Three pivots for a large array segment can be constructed by assembling through swapping a mini array in the middle of the segment, sort it and extract the pivots from the result.

Partitioning variants

A single pivot is typically used for the creation of two partitions. An alternative is the Dutch flag partitioning with three segments having respectively elements less than, equal and larger than the pivot. Two pivots are typically used to handle three segments, although five segments are conceivable also. Three pivots are typically used for managing four segments.

Defense against quadratic explosions

An algorithm that does not come with a *NlogN* worst case complexity guarantee can have as defense spending more preprocessing effort on constructing good pivots. An ultimate defense has been described by Musser [Musser]. It consists of adding a parameter to the quicksort procedure. This parameter gets set initially with a positive integer value that is a function of the initial array size and which denotes the maximum acceptable recursion depth. The parameter value is decreased by one for subsequent recursive invocations. If a recursive invocation encounters a zero parameter value subsequent processing gets delegated to slower heapsort, which guarantees *NlogN* behavior. Musser recommends to set the maximum recursion depth to: *2 * floor(log(initial_array_size))*. Our tests showed that this recommendation needs adjustments. One of our Quicksort implementations switched too often to heapsort on even uniform distributed inputs. We obtained substantial speedups by using instead: *2.5 * floor(log(initial_array_size)).*

Minimizing recursive calls

Quicksort algorithms are considered ‘in-place’, which is true only when ignoring the use of stack space. Prudent versions order recursive invocations by tackling the smallest partition first. In a two partitioning setting this limits stack space to an amount proportional to the 2-base-log of the initial array size. A second (or last) recursive invocation can be avoided by iteration. Some implementations avoid program language provided recursion by managing their own stack.

Moving array elements

Swapping pairs of elements is the standard way of moving array elements into the proper partitions. Three way and four way swaps are possible when more than two partitions are managed. While we have made versions with three way and four way swaps, their performance was often not competitive.

An alternative to swapping was described by Hoare as “single data movement” and was reinvented by us in order to deal with segments having at least three partitions and two gaps. This technique can be applied also to standard quicksort and is easier to explain in that setting. Assume that we have the two partitions Left and Right, a gap between them and an array element stored in a variable X (which was created initially from a ‘hole’ in the Left partition). Testing the X element against the pivot yields the insight that it belongs in Left or Right. If it belongs in Left we exchange an adjacent element to Left in the gap with X and similarly when X needs to be added to Right. The gap narrows after each iteration. The last element in X needs to be added to the proper partition when the gap closes. This is achieved by storing the X element in the hole when it belongs in Left or alternatively a Left-Right boundary element is moved into the hole and the X element is stored in the vacated location. This technique was essential for our SixSort algorithm to keep the coding complexity down.

Timing of insertionsort

There was a time when insertionsort delegation of small segments was *not* done when a small segment was encountered. Instead, such a segment was ignored because insertionsort was run once when all recursive processing was finished. These days insertionsort is usually invoked immediately due to memory caching.

Bias for sorted input

There can be implicit and explicit bias for handling already sorted input. Standard quicksort is an example of an algorithm that has an implicit bias because with a lucky pivot no array elements need to be moved. An algorithm has explicit bias when there is preprocessing infrastructure to recognize and exploit sorted regions in an input segment.

Single thread versus multi thread

Quicksort type algorithms can be parallelized easily. We use the following pattern:

- Extract the arguments for the first call, store them in a task object and add this object to an empty stack
- Create upfront a fixed number of threads, which each execute the same function:
    - If the stack is not empty pop the task object, fetch the arguments and execute the quicksort type function; when done repeat
    - If the stack is empty suspend provided at least one other thread is still active
    - If the stack is empty and there is no active thread then terminate after waking up a sleeping thread, if any
- A quicksort type function that does 1 recursive call and iterates on the second sub-segment is modified so that the largest sub-segment's argument is put in a new task object and pushes it to the stack while this addition produces a notification to the suspended threads, if any; the other segment is executed by the current thread

The termination condition is not that the stack is empty but that all threads are idle. A quicksort type function that produces *N* partitions, adds *N-1* task objects to the stack.

Details of considered algorithms

Quicksort type algorithms used for empirical comparisons are here described in terms of the features given in the previous section as well as with their salient, unique characteristics. To avoid repetitions we use the following defaults:

- Type of array elements: single hardwired type, for example integer, double, pointer, etc.
- Algorithm layers: two consisting of insertionsort and a kind of quicksort
- Number of pivots: one

- Pivot construction: median of three or nine for larger arrays
- Static versus dynamic pivot(s): static, i.e. not changing during a partitioning operation
- Partitioning variants: two partitions
- Defense against quadratic explosion: trust good pivot, not by using the Musser defense
- Minimizing recursive calls by ordering them: not done
- Moving array elements: two way swaps
- Timing of insertionsort: right away when a short segment is encountered
- Explicit bias for sorted input: none
- Single thread versus multithread: single thread.

LQ

We refer with LQ to a quicksort we found in a Linux/glibc GNU library. It was written by Douglas C. Schmidt around 1991 according to a copyright date in the code; the most recent date is 2004. This LQ algorithm (named *_quicksort* in the code) is, we suspect, part of a qsort version that can exploit the availability of multiple cores.

LQ can handle multiple types; it uses only a median of 3 to determine the pivot; it is semi-recursive because it defines its own stack; it ignores small segments and invokes insertionsort once after the top level call returns; and it orders the recursive calls for the processing of the partitions.

B&M Qsort

B&M qsort is part of the C-library. It was developed by Bentley & McIlroy around 1992 and published in [Bentley & McIlroy]. {We were lucky to have access to this publication because we were troubled for years by a defective version in the Cygwin library that exhibited quadratic explosions on instances of their own test-bench.} B&M can handle multiple types. They use a clever coding trick to recognize when comparing an element against the pivot that the element is actually equal to the pivot. If so they store such an element either at the left or right side of the array by swapping. Hence the array layout during partitioning can be diagrammed as:

```
|--------]----------]-----gap------[---------[--------]
    =          <                        >        =
```

When the middle segment is empty, the left and right segments are moved to the middle yielding the new layout:

```
|------------------]|--------------]|------------------]
    <                     =                  >
```

Only recursive calls for sorting the left and right segment are required. While the original publication had two unordered recursive calls, the (defective) version in the Cygwin C-library had the 2nd call replaced by a jump back.

ASPS

ASPS, which stands for Adaptive Symmetry Partition Sort, was developed by Jingchao Chen in 2007 and was described in [Chen]. ASPS is an outlier in the collection of algorithms we describe here. While it has the same interface as B&M's qsort, its internals are radically different. An immediate difference is that it has an explicit bias for sorted input. We ignore this infrastructure and concentrate on the core component Symmetry Partition Sort. It is also atypical and can be characterized loosely as doing bottom up sorting as opposed to quicksort's top down approach. Pictures are likely the best way to give a high level description of SPS. There are two patterns: Left and Right each with their own reducers. A left reducer applied to the Left pattern generates a smaller Left pattern and a smaller Right pattern. Ditto for the right reducer applied to the Right pattern. In pictures, the patterns have a sorted segment *S* and an unsorted segment *U*:

```
Left pattern:
|---------]--------------------------|
     S              U

Right pattern:
|-----------------------[------------|
           U                 S
```

If these patterns are small (size of *S* plus size of *U* less than say 10) then insertionsort can be applied to them, ignoring whether or not the segments *S* are actually sorted or not. The Left pattern reducer starts by taking a bite *V* from *U* while *SL* and *SR* are respectively the left and right halves of *S*:

```
Left pattern:
|----|----][--------]--------------|
  SL   SR       V        U'
```

Segment *V* is moved between *SL* and *VR*:

```
Left pattern:
|----][--------][----]-------------|
  SL      V       SR      U'
```

Using the median of *S* as a pivot *V* is partitioned into *VL* and *VR*:

```
Left pattern:
|----][---|---][-----]-------------|
  SL   VL   VR   SR        U'
```

The segment *SL+VL* is a Left pattern and can be handled by the left reducer. Similarly *VR+SR* is a Right pattern and can be handled by the right reducer. This yields the Left pattern layout where *U'* is smaller in size than *U*:

```
|--------------------]-------------|
          S'                U'
```

The *S'+U'* task is dealt with by taking a bite from *U'*, etc. The right reducer handles the mirror image like the left reducer.

This is the basic version of SPS. The ASPS algorithm has, among other improvements, infrastructure to exploit (semi) ordered ascending or descending inputs, which yields spectacular results in some of our tests.

While the ASPS sorter is ingenious it lacks a key feature from the traditional quicksort variants: quicksort's partitions are large, disjoint array segments and their subsequent processing can be done in parallel. Whether ASPS could be parallelized is unclear, if possible at all.

FourSort

The FourSort algorithm was developed by us around 2010. The code is available at [FourSort]. We will use in the comparisons two versions: an integer only version and a pointer/record version. FourSort has three layers; see the next figure in which the arrows show algorithm member delegation dependencies:

$$\begin{tikzcd}
\text{heapsort} & \text{cut2} \arrow[l] \arrow[r, leftrightarrow] \arrow[d] & \text{dflgm} \\
 & \text{quicksort} \arrow[ul] \arrow[d] & \\
 & \text{insertionsort} &
\end{tikzcd}$$

The member *cut2* is a quicksort version with more overhead than the one below it. *Cut2* uses five array elements to sort them and takes the middle element as the pivot. The first and last elements can be switched with the segment boundary elements when they are respectively smaller or larger than the boundary elements. *Cut2* constructs partitions with the invariants that at the left side are elements less or equal than the pivot while at the right side are elements greater than the pivot. This arrangement fails when, for example, all array elements are equal. Initial failure of an invariant can be recognized easily. If so, processing is delegated to the member *dflgm,* which is a Dutch flag type of algorithm. This algorithm obtains as argument the index position of the pivot. It uses this position to initialize the middle sub-segment immediately. Subsequently it manages the layout:

```
|---------]----gap----[--------]----gap----[--------|
     <                    =                    >
```

Both quicksort members use the Musser defense against quadratic explosions and both as well as *dflgm* order recursive calls. The *dflgm* member uses for the movement of elements regular 2-swaps but also six 3-swaps.

DPQ

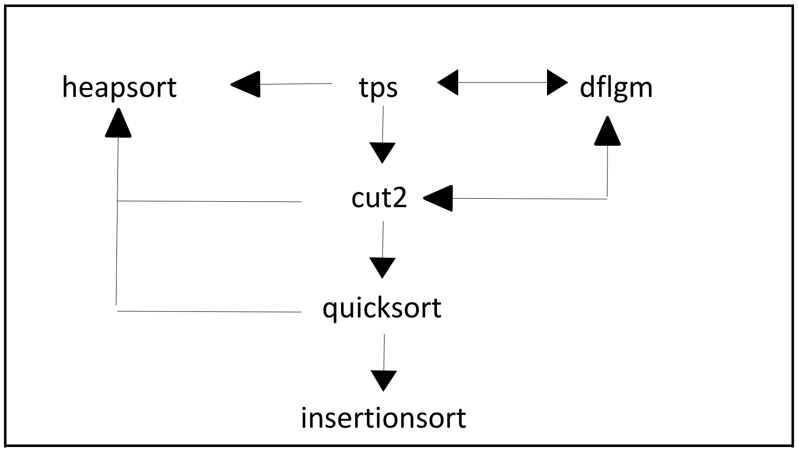
DPQ, which stands for Dual Pivot Quicksort was developed by V. Yaroslavkiy, J. Bentley and J. Bloch around 2009 and has been added to Java around 2011. The code can be found at [DPQ]. This ref has a link to a file with distinct, but virtually identical versions for the following Java types: int, long, short, char, byte, float and double; but not for a generic pointer version. We describe here a shortened version; see below for what we omitted. DPQ is two layered, but with a twist. It analyzes five elements in order to obtain two pivots. When there is at least one duplicate in these five elements segment processing is delegated to a Dutch Flag type of algorithm that takes as single pivot the 3rd of the sorted five elements. Otherwise partitioning is done with two pivots according to the layout:

```
|---------[------------------]----gap----[--------|
    < p1     p1 <= … <= p2                 > p2
```

When the gap is closed the left and right segment are recursively sorted. The middle segment gets a special treatment when its size is greater than 4/7 of the segment's size. If so elements in the middle segment equal to *p1* are swapped to the left and elements equal to *p2* are swapped to the right. (Remaining) middle elements are recursively sorted. (The special treatment of the middle segments appears to be counterproductive in ad hoc tests we did.) The algorithm as described in [DPQ] has explicit bias for sorted input. A preprocessor checks whether the input consists of a limited number of already sorted (ascending, descending or equal) sub-segments. If so mergesort is invoked. We removed this preprocessor to level the playing field in our comparison tests.

FiveSort

The FiveSort algorithm was developed by us around 2010 and is an modification of FourSort. The code is available at [FiveSort]. We will also use in the comparisons two versions: an integer only version and a pointer version. FiveSort has, unlike FourSort four layers; see the next figure in which the arrows show algorithm member delegation dependencies:



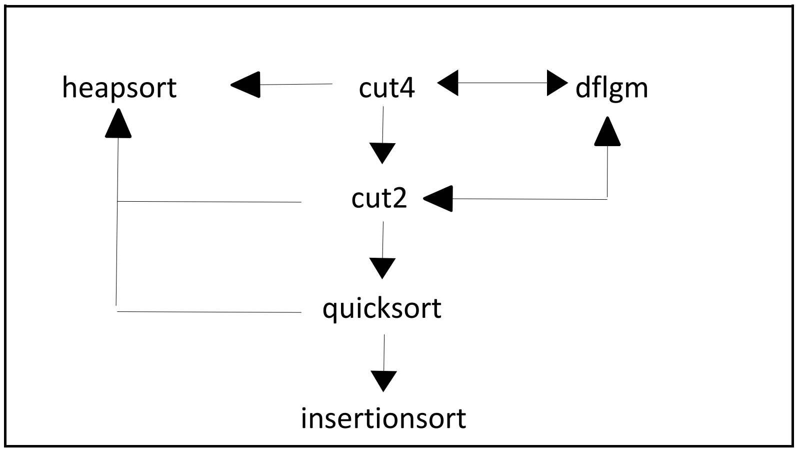
The difference with FourSort is that another layer is added above the *cut2* member. The *tps* member uses like DPQ two pivots. Pivot construction resembles what is done in DPQ. Delegation to *cut2* happens when the array segment size is less than 250. The member *dflgm* is invoked when the two pivots are equal or when the corners do not satisfy the layout invariant. The partition layout of *tps* resembles the one of *dflgm*:

```
|---------]----gap----[--------]----gap----[--------|
  … < p1              p1 <= … <= p2            p2 < …
```

The other design choices are the same as for FourSort.

SixSort

SixSort was developed earlier than FiveSort but has been simplified recently. The code is available at [SixSort]. Again we will use an integer version as well as a pointer version. SixSort has four layers:



The member *cut4* obtains three pivots from assembling through swapping a mini array of at least 20 elements in the middle, sorts them and grabs the ¼, middle and ¾ elements as pivots. The corners of the segment are adjusted when appropriate with the corner elements in the mini array. When the pivots are not different or the corners do not satisfy the layout variant the input segment is delegated to the member *dflgm*. The transitions to members with lower overhead are respectively: *cut4* to *cut2*: 375; *cut2* to *quicksort*: 127; *quicksor*t to *insertionsort*: 10. The partitioning layout in *cut4* has two gaps similar to the one for *tps*:

```
|---------]--gap--[----------]----------]--gap--[--------|
  … <= p1        p1 < … <= p3   p3 < … < p2        p2 <= …
```

The positioning of elements in *cut4* relies on the "single data movement" technique described above. The key insight for *cut4* was to see these movements as the transitions in a finite state machine where a state is a specific configuration and where a transition corresponds with the exchange of a gap element with the value hold in a 'roving variable' and where the next value in the roving variable is checked where it must be deposited. The cycle repeats at the state where the next value will be stored.

Tests for testing the selected algorithms

This section discusses the tests used for comparing the different algorithms. Sorting is a demanding activity. Quality cannot be challenged: all original elements need to show up in the output and a (specified) ordering relationship must be satisfied. The average complexity is basically *NlogN* for all of them, thus they differ only a fixed ratio. Thus testing on uniform distributions allows comparing them faithfully. Still, one can wonder how they perform comparatively on weird input. Constant distributions, ascending and descending distribution are examples. We used these and parametrized them with varying amount of random noise. These distributions can be used to check whether algorithms have (implicit or explicit) bias for already ordered input.

Bentley & McIlroy described a test-bench in [Bentley & McIlroy]. We ran it with the array size 1M. It has five families of distributions. There is a parameter that generates for each family 21 different instances. Each instance can be modified in six different ways. This yields a collection of 630 different distributions. There are two problems with this collection. One of the modifiers sorts the generated distributions. Hence 17% of this collection favors algorithms that have an explicit bias for sorted data. This modifier was eliminated. Two of the families produce distributions that are mostly semi sorted with ascending data. We eliminated these two families also. Instead we created another family *Slopes*; see the Appendix for the code. This yields 105 times four is 420 'weird' distributions.

We use time ratios between two algorithms A and B as follows:

- Determine the number of iterations for a test
- Measure the repeated filling time of an array
- Measure the repeated operation of filling and sorting the array by algorithm A and record for algorithm A the sorting time by subtracting the filling time
- Ditto for algorithm B
- Report the time ratio between algorithm A and B
- (optional) repeat the preceding steps three times and select the median when reasonable or repeat the test

Testing single threaded algorithms can be done with C's *clock* function. Testing multi-threaded algorithms goes similarly but requires the *clock* function to be replaced by actual time measurements using the *gettimeofday* function (and preventing interference by other processes).

Machines used for testing

We used (among others) the following machines:

- Intel Pentium M (p5), 1.73Ghz, WinXP+Cygwin, L1 cache 0,? 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 32 bits, 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.

Comparing single type versus multi type algorithms on integers

We consider here only single pivot algorithms. Since we have available an integer version of FourSort we use that one for the comparison against LQ, B&M and ASPS on integer arrays. We obtain the following results with the abundance of data provided:

- The magnitude of the (obvious) advantage of a single type algorithm
- The diversity of performances of LQ, B&M and ASPS
- The substantial impact of the machine cache architecture on the performances

The reader in a hurry can safely just skim the tables.

Uniform distributions

We used the array sizes: 1M, 2M, 4M, 8M and 16M. Since the timing ratios hardly vary on this range we report here the median of these 5 values (which themselves were obtain by the median of three tests, which themselves were obtained by many accumulations: 32 for 1M size arrays , …, and 2 for 16M arrays). A ratio below one means that FourSort is faster. We tested on three different machines:

Machine/ algorithm	LQ	B&M	ASPS
p5	0.41	0.52	0.56
i3	0.46	0.59	0.59
am	0.34	0.3	0.3

This specific test shows that FourSort has, unsurprisingly, a definite advantage due to being specialized for integers. While the ordering between LQ, B&M and ASPS is undisputed in the rows of the p5 and i3 machines, the ordering changes on the am machine. Machine dependence is also manifest in the columns.

Constant data with an increasing amount of uniform distributed noise

The array size for this test is just 16M. Going down the rows the array with constant values gets more noise data going from 0% to 100%.

Machine/ percentage	p5			i3			am		
%	LQ	B&M	ASPS	LQ	B&M	ASPS	LQ	B&M	ASPS
0	0(*)	0(*)	0(*)	0(*)	0(*)	0(*)	0(*)	0(*)	0(*)
20	0.12	0.55	0.58	0.15	0.62	0.62	0.14	0.32	0.35
40	0.21	0.55	0.58	0.26	0.62	0.63	0.23	0.34	0.35
60	0.29	0.55	0.58	0.34	0.61	0.62	0.28	0.34	0.33
80	0.36	0.54	0.57	0.42	0.6	0.61	0.33	0.31	0.32
100	0.42	0.53	0.56	0.47	0.6	0.61	0.36	0.31	0.31

(*) FourSort's *dflgm* member is too fast on constant data for the used clock-function to produce positive time delta's.

The ratios in the columns for LQ vary substantially in contrast with the other algorithms; hence we notice here again that algorithms cannot be simply compared.

Sorted data with an increasing amount of uniform distributed noise

We use again 16M for the array size. We have here sorted data with an increasing amount of uniform distributed noise:

	p5			i3			am		
%	LQ	B&M	ASPS	LQ	B&M	ASPS	LQ	B&M	ASPS
0	0.4	0.29	12.4	0.42	0.36	12.7	0.2	0.14	4.3
20	0.37	0.47	0.48	0.4	0.52	0.53	0.29	0.25	0.24
40	0.39	0.5	0.52	0.43	0.55	0.55	0.34	0.29	0.28
60	0.41	0.52	0.55	0.46	0.58	0.58	0.36	0.31	0.3
80	0.42	0.53	0.56	0.47	0.59	0.59	0.36	0.31	0.31
100	0.42	0.54	0.57	0.47	0.6	0.61	0.36	0.31	0.31

The results show that ASPS shines with explicit bias for sorted data. However, injecting just 1% of noise brings on the i3 machine the performance ratio down from 12.7 to 0.52.

Inversely Sorted data with an increasing amount of uniform distributed noise

We use again 16M for the array size. Only the row with zero percent noise differs significantly from the results in the preceding table.

	p5			i3			am		
%	LQ	B&M	ASPS	LQ	B&M	ASPS	LQ	B&M	ASPS
0	0.39	0.24	5.1	0.4	0.29	6.3	0.21	0.12	1.76
20	0.38	0.47	0.47	0.4	0.52	0.5	0.3	0.26	0.24
40	0.39	0.49	0.51	0.44	0.55	0.55	0.33	0.29	0.28
60	0.41	0.52	0.54	0.46	0.58	0.59	0.36	0.31	0.3
80	0.42	0.53	0.57	0.47	0.59	0.59	0.36	0.32	0.31
100	0.42	0.53	0.57	0.47	0.59	0.6	0.36	0.32	0.32

The first row with zero noise is where this table differs from the preceding table. The ratios are mostly lower, which suggests that FourSort is less sensitive to the input being reversely sorted than LQ, B&M and ASPS, while ASPS still outperform FourSort on data that is perfectly inversely sorted.

The Bentley-McIlroy test-bench

We compared LQ, B&M and ASPS on the three different machines using the well-known Bentley-McIlroy test-bench. We eliminated the *Plateau* and *Shuffle* families and added the *Slopes* family. Each family has six modifiers. The modifier generating already sorted input was removed because we tested that feature already; hence we get 105 instances per family instead of 120. Here the data on the p5 machine:

Test	#Variants	LQ faster	4S faster	Time ratio	B&M faster	4S faster	Time ratio	ASPS faster	4S faster	Time ratio
Sawtooth	105	0	105	0.18	0	105	0.48	2	103	0.56
Rand2	105	0	105	0.25	0	105	0.6	0	105	0.63
Stagger	105	0	105	0.37	0	105	0.5	0	105	0.54
Slopes	105	0	105	0.3	0	105	0.57	0	105	0.59
Sum	420	0	420		0	420		2	418	

The time-ratio variability of LQ over the different types of families is substantial.
Here the results of the same tests on the i3 machine:

Test	#Variants	LQ faster	4S faster	Time ratio	B&M faster	4S faster	Time ratio	ASPS faster	4S faster	Time ratio
Sawtooth	105	0	105	0.18	0	105	0.51	4	101	0.59
Rand2	105	0	105	0.3	0	105	0.67	0	105	0.68
Stagger	105	0	105	0.39	0	105	0.54	0	105	0.55
Slopes	105	0	105	0.31	0	105	0.58	0	105	0.6
Sum	420	0	420		0	420		4	416	

Time-ratio variability of LQ is substantial here as well.

Here the results of the same tests on the am machine:

Test	#Variants	LQ faster	4S faster	Time ratio	B&M faster	4S faster	Time ratio	ASPS faster	4S faster	Time ratio
Sawtooth	105	0	105	0.14	0	105	0.2	2	103	0.24
Rand2	105	0	105	0.22	0	105	0.31	0	105	0.32
Stagger	105	0	105	0.29	0	105	0.25	0	105	0.27
Slopes	105	0	105	0.23	0	105	0.26	0	105	0.27
Sum	420	0	420		0	420		2	418	

The performance of FourSort on the am machine is again distinctly better. We have provided the 'gory' details of the performances of the algorithms on different distributions and on the different machines to show that there is no uniform best algorithm and that the architecture of machines (their cache sizes) plays a large role in the relative performances.

Key feature of FourSort

The speed up by using type specific sort algorithms, as pursued by the Java community, is not a surprise. Arrays with duplicate elements are troublesome for quicksort type algorithms because they run the risk of quadratic explosions. B&M addresses this by setting aside elements equal to the pivot and swapping them to the middle later. FourSort senses trouble when the corner elements do not conform initially to the asymmetric partition condition and if so delegates the partitioning to *dflgm* a Dutch flag type of algorithm; this algorithm is invoked heavily in the non-uniform distribution tests. Switching cheaply to another algorithm is a distinctly recommended feature.

Comparing single type versus multi type algorithms on records

Thus far we reported about what is common in academics: sorting integers. To cater for sorting in the 'real' world we rewrote FourSort, FiveSort and SixSort so that they can access records through pointers. This requires an additional comparison function argument similar to the signatures of LQ, B&M and ASPS. This section is devoted to selective comparisons using simple records where there are two fields, an integer and a float field while the sorting is done only on the integer field using again uniform distributions.

Switching from arrays with integers to arrays with records changes the comparisons between FourSort and LQ, B&M and ASPS considerably. We use again the array sizes: 1M, 2M, 4M, 8M and 16M and again we report here only the median of these 5 values. A ratio below one, as before, means that FourSort is faster. We tested also on the three different machines:

Machine/ algorithm	LQ	B&M	ASPS	DFLGM
p5	0.78	0.96	1.01	0.91
i3	0.73	0.91	0.94	0.91
am	0.77	0.87	0.95	0.92

The differences among LQ, B&M and ASPS have decreased as well as their differences with FourSort. We will analyze this trend below. We added a column for the DFLGM algorithm. We use the *dflgm* member typically only when a main partitioning algorithms suspects an abnormal distribution. With a wrapper around its core functionality it becomes a 3-layer sorter consisting of the members *dflgm*, *quicksort* and *insertionsort*.

We repeated using the Bentley-McIlroy test-bench for records. Here the data on the p5 machine:

Test	#Variants	LQ faster	4S faster	Time ratio	B&M faster	4S faster	Time ratio	ASPS faster	4S faster	Time ratio
Sawtooth	105	0	105	0.41	14	91	0.79	54	51	0.98
Rand2	105	0	105	0.25	69	36	1.05	97	8	1.11
Stagger	105	0	105	0.61	0	105	0.85	11	94	0.94
Slopes	105	0	105	0.59	30	75	0.88	35	70	0.95
Sum	420	0	420		113	307		197	223	

Noteworthy is that there are many distribution in the test-bench where B&M and ASPS are faster than FourSort.The time-ratio variability of LQ remains remarkable.

Here the results of the same tests on the i3 machine:

Test	#Variants	LQ faster	4S faster	Time ratio	B&M faster	4S faster	Time ratio	ASPS faster	4S faster	Time ratio
Sawtooth	105	0	105	0.38	21	84	0.88	43	62	1.02
Rand2	105	0	105	0.55	67	38	1.06	72	33	1.06
Stagger	105	0	105	0.61	13	92	0.92	16	89	0.94
Slopes	105	0	105	0.46	30	75	0.8	32	73	0.81
Sum	420	0	420		131	289		163	257	

The comparisons on the i3 are slightly better for FourSort than on the p5 machine.

Here the results of the same tests on the am machine:

Test	#Variants	LQ faster	4S faster	Time ratio	B&M faster	4S faster	Time ratio	ASPS faster	4S faster	Time ratio
Sawtooth	105	0	105	0.41	16	89	0.88	52	53	1.01
Rand2	105	0	105	0.56	59	46	1.03	67	38	1.05
Stagger	105	0	105	0.63	12	93	0.93	10	95	0.94
Slopes	105	0	105	0.52	30	75	0.85	32	73	0.87
Sum	420	0	420		117	303		161	259	

The ratios here are comparable with this those on the i3 machine.

Comparing single pivot against two pivot algorithms on integers

This section is devoted to comparing the integer version of FourSort against the integer versions of FiveSort and DPQ; the latter two using two pivots. As discussed earlier the infrastructure for exploiting sorted inputs in DPQ has been eliminated when we translated their

Java version into C.   The next table has the time ratios for FiveSort/FourSort, DPQ/FourSort and FiveSort/DPQ. We use again arrays ranging from 1M to 16M with uniform distributions:

Machine/ competitors	p5	i3	am
FiveSort/FourSort	0.96	0.94	1.02
DPQ/FourSort	0.97	0.9	1.39
FiveSort/DPQ	1.01	1.05	0.73

The first row shows that FiveSort is a bit faster than FourSort on the p5 and i3 machine, but marginally slower on the am machine. The surprise in the second row is the substantially faster performance of the one pivot FourSort against the two pivot DPQ on the am machine. Similarly we see the ratios of two algorithms both using two pivots. FiveSort is marginally slower on p5 and i3 but substantially faster on am.

Comparing single pivot against two pivot algorithms on records

This section has the FiveSort/FourSort ratio with the simple records on arrays 1M-16M with uniform distributions on the different machines; we have added a row with the ratios for integers from the previous section to highlight that on the i3 machine FiveSort does worse for records than for integers:

Machine/type	p5	i3	am
integers	0.96	0.94	1.02
records	0.93-0.92	1.00-0.96	0.97-0.93

We did not bother to rewrite DPQ so that it could handle records because – as we will discus below - two-pivot algorithms cannot be competitive on records.

Comparing single pivot against triple pivot on integers

FourSort specialized for integers outperforms the single pivot competitors and thus we use it in the comparison against the integer version of SixSort. We obtain on uniform distributions the ratios:

Machine/ Array range	p5	i3	am
1M-16M	0.92-0.90	0.94	1.0 (!)

The surprise here is that one pivot FourSort is doing as well as SixSort using three pivots on the am machine. The small code of *cut2* with its tight loops in FourSort yields plausibly better look ahead opportunities on the am machine than for the way larger code of *cut4* in SixSort.

Comparing single pivot against triple pivot on records

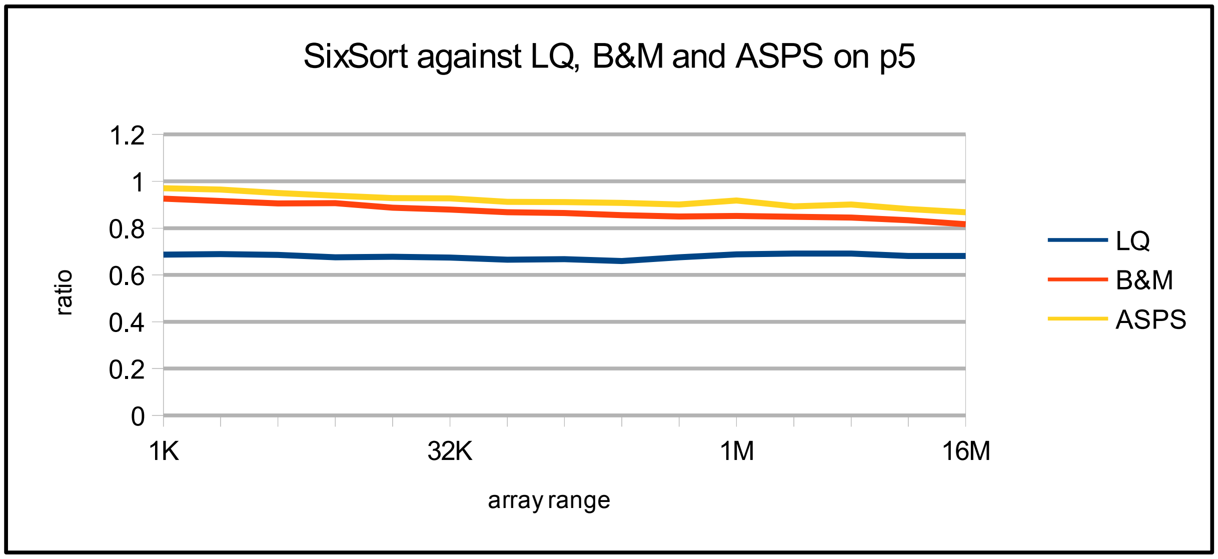
FourSort specialized for records outperforms also the single pivot competitors and thus we use it in the comparison against the records version of SixSort. We obtain on uniform distributions the ratios:

Machine/ Array range	p5	i3	am
1M-16M	0.89-0.86	0.95-0.89	0.92-0.87

Comparing SixSort against LQ, B&M and ASPS on records

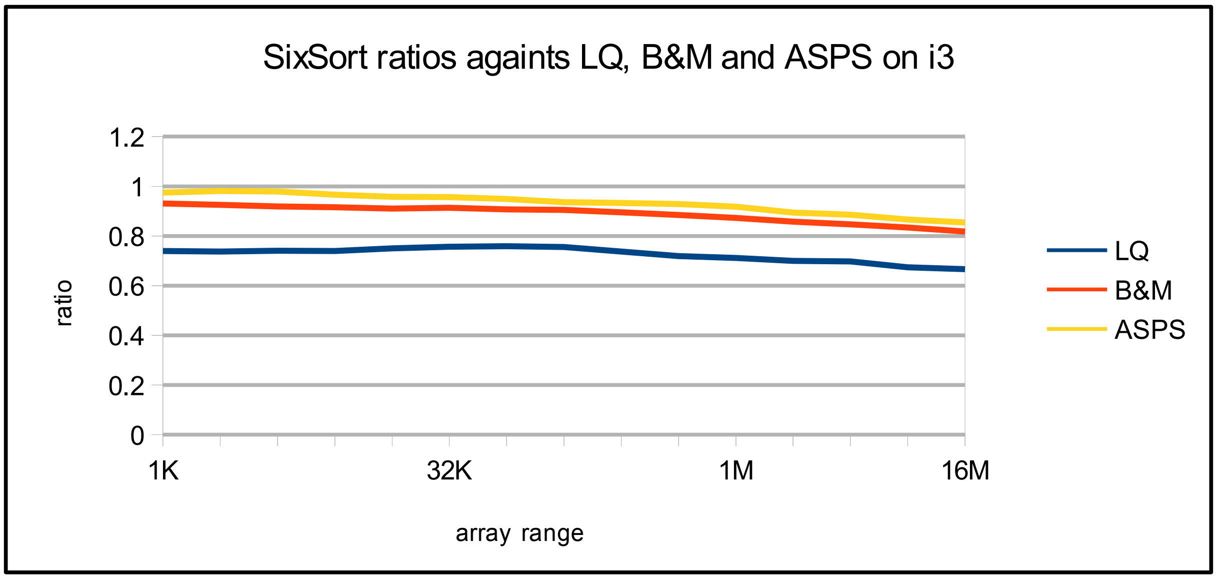
On the p5 machine we get the ratios for arrays in the range 1K-16M:

Array	LQ	B&M	ASPS
1024	0.687	0.926	0.971
2048	0.689	0.915	0.964
4096	0.685	0.906	0.95
8192	0.676	0.906	0.939
16384	0.678	0.887	0.929
32768	0.674	0.880	0.927
65536	0.665	0.868	0.912
131072	0.668	0.865	0.911
262144	0.660	0.856	0.909
524288	0.676	0.850	0.907
1048576	0.688	0.852	0.901
2097152	0.691	0.849	0.894
4194304	0.692	0.845	0.901
8388608	0.681	0.834	0.882
16777216	0.681	0.817	0.868



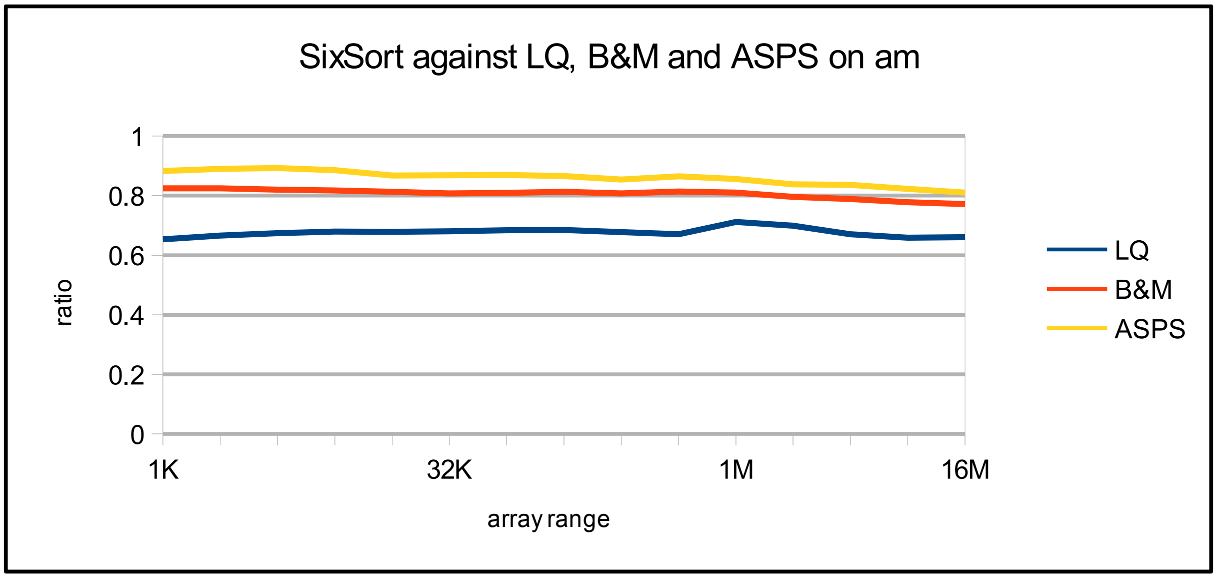
On the i3 machine we get the ratios for arrays in the range 1K-16M:

Array	LQ	B&M	ASPS
1024	0.739	0.931	0.974
2048	0.737	0.925	0.981
4096	0.740	0.919	0.978
8192	0.739	0.916	0.966
16384	0.750	0.910	0.958
32768	0.757	0.914	0.957
65536	0.759	0.907	0.949
131072	0.755	0.905	0.936
262144	0.737	0.895	0.933
524288	0.719	0.884	0.928
1048576	0.711	0.873	0.917
2097152	0.699	0.857	0.895
4194304	0.698	0.847	0.886
8388608	0.674	0.834	0.886
16777216	0.665	0.818	0.855



On the am machine we get the ratios for arrays in the range 1K-16M:

Array	LQ	B&M	ASPS
1024	0.654	0.824	0.883
2048	0.666	0.825	0.890
4096	0.674	0.820	0.892
8192	0.679	0.818	0.885
16384	0.679	0.813	0.868
32768	0.680	0.808	0.868
65536	0.684	0.810	0.869
131072	0.685	0.813	0.866
262144	0.680	0.808	0.854
524288	0.678	0.813	0.865
1048576	0.677	0.810	0.856
2097152	0.670	0.796	0.838
4194304	0.671	0.788	0.836
8388608	0.659	0.777	0.822
16777216	0.660	0.772	0.810



Before discussing the different ratios obtained on integers versus records we report the yields when using parallelism.

Comparing single thread against multithread on integers

Having available a parallel version of SixSort (P6S) for integers we report in this section its performance against single threaded FourSort (4S) and SixSort (6S). We use uniform distributions on, as before, arrays in the range 1M-16M. On the i3 machine we obtain:

#threads/ algorithms	2	3	4
P6S/4S	0.56-0.60	0.49-0.52	0.45
P6S/6S	0.67-0.66	0.60-0.56	0.54-0.49

Given that the i3 machine has only two cpus it is quite amazing that we obtain a speedup barely faster than twofold in the second row.

On the am machine we obtain:

#threads/ algorithms	2	3	4
P6S/4S	0.63-0.61	0.58-0.54	0.81-0.70
P6S/6S	0.63-0.61	0.58-0.54	0.80-0.69

We encounter a surprise on the am machine when we ask for four threads: the ratio *deteriorates* in both rows. L1 cache congestion is a conjecture for the declining performance.

Comparing single thread against multithread on records

We compare in this section (with records) only single threaded FourSort (4S) against the following parallel algorithms: parallel FourSort (P4S), parallel FiveSort (P5S) and parallel SixSort (P6S). On the i3 machine we observed:

#threads/ algorithms	2	3	4
P4S/4S	0.63-0.67	0.53-0.56	0.5
P5S/4S	0.65	0.55-0.53	0.52-0.47
P6S/4S	0.60-0.57	0.51	0.48-0.44

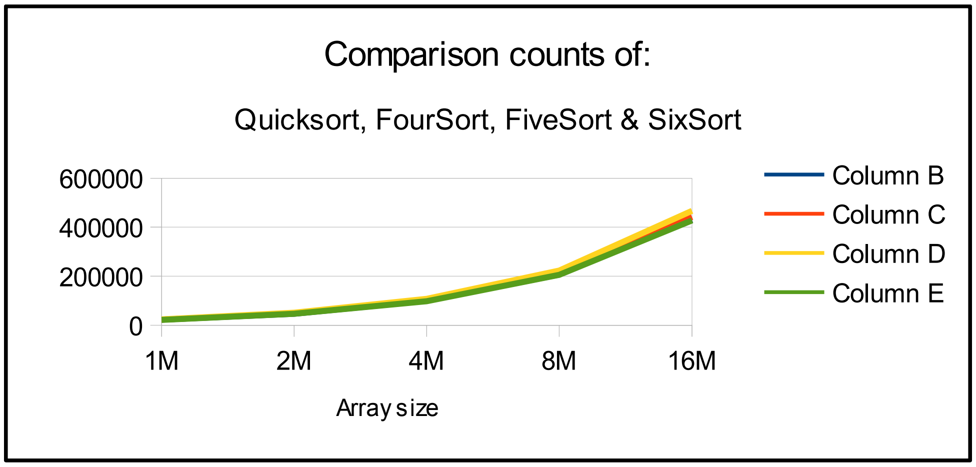
Similarly, we obtain on the am machine:

#threads/ algorithms	2	3	4
P4S/4S	0.64-0.60	0.54-0.48	0.53-0.41
P5S/4S	0.66-0.58	0.57-0.45	0.58-0.42
P6S/4S	0.58-0.51	0.49-0.40	0.41-0.33

There is something strange on both machines with the P5S/4S ratios. The ratios for 1M arrays are not in between those for P4S and P6S. The next section analyzes the problem with two pivot algorithms. The two other parallel algorithms show acceptable speedups, especially on the am machine.

Integer ratios versus record ratios difference

How do we explain the ratios between the algorithms when comparing their performance on integers versus records? We start by showing that the number of comparisons done by algorithms on a 16M array with a uniform distribution is quite similar (and does not depend on whether integers or records are sorted). Here the graph where the values on the y-axis should be multiplied by 1000:



The top outlier of these four graphs is D, which represents FiveSort. Here the underlying data for array size 16M:

Algorithm/ comparisons	Quicksort	FourSort	FiveSort	SixSort
# * 1000	432656	445035	474069	431457
ratio	1.07	1.11	1.18	1.07

The ratios are calculated against the minimum value of 402653184. FourSort does 3% more comparisons than Quiksort because it spends more effort than Quicksort on finding a good pivot as part of the protection against a quadratic degradation. To explain the greater number of comparisons in FiveSort we have to do a mini complexity analysis that assumes perfect pivots and ignores the delegations to simpler algorithms and the overhead of finding a good pivot. We define (see an appendix for the derivation of the approximations):

- *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 of base 2, 3 and 4.
This gives us:

- $T2(N) = N*log2(N)*(A + S/2 + C)$
- $T3(N) = N*log3(N)*(A + S*2/3 + C*5/3)$
- $T4(N) = N*log4(N)*(A + S*3/4 + C*2)$

Since we have log3(N) = 0.6310*log2(N) as well as log4(N) = (1/2)*log2(N) we obtain:

- *T3(N) = N*log2(N)*0.6310*(A + S*2/3 + C*5/3) = N*log2(N)*(A*0.6310 + S*0.4207+C*1.051)*
- *T4(N) = N*log2(N)*(1/2)*(A + S*3/4 + C*2) = N*log2(N)*(A*0.5 + S*0.375+C)*

While T3 has less array access and stores than T2, the number of comparisons is 1.051 more than in T2 (vanilla Quicksort). The measured ratio of comparisons FiveSort/FourSort of 1.06 supports this complexity analysis. This analysis shows as well that an algorithm implementing T4 has a chance to improve on Quicksort since it requires less array access and stores while doing the same number of comparisons.

Still remains the issue how to account for the different ratios between the algorithms when sorting integers versus records. Here the clock-time values for sorting 16M arrays on the p5 machine for the different algorithms:

Algorithm/ type	FourSort	FiveSort	SixSort
integers	9723	6277	5452
records	36341	33848	29848

The SixSort/FourSort ratio for integers at 0.56 increases for records to 0.82. Similar story on the am machine:

Algorithm/ type	FourSort	FiveSort	SixSort
integers	3722	2106	1890
records	11187	10469	8921

The ratio for integers at 0.51 increases for records to 0.80. The formulas above show that during partitioning for SixSort the number of array accesses is half against FourSort, the number of stores is 1/8 less and the number of comparisons is the same. Clearly the relative contribution of comparisons increased when going from integers to records because comparing a pair of integers is cheap in contrast with having to execute a comparison function in the case of records.

This analysis shows also that the two pivot algorithms DPQ and FiveSort, while doing fine on integers, cannot be competitive on records due to the higher ratios of pairwise comparisons as predicted by the complexity analysis and as confirmed by the data. The number of comparisons done by DPQ on 16M arrays came in at 527673*1000. The ratio against the theoretical minimum of 402653184 becomes 1.31 way worse than the 1.18 we measured for FiveSort. The timing ratios for records of DPQ on 16M arrays against SixSort on p5, i3 and am clocked in respectively at 1.26, 1.17 and 1.24. On a Linux Xeon machine the ratio was 1.20.

Conclusion with summary of the findings

Our findings regard quicksort type of algorithms that do partitioning in two, three or four sub-segments and that are single threaded or multi-threaded. Extensive comparisons of single pivot algorithms - well-knows and newer ones - on a battery of distributions generators show that there is not a single algorithm that outperforms the others consistently. The space of permutations remains, even after half a century of research, a major challenge for the different designs of the algorithms considered. Still, a few general assertions are warranted by comparing average speeds over large collections of normal and abnormal distributions.

The algorithms LQ, B&M and ASPS have been designed to handle native types (integers, doubles, floats, fixed length strings, etc.) as well as records accessed by pointers. They pay a heavy price for that general capability. They are outperformed on native type specific sorters, like our integer specific version of FourSort. The pointer/record version of FourSort outperforms them as well due to their more complex array addressing machinery.

Two pivot algorithms, which partition in three segments like DPQ and our FiveSort, are an improvement over single pivot algorithms on integers (and on types where comparisons are cheap) but a complexity analysis and performance data shows that they cannot be competitive on records when comparisons are expensive.

Our three pivot algorithm SixSort outperforms our single pivot algorithm FourSort. Parallel versions of FourSort, FiveSort and SixSort outperform the single threaded versions. See above for the details.

We tested on three machines: an Intel Pentium, and Intel i3 and an AMD FX-8350. Unsurprising is the speed differences on these machines. Noticeable however is that speed ratios differ and there are occasional rank order differences.

We presented an abundance of data in this paper (with newer distribution generators) because we believe that more tests are crucial for triggering progress in this realm.

Where to find the algorithms discussed and more

The site github.com/ddccc contains the following repositories that contain, among others, the files:

- foursort: FourSort.c, ParFourSort.c, UseFourSort.c, UseParFourSort.c
- fivesort: FiveSort.c, ParFiveSort.c, UseFiveSort.c, UseParFiveSort.c
- sixsort: SixSort.c, ParSixSort.c, UseSixSort.c, UseParSixSort.c
- integersorters: compare2.c

Some algorithms can be found in multiple files:
LQ, the Linux version of qsort is in: compare2.c, UseFourSort.c, UseSixSort.c.

B&M, the repaired Cyqwin version of qsort is in: compare2.c, UseFourSort.c, UseFiveSort.c, UseSixSort.c.
ASPS is in: compare2.c, UseFourSort.c, UseSixSort.c.
The file compare2.c has the integer versions of FourSort, FiveSort and SixSort with their members *insertionsort*, *quicksort*, *cut2*, *tps*, *cut4*, *dflgm*. This file has in addition, among others, alternative implementations of *quicksort*, *tps*, *dflgm*, *cut4*, two merge sort versions and the integer version of parallel SixSort.
FourSort with its members is in FourSort.c and similar for FiveSort and SixSort.
Parallel FourSort with its members is in ParFourSort.c and similar for parallel FiveSort and parallel SixSort.
Infrastructure to test, validate, measure and compare algorithms is available in compare2.c, and in the six Use-files.

Acknowledgments

Doug Lea gave the introduction to Joshua Bloch, who provided the pathological input test-bench from [Bentley & McIlroy]. This forced to give way-more attention to avoid quadratic explosions, which led after many variants to the current version of *dflgm* the Dutch flag type of sorter responsible for suspect distributions. Anonymous reviewers convinced us to add the Musser defense to the FourSort members: delegate to heapsort when the recursion depth gets out of hand. Clay Breshears provided the data for comparing parallel FourSort against the sequential version and the differences between the codes generated by the Gnu gcc and the Intel icc compilers as reported in an appendix.

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.

[Dijkstra] Dijkstra, E., "Go To Statement Considered Harmful", *CACM*, Vol 11, No 3, pp 147-148, 1968.

[DPQ] http://www.docjar.com/html/api/java/util/DualPivotQuicksort.java.html

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

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

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

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

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

[Sedgewick] Sedgewick, R., "Implementing Quicksort Programs", *CACM*, Vol 21, No 10, Oct. 1978.

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

Appendix Derivation of the complexity formulas

We give here a complexity analysis for 1, 2 and 3 pivot quicksort type sorting algorithms. We make the following assumptions:
- The array of size N contains a uniform distribution of integers
- We have 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.

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)
  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:
A + S*2/3 + C*5/3
because we must:
access each element
store 2/3 of them (assuming a perfect pivot)
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-segment, 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) =
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*log2(N)*(A*0.6310 + S*0.4207 + C*1.051)

Partitioning an array of size N into four sub-segments takes the time:
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) =
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)

Appendix Intel test

The data in this appendix was obtained with an earlier version of FourSort. We present it here to show the impact that a compiler can have on the performance comparisons. Ratios for parallel FourSort/ sequential FourSort were observed on the Intel multi core lab. Two different compilers yield different results. Below we report the time ratios obtained with the gcc and with the icc Intel C-compiler.

Uniform distributions: on Xeon X5680 Linux with threads 2, 4 and 8:

Array/Threads	2	2	4	4	8	8
Compiler	gcc	icc	gcc	icc	gcc	icc
1M	0.49	0.58	0.38	0.49	0.31	0.40
2M	0.54	0.57	0.38	0.42	0.29	0.33
4M	0.55	0.60	0.37	0.43	0.29	0.33
8M	0.50	0.56	0.36	0.37	0.27	0.26
16M	0.52	0.57	0.33	0.35	0.25	0.25

The data with two threads using the gcc compiler is close to the theoretical gain. It turns out that while the icc ratios are not as good as the gcc ratios, the icc compiler produces way faster codes than gcc. Here two examples:

Two threads, array 2M
Sequential times

gcc	icc	icc/gcc
15.7	8.5	0.54

Parallel times

gcc	icc	icc/gcc
8.4	4.8	0.57

Four threads, array 2M
Sequential times

gcc	icc	icc/gcc
14.0	8.7	0.62

Parallel times

gcc	icc	icc/gcc
5.6	3.6	0.64

Appendix Slopes

The *Slopes* code has been added to the Bentley-McIlroy test-bench. The *tweak* parameter is used to invoke distribution modifiers. We do not use *tweak = 4* because that produces a sorted array.

```
slopes(int *A, int n, int m, int tweak) {
  int k, i, b, ak;
  i = k = b = 0; ak = 1;
  while ( k < n ) {
    if (1000000 < ak) ak = k; else
    if (ak < -1000000) ak = -k;

    A[k] = -(ak + b); ak = A[k];
    k++; i++; b++;
    if ( 11 == b ) { b = 0; }
    if ( m == i ) { ak = ak*2; i = 0; }
  }
  if ( tweak <= 0 ) return;
  if ( tweak == 1 ) { reverse(A, n); return; }
  if ( tweak == 2 ) { reverseFront(A, n); return; }
  if ( tweak == 3 ) { reverseBack(A, n); return; }
  if ( tweak == 4 ) { tweakSort(A, n); return; }
  dither(A, n);
} // end slopes
```