File: C:/bsd/rigel/sort/Papers/Parallel.docx
Date: 2023 October, November
(C) Dennis de Champeaux/ OntoOO

Parallel Quicksort for Referenced Items with 1 and 3 pivots

Dennis de Champeaux
Silicon Valley
2023 November
ddc2@ontooo.com

Abstract
A design pattern using the C pThread package is employed for parallelizing a 1-pivot quicksort version and three 3-pivot versions. Test results on four different machines are reported with the number of threads ranging from 1 to 4. Timing ratios of 1M and 16M arrays show slowdowns against O(NlogN) expectations. Timing ratios of K-threads/1-thread show timing improvements, however less than expected. Rank orders vary over the different machines. More testing on machines used for database processing is recommended.

1 Introduction

A google search with 'parallel quicksort' yields recent numerous references to educational prototype versions, for example: [9.4 Quicksort]. Sample sort from 1970 [SampleSort] operates as follows.

- Create a sample from the array to be sorted with p threads
- Sort the sample and extract p-1 splitters that define p buckets
- Loop over the data, placing each element in the appropriate bucket. (This may mean: send it to a processor, in a multiprocessor system.)
- Sort each of the buckets with quicksort
- Concatenate the buckets

The 3rd step is easy when elements are send to external processors but problematic when in place processing is required.

This paper describes the parallel versions of the sequential 'industrial strength' versions, described earlier in [de Champeaux]. These parallel versions are also specific for referenced items (as explained in [de Champeaux]) and have the following properties:

- Guaranteed O(NlogN) worst case complexity
- Linear complexity on constant input
- Competitive performance with also parallel processing of the first partitioning segment
- Proper ordering of scheduling sub-segment processing to prevent a stack overflow, addressing the equivalent concern of the sequential versions, but with a different solution
- No additional memory is required beyond a stack

Design features of these parallel versions are:

- Using 5 sorting members in the hybrids
- Using two gap array layout in some members
- Reusing the novel technique to obtain better pivots from the sequential version
- Using 4 layers instead of the traditional 2 layers with only insertion sort and sequential quicksort

- Using the C pThread package for a parallel design pattern that has the number of threads as a parameter and can be employed to parallelize any sequential quicksort version as well as similar algorithms that decompose a problem in independent sub-problems.

The road map for the balance is:

- Summary of the sequential versions
- The design pattern using the C pThread package
- Replacing recursive calls and tail iteration with adding task specifications to a stack and tail iteration; sorting of the probe array
- Correctness
- The benefit of parallelizing the first partitioning task
- Conformance to O(NlogN) & the timings when the thread count increases
- Performance data of 1- and 3-pivot versions with multiple threads using 4 different machines
- Discussion of the findings and conclusion

1.1 Miscellaneous

Data is obtained by tests that repeat ten times and the average is taken. Magic numbers that control transitions to other members are obtained with extensive testing.

We used the following equipment for our tests:

-- Intel I3-2310M 2.1 Ghz Win7
-- Intel I5-2410M 2.3 Ghz Win10
-- Intel I5-8256, (I5/u) 1.6 Ghz Ubuntu, 6 MB Intel® Smart Cache
-- AMD fx-8350 (AMD), 4 Ghz, Win8.1+Cygwin, L1 cache 4x64Kb instruction + 8x16Kb data, L2 cache 4x2Mb, L3 cache 8Mb

All versions are available in [GitHub].

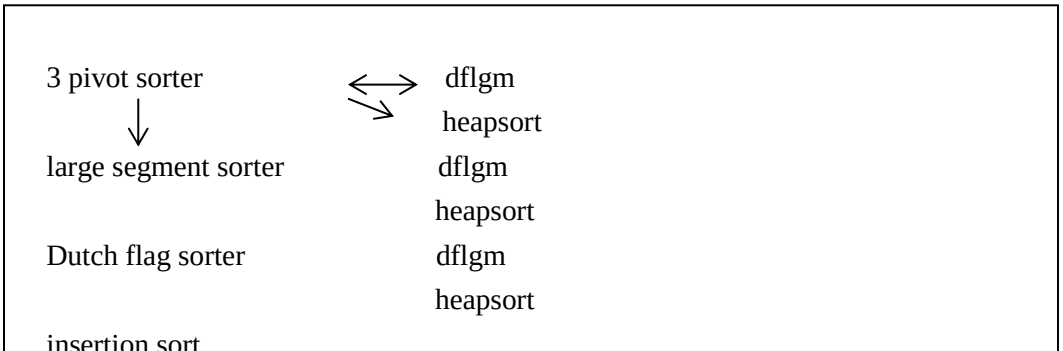
2 Summary of the sequential versions

Effective sorting using quicksort of native types and referenced types requires different algorithm due to the cost difference of the comparison operation: being 'free' for native types and arbitrary expensive for referenced types; see the cross tests that confirms this insight in [de Champeaux].

The control flow of the sequential versions is shown in Figure 1. Pivots are constructed in all members at the left hand side except in insertion sort. The Dutch Flag sorter uses the ninether – taking the medians of up to nine members of a segment. The other members assemble in the middle of a segment a small probe segment using pairwise swapping of elements equally spaced from the segment. Sorting the probe allows extracting one or three pivots. A check is made that the probe can be used for initialing the sub-segments before partitioning starts so that they can act as sentinels for supporting fast loops. Otherwise the segment is 'suspect' and may have too many equal elements. If so, the segment is delegated to *dflgm*. That module uses an array layout with two gaps while the elements during partitioning satisfy the invariant:

less than the pivot – *gap* – equal to the pivot – *gap* – larger than the pivot

This arrangement guarantees that constant segments are handled with linear complexity. The heapsort member is part of the hybrids to guarantee O(NlogN) complexity using the Musser defense [Musser].



The 1-pivot large segment module C2LR/cut2lr uses the [Edelkamp] technique for moving elements around. Instead of the traditional way of pairwise swaps, the indices of elements to be 'thrown over the wall' to the right are registered in a small (200 size) auxiliary array. Similarly for elements at the right side that must be transferred to the left. Bulk swaps move over the elements in the two auxiliary arrays. This reduces cache failures. Some of our hybrid members use the single element moving technique instead of pairwise/ bulk swapping. In short it entails two variables *x* and *y* for which the sub-segment is known where *x* must be added. Copy in *y* a gap element adjacent to the target sub-segment. Drop *x* in the gap location, assign *y* to *x*, determine the new target sub-segment for *x* and repeat.

Table 1 summarizes features of the sequential versions that are used by the parallel versions when a sub-segment is small.

Name	# pivots	#gaps	Moving elements	Control/ coding
C2LR/cut2lr	1	1	bulk swap	traditional
C4/cut4	3	2	single element	finite state machine
C7/cut7	3	1	re-relocation	traditional
CD4/cut4d	3	2	traditional	traditional

Table 1. Properties of the sequential versions

3 Design pattern using the C pThread package

This package allows specification of any number (within reason) of threads. The design pattern for parallelization of the operations assumes the notion of a task that contains the arguments for the partitioning of an array segment: the array, the segment boundary indices, the comparison function and a depth-limit. The latter argument is used to ascertain whether a delegation to heapsort is necessary to guarantee the O(NlogN) complexity. Tasks are stored on a stack. An addition is protected within a lock- and unlock-mutex region. A signal is also send inside the region (after the push) to all waiting threads to notify them of the addition , if any. All threads are created with the mandate to execute a specific *sortThread* procedure. Example code of this procedure is in the Appendix sortThread. It contains a loop in which a thread is forced to wait when the stack is empty and the number of waiting threads is less than the number of threads. A waiting thread will be activated when an item is added to the stack. When the

stack is empty and the number of waiting threads is equal to the number of threads the thread will terminate after it has notified the waiting threads, if any. A thread that has succeeded to pop a task will subsequently obtain the arguments from the task to launch a partitioning operation.

Partitioning the first segment in parallel is using a different 'trick'. Five elements are selected from the segments, are sorted through a network and are reassigned. Through additional swaps the aim is to obtain the configuration as in Figure 2 with the pivot T.

```
|----------------------------------------|----------------------------------------|
lo                                      e3                                      hi
A[lo] < T                            A[e3] = T                            T <= A[hi]
```

Figure 2. Target configuration for partitioning the first segment in parallel.

If this configuration cannot be obtained, partitioning of the first segment in parallel is abandoned, a task is created for the *lo-hi* segment, which is put on the stack and subsequent processing is launched. Otherwise two tasks are created for partitioning the left and right sub-segments. Two threads are created for these two tasks, which yield the configuration in Figure 3.

```
          LL            LR            RL            RR
|----------------------|------------------|------------------|------------------|
lo                  i1              e3              i2                  hi
A[lo] < T                        A[e3] = T                        T <= A[hi]
```

Figure 3. The sub-segments LL and RL have elements less than the pivot and the elements in LR and RR have elements greater or equal than the pivot.

After swapping the elements of the smallest one of LR and RL to the other side we obtain two segments for which we create two partitioning tasks for the left and right sub-segments, add these to the stack, and launch further processing by creating new threads,

4 Replacing recursive calls and tail iteration & sorting of the probe array

Assume that we obtain in the sequential case with one pivot two partitions A and B. To avoid stack overflow we proceed with:

```
if |A| <= |B| then { recurs on A and tail iterate on B }
else { recurs on B and tail iterate on A }
```

The parallel case is handled with:

```
if |A| <= |B| then { put B on the stack and tail iterate on A }
else { put A on the stack and tail iterate on B }
```

The Appendix 'Recursive calls and tail iteration' shows example code.

We obtain with 3 pivots the sub-segments layout with four regions:

```
A B C D
```

The parallel case is handled with:

```
if |A B| <= |C D| then {
    if |C| <= |D| then { put D on the stack; put C on the stack }
    } else { put C on the stack; put D on the stack }
    put B on the stack, tail iterate on A // their order is irrelevant
```

} else proceed similarly when |C D| < |A B|

Sorting of the probe array is done by a sequential sorter in the sequential versions. Replacing that sorter by a parallel sorter was a mistake for which recovery took too long; the sorter's result is needed immediately.

5. Correctness

Correctness of our versions depends on two different components: the parallel versions of the codes of the sequential versions and the code based on the pThread package that manages the partitioning tasks. The former reduces to the question of the correctness of the sequential version (which we have discussed in [de Champeaux]) because the only difference is that recursive calls and tail iterations are replaced by other tail iterations and by the execution of tasks previously stored on a stack.

The correctness of the *sortThread* procedures is challenging indeed due to the usage of lock- and unlock-mutex, conditional signaling and conditional wait operations. One needs more than Hoare logic for this challenge. A treatment that goes further than our narrative in section 3 is beyond the scope of this paper.

6 The benefit of parallelizing the first partitioning task

It is relatively easy to enable and disable parallelizing the partitioning of the first segment. The timing ratio advantage of parallelizing for the 1-pivot C2LR/cut2lr is 1.9%. The advantage for the three 3-pivot versions ranged from 4.3% to 5.4%.

7 Conformance to O(NlogN) & the timings when the thread count increases

Our algorithms have worse case complexity of O(NlogN). In this section we measure and check whether the performance ratio of 16Mlog(16M)/Mlog(M) =19.2 holds for our versions for the # threads ranging from 1 to 4. The table elements with time data contain also the ratio of thread K/ thread 1. The ideal ratios would be 0.5, 0.33 and 0.25. This data was obtained on the AMD desk top machine. The time entries are seconds. Timings in **bold** are the best against the other versions.

7.1 1 pivot C2LRp

# threads	1M time	16M time	Ratio
1	**0.249** | 1	**6.02** | 1	24.2
2	**0.155** | 0.62	**3.87** | 0.64	25.0
3	**0.124** | 0.50	**3.11** | 0.52	25.1
4	**0.107** | 0.43	**2.86** | 0.47	26.7

Table 2 Measured ratio of: 16M time/ 1M time for C2LRp.

7.2 3 pivot C4p

# threads	1M time	16M time	Ratio
1	0.254 | 1	6.31 | 1	24.8
2	0.180 | 0.71	4.22 | 0.67	23.4
3	0.142 | 0.56	3.31 | 0.52	23.3
4	0.121 | 0.48	2.87 | 0.46	23.7

Table 3 Measured ratio of: 16M time/ 1M time for C4p

7.3 3 pivot C7xp

# threads	1M time	16M time	Ratio
1	0.267 | 1	6.74 | 1	25.2

2	0.160 | 0.60	3.97 | 0.59	24.8
3	0.128 | 0.48	3.11 | 0.46	24.3
4	0.110 | 0.41	2.75 | 0.41	25.0

Table 4 Measured ratio of: 16M time/ 1M time for C7xp.

7.4 3 pivot CD4p

# threads	1M time	16M time	Ratio
1	0.306 | 1	7.27 | 1	23.7
2	0.180 | 0.59	4.20 | 0.58	23.3
3	0.142 | 0.46	3.31 | 0.45	23.3
4	0.121 | 0.39	2.88 | 0.40	23.8

Table 5 Measured ratio of: 16M time/ 1M time for CD4p

7.5 Preliminary observations

The expected O(NlogN) complexity ratio for 1M and 16M arrays is surpassed uniformly by the 1-pivot version and by the three 3-pivot versions. The measured time for 16M arrays is approximately a factor of 1.3 over the required time. Increasing the thread count shows also slowdowns against the ideal ratios. We conjecture that cache failures increase when the array size increases and when the number of locations where the array gets modified increases. Strangely enough CD4p is the slowest while having the best ratios for the threads 2, 3 and 4. We reported in [de Champeaux] that the 1-pivot sequential C2LR/cut2lr version outperformed the other 3-pivot versions on the AMD machine. The parallel version outperforms the others again on this machine.

8 Performance data of 1- and 3-pivot versions with multiple threads using 4 different machines

This section has timing data on 16M arrays for the different versions on four different machines. The best timing in a column is in **bold**.

Version	1 thread	2 threads
C2LRp	**6.00**	**3.87**
C4p	6.34	4.24
C7xp	6,80	3.98
CD4p	7.27	4.26

Table 6. Tests on AMD Win 8

Version	1 thread	2 threads
C2LRp	11.8	7.39
C4p	**9.95**	7.57
C7xp	10.0	**7.27**
CD4p	11.0	8.44

Table 7. Tests on I3 machine Win7

Version	1 thread	2 threads
C2LRp	8.93	5.84
C4p	8.01	5.50
C7xp	**7.87**	**5.13**
CD4p	8.53	5.50

Table 8. Tests on I3 machine Win10

Version	1 thread	2 threads
C2LRp	7.53	5.12
C4p	7.39	4.69
C7xp	**7.01**	**4.30**
CD4p	7.57	4.71

Table 9. Tests on I5 machine Ubuntu

9 Discussion of the findings and conclusion

Our design pattern for parallelizing applied successfully to our sequential versions and required only minimal modifications. Replacing recursion and tail iteration has been explained in section 4. We suspected earlier that the claim of O(NlogN) complexity was 'breached' when the array sizes increase. The data in section 7 confirms our suspicion. Similarly we had suspected that increasing the thread count will not provide the expected speed ups. The data in section 4 confirms this suspicion.

Still all versions were faster with more threads on the four machines we tested. However, time testing our 1- and 3-pivot versions on multiple machines has shown *again* that rank orders vary – as we had observed for the sequential versions in [de Champeaux]. If we assume that all four machines have equal 'votes' then we can declare that C7xp is the 'winner'. If instead we assume that the AMD machine is most representative for equipment where 'real' sorting is occurring, we are back at the possibility that the 1-pivot C2LR/cut2lr is still the winner. Hence we request again to run our tests on 'serious' equipment.

Quicksort was created over 60 years ago. It was originally a non-hybrid. It became and was a hybrid with only insertion sort for a long time. The current versions have minimally 4 members. This suggests that adding more members could be an enhancement; this would also increase the layer count. The insertion sort member of the hybrids is called the most often and an improvement would be helpful. The Isort file has three versions. A hybrid of them could be an opportunity.

Battling cache failures more could be another venue to explore. A correctness proof for the task manager using the C pThread package would be 'fun'. Considering quicksort being a victim of the memory hierarchy could lead to 'fresh anger': a trigger for creativity. Rethinking the hardware was already done in 2003, [Tsigas]. Its title: "A Simple, Fast Parallel Implementation of Quicksort and its Performance valuation on SUN Enterprise 10000" should be inspiring.

We repeat that the quicksort wiki needs an update. The worst case complexity becomes O(NlogN) with the Musser defense, and the 1997 paper by David Musser needs to be covered, [Musser].

PS This paper may be sobering for those who believe that parallelism will be the way forward when the processors cannot be made faster.

References

[9.4 Quicksort] http://users.atw.hu/parallelcomp/ch09lev1sec4.html

[de Champeaux] de Champeaux, D., "Hybrid Quicksort for Referenced Items with 1 and 3 pivots", submitted 2023.

[Edelkamp] Edelkamp, *Stefan; Weiß, Armin (22 April 2016). "BlockQuicksort: How Branch Mispredictions don't affect Quicksort".*
https://arxiv.org/abs/1604.06697

[GitHub] https://github.com/ddccc/C7p

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

[SampleSort] Frazer, W. D.; McKellar, A. C. (1970-07-01). "Samplesort: A Sampling Approach to Minimal Storage Tree Sorting". *Journal of the ACM*. **17** (3): 496-507.

doi:10.1145/321592.321600. S2CID 16958223.

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

Appendix sortThread

We provided the annotated code for the sortThread1 procedure that is used for scheduling the threads for the module Cut2lr2.

```
void *sortThread1(void *AAA) { // AAA is not used
  struct task *t = NULL;
  for (;;) {
    pthread_mutex_lock( &condition_mutex2 );
       sleepingThreads++;
       while ( NULL == ( t = pop(ll) ) &&      // attempt to fetch a task
              sleepingThreads < NUMTHREADS ) {
           // wait when no task obtained and actions are ongoing
           pthread_cond_wait( &condition_cond2, &condition_mutex2 );
       }
       if ( NULL != t ) sleepingThreads--;
    pthread_mutex_unlock( &condition_mutex2 );
    if ( NULL == t ) { // terminate but notify other threads first
      pthread_mutex_lock( &condition_mutex2 );
      pthread_cond_signal( &condition_cond2 );
      pthread_mutex_unlock( &condition_mutex2 );
      break;
    }
    // fetch the arguments of the task
    void **A = getA(t);
    int n = getN(t);
    int m = getM(t);
    int depthLimit = getDL(t);
    int (*compare)() = getXY(t);
    free(t);
    // launch the partitioning task:
    cut2lrpc(A, n, m, depthLimit, compare);
  }
  return NULL;
} // end sortThread1
```

Appendix Recursive calls and tail iteration

We show here the sequential and parallel codes of C2LR and C2LRp. The commented out lines are the sequential code. Sequential codes have recursive calls; the parallel codes have additions to a stack.

```
    if ( (I - lo) < (hi - J) ) { // smallest one first
      // cut2lrpc(A, lo, J-1, depthLimit, compareXY);
      // lo = J;
      addTaskSynchronized(ll, newTask(A, J, hi, depthLimit, compareXY));
      hi = J-1;
      goto Start;
    }
    // cut2lrpc(A, J, hi, depthLimit, compareXY);
    // hi = J-1;
    addTaskSynchronized(ll, newTask(A, lo, J-1, depthLimit, compareXY));
    lo = J;
```

```
goto Start;
```