Fast parallel sort on x86 and A64fx

Jun Makino

Internal Seminar May 25, 2022

Talk Summary

- I have discussed a bit about parallel samplesort(Sept 22, 2021) and SIMD quicksort (Mar 3 2022).
- Combination of them worked well, but parallel performance was not ideal.
- minimized the memory access of single-core sort routine using the idea of samplesort.
- Achieved pretty good performance on both x86 and A64fx.
- Will be tested with FDPS.

Talk plan

1. Parallel sort
2. Scalability of parallel sort
3. Theoretically better approach
4. Our implementation
5. Performance
6. Summary

Parallel sort

- **In this talk, I discuss intra-node parallel sort.**
- **The performance of `std::sort` is pretty good for single-core sort, but we need multi-core sort (for FDPS, in particular on A64fx).**
- **C++17 supports "Parallel STL" which you can use on GCC9 or later, and of course on icc, but not on Fujitsu C++ compiler for A64fx.**

Scalability of parallel sort

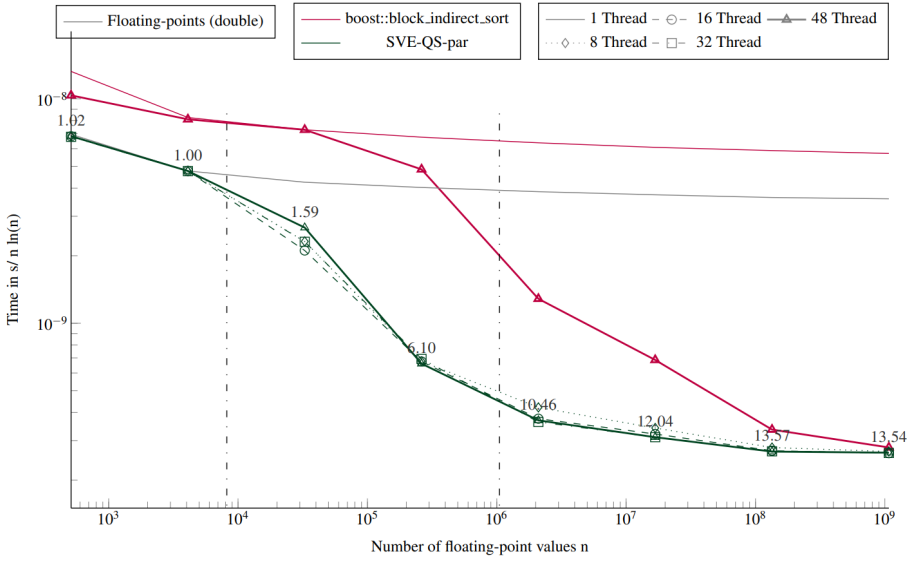


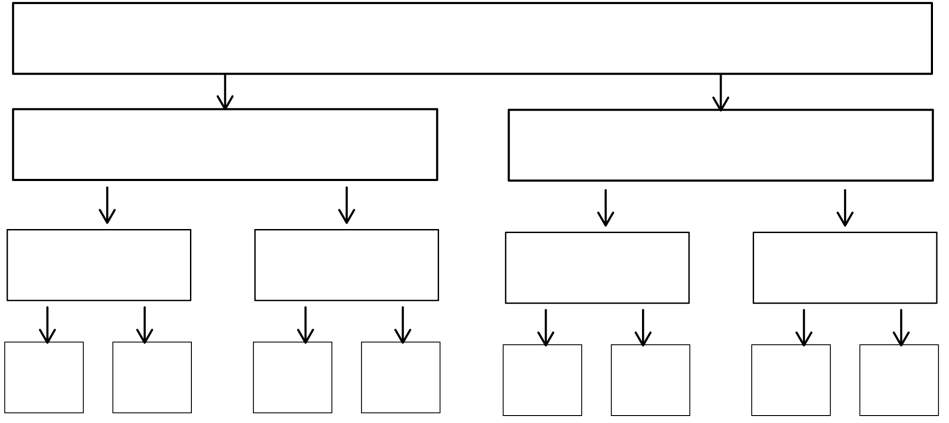
Figure 7 Execution time to sort large arrays (in parallel). Execution time divided by n ln(n) to sort in parallel arrays filled with random values with sizes from 512 to ∼109 elements. The execution time is obtained from the average of 5 executions with different values. The speedup of the parallel SVE-QS-par against the sequential execution is shown above the lines for 16 and 48 threads. The vertical lines represent the caches relatively to the processed data type (- for the integers and .-. for the floating-points).
Full-size DOI: 10.7717/peerjcs.769/fig-7

A fast vectorized sorting implementation based on the ARM scalable vector extension (SVE), B. Bramas, 2022, Figure 7

Sorting FP64 numbers. 13 times faster than single-core with 16, 32, and 48 threads.

Algorithm: Parallel Quicksort.

Parallel Quicksort



At each stage, the array is divided to (roughly) two halves.

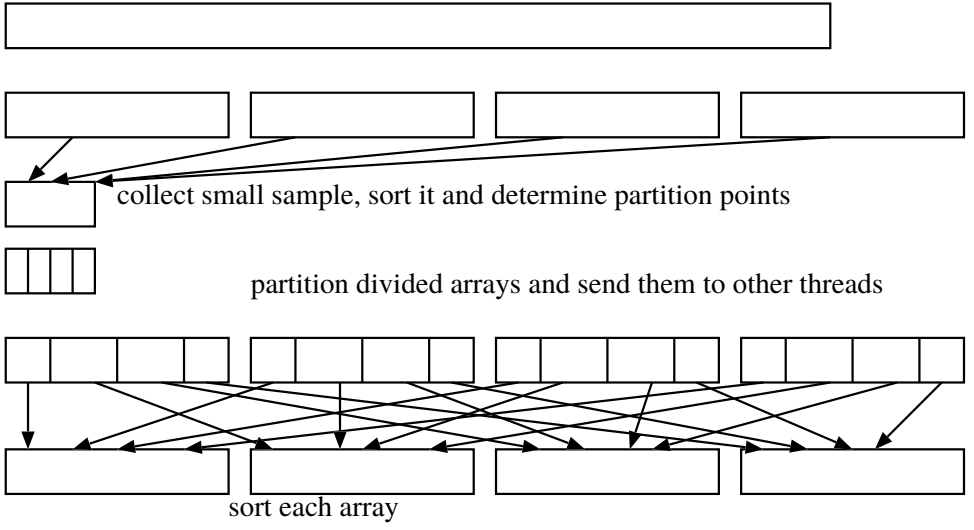
1, 2, 4, ... threads can be used at each stage.

Speedup factor is limited by first few stages which are not well parallelized.
Even if we have 20 stages (10^6 elements) and infinitely many cores, speedup will be limited to $10 (= 20/(1 + 0.5 + 0.25 + ...)$
Parallel merge sort suffers the same problem.

Theoretically better approach

- **Parallel sample sort**
- **Parallel merge sort (not discussed today)**

Parallel sample sort



Sample sort: Generalization of quicksort

- **partition to n parts ($n > 2$)**
- **First stage: $n =$ the number of threads**
- **Use sampling to determine partition points**

Parallel version: ALL steps except for the sorting of small sample array are parallelized

Our implementation

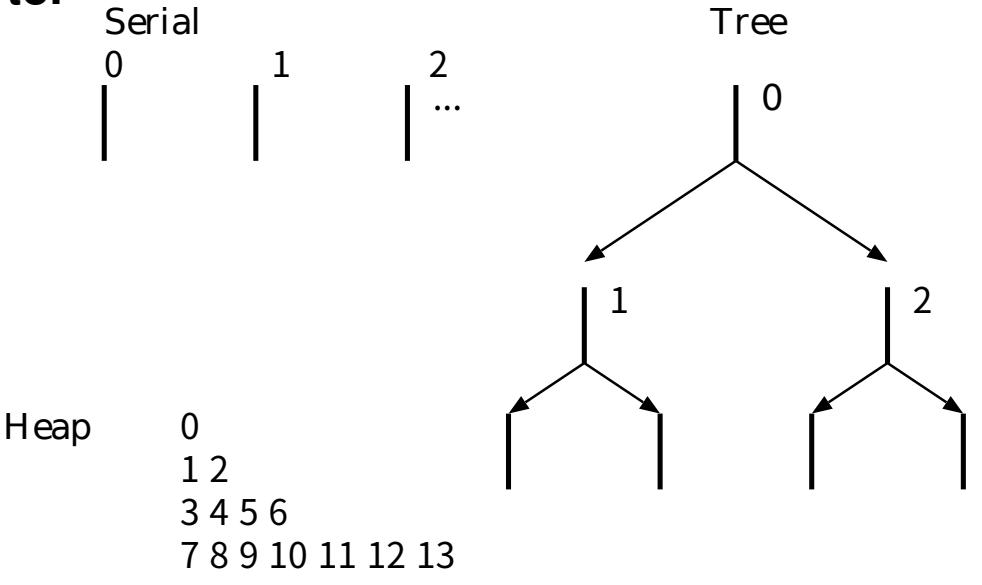
- **https://github.com/jmakino/sortlib**
- **Use key-index array for sort.**
- **Fast n-way partitioning using heap**
- **Sample sort also for single-core sort**

Key-index sort

- What we want to sort: particles, sort key: Morton key
- Sorting particles directly would cause lots of memory copy
- Instead, first create an array of struct of key + original array index, sort it according to key and reorder the particle array using the sorted index.
- Parallel reordering requires two loops (reorder and copy back)

Fast n-way partitioning using heap

For each element, we need to determine which of n partitions it should go to.



- **Naive implementation: $O(n)$ operations.**
- **Tree-based implementation: $O(\log n)$ operations.**
- **Heap-based: $O(\log n)$ but no pointer access.**

Keep tree data in 1d array. Use loop structure to access tree.

Code (not quite optimal...)

```
bool unfilled = false;
for(ilevel=0;ilevel <nlevel; ilevel++){
    if (ipart > n-1) {
        unfilled = true;
    }else{
        int inc = data <= tree[ipart]? 0:1;
        ipart = ipart*2+1+inc;
    }
}
int offset = (1<<nlevel)-1;
if (unfilled){
    offset = offset -n -1;
}
return ipart - offset;
```

Sample sort also for single-core sort

- **Our initial implementation used `std::sort` for single-core sort**
- **It turned out that parallel performance degrades for large array.**
- **The bandwidth of the main memory for the quicksort used in `std::sort` limits the parallel speedup.**
- **For single-core sort part, when the array is large, first divide ot to smaller blocks (size 16k) using samplesort algorithm, so that new blocks fit to L2 cache (and to L1 cache after a few steps)**

Performance

Size of test data struct: 160 bytes, sort key size: 64 bits (128 bits also implemented). 1M elements

A64fx performance

# threads	time(s)
1(std::sort)	0.34
2	0.138
4	0.081
6	0.063
8	0.047
12	0.041
24	0.019
36	0.013
48	0.011

Xeon Gold 6140 performance (cplab0)

# threads	parallel stl time(s)	samplesort time(s)
1(std::sort)	0.172	0.172
2	0.180	0.102
4	0.095	0.056
9	0.039	0.031
18	0.027	0.025
36	0.023	0.020

Performance

- **On A64fx, speedup is observed up to 48 threads. The sSpeedup factor for 48 threads is around 30.**
- **On Xeon, even with 36 threads the sppeedup factor is around 8, but generally faster than parallel STL.**
- **Single-core performance of A64fx is one half of that of Xeon.**
- **A rather rare example that A64fx is actually faster than Xeon for non-trivial parallel operation.**

Summary

- I have discussed a bit about parallel samplesort(Sept 22, 2021) and SIMD quicksort (Mar 3 2022).
- Combination of them worked well, but parallel performance was not ideal.
- minimized the memory access of single-core sort routine using the idea of samplesort.
- Achieved pretty good performance on both x86 and A64fx.
- Will be tested with FDPS.