A64fx 上での高速並列ソート

牧野淳一郎

富岳計算宇宙惑星・計算資源利用に関する課題内ワークショップ

2022//6/28

概要

- **A64fx** 上でのソート **(**キーが **128** ビットくらいでもうちょっと大きい構造体をソート**)** が並列でそこそこ速くなるような実装を書いた。
- スレッド並列化はサンプルソート、スレッド内はサンプルソートと **SIMD** 化されたクィックソートとバイトニックの組合せ。
- そこそこ性能でるようになった。論文になってる他のよりは速い。
- **FDPS** にもいれる予定である。

Talk plan

1. Parallel sort
2. Scalability of parallel sort
3. Theoretically better approach
4. Our implementation
5. Parallel quicksort and bitonic sort
6. Performance
7. 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

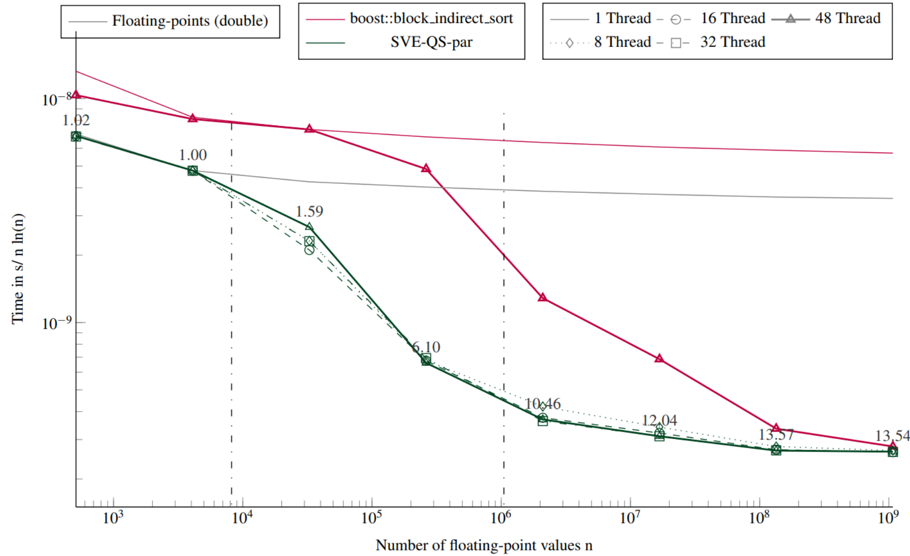


Figure 7 Execution time to sort large arrays (in parallel). Execution time divided by n ln(n) to sort in parallel arrays filled with random values with sizes from 512 to ∼109 elements. The execution time is obtained from the average of 5 executions with different values. The speedup of the parallel SVE-QS-par 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).

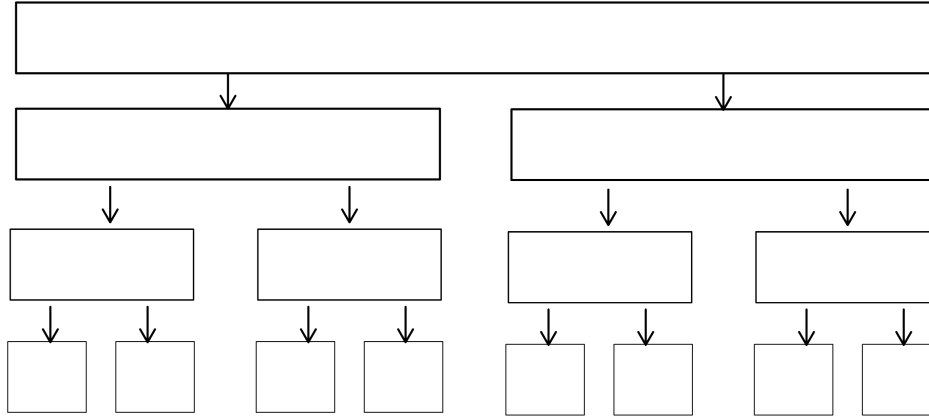
Full-size DOI: 10.7717/peerjcs.769/fig-7

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.

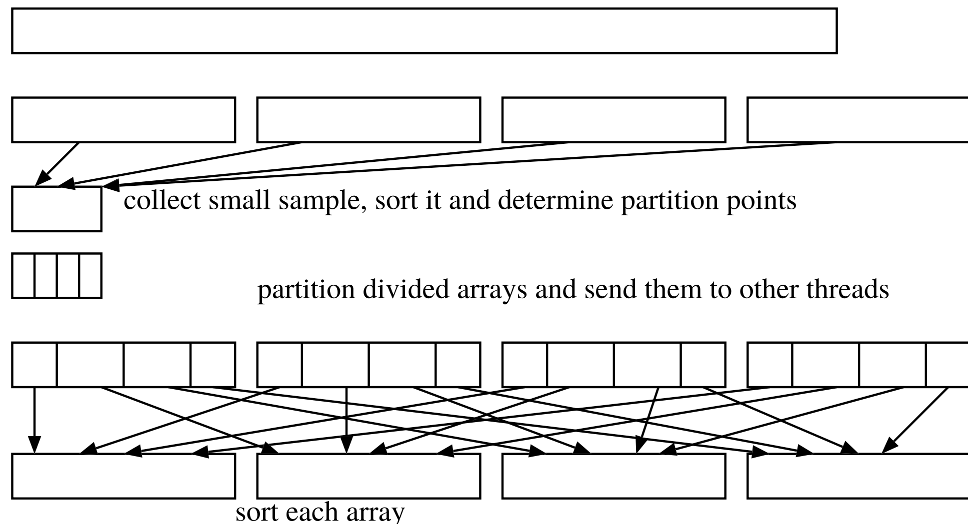
1, 2, 4, ... threads can be used at each stage.

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

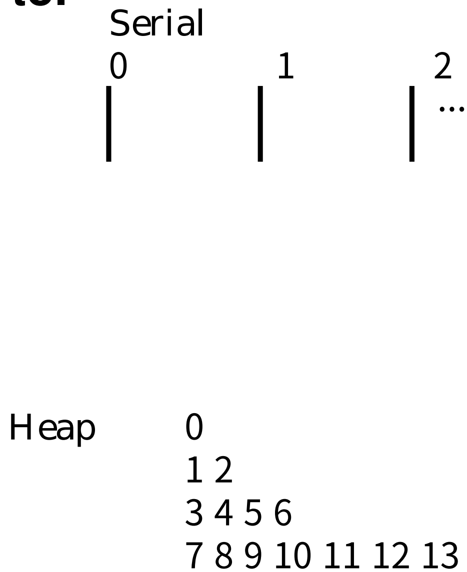
- https://github.com/jmakino/sortlib
- Use key-index array for sort.
- Fast n-way partitioning using heap
- Sample sort also for single-core sort

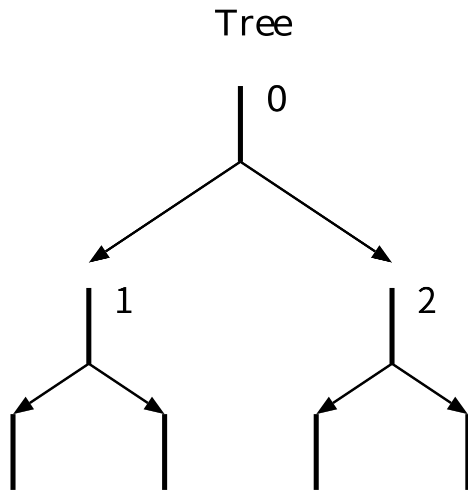
Key-index sort

- 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 it to smaller blocks (size 16k) using samplesort algorithm, so that new blocks fit to L2 cache (and to L1 cache after a few steps)**

SIMD Quicksort

- Sorting is used in many, many applications
- We use sorting to construct trees in FDPS. This is actually the most time-consuming part of the calculation other than tree traversal and interaction calculation.
- However, what are available are not quite the fastest.
  - std::sort is a sequential quicksort (actually an introsort, a combination of quicksort and heapsort)
  - Parallel sorts are not quite fast
  - SIMD instructions are not used

Aren't there existing implementations?

- There are many papers on implementation of fast sorting algorithms using SSE, AVX, AVX2, AVX512 and even SVE.
- However, almost all of them cannot be used as a replacement of std::sort.
  - They just sort 32-bit integers
  - What we want to do is to sort, for example, particles using the Morton key.
  - Actually, with usual interface of std::sort [or qsort(3)], we cannot use SIMD sort. They only supply a comparison function.
- We can still make generic interface if we provide a function to generate integer keys from data to be sorted (my samplesortlib `https://github.com/jmakino/sortlib` uses this interface)

Basics of Quicksort

1. We have array a with n elements.
2. Pick one “pivot” value from these n elements
3. divide a into two (or three) parts. The left part contains all values smaller than pivot, the right part larger than, and the middle part equal to.
4. Apply steps 2 and 3 recursively to left and right part. If the number of element is one, do nothing.

An example

```
input:  2 5 1 4 9 5 2 1 6 1

pivot: 2:   1 1 1   2 2   5 4 9 5 6

pivot 1: 1 1 1, (2 2 part finished)  pivot:5  4  5 5  9 6

(1 1 1 part finished)                         pivot 9 : 6 9

result: 1 1 1 2 2  4 5 5 6 9
```

Example Source code

(not quite sure why and how this works, though)

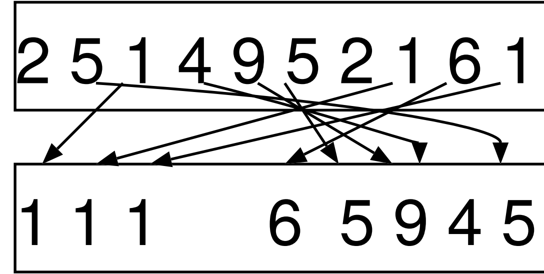
```
void sort_int64_array(int64_t* r, int lo, int up)
{
    int i, j;
    int64_t tempr;
    while ( up>lo ) {
        i = lo;
        j = up;
        tempr = r[lo];
        /*** Split data in two ***/
        while ( i<j ) {
            for ( ; r[j]> tempr; j-- );
            for ( r[i]=r[j]; i<j && r[i]<=tempr; i++ );
            r[j] = r[i];
        }
        r[i] = tempr;
```

Example Source code(continued)

```
        /*** Sort recursively, the smallest first ***/
        if ( i-lo < up-i ) {
            sort_int64_array(r,lo,i-1);  lo = i+1;
        }else{
            sort_int64_array(r,i+1,up);  up = i-1;
        }
    }
}
```

This code requires no additional memory (might have been important in 1960s, when the main memories of computers were small

SIMD partition algorithm



Here, we start from a simple algorithm which require additional working memory.

1. **We have array a with n elements.**
2. **Pick one “pivot” value from these n elements**
3. **prepare an empty array b of size n.**
4. **For each element of array a, if it is smaller than the pivot, put it in the leftmost free location in b. If larger, rightmost. If equal, do nothing**
5. **copy left and right parts of array be back to array a.**
6. **Apply steps 2-5 recursively to the left and right part of array a.**

SIMD partition algorithm(continued)

We can use SIMD instructions to implement the algorithm described in the previous slide. Within an SIMD word,

1. Mark values less than the pivot
2. Move these values to the left side (in a separate SIMD word)
3. Mark values larger than the pivot
4. Move these values to the right side (in yet another separate SIMD word)
5. copy smaller values to leftmost free locations of array b
6. copy larger values to rightmost free locations of array b

All steps can be done using SIMD instructions

Current implementation with SVE

https://github.com/jmakino/simdsort

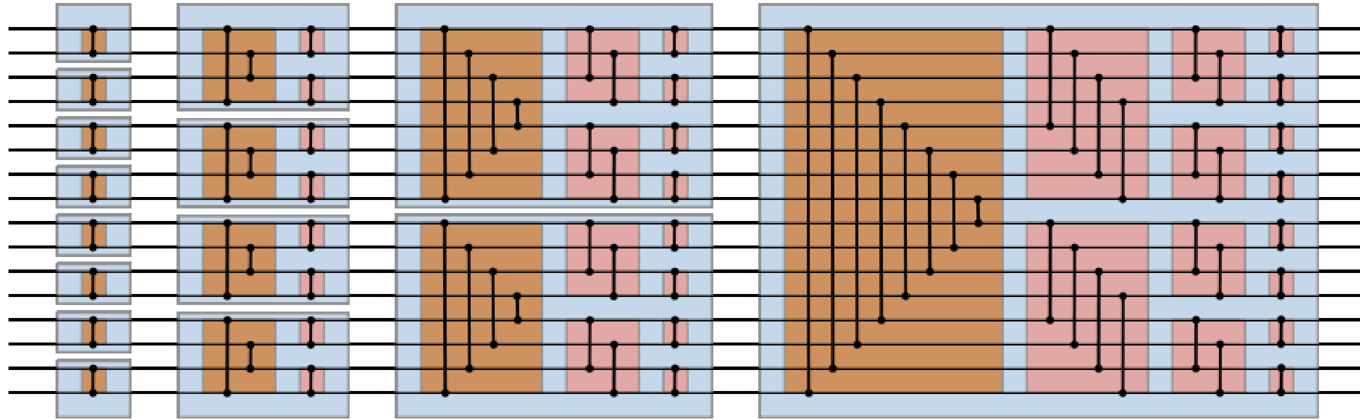
```
for(int i8=0;i8<n;i8+=nsve){
    svbool_t pmask= svwhilelt_b64(i8,n);
    svint64_t val =svld1_s64(pmask, work+i8);
    svbool_t maskl =  svcmpgt_s64(pmask, pivotv,val);
    svbool_t masku =  svcmpgt_s64(pmask, val, pivotv);
    int dl = svcntp_b64(pmask, maskl);
    int dh = svcntp_b64(pmask, masku);
    svst1_s64(pmask, data+l+1,  svcompact_s64(maskl, val));
    svbool_t maskustore = svwhilelt_b64(0,dh);
    svst1_s64(maskustore, data+h-dh,  svcompact_s64(masku, val));
    l+=dl;
    h-=dh;
}
```

Remarks

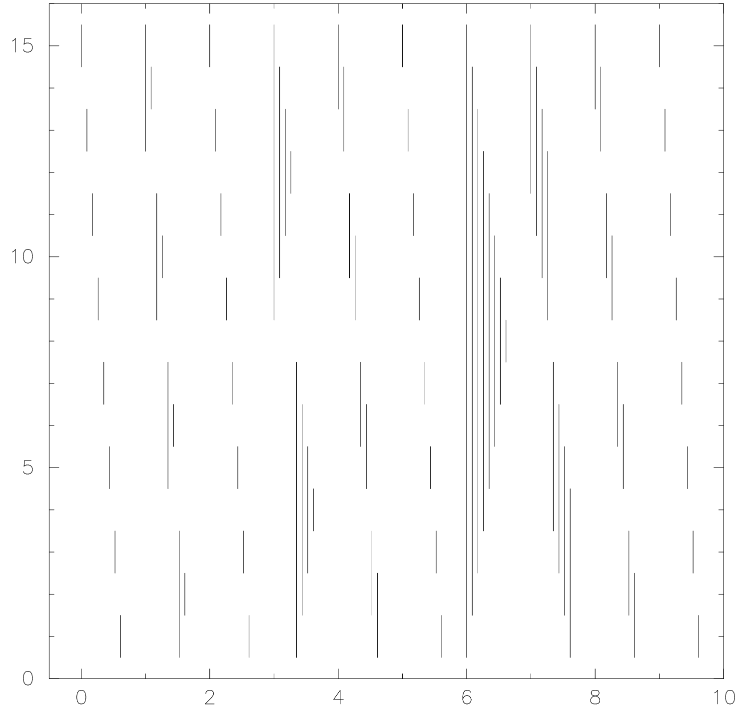
- The size of array (in particular during recursion) is not always an integer multiple of the SIMD width. Therefore we need to take care of the residual part. With SVE this can be done through the use of mask.
- SVE has very rich set of opertions and thus the code looks simple and runs fairly efficiently.
- Actual code is for three 64-bit int values and thus a bit complicated.

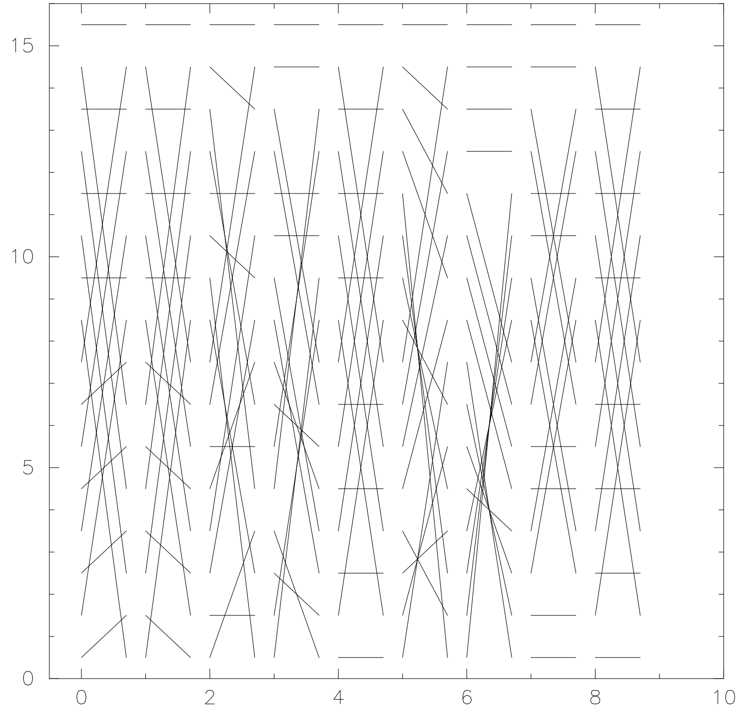
Bitonic sort

- One of “sorting networks” — Can be implemented with SIMD instructions without any conditional branch.
- Calculation cost: $O(n(\log n)^2))$



Bitonic sort cont'd





Sorting network (left) and data move within two SIMD registers(right)

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

- **A64fx** 上でのソート (キーが **128** ビットくらいでもうちょっと大きい構造体をソート) が並列でそこそこ速くなるような実装を書いた。
- スレッド並列化はサンプルソート、スレッド内はサンプルソートと **SIMD** 化されたクィックソートとバイトニックの組合せ。
- そこそこ性能でるようになった。論文になってる他のよりは速い。
- **FDPS** にもいれる予定である。