Master Of Science in Informatics at Grenoble
Parallel, Distributed, and Embedded Systems

Parallel Systems

(1st Session)

Vincent Danjean, Arnaud Legrand

January 26th, 2015

Important information.
Read this before anything else!

▷ Any printed or hand-written document is authorized during the exam, even dictionaries. Books are not allowed though.
▷ Please write your answers to each problem on separate sheet of papers.
▷ The different problems are completely independent. You are thus strongly encouraged to start by reading the whole exam. You may answer problems and questions in any order but they have to be written in the order on your papers.
▷ All problems are independent and the total number of points for all problems exceeds 20. You can thus somehow choose the problems for which you have more interest.
▷ The number of points alloted to each question gives you an indication on the expected level of details and on the time you should spend answering.
▷ Question during the exam: if you think there is an error in a question or if something is unclear, you should write it on your paper and explain the choice you made to adapt.
▷ The quality of your writing and the clarity of your explanations will be taken into account in your final score. **The use of drawings to illustrate your ideas is strongly encouraged.**

I. Parallel Quick Sort ($\approx$ 1h00)

Question I.1. ***Preliminary question***

We consider n independent tasks $t_1, \ldots, t_n$, with n very large. For $1 \leq i \leq n$, the task t_i performs w_i operations. The values w_i are unknown, but are assumed to be bounded by two constants c and c' : $c \leq w_i \leq c'$.

We consider a multicore machine with p identical cores.

a) *We first assign $\frac{n}{p}$ tasks to each core. Prove that the execution time T_p verifies $\frac{n}{p}c \leq T_p \leq \frac{n}{p}c'$. Exhibit a worst case for the w_i together with an assignment such that the execution time T_p is close to $\frac{c'}{c}\frac{\sum_{i=1}^{n} w_i}{p}$.*

b) *Write a implementation of a Parallel For Loop (in a language like Cilk, Athapascan/Kaapi, OpenMP, MPI...) such that the execution time T_p verifies $T_p \leq \frac{\sum_{i=1}^{n} w_i}{p} + O(\log n)$ with high probability: note that this time should be achieved whatever the values of the constants c, c' and the w_i are. Analyze the work and depth of this algorithm (justify very briefly).*

Algorithm 1 Par-Partition$(A[q:r], x)$

Input: An array $A[q:r]$ of distinct elements and an element x from $A[q:r]$

Output: Rearrange the elements of A[q : r], and return an index $k \in [q, r]$, such that all elements in $A[q : k-1]$ are smaller than x, all elements in $A[k+1:r]$ are larger than x, and $A[k] = x$.

```
 1: n ← r − q + 1
 2: if n = 1 then
 3:     return q
 4: end if
 5: Array B[0 : n − 1], lt[0 : n − 1], gt[0 : n − 1]
 6: parallel for i ← 0 to n − 1 do
 7:     B[i] ← A[q + i]
 8:     if B[i] < x then lt[i] ← 1 else lt[i] ← 0
 9:     if B[i] > x then gt[i] ← 1 else gt[i] ← 0
10: end parallel for
11: lt[0 : n − 1] ← Par-Prefix-Sum(lt[0 : n − 1], +)
12: gt[0 : n − 1] ← Par-Prefix-Sum(gt[0 : n − 1], +)
13: k ← q + lt[n − 1]
14: A[k] ← x
15: parallel for i ← 0 to n − 1 do
16:     if B[i] < x then
17:         A[q + lt[i] − 1] ← B[i]
18:     else if B[i] > x then
19:         A[k + gt[i]] ← B[i]
20:     end if
21: end parallel for
22: return k
```

Question I.2. ***"Par-Partition" Algorithm Analysis***

a) *Using the following entry $A = [9, 5, 7, 11, 1, 3, 8, 14, 4, 21]$ and $k = 8$, explain with schema how this program works*

b) *We recall that **Par-Prefix-Sum** can be implemented with a parallel algorithm with $O(n)$ operations and a depth of $O(log^2 n)$. We also consider that memory allocations without initialization are done in constant time.*

Analyze the complexity of each part of the provided algorithm, both in term of number of operations and depth

c) *Conclude about the global complexity of the algorithm (still both in term of number of operations and depth)*

Question I.3. ***Randomized Parallel QuickSort***

Algorithm 2 Par-Randomized-quickSort($A[q:r]$)

Input: An array $A[q:r]$ of distinct elements
Output: Elements of $A[q:r]$ sorted in ascending order

1: $n \leftarrow r - q + 1$
2: **if** $n \leq 30$ **then**
3:     sort $A[q:r]$ using any sorting algorithm
4: **else**
5:     select a random element x from $A[q:r]$
6:     $k \leftarrow$ **Par-Partition**($A[q:r], x$)
7:     **spawn Par-Randomized-QuickSort**($A[q:k-1]$)
8:     **Par-Randomized-QuickSort**($A[k+1:r]$)
9:     **sync**
10: **end if**

Give the complexity, both in term of number of operations and depth, of the "Par-Randomized-QuickSort" algorithm in the best case. Remark: you need to explain what is the best case but you do not need to prove this fact.

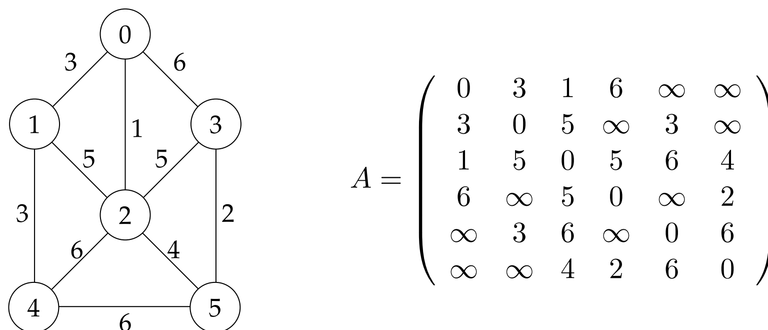
Question I.4. ***Data movements***

a) *What is the pattern of memory allocations if these algorithms?*

b) *Propose some ideas to improve performance of these algorithms in term of memory allocation and memory access*

II. Building spanning trees ($\approx$ 40 min)

In this problem, we consider a connected weighted undirected graph represented as a symmetric matrix A. If i and j are connected then $A[i,j] > 0$ and $A[i,j]$ denotes the weight of the connection between i and j. If i is not connected to j, then we assume that $A[i,j] = \infty$. The following figure depicts a simple but typical graph and its matrix representation.



$$A = \begin{pmatrix} 0 & 3 & 1 & 6 & \infty & \infty \\ 3 & 0 & 5 & \infty & 3 & \infty \\ 1 & 5 & 0 & 5 & 6 & 4 \\ 6 & \infty & 5 & 0 & \infty & 2 \\ \infty & 3 & 6 & \infty & 0 & 6 \\ \infty & \infty & 4 & 2 & 6 & 0 \end{pmatrix}$$

We propose to study how to parallelize Prim's algorithm that build a minimum spanning tree (MST) of a graph, i.e., a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

Prim's algorithm is a greedy algorithm that starts with an empty spanning tree. Intuitively, there are two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

Here is a pseudo code (in C style) of a sequential version of this algorithm.

```
 1| list prim_MST(int n , double A[][]) {
 2|   list_t MST = {};                             /* an empty list */
 3|   double d[n] = [inf, inf, .., inf];        /* inf means infinite */
 4|   int v[n] = [-1, -1, ..., -1];   /* v[j] is s.t A[v[j],j] = d[j] */
 5|   int r = 0;
 6|   for(step=0; step<n-1, step++) {
 7|     for(j=0; j<n; j++) {          /*     d[j] <- min(d[j],A[r,j]) */
 8|       if(A[r,j]<d[j]) {
 9|          d[j]=A[r,j];
10|          v[j]=r;
11|       }
12|     }
13|     r = index such that d[r] is minimal and non-zero;
14|     Add (r,v[r]) in MST;
15|   }
16|   return MST;
17| }
```

Question II.1. *Apply the previous algorithm to build a minimum spanning tree of the simple graph given earlier. You will detail the state of* `d`*,* `v`*,* `r`*, and* `MST` *at every step.*
(The goal of this question is to make sure you understand how the algorithm works so that you can answer the next question.)

Question II.2. *Propose a way to parallelize this algorithm on a ring of processors. You will detail how the different variables should be distributed among the processors and you will give a performance model for your algorithm.*

III. Efficient Sorting on a Recent Accelerator ($\approx$ 1h15)

The article that can be found in the appendix is an article published in a workshop of the Super-Computing conference in 2013. Some parts have been removed for clarity. This document reports investigations on efficiently implementing the sorting operation on the recently released Intel Xeon Phi. Unlike classical accelerators like GPU, this accelerator does not require major code rewriting. However, obtaining good performances can be challenging.

The goal of this section is to evaluate both your ability to quickly comment articles, figures and experimental protocol (question III.2) and to connect some of the fundamentals presented during the lecture with a recent research article (question III.1).

> **You should thus not try to understand all the details of this article. It is not a problem if some parts remain unclear. You should rather try to understand the most important parts.**
>
> **You should thus first carefully read the following questions before engaging yourself into the reading of the whole article.**

Question III.1. *After reading through the article explain:*

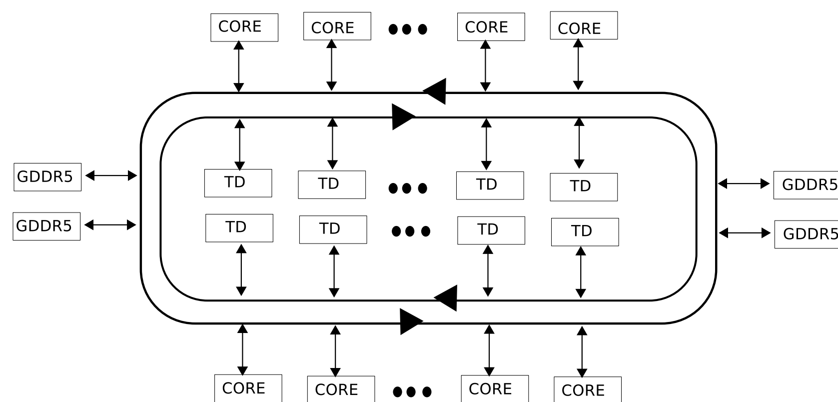
1. *Why do the authors focus on register sort (in a few lines)?*
2. *Comment and interpret the results given in Table 4 (at most 15 lines).*
3. *What are the main issues the authors have to face? (20 lines plus drawings if needed)*
4. *What is the general architecture of the whole algorithm and what do you think about it? (15 lines plus drawing)*

If you consider that some parts are unclear, do not hesitate to mention them and formulate hypothesis on what you understand or on how you think things may actually work.

Question III.2. *List all the aspects of the performance evaluation of Section 4 that you think should or could be improved.*

Appendix

Here is a short description of the Xeon Phi accelerator in addition to the article.

> The Intel Xeon Phi coprocessor is a many-core system based on the Intel MIC (Many Integrated Core) architecture. The current commercial Xeon Phi (5110P) has 60 simplified Intel CPU cores running at 1056 MHz and supports 4 threads per core with hyperthreading (thus, 240 threads in the die). Each core has a vector unit with 64 byte registers featuring a new vector instruction set known as Intel Initial Many Core Instructions (IMCI), also known as AVX-512, and which is described in the following article. The cache memory in each core is arranged in a 32 kb L1 data cache, 32 kb L1 instruction cache, and a private 512 kb L2 unified cache which is kept coherent by a distributed tag directory system (DTDs). Cores are arranged on a bidirectional ring bus that provides high scalability to which other components like memory controllers or tag directories are also connected.
>
> 
>
>
> The previous figure represents the basic architecture of the Xeon Phi including cores, bus, memory controllers and tag directories. There are 64 tag directories (TD) connected to the ring and the address-mapping to the tag directories is based on hash functions over memory addresses, leading to an even distribution around the ring. The memory controllers, also connected to the ring, provide access to the GDDR5 memory (8 GB of global memory). The coprocessor runs a simplified Linux-based OS in one of the cores. The main advantage of the Xeon Phi over other accelerators or coprocessors is that it provides the well-known x86 ISA and memory model, hence the programming effort is just focused on how to better exploit performance, but it can be done with known techniques and languages such as OpenMP or MPI. The Xeon Phi can also be used as a mere coprocessor in which the host offloads code to be accelerated, or as an independent unit that runs a whole application or that communicates in a symmetric manner with the host.

Register Level Sort Algorithm on Multi-Core SIMD Processors

Tian Xiaochen, Kamil Rocki and Reiji Suda
Graduate School of Information Science and Technology
The University of Tokyo & CREST, JST
{xchen, kamil.rocki, reiji}@is.s.u-tokyo.ac.jp

ABSTRACT

State-of-the-art hardware increasingly utilizes SIMD parallelism, where multiple processing elements execute the same instruction on multiple data points simultaneously. However, irregular and data intensive algorithms are not well suited for such architectures. Due to their importance, it is crucial to obtain efficient implementations. One example of such a task is sort, a fundamental problem in computer science. In this paper we analyze distinct memory accessing models and propose two methods to employ highly efficient bitonic merge sort using SIMD instructions as register level sort. We achieve nearly 270x speedup (525M integers/s) on a 4M integer set using Xeon Phi coprocessor, where SIMD level parallelism accelerates the algorithm over 3 times. Our method can be applied to any device supporting similar SIMD instructions.

Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: PROCESSOR ARCHITECTURES—*Single-instruction-stream, multiple-data-stream processors (SIMD)*

General Terms

Algorithms, Performance, Design

Keywords

SIMD, Parallel, Sort, Xeon Phi, Register, Irregular

1. INTRODUCTION

Multi-core architecture has been widely used in state-of-art processors. For example, NVIDIA's Tesla K20 GPU has 14 Stream Multiprocessors(SMX) and Intel's Xeon Phi has 60 cores. Xeon Phi supports 512 bits SIMD instruction i.e. AVX-512 which can process 16 integers (4 bytes) simultaneously. GPU employs a similar architecture: single-instruction-multiple-threads(SIMT). On K20, each SMX has 192 CUDA cores which act as individual threads. Even a desktop type multi-core x86 platform such as i7 Haswell CPU family supports AVX2 instruction set (256 bits). Developing algorithms that support multi-core SIMD architecture is the precondition to unleash the performance of these processors. Without exploiting this kind of parallelism, only a small fraction of computational power can be utilized.

Parallel sort as well as sort in general is fundamental and well studied problem in computer science. Algorithms such as *Bitonic-Merge Sort* or *Odd Even Merge Sort* are widely used in practice. However, implementing sort algorithm on SIMD processors efficiently, especially on Xeon Phi or x86 CPU remains a challenging job. Until now, many algorithms have been proposed for GPU or CPU. GPU version of the algorithm cannot be directly transplanted a processor supporting 512-bit wide vector instructions such as Xeon Phi. Nvidia GPUs have fast, explicitly manageable on-chip shared memory and their cores are more functional and execute individual threads concurrently with SIMT synchronization on hardware level. Programming Xeon Phi poses a different challenge - the instruction restrictions of memory accesses (discussed in Section 3.1) needed during SIMD sort or merge.

In this paper, we focus on developing parallel sort and merge algorithms only in register, under the constraints of memory accessing model within registers. We propose two in-register sort/merge algorithms which only take a constant number of registers (2 or 3) no matter how long SIMD instruction is. Then, we present theoretical and empirical analysis of the execution time. We chose AVX-512 and Xeon Phi as our main experiment environment as a extreme case of SIMD parallelism combined with restricted access patterns and limited bandwidth. We also implemented the algorithm on Intel E5-2670 with AVX instructions. The same algorithm can be derived for other SIMD processors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC13 November 17-21, 2013, Denver CO, USA
Copyright is held Copyright is held by the owner/author(s).
ACM 978-1-4503-2503-5/13/11
http://dx.doi.org/10.1145/2535753.2535762.

2. RELATED WORK

Comparison based sort is a more general sort algorithm. $\Omega(nlog(n))$ is the lower bound of its time complexity. Many theoretical results have been published in the field of parallel sorting. *Bitonic-Merge Sort*[3] or *Odd-Even Merge Sort*[8] are just two algorithms usually used in practice. They are also sorting networks traversed in $O(log^2(n))$ steps. Though $log(n)$-step algorithm has been invented[9], due to the large overhead it is hardly used in practice.

There is a large number of research papers which describe implementing parallel sort. We classify the algorithms into two types: *Register Level Sort* and *Multi-Core Level Sort*. This classification is made by the way of synchronization in processors. In register level, SIMD instructions play the role of cores. Synchronization is not necessary(GPU may need synchronization explicitly) since SIMD computation is naturally synchronized after each instruction. In the situation of Multi-Core level, communication happens in slow cache or RAM. The cost of synchronization becomes more expensive.

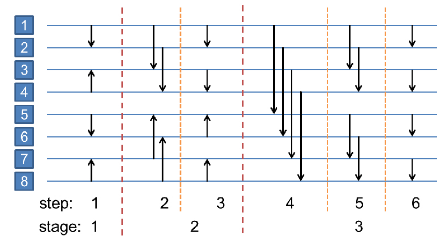


Figure 2: Bitonic-Merge sorting network of 8 element.

3. OUR APPROACH

First, let's define K as the number of elements that one vector instruction can handle and two available registers as $R1$ and $R2$. *Bitonic-Merge Sort* is a often used as a parallel

sort algorithm. Figure 2 shows the sorting network of 8 elements.

The number in the box represents the indices of element. An arrow represents a comparison of a pair of elements, storing the larger element where the arrow is pointing and the smaller element on the opposite side. Figure 3 shows an example of *Bitonic-Merge Sort*.

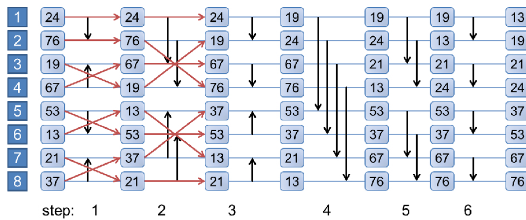


Figure 3: An example of Bitonic-Merge sort.

3.1 AVX-512 Capability of Comparison

To implement Bitonic-Merge Sorting network using recent Intel's vector extensions, it is necessary to ravel out what AVX-512 can do in a sort problem. On GPU, CUDA core seems to more flexible, since one core corresponds to one real thread from the programming point of view (CUDA, OpenCL and etc.). On the other hand, programming using AVX-512 is more like assembling instructions together in a procedural way. From the hardware perspective, GPU has on-chip shared memory which makes CUDA random accesses memory possible. Therefore memory accessing pattern of CUDA core is PRAM. When we consider AVX-512, vector register has to load data from main memory or cache before calculation in a vector-manner, meaning chunks of memory at once. Many operations have to be executed between two registers. So memory access pattern is not PRAM.

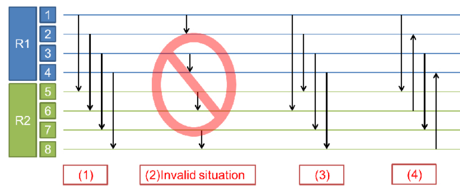


Figure 4: Capability of AVX-512 instruction set

Let's assume that some data is loaded into registers $R1$ and $R2$. Figure 4 shows the capability of AVX-512. Notice that the second operation is invalid, because elements in the same register can not be compared with each other. With permutation operation (e.g. _mm512_shuffle_epi32) and mask operation (e.g. _mm512_mask_min_epi32), it is possible to do comparison like (3) and (4). Corresponding pseudocode is shown in Table 1. We can conclude from the discussion that: A comparison is valid on AVX-512 if each pair of elements be is placed in two separate registers and compare option is applied between them. Our goal is to generate the sorting network like in Figure 2 and at the same time satisfy the memory access constraints. Then, the program can be hard-coded from generated network with AVX-512 instructions. Intuitively the fewer instructions the code has, the faster the program becomes. We propose two methods to generate new sorting network from Bitonic-Merge sorting network.

Table 1: Capability of AVX-512

Situation	Pseudocode
(1)	R1' = SIMD_min(R1,R2); R2' = SIMD_max(R1,R2);
(2)	(invalid)
(3)	R2' = permute(R2,<2,1,3,4>); R1' = SIMD_min(R1,R2); R2' = SIMD_max(R1,R2);
(4)	R1' = SIMD_mask_min(R1,R2,0b1010); R2' = SIMD_mask_min(R1,R2,0b0101); R1' = SIMD_mask_max(R1,R2,0b0101); R2' = SIMD_mask_max(R1,R2,0b1010);

3.2 1-Register Method

The *1-Register Method* does not mean that we are only using one register, but it means sorting such number of elements that can be stored in one register at the same time, i.e. K elements. Let's assume that the data is loaded into $R1$ initially. The target is to get elements which are stored in $R1$ to be compared with each other. The idea is to make a copy of $R1$, for instance copy it to $R2$. In such a way, elements can be compared with each other. The right-hand part of Figure 5 shows the example of 4 elements sorting networking using 1-register method. The left-hand part of the figure represents the 4 elements sorting network of bitonic merge. Obviously, the left-hand side and the right-hand side parts follow the same logic. The difference is *1-Register Method* compares each pair of elements twice. The order of permutation is generated from bitonic merge sorting network.

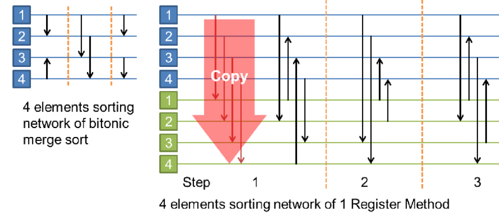


Figure 5: 1-Register Method Sorting Network

Let's define x as the position of element and $f_i(x)$ as the corresponding position should be compared to at ith step. $f_i(x)$ can be calculated from normal bitonic merge sort. Then we have code as in Algorithm 1 using AVX-512 instructions. There are 10 steps in total, and each step contains 3 AVX-512 instructions.

Algorithm 1 1-Register Method Sorting Algorithm

1: $__m512i\ a, b;$
2: $b = _mm512_shuffle_epi32(a, _MM_PERM_CDAB);$
3: $a = _mm512_mask_min_epi32(a, 0x6996, a, b);$
4: $a = _mm512_mask_max_epi32(a, 0x9669, a, b);$
5: ... ▷ 3 instructions in one step. The rest of the steps are similar.

3.3 2-Register Method

One disadvantage of the previous approach is that the elements need to be compared twice. Therefore, we developed the *2-Register Method* to avoid it. The 2-Register Method sorts $2K$ elements at once each time. At first, the data is loaded into registers $R1$ and $R2$. However, this time, bitonic merge sorting network scheme is applied directly. Unfortunately, bitonic merge cannot be simply applied even in the first step (leftmost arrows in Figure 6).

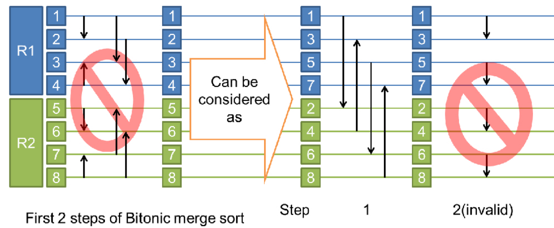


Figure 6: Invalid situation of first 2 steps

Since initially the dataset is unsorted at all, it can be considered as being in any order. In the second step, invalid situation would occur again if the sorting network is applied directly. To solve this problem, the position of the elements should be prepared for the next step so that no invalid situation occurs. The idea is to **look one step ahead**. Before introducing the *one step ahead* approach, it is better to describe how to perform exchanging the elements at the same time with comparing them. Doing comparison and then exchange of a pair of elements is equivalent to doing the comparison in the opposite direction. Figure 7 shows this procedure.

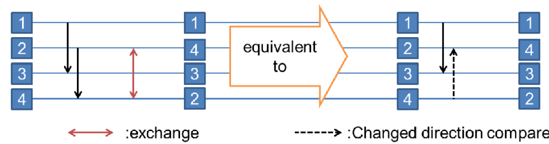


Figure 7: Exchange a pair of two elements

If the following step requires a pair of elements to be compared, and they are in the same register in the current step, they should be exchanged with one another, so that they are in different registers in the next step.

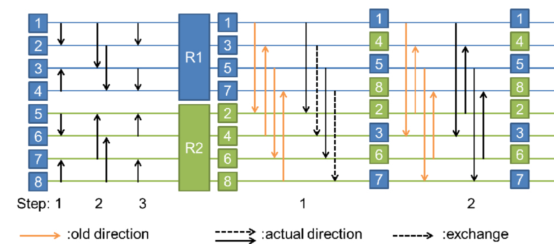


Figure 8: Solution of invalid situation

Figure 8 shows an example how the invalid situation be avoided in the first two steps. The whole sorting network is given in Figure 9. In this figure, a dashed black arrow corresponds to exchanging positions of the elements.

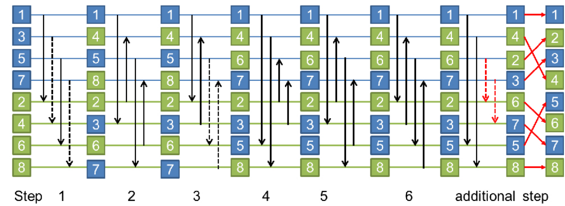


Figure 9: Capability of AVX-512 instruction set

The essential sorting program code is similar to the 1-Register Method, and it is presented as Algorithm 1.

Algorithm 3 2-Register Method

```
1: __m512i a, b, a2;
2: a2 = _mm512_mask_min_epi32(a, 0x6996, a, b);
3: a2 = _mm512_mask_max_epi32(a, 0x9669, a, b);
4: b = _mm512_mask_max_epi32(a, 0x6996, a, b);
5: b = _mm512_mask_min_epi32(a, 0x9669, a, b);
6: b = _mm512_shuffle_epi32(b, _MM_PERM_CDAB);
7: ...    ▷ five instructions consist of one step. The rest of
   the steps are similar.
8: ...                                  ▷ additional step:
9: a = _mm512_min_epi32(a2, b)
10: b = _mm512_max_epi32(a2, b)
11: permute a and b to correct order.
```

Because the procedure is exactly the same as Bitonic-merge sort, the theoretical time complexity has no difference in these two methods. Since 2-Register Method only uses 1 permute instruction per step, it saves 1 instruction in each given step. However, the 2-Register Method does not keep the order of elements in registers, so it needs one more step to rearrange the elements. Table 2 shows a comparison of the two methods regarding instruction count.

Table 2: Comparison of 1-Register and 2-Register methods

Algorithm	1-Register Method	2-Register Method
Set Length	K	$2K$
Number of steps	$\frac{log(K)(log(K)+1)}{2}$	$\frac{log(2K)(log(2K)+1)+2}{2}$
Intructions/step	3	5
Total instructions[a]	86	77

[a] When sorting 32 elements and $K = 16$ (Xeon Phi implementation)

3.4 Register Merge

Merging two sorted sequence is an essential part of the merge sort algorithm. Sequential merge algorithm can be done in $O(n)$ time complexity, where n is defined as the length of data. However, in the parallel case, it is very hard to achieve linear speedups in practice, e.g. merging K elements with K cores in $O(1)$ time. Needless to say efficient merging using SIMD Instructions is even harder. With bitonic merge, $O(Klog(K))$ work complexity, it is possible to merge K elements in $O(log(K))$ steps. The procedure of merging in registers is exactly the same as bitonic merge, which is also the last stage of bitonic merge sort. There are two methods in this case too: 1-Register and 2-Register. Since it is only the last stage of bitonic merge sort, it shares some common code with register-level sort. Input is given a two sorted sequences. One needs to be sorted in descending and the other in ascending order. Applying SIMD parallelism to merge longer sequence (e.g.longer than K) needs a little bit more work.

Thus all SIMD components of the algorithm are prepared. The rest is to assemble them together.

3.5 Implementation

Figure 10 shows the overview of the whole algorithm. The subroutines are divided according to the size of data. Generally the outer algorithm is merge sort algorithm. Firstly, each core sorts a part of data recursively (the orange color merge sort procedure in Figure 10). When the size of data is small enough, register-level sort is employed. Alternately sorted subsequence should be merged recursively with the algorithm introduced in Section 3.4. When each core finishes sorting its own data, merge procedure becomes multi-core-level merge (cyan color merge sort procedure in Figure 10). This subroutine is in charge of merging two sorted subsequences in parallel using multiple cores. Synchronization between multi-cores is required. Here our algorithm uses Merge Path algorithm[4] to help dividing sequences, so that each core can do merging independently. Merge Path algorithm is a parallel merge algorithm which uses binary search in order to divide two sequences into pairs of subsequences. Then each core can merge such two short subsequence. It is highly balanced algorithm because the partitioning guarantees that the total length of any pair of subsequences are the same. Hence each core process the same mount of data. Merge Path algorithm is used only for partitioning, but it

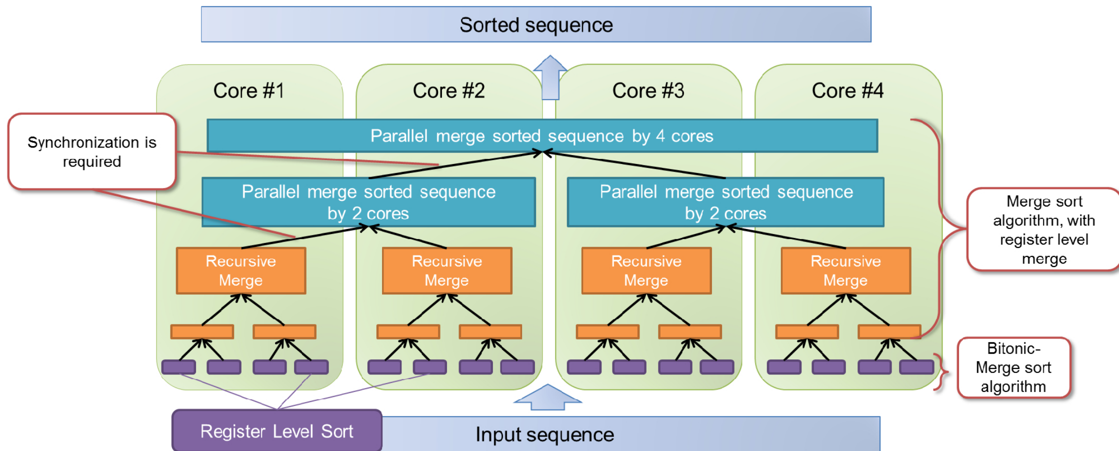


Figure 10: Algorithm Overview

contributes to extra computation which becomes the main overhead of our algorithm. Recursive merging is being done by multiple cores and whole data is finally sorted.

4. PERFORMANCE AND ANALYSIS

In Figure 10, Merge Sort (orange) and Register Sort (purple) are performed within a single core

Parallel Merge requires synchronization between multiple cores

Implementation and empirical experiments are done on Xeon Phi 5100 series. Data for experiment is generated randomly, distributed uniformly in range from 0 to $2^{31} - 1$. Table 4 shows speedup provided by SIMD and multi-core respectively, when sorting 4 million integer elements.

Table 4: Performance of our sort algorithm using Xeon Phi

Configuration	Speed Up	Time(sec)
Sequential merge sort	1.0	2.3
Sequential merge sort w/ SIMD	3.7	0.61
240-thread merge sort w/o SIMD	79	0.029
240-thread merge sort w/ SIMD	291	0.0079

The performance is measured by how much data (how many elements) can be sorted per second conventionally. Since comparison sort takes $O(nlog(n))$ time complexity, the performance will drop with the growth of data length n. Our algorithm achieves the peak performance when the data size is 4 million, which is about 525M integer elements per second. In our opinion, the architecture of Xeon Phi, particularly it's memory bandwidth and cache hierarchy are responsible for this phenomenon. Further experiments and optimization of the algorithm may be needed in order to fully understand the performance peak occurring for a particular number of elements. Figure 11 compares our result with other platforms and algorithms. Compared to the fastest, but limited radix-sort, our algorithm performs almost as well. It is only 13% slower for that particular number of elements with performance slightly dropping as we increased the dataset size. To show the generality of the algorithm,

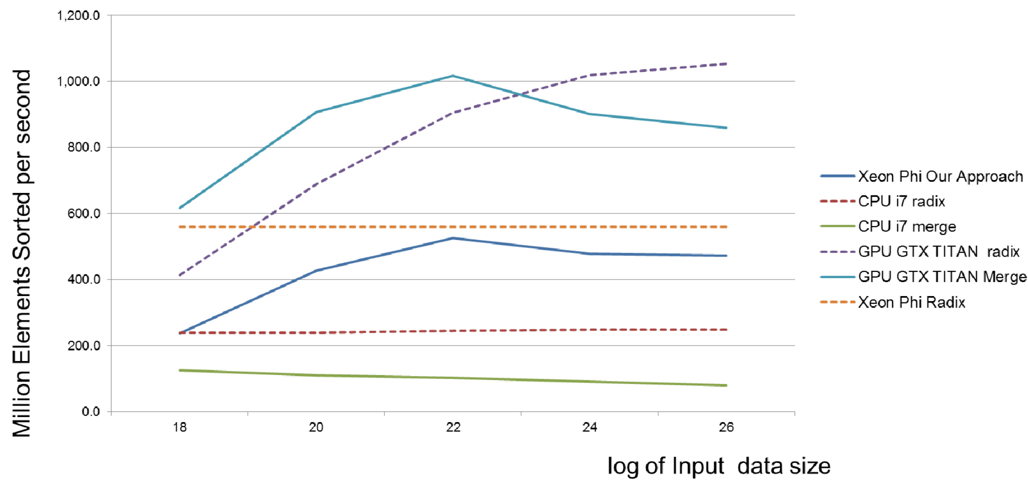


Figure 11: Performance compare to other algorithms and platforms

we implement float data sorting program on Xeon Phi and Intel E5-2670 with AVX, shown in Figure 12. Sorting float takes about 2 times longer than integer on Xeon Phi, due to difference in hardware. It proves the main advantage of our algorithm, meaning its generality regarding the data type (since it is based on comparison sort, as opposed to radix-sort).

5. CONCLUSIONS

This paper explored the issue of parallel sort, a fundamental computer science problem, on a class of SIMD processors. Main contribution is the proposed register level sort combined with merge algorithm. The key idea of our approach is changing the bitonic merge sorting network in order to satisfy the constraint of memory accesses within registers. Our algorithm uses constant number of registers to sort SIMD instruction length data, which exposes strong scalability. Generation of the sorting network has the same time complexity as bitonic merge sort, which can handle long vector instruction situation. Furthermore this algorithm can be generally employed in SSE, AVX or any similar instruction set.

We also implemented the algorithm on Xeon Phi combined with merge sort algorithm. Empirical performance results showed promising speed which is not far from the fastest radix sort implementation, while maintaining the generality of comparison sort. We achieve nearly 270x speedup (525M integers/s) on a 4M integer set using Xeon Phi coprocessor, where SIMD level parallelism accelerates the algorithm over 3 times. Our method can be applied to any device supporting similar SIMD instructions.

6. FUTURE WORK

Our algorithm is able to apply Bitonic-Merge sorting network on SIMD instruction processor like AVX-512. Other types of sorting network (e.g. Odd-Even or even less step algorithm) should also be possible to be adapted to AVX-512. To verify this, more work is required. Usually, not only numbers need to be sorted. Sorting keys with values (e.g. indices or address) is widely used. Efficient implementations may need more work in order to apply SIMD instructions like AVX-512. We would also like to perform experiments using other hardware, such as AVX2 supporting CPUs and run comparative benchmark on many machines using different data types.

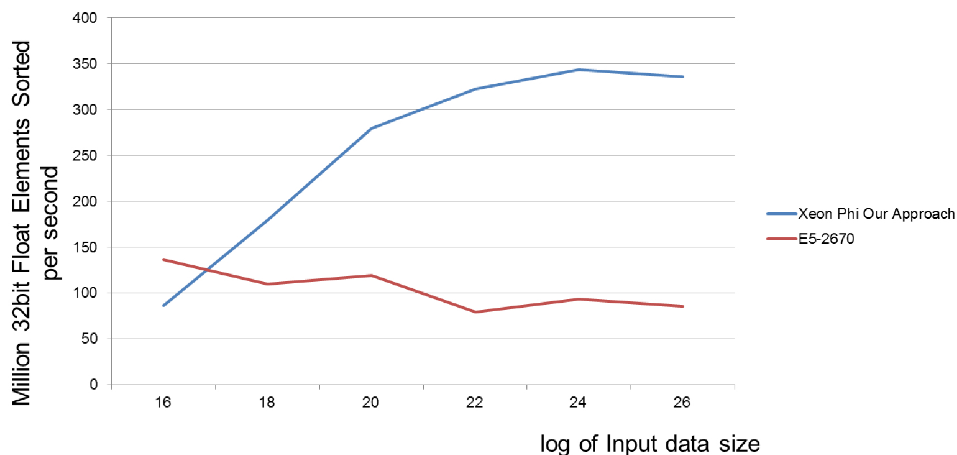


Figure 12: Float elements sorting performance

7. REFERENCES

[1] Jatin Chhugan et al. Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture Proceedings of the VLDB Endowment, Volume 1 Issue 2, August 2008, pp. 1313-1324

[2] Nadathur Satish et al. Fast Sort on CPUs, GPUs and Intel MIC Architectures Technical Report, 2010

[3] K. E. Batcher Sorting Network and Their Applications AFIPS '68 (Spring) Proceedings of the April 30–May 2, 1968, spring joint computer pp. 307-314

[4] Saher Odeh et al. Merge Path - Parallel Merging Made Simple Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International. pp. 1611 - 1618

[5] Nadathur Satish et al. Designing Efficient Sorting Algorithms for Manycore GPUs Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on Parallel and Distributed Processing

[6] Nadathur Satish et al. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort SIGMOD '10 Proceedings of the 2010 ACM SIGMOD International Conference on Management of data pp. 351-362

[7] Timothy Furtak, Jose Nelson Amaral and Robert Niewiadomski Using SIMD Registers and Instructions to Enable Instruction-Level Parallelism in Sorting Algorithms SPAA '07 Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures pp. 348-357

[8] D.E. Knuth The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition. Addison-Wesley, 1998. ISBN 0-201-89685-0. Section 5.3.4: Networks for Sorting, pp. 219-247

[9] Miklos Ajtai et al. An nlog(n) sorting network STOC '83 Proceedings of the fifteenth annual ACM symposium on Theory of computing pp. 1-9

[10] Hiroshi Inoue et al. AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors PACT '07 Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques pp. 189-198

[11] Bronislava Brejov Analyzing variants of Shellsort Information Processing Letters Volume 79, Issue 5, 15 September 2001, pp. 223-227

[12] Naga K. Govindaraju et al. GPU:TeraSort: High Performance Graphics Co-processor Sorting for Large Database Management SIGMOD 2006 Chicago, Illinois, USA

[13] Michael Herf, Dec 2001 Radix Tricks: Retrieved Oct 13rd, 2013 from: http://stereopsis.com/radix.html

[14] Andrew Davidson et al. Efficient Parallel Merge Sort for Fixed and Variable Length Keys Innovative Parallel Computing(InPar) ,2012 San Jose, USA

[15] Erik Sintorn and Ulf Assarsson Fast parallel GPU-sorting using a hybrid algorithm Journal of Parallel and Distributed Computing Volume. 68, Issue 10, October 2008, pp. 1381–1388

[16] Gianfranco Bilardi and Alexandru Nicolau Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared-Memory Machines SIAM, Journal on Computing, 1989, Volume 15 Issue 2, pp. 216-228