From Sand to Flour: The Next Leap in Granular Computing with NanoSort

Theo Jepsen*†, Stephen Ibanez*†, Gregory Valiant*, and Nick McKeown*†
**Stanford University* †*Intel*

Abstract

The granularity of distributed computing is limited by communication time: there is no point in farming out smaller and smaller tasks if the communication overhead dominates the decrease in processing time due to the added parallelism. In this work, we leverage the low communication latency of a new NIC/CPU hardware design, the nanoPU, to explore a new extreme of granularity in distributed computation, where a problem is partitioned into tens of thousands of nanosecond-scale tasks.

To evaluate the feasibility and practicality of extremely fine-grained computing, we built NanoSort, a distributed sorting algorithm running on the nanoPU. Using cycle-accurate FireSim simulations of 65,536 nanoPU cores, we show that NanoSort can sort 1M keys in 68μs, an order of magnitude faster than MilliSort, the current state-of-the-art.

1 Introduction

There are two main ways to speed-up compute workloads: use beefier CPUs with higher clock speeds; or split the workload into smaller, finer-grained pieces that run in parallel.

With the slowing of Dennard scaling, clock speed is not increasing as fast as it once did, the general trend is towards increased parallelism. Over the past twenty years, we have seen parallelism increase with multicore CPUs,from dual-core, to quad core, to 64-core CPUs. It is not uncommon to see a single server with hundreds of cores (e.g., 224 cores per server [30]). To further increase parallelism, workloads frequently expand beyond a single server, to racks of servers connected by high-speed networks. As we add more cores, however, the cost of inter-core communication increases: from O(ns) within the same CPU, to O(10ns) between CPUs on the same server, and to O(10μs) across the network.

As Gene Amdahl said in 1967 "the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used" [3]. In our context, as we break the overall workload into smaller and smaller compute tasks to be processed in parallel, we eventually run into the overhead cost of communication between the cores. If we make the tasks too small, the completion time is dominated by the communication time between CPU cores. Ultimately, if we break the job into tiny tasks, farmed out to a huge number of CPU cores, the completion time will be dominated by the communication time. The communication time limits the degree of parallelism.

We have seen this trend with one of the most executed [23, 42] and studied [1, 6, 42] workloads in computer science[1]: sorting. Initially, the biggest speedups were achieved by scaling *up* to beefier CPUs. Then, systems began scaling *out* to multiple cores. In 1986, Tsukerman et al. FastSort [36] sorted 1M records in 3600 seconds with 2 cores. In 1989, it took Baugsto et al. 180 seconds with 16 cores, and 40 seconds with 100 cores [9]. In 1993, Nyberg et al. sorted in 7 seconds using AlphaSort [31] with 3 cores.

When the single-server limit was reached, they started using multiple servers connected by the network. In 2002, Popovici et al. [33] sorted in 440 milliseconds with 32 servers, each with two P3 CPUs. Then, they started using racks of servers, but optimized for throughput, not latency. In 2009, Hadoop used 3,452 nodes with 2 quadcore CPUs each [39]. In 2016, TencentSort [17], the current record holder of the GraySort "Daytona" (sorting throughput category) used 512 nodes with 20 cores each. These large scale systems were trading-off *latency* in favor of *throughput*. Recently, MilliSort [27] tried pushing the limits of large-scale, fast (low-latency) sorting, by using RAMCloud, a high performance networking subsystem. They achieved an impressive speed-up with finer granularity: they sorted 1M records in 1ms using 2,240 cores. But they were fundamentally limited by Amdahl's law. When MilliSort's cost of network communication (using RAMCloud with a 5μs RTT) dominated the runtime, they could no longer decrease the size of compute tasks, limiting the scale-out.

The networking community has been working to reduce

[1] Sorting accounted for a quarter of computing time in 1984 [42].

the communication overhead. Recently we have seen several designs that reduce latency, especially at the end-host, which is responsible for most of the of the latency [10]. By optimizing the software stack using currently available network cards (NICs), eRPC [20] reduces the wire-to-wire latency to 850ns. NeBuLa [40] presents a hardware design that delivers packets straight from the NIC to the CPU L1 cache, with a latency of 100ns. These systems bring two main benefits: lower overhead and reduced latency. To reduce the mean and *tail* latency, the nanoPU [16] offloads the entire RPC stack to NIC hardware and provides a dedicated *fast path* directly between the network and CPU register file. The nanoPU's wire-to-wire round-trip latency through a core is just 69ns with almost no additional tail latency.

Generally, we decide to stop increasing parallelism when the limits of the underlying communication system are reached. For example, MilliSort reached this limit with 2,240 cores. Since the nanoPU has reduced the communication latency by an order of magnitude, from microseconds to nanoseconds, it may be able to push the limit further. We now have the opportunity to take granularity to a new extreme. This raises the question: ***if** we had an infinite number of cores, how much of a speed-up would we get?*

When communication latency is on the order of microseconds, the workload takes O(ms). For example, with a RTT of 5μs and 10 communication steps, the cost will be 50μs, not even including the compute time. If the communication latency is reduced to nanoseconds, will workloads complete within microseconds?

Using a network stack with low communication overhead is essential, but not the sole requirement to enable fine-granularity and massive parallelism. An application must be amenable to parellilization, particularly, it must be possible to split the application into fine-grained tasks.

Increasing granularity to the extreme presents three main challenges. First, we have to shrink the task size as much as possible, but not too small, as it will have to pay the price of constant overheads. Also, as tasks get smaller, we need more parallel tasks, which increases the amount of communication. Second, the tasks work dependency graphs must be structured in such a way to maximize system utilization. If tasks have too many dependencies, they may spend too much time waiting, reducing the overall parallelism. Third, the load must be evenly distributed across cores. This is especially important for sparse workloads, which are unpredictable, so must be partitioned in such a way to reduce variance. Otherwise, imbalanced partitioning will lead to some tasks becoming stragglers.

We believe that there are two key ideas for addressing these challenges of granular computing: reduce overhead and use efficient algorithms. New low-latency, predictable networking stacks, are essential for reducing the overheads. This enables tasks to send and receive many small messages, enabling finer granularity. It is also necessary to design algorithms that overcome the imbalance of partitioning the data, as well as structuring the communication, to reduce the length and duration of the critical path in the work dependency graph.

To demonstrate how these key ideas can push the limits on fine-grained computing, we explore the problem of distributed sorting. Sorting is a good problem to study because: it is a familiar problem that is well understood and easy to describe [6]; it is industrially applicable, from analytics to graphics; and its unpredictable nature makes it non-trivial to implement as a distributed algorithm that stresses system resources [35].

We designed a new sorting algorithm, NanoSort. NanoSort runs on a cluster with tens of thousands of nanoPU cores using efficient communication patterns, which minimize runtime variance. The nanoPU provides very low overhead communication with just 69ns wire-to-wire latency. Such low overhead makes it possible for NanoSort to use very small tasks, on the order of 100s of nanoseconds. The NanoSort algorithm is recursive, so it breaks the problem into smaller sub-problems in each iteration. By fanning out quickly, the critical path (the depth in the work dependency graph) is minimized, reducing the overall runtime. The nanoPU's predictable latency, together with NanoSort's partitioning, minimizes the variance. The predictable latency ensures tasks complete in time, while the balanced partitioning balances the work across all the tasks.

To see whether NanoSort can scale, we used hundreds of Amazon EC2 instances to run cycle level simulations of thousands of nanoPU cores. We ran the GraySort 1M sorting benchmark, which measures the end-to-end time to sort 1M records. Using 65,536 cores, NanoSort sorts 1M records in 68μs, which is an order of magnitude faster than MilliSort's 1ms record.

Overall, we make the following contributions:

- Explore the design space for distributed algorithms to efficiently use new low latency communication stacks;
- Introduce NanoSort, a sorting algorithm designed for low-latency, fine-grained computing; and
- Perform a large scale, cycle level simulation of NanoSort sorting 1M keys in 68μs, an order of magnitude faster than the state of the art.

The rest of the paper is structured as follows. The next section provides background on existing network stacks and new low latency network stacks. Section 3 explains how workloads can be restructured for even finer-grained tasks. Sections 4 and 5 describe the design and implementation of our new sorting algorithm, NanoSort. Section 6 evaluates the parameters that affect fine-grained computing, as well as a large scale simulation of NanoSort.

System	Median latency (ns)
eRPC	850
NeBuLa	100
nanoPU	69

Table 1: Median wire-to-wire loopback latency for three end-host network stacks.

- Scan 1K 8-byte words in L1 cache
- Sort 40 8-byte keys (Section 6)
- Travel 300m at the speed of light
- Receive 2Kbyte on a 200Gbps NIC
- Process 118 8-byte loopback nanoRequests [16]
- Perform 2 MICA R/W operations [16]
- Perform 4 set algebra intersections [16]

Figure 1: Examples of what can be accomplished in under 1 μs (computations using a nanoPU core).

2 Low Latency Communication Systems

In recent years, there has been a significant amount of research effort in both industry and academia to design low latency, high throughput communication systems. Systems like Shinjuku [19] and Shenango [32] recognize the high overheads associated with the Linux network stack and opt to use a custom data-plane operating system that is designed to efficiently schedule network requests and application threads across cores. The authors of eRPC [20] demonstrate that it is possible to achieve very low latency RPCs using commodity NICs by leveraging a number of clever common case optimizations. Many commercial NICs include support for remote direct memory access (RDMA) which enables low latency by performing zero-copy communication between application buffers on remote systems and by offloading the transport layer into NIC hardware. Another approach to optimize communication performance is to design new datacenter transport protocols. NDP [15] and Homa [28] are two recent datacenter transport protocols that achieve low latency communication by leveraging receiver-driven solicitation, and in-network support for packet trimming and priority queueing. One of the most promising and ambitious approaches we have seen to optimize communication performance is to integrate the CPU and NIC, thus bypassing PCIe. NeBuLa [40] and the nanoPU [16] are two such systems that take this approach. The nanoPU is able to achieve a wire-to-wire latency of just 69ns, with very little variance, and each core is able to process over 100M requests per second. While the nanoPU is a research prototype, we choose to use it as a platform for fine-grained computing because we believe its performance characteristics are representative of future datacenter network infrastructure. Additionally, an open source nanoPU prototype is readily available [29] and can be used to implement large scale application on Amazon cloud. The following section describes some of the key characteristics of the nanoPU that allow it achieve such high performance.

2.1 nanoPU Background

The nanoPU is a hero experiment intended to push the performance limits of the host network stack. It is designed to minimize average and tail latency as well as the CPU overhead associated with network communication. By doing so, it enables applications threads on remote cores to communicate quickly and predictably at high message rates. The nanoPU "fast path" uses a novel approach to deliver data directly between the network and the CPU register file - bypassing PCIe, caches, and the memory hierarchy entirely. In order to minimize tail latency, the nanoPU fast path implements the network stack's key resource scheduling decisions in efficient hardware: (1) load balancing of network messages across cores and (2) scheduling of application threads on cores. Furthermore, the nanoPU implements latency-optimized network transport protocols in programmable hardware thus reducing the CPU overhead associated with network communication Additionally, the nanoPU's simple two-register interface enables applications to send and receive small messages at very high rates.

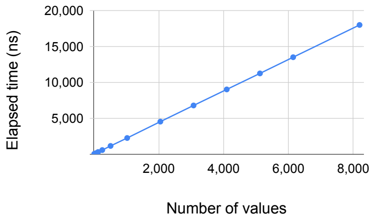
3 Extremely Granular Computing

Current systems have tasks (e.g. RPCs) that run for hundreds of microseconds [4, 8, 14, 19, 32]. One reason for long-lived tasks is to justify the communication cost of invocation: if it takes microseconds [20] to invoke the task, it should run for microseconds to amortize the communication cost.

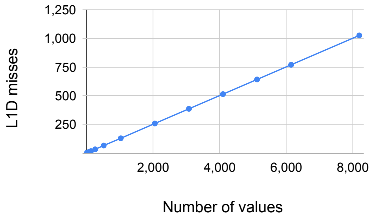
As the communication latency and overhead decrease, it is worth farming out smaller, shorter-lived tasks. With the 69ns wire-to-wire communication latency of the nanoPU, tasks only need to run for hundreds of nanoseconds to amortize the communication..

A time budget on the order of hundreds of nanoseconds may seem too short to perform useful work for an application, especially on a general purpose CPU. Figure 1 shows examples of common operations and tasks that complete within 1μs on a 3.2GHz in-order RISC-V Rocket core. When data is cache resident, which is the case for many of those operations, most of the time is spent on "useful" compute cycles, instead of waiting for memory I/O. Alone, a single small task may not be able implement a meaningful application. However, together, thousands of fine grained tasks can execute a large job, harnessing the aggregate CPU cycles and cache bandwidth.

There are two main benefits of using fine-grained tasks. First, by breaking a large job into smaller pieces, it is possible to increase parallelism. It lets us re-shape the compute core-



(a) Runtime



(b) Cache misses

Figure 2: Using a single core to find the minimum from an increasing number of values.

hours: we can run the same number of core-hours in less time, going from "deep" to "wide". Second, the compute resources can be scaled to the precisely match the size of the job: we can allocate compute at the granularity of cores or threads, instead of servers or entire racks.

Parallelism doesn't come "for free" though. Job completion time depends on the depth of the task dependency graph, which is a graph of the communication pattern between the tasks defined by the algorithm (e.g., Figure 3). The deeper the graph, the longer the execution. To reduce the execution time, the graph can be reshaped to be shallower and wider. Wider means more parallelism is possible, which requires smaller compute tasks. In turn, it requires more communication, so it is essential that the communication overhead be low. The width and depth of the dependency graph determine the amount of communication as well as the size of the tasks.

3.1 A Simple Example: MergeMin

To illustrate the performance implications of a dependency graph's width vs. depth, we present a simple example: finding the minimum value from a list of numbers. This could be done sequentially, scanning the numbers while keeping a running minimum. However, as we can see in Figure 2 this is limited by the processing speed of a single core, which in turn is limited by the I/O throughput, or cache misses (Figure 2b).

By using more workers, this process can be parallelized. It cannot be fully parallelized, however; it requires some communication between the workers to merge their minimum values. This can be done using a merge tree, as shown in Figure 3. At each level in the tree, a worker receives the minimum value of one or more other workers; in turn, the worker

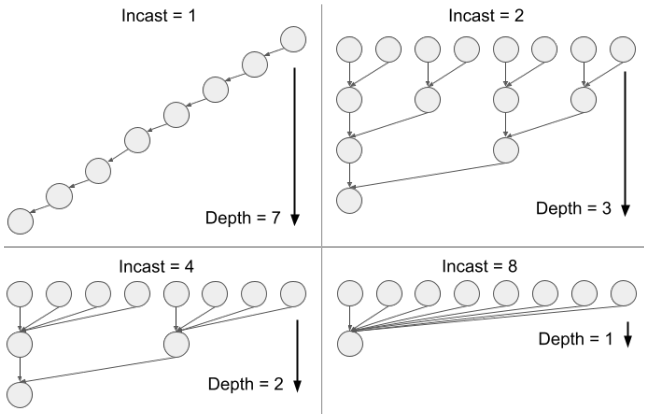


Figure 3: Lower incast results in deeper work trees

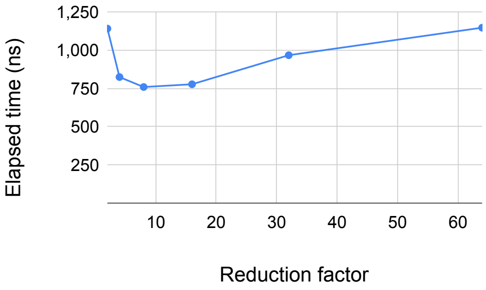


Figure 4: The runtime for MergeMin with increasing incast size (64 cores, 128 values per core).

merges these minimums, and passes it down the tree.

This is repeated until reaching the root of the tree, where a single worker merges the final minimum values. There are two extremes of this merge tree: either very deep, or very shallow. This determines the "incast": the number of values that are received and merged at each level in the tree. A very shallow tree has large incast. This increases the latency for each level of tree, because the merger worker has to receive and process more minimum values. A deep tree has smaller incasts, so each level is faster, but there are more levels. As we can see in Figure 4, there is a trade-off between width and depth, which has sweet spot with an incast size of 8.

The runtime depends on the choice of two parameters: the granularity (i.e. the amount of work per worker, the inverse of the number of workers), and the depth of the tree (determining the incast size). The two parameters cannot be chosen independently. For example, if the granularity (number of workers) is increased, the tree becomes wider. This will cause the tree to become deeper, unless the incast size is increased.

3.2 Building Applications with Fine-Grained Tasks

Although many applications are amenable to being implemented with granular computing, applications that have real-time latency requirements benefit the most from fine-grained tasks. In some cases, such as interactive systems, it is more important to provide a result in a timely fashion, rather than optimize resource utilization (efficiency, throughput). Applications that have these requirements, and that are highly parallelizable, are the best suited for granular computing.

Fine-grained computing is an especially good fit for interactive applications, like real time data analytics and graphics processing. Stream processing systems like Spark ingest large amounts of data and must quickly produce a result. Many of its basic operations, like mapping (transforming) and reducing (merging) data are easily parallelizable. Graphics processing, like ray tracing, also has many operations that can be parellelized. For example, constructing the bounding volume hierarchy (BVH) requires sorting the objects in the scene. Web search also has interactive requirements, and can be executed with fine-grained tasks using set algebra operations [16].

Many of these applications exhibit the Map Reduce [11] pattern, which is is a natural fit for granular computing. The first step is to transform the input data, which can generally be done independently, in parallel, by all the nanoTasks running on different cores. The reduce step, however, requires coordination among the cores. The design of the reduce (merge) has important implications for how the granular program is implemented.

It is essential not only to minimize the size of individual tasks, but also the critical path through the dependency graph. This requires the granular program to follow some design principles:

- Asynchronous communication. Tasks should not block waiting for responses after sending out a message. It is better to use the "fire and forget" communication model and build synchronization into the algorithm.
- Decentralized decision-making. Workers should make decisions without a global coordinator, which could become a bottleneck in the task dependency graph. This also allows multiple threads to progress through the dependency graph in parallel.
- Tiny tasks. Each task should be as small as possible and entirely cache-resident.
- Even load distribution. In expectation, all tasks should process the same amount of data and send/receive the same amount of messages, independently of the workload.

Some of these guiding principles may seem obvious, but there are some nuances when actually implementing them. In the next section we discuss how they can be applied to a specific application: sorting.

4 NanoSort Design

Over the years, the requirements and constraints faced by sorting algorithms have continued to evolve [1, 6, 23, 42]. In the early days, sorting was CPU bound, which was suitable for algorithms that do arbitrary comparisons (e.g., InsertionSort, BubbleSort, QuickSort). Then, it was limited by disk, so external sorting algorithms were designed to optimize I/O (e.g., reducing disk seeks [1]). More recently, as sorting has scaled out to clusters [17, 35], distributed systems have adopted various forms of BucketSort [42].

MilliSort is a bucket based sorting algorithm that minimized network communication by dividing the algorithm into two parts: partitioning and shuffling. During partitioning, a subset of the nodes pick bucket boundaries, which are disseminated to all other nodes. During the shuffle, the boundaries are used to route records to the final node which is responsible for a bucket.

These distributed sorting algorithms like MilliSort were designed to minimize communication, in order to avoid network overhead. Now that systems like the nanoPU provide low-overhead communication, it is no longer necessary to *minimize* communication. It may be actually beneficial to communicate *more*. In the next section we describe the high-level design of such an algorithm that takes advantage of cheap communication.

4.1 Extremely Granular Sorting

There is no black-box way to convert an existing algorithm into one that can be efficiently run on the nanoPU; to some extent, the algorithm must be designed from scratch with an eye towards balancing the per-node compute and communication times. Nevertheless, this algorithm design problem need not be daunting. In the case of nanoSort, we arrived at a simple and natural modification of quicksort, guided by the following the general design principles of minimizing unnecessary communication and exposing high-level "dials" (or "knobs") that tune the tradeoff between compute-per-node and communication.

We now describe nanoSort. For the sake of simplicity and clarity, throughout we assume that the set of keys are all distinct. By design, all operations in nanoSort are comparison-based, and hence the execution depends only on the *relative* ordering of the keys, as opposed to their actual values. For this reason, the runtime will not depend on the distribution of the underlying keys (e.g. uniformly distributed, vs clustered/heavy-tailed, etc.).

NanoSort is a clean, recursive sorting algorithm that is a variant of quicksort. At each level of recursion, the goal will be to split up the set of keys into b "buckets" such that: 1) all the keys in bucket i are less than those in bucket $i+1$, and 2) the number of keys assigned to each bucket is roughly equal for all buckets. We will then recurse on each bucket.

Crucially, after a given step of recursion, no communication will be required between nodes corresponding to different buckets.

To motivate the specific design decisions (beyond simply setting the parameters such as the number of buckets at each level of recursion, b, and the average number of keys per node), we briefly discuss key considerations.

4.2 Balanced Buckets and Balanced Nodes

As with any variant of quicksort, one hopes that at each step of recursion, the number of keys assigned to each bucket is fairly uniform, since larger buckets would require either deeper recursion or more time per step of recursion. Beyond this, in our distributed setting, if some buckets are much larger than others, we would need to ensure that the number of nodes allocated to a given bucket is proportional to the number of keys in the bucket. If that were not the case, if each of the b buckets corresponded to the same number of nodes, then nodes corresponding to buckets with more keys would end up incurring significantly more incasts. Combined with the greater number of recursive steps, this would be especially debilitating. Of course, to ensure that a bucket is allocated a proportional number of nodes would require counting (exactly, or approximately via sampling) the number of keys in each bucket.

To avoid these complications that would come with significant variation in the bucket sizes at each level of recursion, we opt to expend slightly more communication to ensure fairly uniform buckets. This is achieved via a two-pronged approach. First, we ensure that the bucket sizes will be tightly concentrated about their expectations—namely that we will not expect much variation due to randomness in the execution of the algorithm—and then we are careful to ensure that the expected size of all buckets are equal at a given step of the recursion. The first step is accomplished via a "median-tree" that approximates the median while being fairly communication efficient. The second step uses no additional communication but requires some care. We describe these two components below.

For concreteness, suppose our goal is to define $b = 10$ buckets by selecting $b-1$ "pivots", $p_1 \leq p_2 \leq \ldots \leq p_9$. A naive and communication-efficient way to select these pivots is simply to select 9 pivots uniformly at random (without replacement) from the set of keys. This scheme has the fortuitous property that the expected quantile of p_i is exactly $i/10$ (for example, in expectation, p_2 will be larger than 20% of the keys). The variance of the quantiles of these pivots (and hence the variance of the corresponding bucket sizes), however, is problematic. Indeed, there is a good chance that the largest of the resulting buckets would have more than twice as many keys as the smallest.

This issue is easy to fix in a communication-cheap manner via a median-tree. Suppose we have N nodes that each receive 9 uniformly random keys, and let p_i^j denote the ith smallest key received by the jth node. If we define $p_i = median(p_i^1, \ldots, p_i^N)$, then the variance of the quantile of p_i is $O(1/N)$, corresponding to a standard deviation of $\approx 1/\sqrt{N}$. This could be made arbitrarily small by choosing N to be large, though at the expense of significant communication in the computation of the median. Instead of computing the median exactly, we approximate the median via a median tree: for example, if $N = 10^2$, we can partition the N nodes into sets of 10, and have each set of 10 report their medians to a designated node, which then aggregates the 10 reported medians (one from each set) by taking the median. Similarly, for $N = 10^3$, a similar mechanism will result in the median-of-medians-of-medians, requiring only 3 rounds of communication. This sort of trick maintains much of the accuracy of the overall median, while reducing the communication from linear in N, to logarithmic. The issue with the above strategy, is that the expected sizes of the buckets is no longer equal. If a node receives 9 keys drawn uniformly at random from the pool of keys, it is the case that the expected quantile of the smallest received key is 10%. If, however, we have N nodes that each receive 9 keys drawn uniformly from the set of keys, then the expected quantile of the *median* of the smallest keys that each node received is not 10%, but $\approx 7.5\%$. Phrased differently, if we let D denote the distribution corresponding to the quantile of the smallest key in a set of 9 keys chosen uniformly at random from the entire set of keys, then the *expectation* of D is exactly 10%, but the *median* of D is $\approx 7.5\%$. This discrepancy between 10% and 7.5% means that the bucket corresponding to keys that are at most p_1 would be 25% smaller than we would hope. After multiple rounds of recursion, this discrepancy is compounded multiplicatively with each recursion.

To fix this issue, we simply need to take into account that we are concerned with the median, rather than the expectation. For example, suppose N nodes each receive 9 keys drawn uniformly at random from the set of keys, and the ith node sets p_1^i to be its smallest key with probably 0.8 and its *second smallest* key with probability 0.2, then the expected quantile of the median of these N numbers is $\approx 10\%$. Analogous modifications can similarly be made for the remaining buckets. In this way, our algorithm is able to "fix" the expected quantiles of all the bucket sizes in this manner. Figure 5 illustrates the expected bucket sizes for the naive pivot selection, and two other protocols.

The PivotSelect routine described below gives our specific instantiation of this approach for the setting we implemented where the number of buckets is 16.

Below we provide a more formal description of nanoSort. We will require the number of nodes at our disposal to be a power of the number of buckets that is used at each level of our quicksort-like recursion.

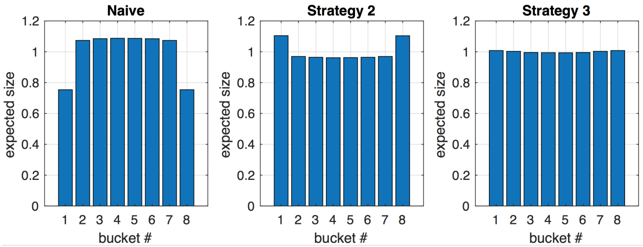


Figure 5: Comparison of the expected bucket sizes for three pivot selection strategies in the setting where the number of buckets is 8, and the number of received keys is also 8. Naive strategy: Select 7 pivots uniformly without replacement from the 8 keys. Strategy 2: Sort the keys $k_1 \leq k_2 \leq \ldots \leq k_8$, and with probability return $k_1, k_2, \ldots, k_7$, and with probability $1/2$ return $k_2, k_3, \ldots, k_8$. Strategy 3: Randomize between the previous two strategies as follows: With probability $1/4$ return the pivots selected by the "Naive" strategy, and with probability $3/4$ return the pivots selected by "Strategy 2".

5 Implementation

We implemented granular computing applications, including NanoSort, as nanoPU programs. In this section we describe how we implemented the programs and the simulation cluster that we use in our evaluation (Section 6).

5.1 nanoPU FireSim Simulation

The nanoPU is a SoC that extends the open-source RISC-V Rocket-Chip SoC generator [5]. The Rocket core is a simple five-stage, in-order, single-issue processor. We use the default Rocket core configuration: 16KB L1 instruction and data caches, a 512KB shared L2 cache, and 16GB of external DRAM memory.

We use the Verilator [41] cycle-accurate simulator running on top of FireSim [21]. FireSim enables us to run large-scale, cycle-accurate simulations with tens of thousands of nanoPU cores using AWS EC2 instances [2]. We simulate a target clock rate of 3.2GHz—all reported results are in terms of this target clock rate. The simulated servers are connected by switch models implemented in C++ running on the AWS x86 host CPUs.

We used the same network topology for all experiments, except for where noted otherwise. We used a two layer full-bisection topology with 200Gbps links. Each leaf switch has 64 downlinks to nanoPU NICs, and 64 uplinks to core switches. The link latency is 43ns and the switching latency is 263ns.

For our large-scale simulation, we ran 65,536 nanoPU cores using 264 EC2 instances. Of the instances, 256 simulated the nanoPU and leaf switches, while 8 instances simulated the spine switches.

The original FireSim uses AWS F1 instances to accelarate the simulation with FPGAs. Running a simulation at this scale with AWS F1 instances would be prohibitively expensive. We found that the NanoSort execution time is so short, that it is not worth the overhead of provisioning and flashing the FPGA. Instead, we used cheaper general-purpose AWS compute instances running the Verilator software-based simulator. This required significant changes to FireSim, which we are contributing back to the community.

PivotSelect (16 Buckets)

Input parameters: $n = number\_keys\_at\_node$, $num\_buckets$.

Output: 15 pivots $p_1 \leq p_2 \leq \ldots \leq p_{15}$.

1. Sort the n keys $k_1 < k_2 < \ldots < k_n$.
2. If $n = 16$: With probability $1/4$ select 15 pivots uniformly at random without replacement from $k_1, \ldots, k_{16}$ and return them (sorted). With probability $3/8$ return $k_1, k_2, \ldots, k_{15}$, and with probability $3/8$ return $k_2, k_3, \ldots, k_{16}$.
3. If $n < 16$: Select $16 - n$ keys uniformly at random from $\{k_1, \ldots, k_n\}$ and duplicate them to form a list of 16 keys, then select pivots as in the $n = 16$ setting.
4. $n \in \{17, 18, \ldots, 31\}$: Select a subset of 16 keys uniformly at random, and run the $n = 16$ protocol.
5. $n = 32$: With probability $1/2$ return the keys indexed by the set $[1,3,6,8,10,12,14,16,18,20,22,24,26,27,29]$, and with probability $1/2$ return the keys indexed by the set $[4,6,7,9,11,13,15,17,19,21,23,25,27,30,32]$.
6. $n > 32$: Select a subset of 32 keys uniformly at random, and run the $n = 32$ protocol.

5.2 Applications

We implemented microbenchmarks, MilliSort and NanoSort programs in C using the nanoPU's register-based communication API. All the programs rely on common code for initialization and communication, as well as for the sort benchmark.

Communication A common property of both algorithms is that they have "phases" or "steps". Cores do not progress through the steps in lockstep, so cores may be in different steps from each other at the same point in time. Thus, a core may receive messages for a subsequent step before it has completed its current step. This requires *reordering* messages, so that messages for the current step are processed before the subsequent step.

Ideally the NIC would provide a reordering mechanism to ensure that messages for the same step are delivered together. This behaviour could be approximated by associating each step with a different L4 port, which would associate different steps with different NIC RX queues. This may lead to poor RX queue utilization. Instead, we implemented a reordering buffer in software.

NanoSort
Input parameters: *num_keys*, *num_buckets*, $num\_nodes = num\_buckets^r$.

Initial Step—Random Shuffle: After this step, the keys will be randomly partitioned among the nodes subject to each node having num_keys/num_nodes keys.

1. If $num\_nodes = 1$ then sort the keys
2. Otherwise (ie $num\_nodes = num\_buckets^r$ for $r \geq 1$):
    - For each node i, it sorts its keys, and extract a list of $b-1$ pivots $p_1^i \leq p_2^i \leq \ldots \leq p_{b-1}^i$ from its set of keys via the PivotSelect routine.
    - Pivots $p_1, \ldots, p_{b-1}$ are selected via $b-1$ median-trees, where the jth median tree aggregates the p_j^i's computed by each node to compute p_j. The in-cast (ie fan-in) and depth of the median-tree are chosen appropriately to balance the compute time and the variance-reduction.
    - The pivots $p_1, \ldots, p_{b-1}$ define b buckets (the first bucket consists of values less than p_1, the next consists of values between p_1 and p_2, etc.) The pivots are broadcast to the nodes, and the nodes are partitioned into b equal sized sets (the first num_nodes/b nodes are in one set, etc). Every node sends each of its keys to a uniformly random node in the appropriate bucket: for each key, if it is in bucket i, it is sent to a node chosen uniformly at random from the nodes in the ith partition.
3. Recursively apply the algorithm (starting at Step 1) to each of the b buckets.

Sort benchmark We perform the sorting according to the GraySort benchmark [39]. Just like in MilliSort's implementation [27], the cluster is pre-loaded with the sorting records before starting the benchmark. The benchmark specifies that each record be 100 bytes (a 10-byte key and a 90-byte value). In order to have all the data aligned to 8-byte words for the RISCV architecture, we deviate slightly from the specification by using 104-byte records (8-byte key and 96-byte value). The data on each core is structured with all the keys in a contiguous block of memory, followed by all the values, in corresponding order. When cores exchange keys, they also include the original core ID of each key. This way, after all the keys are sorted, the cores know where to find the original record (including the value) for each of their keys. At the end of the benchmark, the records are redistributed across the cores in the cluster, with the keys and values in sorted order in separate, contiguous memory regions.

5.3 Multicast Support

Although current commodity switches support multicast replication, they do not ensure *reliable* delivery. Naively layering reliability on top of multicast is not so straight forward though, as it leads to scalability problems such as the ACK implosion problem [26], where the sender is flooded with an incast of acknowledgement messages. However, promising results in recent efforts [37] on in-network aggregation of ACKs suggest that *efficient* reliable multicast is a not-so-distant reality.

Thus, since we believe it is reasonable to assume that the network can provide reliable multicast, we extended FireSim's software switch to provide reliable multicast. When a switch receives a multicast packet, it caches it before replicating it to all members of the multicast group. The cached packet can then be sent in response to a NACK or a timeout.

6 Evaluation

We evaluate the conjecture that with lower communication overhead, it is worth farming out small pieces of compute. Our main research questions are:

- With low communication overhead, what are the smallest tasks possible?
- What is the importance of network characteristics, like latency, tail latency, and group communication?
- Can large-scale extreme granular computing speed-up an application like sorting?

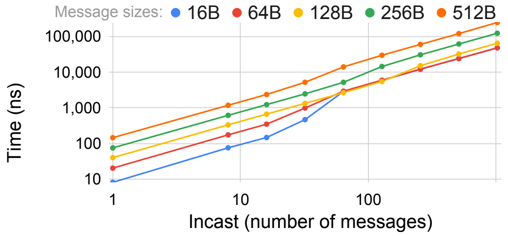


Figure 6: Time to receive increasing number of messages.

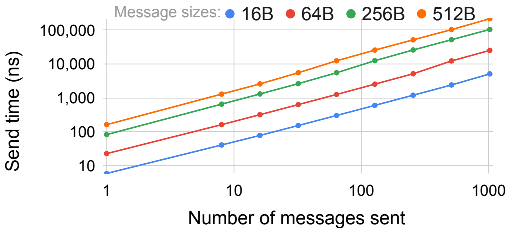


Figure 7: Time to send increasing number of messages.

We first run some microbenchmarks to measure the effect of parameters on the cost of performing small operations locally, as well as sending and receiving messages. Then, we show how these parameters are used in the design space of sorting algorithms.

6.1 Microbenchmarks

In this section we explore the parameters that are necessary for minimizing the granularity of tasks. This includes the cost of local compute, as well the overhead for various communication patterns. These determine the ratio between compute and communication. We use this to answer the question: how does the compute/communication ratio affect application runtime?

We ran experiments to evaluate the MergeMin example described in Section 3.1. Examining this simple example helps us understand the parameters in the design space, before examining the more complex NanoSort algorithm in the next section.

First, we measure the individual parameters used for the compute ratio: we look at the runtime for local operations (finding the minimum value and sorting), and communication operations (receiving/sending messages).

Figure 2a shows the time for a single RISC-V core to find the minimum value from a list of integers. The min operation is not compute bound (it consists mostly of comparison operations), but it is limited by the I/O, as we can see from the cache miss rate in Figure 2b. It takes 18us to find the minimum of 8,192 values. This is clearly too long for a nanoTask that should complete within hundreds of nanoseconds.

By using many cores in parallel, we can leverage the aggregate I/O from all the cores' caches. Ideally, with 64 cores, it should take $\frac{1}{64}^{th}$ of 18us (281ns) to find the minimum value. However, this does not include communication. When structuring communication using the merge tree described in Section 3.1, we must consider the overhead for each core to receive messages.

Figure 6 shows the time for a single nanoPU core to receive an increasing number of messages of various sizes. It takes about 8ns to receive a single 16-byte message, and 400ns to receive 64 messages. If one core were to receive and the min values from 63 other cores, it would take about 400ns.

Figure 4 shows that there is a sweet spot for the number of values to merge at each level of the merge tree. At one extreme, with an incast of 1, the tree reduces to a straight line (see Figure 3) where the runtime is dominated by propagation delay. At the other extreme, with an incast of 64, the runtime is bottle-necked by a single core receiving and merging all the values. With an incast of 8, the tree has two levels, which is the right trade-off between tree width and tree height: it finds the minimum value in 750ns.

Although MergeMin is a simple example, it provides insights into the design space and the parameters that affect performance. It is important to carefully find the appropriate balance of communication and per-core computation. We must keep this in mind when considering the design of more complex applications, as we will describe in the next section with sorting.

6.2 Sorting Benchmark

We evaluate the effect of the parameters described above on the performance of distributed sorting algorithms. We start with a microbenchmark for sorting keys locally on a single core, then we evaluate an existing distributed sorting algorithm, MilliSort. Finally, we evaluate parameters in the design space of NanoSort, and present results from large scale simulations with 65,536 cores in the next section.

6.2.1 Local Sort

Figure 8 shows the time for a single RISC-V core to sort an increasing number of keys (integers). Each point is the average of 10 runs. We cleared the cache before each run, so that they would all experience the effects of the cache hierarchy. It takes over 30us to sort 1,024 keys, which is too long for a nanoTask. The appropriate number of keys for a nanoTask appears to be at most 64 keys.

6.2.2 MilliSort

Figure 9 shows the time for MilliSort to sort an increasing number of cores with 16 keys per core. The partitioning phase of the MilliSort algorithm picks the boundaries for each of the final buckets, which equals the number of cores; the more cores, the more bucket boundaries to pick. Thus, MilliSort's partition time increases with the number of cores, causing it

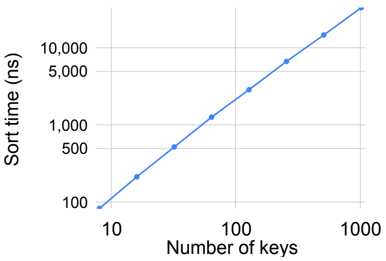


Figure 8: Sorting locally with a single core.

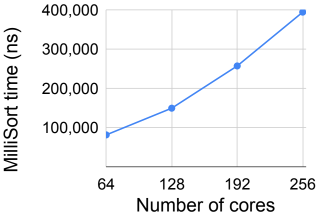


Figure 9: MilliSort runtime on nanoPU with increasing number of cores (4,096 keys, 4 incast)

to scale poorly to hundreds of cores. As we see in Figure 9, MilliSort's runtime jumps from 61us with 64 cores, to 400us with 256 cores.

To decrease MilliSort's runtime, we tried varying its key parameter: the "reduction factor", which controls the number of pivot sorters per core (essentially, the incast). Figure 10 shows that increasing MilliSort's reduction factor causes a slowdown. This is because each pivot sorter must process a larger incast.

6.2.3 NanoSort

We evaluate the effect of NanoSort "knobs", as well as the importance of network characteristics, including low switching latency, low tail latency and multicast support. Finally, we show the NanoSort runtime using 65,536 cores.

Tuning NanoSort knobs. Section 4 describes "knobs" that affect the performance of NanoSort: the bucket size, keys per core and total number of cores.

We ran NanoSort with a varying number of buckets, and fixed number of cores and keys per core. Figure 11a shows that with 4,096 cores and 32 keys/core, using 4, 8 or 16 buckets has similar performance. This is despite the fact that they generate different amounts of network traffic (Figure 11b). This is because, as shown with the microbenchmarks in Section 6.1, there is a trade-off between the tree depth and width.

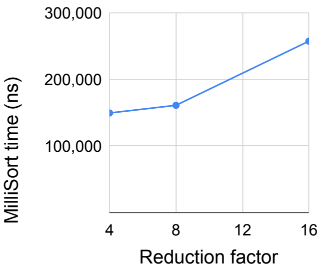
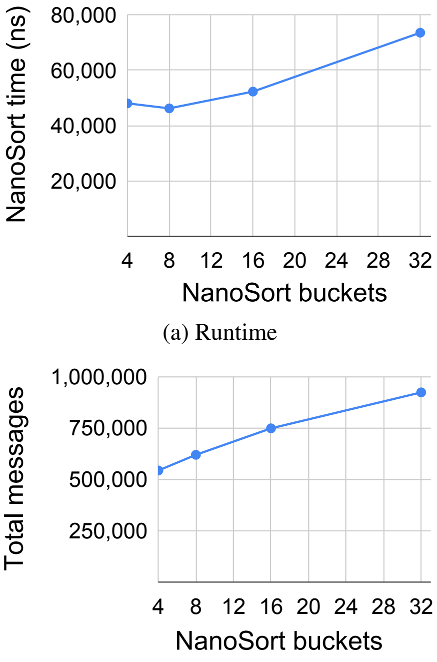


Figure 10: MilliSort runtime on nanoPU with different incast sizes (128 cores, 4,096 keys)



(a) Runtime

(b) Traffic

Figure 11: Effect of varying the number of buckets (4096 cores, 32 keys/core, 263ns switch latency).

We ran NanoSort on 4,096 cores with an increasing number of keys, as shown in Figure 12. As the number of keys increases along the x-axis, as does the number of keys per core (e.g. 16K keys uses 4 keys/core, and 262K keys uses 64 keys/core). The runtime seems to increase linearly, which is because the work per core increases proportionally to the number of keys.

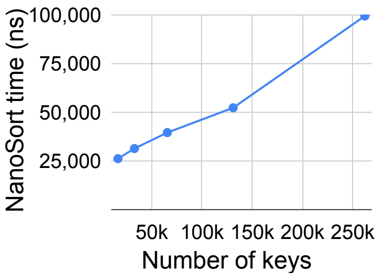


Figure 12: Sorting an increasing number of keys with a fixed number of cores (4,096 cores, 263ns switch latency).

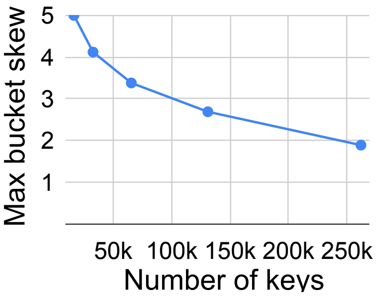


Figure 13: Skew reduces with an increasing number of keys per core (4,096 cores, 263ns switch latency).

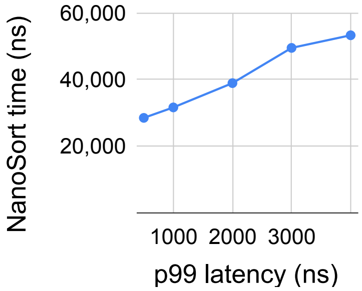
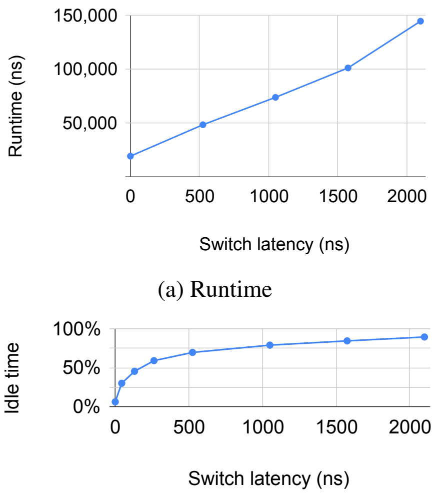


Figure 14: Effect of tail latency (256 cores, 8 buckets, 32 keys/core, 263ns switch latency)

As the total number of keys increases, the maximum skew of the final buckets decreases (Figure 13). This is because the keys per core increases, so each core has better "visibility" of the distribution, and is able to pick better medians for the bucket boundaries.

Switching latency. Figure 15 shows the effect of the switching latency on NanoSorting 1K keys with 64 cores. As expected, as the switching latency increases, so does the runtime. This is because the cores spend more time idling, waiting for messages, as we can see in Figure 15b.

Tail latency. Not only is the minimum communication time (switching latency) important, but also the tail. We ran NanoSort on 256 cores with 131K keys, and injected an additional latency to 1% of messages (i.e. the 99th percentile tail latency). Figure 14 shows how increasing this 99th percentile latency affects runtime. With a p99 latency of 4000ns, NanoSort takes 53μs, double the baseline time of 26μs. When so many messages are sent (38K messages in this experiment) it is inevitable that the tail latency will be experienced. This demonstrates that it is crucial that the latency of the tail be minimized.



(a) Runtime

(b) Idle time

Figure 15: Effect of switching latency on NanoSort (64 cores, 16 keys/core).

Multicast support. We ran NanoSort on 4,096 cores with and without multicast support in the network. Without multicast, when a core picks a median, it must individually send it to all other nodes in its group. In the first level of recursion, this means sending the same message to all other 4,096 nodes. The the effect is clear: without multicast, NanoSort takes 96μs; adding multicast support reduces the number of messages sent by 18%, and the runtime to 40μs (2.4x speedup).

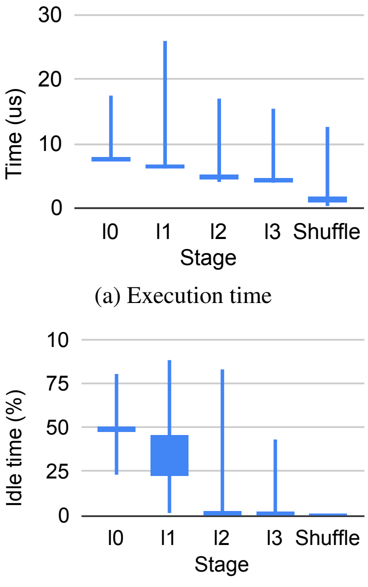
6.3 Datacenter-Scale NanoSorting

To see whether NanoSort can scale to thousands of cores, we ran a large scale simulation of 65,536 nanoPU cores using 4,224 AWS EC2 vCPUs. We measured the time for NanoSort to sort 1M keys (16 keys per node and 16 buckets). Of 10 runs, all took less than 78μs, with an average time of 68μs (4.127μs standard deviation). For comparison, this is an order of magnitude faster than MilliSort running on a HPC cluster, which sorts 0.9M keys in 1ms [27]. The state of the art sorting algorithm for multicore systems, IPS^{4}o, sorts 1M keys in 2ms [7].

Execution breakdown Figure 16a shows the distribution of time spent on each stage for all the cores. We can see that there is some variance, but it is modest. The first stage (level of recursion 0) is the fastest. It also has the least variance, because all the workers start at the same time, have exactly the same number of initial keys, and perform the same operation (send/receive each key). The variance in the other stages is not due to compute time, but the time the core spends idle, waiting for messages from other cores (possibly still in previous stages). Figure 16b shows the distribution of idle times for the cores on each stage. There is less variance in idle time in the first stages. In the last stage, shuffling, the cores are

	CPU	Cores	SMT	1M sort (μs)	Tput (records/ms/core)
NanoSort	RISC-V Rocket @ 3.2GHz	65,536	1	68	224
MilliSort	Xeon Gold 6148 @ 2.4GHz	2,240	1	1000	1297
TencentSort	IBM POWER8 @ 2.9GHz	10,240	8	N/A	1977
CloudRAMSort	Xeon X5680 @ 2.9GHz	3,072	2	N/A	707

Table 2: Per-core efficiency comparison. The throughputs are measured from sorting different numbers of records on each system: 1M with NanoSort, 26M with MilliSort, 10B with TencentSort and CloudRAMSort.



(a) Execution time

(b) Idle time

Figure 16: Execution breakdown for NanoSort with 65,536 cores.

sending and receiving the final values, so they are constantly active until completion.

Efficiency Table 2 compares the sorting throughput of NanoSort to MilliSort and other systems in the GraySort benchmark. There are multiple reasons why NanoSort has a lower per-core throughput than the other systems: it is sorting fewer records (1M) than the other systems; it has wimpier cores without SMT (8x less than TencentSort); and it is using many more cores. There is a trade-off between latency and throughput: sorting with a tight time budget comes at the expense of reduced efficiency.

7 Related Work

NanoSort explores a new extreme of granular computing using new low-latency communication designs and sorting algorithms.

Distributed sorting algorithms. Parallel sorting algorithms were designed for multi-core systems, and were initially limited by disk I/O [1,42,43]. AlphaSort [31] addresses this bottleneck with cache-sensitive external sort. Various sampling-based techniques have been proposed for sorting on a single machine: IPS^{4}o [7] describes a distribution sort optimized for multi-core machines and LearnedSort [24] creates a hierarchecal model to determine bucket boundaries. Manycore GPUs can be used to accelerate radix and merge sorts [38]. Some systems are designed for high-throughput (as opposed to low-latency): CloudRAMSort [22], TritonSort [35] and TencentSort [17] perform large scale sort using hundreds of machines.

Low-latency communication. eRPC [20] reduces RTT with off-the-shelf NICs. The nanoPU [16] and NeBULA [40] prototype new low-latency NIC/CPU codesigns. While NeBULA delivers messages to the cache, the nanoPU delivers messages straight to the CPU register file, with has a lower latency. Shinjuku [19], Shenango [32] and Zygos [34] explore OS scheduling to reduce request latency.

Granular computing. Our work addresses the questions raised by Lee et. al in their position paper on granular computing [25]. Other work has investigated increasing parallelism [12,13,18] for interactive applications and for handling “bursts”, like millisort [27].

8 Conclusion

The datasets processed by real-time systems is growing in size. If they are to continue providing interactive results, they must scale out, increasing parallelism and in turn, granularity. As granularity increases, the communication overhead and coordination dominates the processing time. We show how new network design and new algorithms can be leveraged to reduce the communication overhead and scale out with the dataset.

References

[1] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, sep 1988.

[2] Amazon ec2 instance types. `https://aws.amazon.com/ec2/instance-types/`. Accessed on 2022-01-12.

[3] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[5] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[6] Owen Astrachan. Bubble sort: An archaeological algorithmic analysis. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, page 1–5, New York, NY, USA, 2003. Association for Computing Machinery.

[7] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-Place Parallel Super Scalar Samplesort (IPSSSSo). In *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[8] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[9] B. A. W. Baugstø, J. F. Greipsland, and J. Kamerbeek. Sorting large data files on pooma. In *Proceedings of the Joint International Conference on Vector and Parallel Processing*, CONPAR 90, page 536–547, Berlin, Heidelberg, 1990. Springer-Verlag.

[10] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 65–77, New York, NY, USA, 2021. Association for Computing Machinery.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[12] Sadjad Fouladi, Dan Iter, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. A thunk to remember: make-j1000 (and other jobs) on functions-as-a-service infrastructure, 2017.

[13] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-latency Video Processing Using Thousands of Tiny Threads. In *USENIX NSDI*, 2017.

[14] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[15] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42, 2017.

[16] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 239–256. USENIX Association, July 2021.

[17] Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao, Mark R Nutter, and Jeremy D Schaub. Tencent sort. *Technical Report*, 2016.

[18] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *ACM SoCC*, 2017.

[19] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 345–360, 2019.

[20] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 1–16, 2019.

[21] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42. IEEE, 2018.

[22] Changkyu Kim, Jongsoo Park, Nadathur Satish, Hongrae Lee, Pradeep Dubey, and Jatin Chhugani. Cloudramsort: Fast and efficient large-scale distributed ram sort on shared-nothing cluster. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 841–850, New York, NY, USA, 2012. Association for Computing Machinery.

[23] Joseph B. Klerlein and Curtis Fullbright. A transition from bubble to shell sort. In *Proceedings of the Nineteenth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '88, page 183–184, New York, NY, USA, 1988. Association for Computing Machinery.

[24] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. The case for a learned sorting algorithm. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1001–1016, New York, NY, USA, 2020. Association for Computing Machinery.

[25] Collin Lee and John Ousterhout. Granular computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 149–154, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Brian Neil Levine and Jose J Garcia-Luna-Aceves. A comparison of reliable multicast protocols. *Multimedia systems*, 6(5):334–348, 1998.

[27] Yilong Li, Seo Jin Park, and John Ousterhout. MilliSort and MilliQuery: Large-Scale Data-Intensive computing in milliseconds. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 593–611. USENIX Association, April 2021.

[28] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

[29] The GitHub link omitted for double-blind review.

[30] New intel xeon platinum 9022 server packs 7 616 cores in a rack. https://www.crn.com/news/data-center/new-intel-xeon-platinum-9200-server-packs-7-616-cores Accessed on 2022-03-12.

[31] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. Alphasort: A cache-sensitive parallel external sort. *The VLDB Journal*, 4(4):603–627, 1995.

[32] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 361–378, 2019.

[33] Florentina Popovici, John Bent, Brian Forney, Andrea Arpaci-Dusseu, and Remzi Arpaci-Dusseau. Datamation 2001: A sorting odyssey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2002.

[34] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[35] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. TritonSort: A balanced Large-Scale sorting system. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.

[36] Betty Salzberg, Alex Tsukerman, Jim Gray, Michael Stuewart, Susan Uren, and Bonnie Vaughan. Fastsort: A distributed single-input single-output external sort. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, page 94–101, New York, NY, USA, 1990. Association for Computing Machinery.

[37] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on*

Networked Systems Design and Implementation (NSDI 21), pages 785–808. USENIX Association, April 2021.

[38] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, IPDPS '09, page 1–10, USA, 2009. IEEE Computer Society.

[39] Sort benchmark home page. https://sortbenchmark.org/. Accessed on 2022-01-21.

[40] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. The nebula rpc-optimized architecture. Technical report, EcoCloud, EPFL, 2020.

[41] Verilator. https://www.veripool.org/wiki/verilator. Accessed on 2020-01-29.

[42] J. Vitter and E. Lindstrom. The design and analysis of bucketsort for bubble memory secondary storage. *IEEE Transactions on Computers*, 34(03):218–233, mar 1985.

[43] Jeffrey Scott Vitter. *Algorithms and data structures for external memory*. Now Publishers Inc, 2008.