Two-Level Main Memory Co-Design: Multi-Threaded Algorithmic Primitives, Analysis, and Simulation

Michael A. Bender^a, Jonathan W. Berry^{b,*}, Simon D. Hammond^b, K. Scott Hemmert^b, Samuel McCauley^a, Branden Moore^b, Benjamin Moseley^c, Cynthia A. Phillips^b, David Resnick^b, Arun Rodrigues^b

^a*Stony Brook University, Stony Brook, NY 11794-2424 USA*

^b*Center for Computing Research, Sandia National Laboratories, Albuquerque, NM 87185 USA*

^c*Washington University in St. Louis, St. Louis, MO 63130 USA*

Abstract

A challenge in computer architecture is that processors often cannot be fed data from DRAM as fast as CPUs can consume it. Therefore, many applications are memory-bandwidth bound. With this motivation and the realization that traditional architectures (with all DRAM reachable only via bus) are insufficient to feed groups of modern processing units, vendors have introduced a variety of non-DDR 3D memory technologies (Hybrid Memory Cube (HMC), Wide I/O 2, High Bandwidth Memory (HBM)). These offer higher bandwidth and lower power by stacking DRAM chips on the processor or nearby on a silicon interposer. We will call these solutions “near-memory,” and if user-addressable, “scratchpad.” High-performance systems on the market now offer two levels of main memory: near-memory on package and traditional DRAM further away. In the near term we expect the latencies near-memory and DRAM to be similar. Thus, it is natural to think of near-memory as another module on the DRAM level of the memory hierarchy. Vendors are expected to offer modes in which the near memory is used as cache, but we believe that this will be inefficient.

In this paper, we explore the design space for a user-controlled multi-level main memory. Our work identifies situations in which rewriting application kernels can provide significant performance gains when using near-memory. We present algorithms designed for two-level main memory, using divide-and-conquer to partition computations and streaming to exploit data locality. We consider algorithms for the fundamental application of sorting and for the data analysis kernel k -means. Our algorithms asymptotically reduce memory-block transfers under certain architectural parameter settings. We use and extend Sandia National Laboratories’ SST simulation capability to demonstrate the relationship between increased bandwidth and improved algorithmic performance. Memory access counts from simulations corroborate predicted performance improvements for our sorting algorithm. In contrast, the k -means algorithm is generally CPU bound and does not improve when using near-memory except under extreme conditions. These conditions require large instances that rule out SST simulation, but we demonstrate improvements by running on a customized machine with high and low bandwidth memory. These case studies in co-design serve as positive and cautionary templates, respectively, for the major task of optimizing the computational kernels of many fundamental applications for two-level main memory systems.

Keywords: two-level memory, high-bandwidth memory, sorting, k -means clustering

1. Introduction

Recently vendors have proposed a new approach to improve memory performance by increasing the bandwidth between cache and memory [1, 2]. The approach is to bond memory directly to the processor chip or to place it nearby on a silicon interposer. By placing memory close to the processor, there can be a higher number of connections between the memory and caches, enabling higher bandwidth than current technologies. While the term *scratchpad* is overloaded

*Corresponding author

Email addresses: bender@cs.stonybrook.edu (Michael A. Bender), jberry@sandia.gov (Jonathan W. Berry), sdhammo@sandia.gov (Simon D. Hammond), kshemme@sandia.gov (K. Scott Hemmert), smccauley@cs.stonybrook.edu (Samuel McCauley), bjmoor@sandia.gov (Branden Moore), moseleyb85@gmail.com (Benjamin Moseley), caphill@sandia.gov (Cynthia A. Phillips), drresni@sandia.gov (David Resnick), afrodni@sandia.gov (Arun Rodrigues)

within the computer architecture field, we use it throughout this paper to describe a high-bandwidth, local memory that can be used as a temporary storage location. Note that this term also refers to high speed internal memory used for temporary calculations [3, 4] which is a different technology than that discussed in this paper.

The scratchpad cannot replace DRAM entirely. Due to the physical and cost constraints of adding the memory directly to the chip, the scratchpad cannot be as large as DRAM, although it will be much larger than cache, having gigabytes of storage capacity. Since the scratchpad is smaller than main memory, it does not fully replace main memory, but instead augments the existing memory hierarchy.

The scratchpad has other limitations besides its size. First, the scratchpad does not significantly improve upon the latency of DRAM. Therefore, the scratchpad is not designed to accelerate memory-latency-bound applications, but rather memory-bandwidth-bound ones. Second, adding a scratchpad does not improve the bandwidth between DRAM and cache. Thus, the scratchpad will not accelerate a computation that consists of a single scan of a large chunk of data that resides in DRAM.

This gives rise to a new multi-level memory hierarchy where two of the components—the DRAM and the scratchpad—work in parallel. We view the DRAM and scratchpad to be on the same *level* of the hierarchy because their access times are similar. There is a tradeoff between the memories: the scratchpad has limited space and the DRAM has limited bandwidth.

Since the scratchpad does not have its own level in the cache hierarchy, when a record is evicted from the cache there is an algorithmic decision whether to place the record in the scratchpad or directly into the DRAM. In the currently-proposed architecture designs, this decision is user-controlled.

Under the assumption that the memory is user-controlled, an algorithm must coordinate memory accesses from main memory and the scratchpad. Ideally the faster bandwidth of the scratchpad can be leveraged to alleviate the bandwidth bottleneck in applications. Unfortunately, known algorithmics do not directly apply to the proposed two-level main-memory architecture. The question looms, can an algorithm use the higher bandwidth scratchpad memory to improve performance? Unless this question is answered positively, the proposed architecture, with its additional complexity and manufacturing cost, is not viable.

Our interest in this problem comes from *Trinity* [5], the latest NNSA (National Nuclear Security Administration) supercomputer architecture. This supercomputer uses the Knight’s Landing processor from Intel with Micron memory. This processor chip uses such a two-level memory [2]. The mission for the Center of Excellence for Application Transition [1], a collaboration between NNSA Labs (Sandia, Los Alamos, and Lawrence Livermore), Intel, and Cray, is to ensure applications can use these new architectures.

1.1. Results

In this paper we introduce an algorithmic model of the scratchpad architecture, generalizing existing sequential and parallel external-memory models [6, 7]. (See Section 9 for a brief background on the architectural design and motivation for the scratchpad.) We introduce theoretically optimal scratchpad-optimized, sequential and parallel sorting algorithms. We report on hardware simulations varying the relative-bandwidth parameter. These experiments suggest that the scratchpad will improve running times for sorting on actual scratchpad-enhanced hardware for a sufficient number of cores and sufficiently large bandwidth improvement. We also study the data analysis technique k -means clustering for contrast.

Theoretical Contributions

We give an algorithmic model of the scratchpad, which generalizes existing sequential and parallel external-memory models [6, 7]. In our generalization, we allow two different block sizes, B and ρB ($\rho > 1$) to model the bandwidths of DRAM and the larger bandwidth of the scratchpad. Specifically, $\rho > 1$ is the relative increase in bandwidth of the scratchpad in comparison to DRAM.

We give scratchpad-aware algorithms for two basic problems: sorting and k -means clustering. In both cases, we obtain asymptotic reductions in memory block transfers compared to conventional algorithms. However, we find that our sorting algorithms are more practical than our k -means algorithms. Our experience in designing and experimenting with these algorithms has led us to enumerate several basic properties of successful scratchpad-aware algorithms. We present these in Section 1.1.

We exhibit, under reasonable architectural assumptions of the scratchpad, sorting algorithms that can achieve speedups proportional to ρ over the state-of-the-art sorting algorithms that use only far DRAM. We begin by introducing a sequential algorithm for sorting, and then generalize to the multiprocessor (single node) setting. We complement

this result by giving a matching lower bound in our theoretical model. Our algorithms supply theoretical evidence that the proposed architecture can indeed speed up fundamental applications.

Algorithm Engineering Contributions

For an application to benefit from a scratchpad-enhanced memory architecture, good data locality is necessary, but it is not sufficient. For example, consider a program that scans a large array once. This program has great data locality. However, running this program on a memory system with a scratchpad will not help. Moving the data to the scratchpad before computation is effectively just an extra copy.

The following definitions help describe the conditions under which a scratchpad-enhanced memory can help. A computation is *memory bound* if the limiting factor in its running time is the time to feed data to the processors. Adding more compute power does not improve runtime in any significant way, but delivering data faster will. A process is *compute bound* if the limiting factor is the total processing power. A compute-bound computation might run faster if given more or faster processors, but is unlikely to benefit from faster memory.

Intuitively, a computation can be helped by a scratchpad-enhanced memory architecture if that computation has the following properties.

1. **(Memory bandwidth boundedness)** In the absence of scratchpad, *any* variant of the computation is memory bound. This includes any newly discovered algorithm designed to optimize for two-level memory.
2. **(Scratchpad chunking is appropriate)** The input data can be decomposed into portions that are larger than aggregate on-node cache, smaller than scratchpad, and independent. Each portion can be gathered into scratchpad from far memory, then processed in scratchpad. Partial results can be combined without excessive overhead.
3. **(Scratchpad chunk reuse)** After being copied to scratchpad, each chunk needs to be scanned sufficiently many times to amortize the cost of the copy.
4. **(Final check)** The new algorithm is not cache-chunkable. If the new algorithm works when keeping chunks in cache, near-memory is superfluous.

We believe that these properties are necessary and sufficient to warrant a rewrite of any fundamental application kernel possessing them. However, we also emphasize that the lack of any one of these properties disqualifies such an effort. A new algorithm may seem ideal for scratchpad. However, if it can exploit cache just as well, there is no benefit from scratchpad. Algorithm designers must understand the divide-and-conquer strategy of a scratchpad-aware algorithm well in order to argue that it will produce any significant speedup. We will provide discussions of these properties for each of our algorithmic case studies, noting in advance that the case for rewriting sorting kernels is quite strong, while that for k -means is much less so. We include the latter for pedagogical purposes. Understanding these contrasting case studies helped us state the three properties above.

Empirical contributions

After theoretically establishing the performance of our algorithms, we performed empirical evaluations. The scratchpad architecture has yet to be generally available, so we could not perform experiments on a deployed system. Instead, we employ two alternatives: (1) we extended Sandia National Laboratories' SST [8] simulator to allow for a scratchpad memory architecture with variable bandwidth, and ran simulations over a range of possible hardware parameters, and (2) we customize a single-node machine to mimic two-level memory. The sorting instances are generally small enough for approach (1), though we expect better results from larger instances (see Section 5). It is more difficult to find instances of k -means that benefit from scratchpad; these are larger and require approach (2).

In both cases, our empirical results corroborate the theoretical results. For sorting, we compare our algorithm to the well-established parallel GNU multiway merge sort [9]. In simulation, we show a definite reduction in running time for our algorithm when increasing the bandwidth from two to sixteen times that of far DRAM. We also show that in the when the application is not memory bound, we do not achieve improved performance.

Our empirical and theoretical results together show the first evidence that the scratchpad architecture could be a feasible way to improve the performance of memory-bandwidth-bound applications.

Algorithms and architecture co-design

This work is an example of co-design. The possible architectural design motivates algorithm research to take advantage of the new architectural features. The results help determine ranges of architectural parameters where our scratchpad-aware algorithms will be practical.

Figure 1: The scratchpad memory model.

In particular, we find that the bandwidth-expansion parameter ρ introduced in the theoretical model also is an important performance parameter in our simulations. We estimate the ratio of processing capability to memory bandwidth necessary for sorting to become memory-bandwidth bound. From this, we estimate the number of cores that must be on a node at today’s computing capability for the scratchpad to be of benefit. The core counts and minimum values of ρ could guide vendors in the design of future scratchpad-based systems.

The SST extension, designed and implemented by computer architects, is a co-design research tool. SST is an open source, modular framework for designing simulations of advanced computing systems [8, 10]. This extension will become available in SST, enabling algorithms researchers to test scratchpad algorithms before obtaining hardware such as Knights Landing.

The experimental results we report compare redesigned kernels run with two-level memory to original versions run on conventional systems with only one level of memory: far DRAM. These results do not consider the original versions as they might run on a future system with two-level memory. Such versions might benefit, for example, from scratchpad-as-cache, and/or via compiler pragmas or systems software that direct temporary data structures to scratchpad. Such experiments would be necessary to inform the costly decision to redesign a kernel. The trade-off between this development cost and the true performance boost is therefore beyond the scope of this paper. However, we do claim that verifying the design properties we state is a prerequisite to user-level redesign efforts aimed at exploiting scratchpad.

2. Algorithmic Scratchpad Model

In this section, we propose an algorithmic model of the scratchpad, generalizing the external-memory model of Aggarwal and Vitter [6] to include high- and low-bandwidth memory.

Scratchpad model

Both the DRAM and the scratchpad are independently connected to the cache; see Figure 1. The scratchpad hardware has an access time that is roughly the same as that of normal DRAM—the difference is that the scratchpad has limited size but higher bandwidth. We model the higher bandwidth of the scratchpad by transferring data to and from the DRAM in smaller blocks of size B and to and from the scratchpad in larger blocks of size ρB , $\rho > 1$.

We parameterize memory as follows: the cache has size Z , the scratchpad has size $M \gg Z$, and the DRAM is modeled as arbitrarily large. We assume a tall cache; that is, $M > B^2$. This is a common assumption in external-memory analysis, see e.g. [11, 12, 13, 14].

We model the roughly similar access times of the actual hardware by charging each block transfer a cost of 1, regardless of whether it involves a large or small block. Performance is measured in terms of block transfers. Computation is modeled as free because we consider memory-bound computations.

Note that the scratchpad block size, ρB , in the algorithmic performance model need not be the same as the block-transfer size in the actual hardware. But for the purpose of algorithm design, one should program assuming a block size of ρB , so that the latency contribution for a transfer is dominated by the bandwidth component.

In the following, we say that event E_N on problem size N occurs *with high probability* if $\Pr(E_N) \geq 1 - \frac{1}{N^c}$, for some constant $c > 0$.

- **Processor of 256-cores split into quad-core groups**
 - Cores run at 1.7GHz
 - Each core has a private 16KB L1 data cache
 - Each core group shared a 512KB L2 cache
 - Each core group has a 72GB/s connection to the on-chip network
- **Far (Capacity) Memory (e.g., DRAM)**
 - 1066MHz DDR configuration
 - 4 channels
 - 36GB/s connection per channel to the on-chip network
 - Approx. 60GB/s of STREAM Triad Performance
 - Block size 64 bytes
- **Scratchpad Near Memory**
 - 500MHz clock, 50ns constant access latency
 - 8, 16 or 32 channels of memory (2X, 4X and 8X bandwidth)
 - Block size 64 bytes

Figure 2: Simulation system parameters.

Putting the scratchpad model in context

The external-memory model of Aggarwal and Vitter [6], assumes a two-level hierarchy comprised of a small, fast memory level and an arbitrarily large second level. Data is transferred between the two levels in blocks of size B .

Because the scratchpad model has two different bandwidths but roughly similar latencies, we have extended the external-memory model to have two different block sizes, each with the same transfer cost; see Figure 1.

There do exist other memory-hierarchy models, such as the cache-oblivious model [14, 15], which apply to multi-level memories. However, the different bandwidths but similar latencies means that the cache-oblivious model does not seem to apply. Instead, the scratchpad model can be viewed as a three-level model, but where two of the levels are in parallel with each other.

In our experiments, we simulated a multi-level memory system with the parameters shown in Figure 2.

Background on external-memory sorting

Aggarwal and Vitter [6] first validated their external-memory model using sorting. In Section 3 we will follow their example for the scratchpad model.

The following well-known theorems give bounds on the cost of external-memory sorting.

Theorem 1 ([6]). *Sorting N numbers from DRAM with a cache of size Z and block size L (but no scratchpad) requires $\Theta((N/L) \log_{Z/L}(N/L))$ block transfers from DRAM. This bound is achievable using multi-way merge sort with a branching factor of Z/L .*

Theorem 2 ([6]). *Sorting N numbers from DRAM with a cache of size Z and block size L (but no scratchpad) using merge sort takes $\Theta((N/L) \lg(N/Z))$ block transfers from DRAM.*

In the analysis of sorting algorithms in the following sections, we will sometimes apply these theorems with $L = B$ and sometimes apply them with $L = \rho B$.

3. Sorting with a Scratchpad

We begin our design of a scratchpad-aware algorithm for sorting by considering our design properties.

Pre-validation of scratchpad design properties for sorting

1. (Memory bandwidth boundedness)

We the hypothetical, but realistic multi-level memory system described in Figure 2. In this system, we predict that sorting will be memory bound based on the following simple computation:¹ (1) given the available processing rate and the expected amount of work that needs to be done, compute the expected processing time, and (2) given the available memory bandwidth on/off-chip and the expected amount of memory transfers required, compute the expected time required for memory access. If the expected processing time is much less than the expected memory-transfer time, the application is memory-bandwidth bound.

Let x be the processing rate in operations per second (in our case, integer comparisons in the sort). Let y be the memory bandwidth between off-chip memory and on-chip cache (in array elements per second). Further, let N be the number of elements to sort and Z be the number of blocks in on-chip memory (i.e., L1, L2, L3 cache). Up to constants, we derive the inequality:

$$\frac{N \log N}{x} < \frac{N \log N}{y \log Z},$$

because there are $N \log N$ comparisons in the algorithm and the minimum aggregate memory transfer is $N \frac{\log N}{\log m}$ [6]. Before plugging in the numbers, we can rearrange the inequality:

$$y \log Z < x.$$

Thus, the sorting instance size is not a factor in this computation. In our experimental setup, we would have (roughly), $Z \approx 10^6$, $x \approx 10^{10}$, and $y \approx 10^9$. Ignoring constants, we see that these quantities are comparable:

$$10^9 \log 10^6 \approx 10^{10}.$$

2. **(Scratchpad chunking is appropriate)** We design and analyze a specialized hybrid of sample-sort and bucket-sort. Our algorithm is a comparison-based method that sorts integers. The first phase of the algorithm sorts independent chunks of data, and the second phase assembles partial results. The crucial feature of the algorithm that leads to good performance is the reassembly of sub-problems. This can be done by transferring scratchpad-sized chunks between DRAM and scratchpad.
3. **(Scratchpad chunk reuse)** After being copied to scratchpad, each chunk of elements can be sorted by a conventional multi-threaded sorting algorithm. This entails many passes over the data, amortizing the copy.
4. **(Final check)** If we were to decrease the size of these chunks so that they fit in cache, the small subproblems would produce many small results. This would minimize the advantage of sorting large chunks in scratchpad and maximize the disadvantage of assembling many partial results.

Scratchpad-aware algorithms for sorting

This section gives an optimal randomized algorithm for sorting with one processor with a scratchpad. This algorithm generalizes multiway merge sort and distribution sort [6], which in turn generalize merge sort and sample sort [16].

We are given an array $A[1, \dots, N]$ of N elements to sort, where $N \gg M$. For simplicity of exposition we assume that all elements in A are distinct, but this assumption can be removed.

The algorithm works by recursively reducing the size of the input until each subproblem fits into the scratchpad, at which point it can be sorted rapidly. In each recursive step, we “bucket” elements so that for any two consecutive buckets (or ranges) R_i and R_{i+1} , every element in R_i is less than every element in R_{i+1} . We then sort each bucket recursively and concatenate the results, thus sorting all elements.

3.1. Choosing Bucket Boundaries

To perform the bucketing, we randomly select a sample set X of $m = \Theta(M/B)$ elements from $A[1, \dots, N]$ and sort them within the scratchpad. (We assume sampling with replacement, but sampling without replacement also works.)

The exact value of m is chosen by the algorithm designer, but must be small enough to fit in the scratchpad.

Our implementation uses multiway mergesort from the GNU C++ library to sort within the scratchpad [17]. Other sorting algorithms could be used, such as quicksort. If ρ is sufficiently large, either sorting algorithm within the scratchpad leads to an optimal algorithm (see Theorem 6); however, the value of ρ based on current hardware probably is not large enough to make quicksort practically competitive with mergesort.

¹ We thank Marc Snir for personal communication suggesting when applications become bandwidth bound.

Corollary 3. *Sorting x elements that fit in the scratchpad uses $\Theta((x/\rho B) \log_{Z/B} x/B)$ block transfers using multi-way merge sort with a branching factor of Z/B , or $\Theta((x/\rho B) \lg(x/Z))$ in expectation using quicksort. Both algorithms use $O(x \lg x)$ work.*

3.2. Bucketizing

We are given the sample set $X = \{x_1, x_2, \dots, x_m\}$ ($x_1 < x_2 < \dots < x_m$) of sorted elements and input array A . Since $|X| = m$, X fits in the scratchpad, and A does not.

Our objective is to place each element $A[i] \in A \setminus X$ into bucket R_j if $x_j < A[i] < x_{j+1}$. (We may assume that $x_0 = -\infty$, and $x_{m+1} = \infty$.)

We perform a linear scan through A , ingesting $M - \Theta(m)$ elements into the scratchpad at a time. All of X remains in the scratchpad for the entire scan.

Once each new group of elements is loaded into the scratchpad, we sort them (along with X). Their position relative to the elements of X indicate which bucket they are stored in. The resulting bucketed elements get written back to DRAM, while the set X remains in the scratchpad. We call this process a *bucketizing scan*.

Lemma 4. *Each bucketizing scan costs:*

1. $O(N/B)$ block transfers from DRAM,
2. $O(N/(\rho B) \log_{Z/(\rho B)} M/(\rho B))$ block transfers from the scratchpad, and
3. $O(N \lg M)$ operations in the RAM model.

Proof. We select and sort the set X of sampled points in the scratchpad (necessary for mergesort), and then bucketize the rest of A . Assume for simplicity that $m \leq M/2$.

Sampling X requires $O(m)$ block transfers and $O(m)$ work, given constant time to choose a random word. Sorting X requires $O(m \lg m)$ work and $O(\frac{m}{\rho B} \log_{Z/B} \frac{m}{B})$ block transfers (via Corollary 3).

To bucketize, we read all elements from DRAM to the scratchpad in blocks of size $M - \Theta(m) \geq M/2$. This requires $O(M/B)$ block transfers from DRAM. The algorithm sorts each of these blocks in $O(M/(\rho B) \log_{Z/(\rho B)} M/(\rho B))$ scratchpad block transfers and requires $O(M \lg M)$ work to decide their buckets. There are $O(N/M)$ such groups, so this requires $O(N/B + N/(\rho B) \log_{Z/(\rho B)} M/(\rho B))$ total block transfers. We write each bucket to a separate piece of DRAM memory. This does not increase the cost: were the writing done to a contiguous location, it would require $\Theta(M/B)$ block transfers as the reading does. By writing the elements in $O(m) = O(M/B)$ pieces, we may incur up to two extra block transfers per piece: one to start each bucket, and perhaps another to end it. Thus the total number of transfers to write buckets to DRAM (even in separate pieces) is $\Theta(M/B)$. $\square$

3.3. Bounding the Recursion Depth for Sorting

We now conclude our analysis of scratchpad sorting. Recall that the algorithm successively scans the input, refining it into progressively smaller buckets. Lemma 4 from the previous section gave a bound of the complexity of each scan.

In this section, we first show that the random samples are good enough so that with high probability each bucket is small enough to fit into cache after $O(\log_m(N/M))$ bucketizing scans. Then we give a bound on the sorting complexity.

This randomized analysis applies to any external-memory sort. However, to the best of our knowledge, this kind of analysis does not appear in the literature. Previous external memory sorting algorithms were deterministic. However, randomized algorithms are practical.

Lemma 5. *Each bucket fits into cache after $O(\log_m(N/M))$ bucketizing scans with high probability.*

Proof. Each newly-created bucket results from a *good* or *bad* split depending on its size. If a split makes a new bucket at least $\sqrt{m}$ factor smaller than the old one we call this a good split; otherwise it is bad.

We show that in a sufficiently large set of $O(\log_m(N/M))$ splits, there are $3 \log_m(N/M)$ good splits with high probability. Thus, after $O(\log_m(N/M))$ scans, each bucket is involved in at least $3 \log_m(N/M)$ good splits, and fits in scratchpad.

We first show that the probability of a bad split is exponentially small in $\sqrt{m}$. Let R_i be the bucket being split. For simplicity, assume R_i lists the elements in sorted order, though the bucket itself may not be sorted yet. If a subbucket starting at j came from a bad split, no element in $\{R_i(j), R_i(j+1), \dots, R_i(j + (|R_i|/\sqrt{m}/m))\}$ is sampled to be in X . This happens with probability at most $(1 - \sqrt{m}/m)^m \approx e^{-\sqrt{m}}$.

We next show that of any $(3/2)c \log_m(N/M)$ splits on a root-to-leaf path of the recursion tree, at most $c \log_m(N/M)$ are bad with high probability. Thus there are at least $(c/2) \log_m(N/M)$ good splits; choosing $c \geq 20$ proves the lemma. Our proof applies for any positive α with a sufficiently large constant c : intuitively, this means that more bucketizing scans (i.e. cases with high c) occur with smaller probability. For the remainder of the proof, assume that $c \geq \max\{20, 4\alpha\}$.

By linearity of expectation, the expected number of bad splits is at most $(3/2)ce^{-\sqrt{m}} \log_m(N/M)$. Applying Chernoff bounds [18] with $\delta = (2/3)e^{\sqrt{m}} - 1$, we obtain

$$\Pr \left[\# \text{ bad splits} > c \log_m \frac{N}{M} \right] \leq \left(\frac{e}{2e^{\sqrt{m}}/3} \right)^{c \log_m \frac{N}{M}}.$$

We want to show that this is bounded above by $1/N^\alpha$ for some positive constant α . Taking the log of both sides of the inequality, this simplifies to

$$c((\sqrt{m} - 1) \lg e - \lg 3/2) \log_m N/M \geq \alpha \lg N.$$

Now there are two cases. First, assume $M \geq \sqrt{N}$. Recall the tall cache assumption, that $M > B^2$. Then $\sqrt{m} = \sqrt{M/B} \geq M^{1/4} \geq N^{1/8}$. Substituting, we obtain $c((N^{1/8} - 1) \lg e - \lg 3/2) \log_m N/M \geq \alpha \lg N$, which is true for sufficiently large N since $c \geq \alpha$.

On the other hand, let $M < \sqrt{N}$. Note that for $m \geq 2$, $((\sqrt{m} - 1) \lg e - \lg 3/2) / \lg m \geq 1/2$. Then for $c \geq 4\alpha$,

$$c((\sqrt{m} - 1) \lg e - 2) \log_m \frac{N}{M} \geq \frac{c}{2} \lg \frac{N}{M} \geq \alpha \lg N.$$

This shows that each bucket is sufficiently small with high probability after $O(\log_m N/M)$ bucketizing scans. To show that all must be small enough, take the union bound to obtain a probability no more than $1/N^{\alpha-1}$. Performing the analysis with $\alpha' = \alpha + 1$ gives us the desired bound. $\square$

Theorem 6. *There exists a sorting algorithm that performs $\Theta\left(\frac{N}{B} \log_{M/B} \frac{N}{B} + \frac{N}{\rho B} \log_{Z/\rho B} \frac{N}{B}\right)$ block transfers with high probability to partition the array into buckets, all of size $\leq M$. This bound comprises $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ block transfers from DRAM and $O(\frac{N}{\rho B} \log_{Z/\rho B} \frac{N}{B})$ high bandwidth block transfers from the scratchpad. There is a matching lower bound, so this algorithm is optimal.*

Proof. By Lemma 5 there are $O(\log_m N/M) \leq O(1 + \log_m N/M) = O(\log_{M/B} N/B)$ bucketizing scans; multiplying this by the block transfers per scan in Lemma 4 gives us the desired bounds. Note that once the buckets are of size $\leq M$, we can sort all of them with $O(\frac{N}{\rho B} \log_{Z/\rho B} \frac{M}{\rho B})$ additional block transfers by Corollary 3; this is a lower-order term.

To obtain the lower bound we will combine two lower bounds from weaker models. First consider a model where computation can be done in the scratchpad as well as in cache. Then the standard two-level hierarchy lower bound applies, and sorting requires $\Omega(\frac{N}{B} \log_{M/B} \frac{N}{B})$ block transfers from DRAM (since $M + Z = O(M)$). This bound is the same as the one given in Theorem 1, and can be found in [6].

Consider a second model where the DRAM allows high-bandwidth block transfers. Again, the standard two-level hierarchy lower bound applies, and sorting requires $\Omega(\frac{N}{\rho B} \log_{Z/(\rho B)} \frac{N}{\rho B})$ high-bandwidth block transfers.

Both of these bounds must be satisfied simultaneously. Thus, all together we require $\Omega(\frac{N}{B} \log_{M/B} \frac{N}{B} + \frac{N}{\rho B} \log_{Z/\rho B} \frac{N}{\rho B})$ block transfers. Since $(N/\rho B) \log_{Z/\rho B} \rho < N/B$, this simplifies to $\Omega\left(\frac{N}{B} \log_{M/B} \frac{N}{B} + \frac{N}{\rho B} \log_{Z/\rho B} \frac{N}{B}\right)$. $\square$

The following corollary explains how the sorting algorithm performs when quicksort is used within the scratchpad, and follows by linearity of expectation. The corollary shows that the algorithm remains optimal as long as the bandwidth of the scratchpad, determined by ρ , is sufficiently large.

Corollary 7. *An implementation of sorting using quicksort within the scratchpad uses $O(\frac{N}{B} \log_{M/B} \frac{N}{B} + \frac{N}{\rho B} \lg \frac{M}{Z} \log_{M/B} \frac{N}{B})$ block transfers in expectation. This is optimal when $\rho = \Omega(\lg M/Z)$.*

We conclude this section by showing how to use our new theory to make concrete predictions of the reduced DRAM usage by our algorithm.

Theorem 1 predicts $\Theta((N/L) \log_{Z/L}(N/L))$ block transfers from DRAM (no scratchpad), where L is the block size. For DRAM, this is B . Theorem 6 predicts $\Theta((N/B) \log_{M/B}(N/B))$ block transfers from DRAM by our theoretical sorting algorithm with scratchpad. Let's examine the ratio:

$$\frac{(N/B) \log_{M/B}(N/B)}{(N/B) \log_{Z/B}(N/B)}.$$

Let $N/B = a$, $Z/B = c$, and $(M/B) = (Z/B)^b$, and assume that the hidden constants are similar. The ratio of interest becomes:

$$\frac{\log_{(c^b)} a}{\log_c a} = \frac{1}{b}.$$

With realistic values, $Z \approx 2^{25}$ (64 L2 caches of size 512K), $M \approx 2^{31}$ (2GB scratchpad), and $B = 2^6$, we need to solve for b :

$$2^{25} = (2^{19})^b,$$

so $25 = 19b$, and $b = \frac{25}{19}$. We therefore expect the ratio of DRAM access by our algorithm to that of a classical one to be about 0.75. In Section 5 we report empirical ratios of 0.3 to 0.4.

4. Sorting in Parallel

In this section, we show how to sort using a scratchpad that is shared by multiple CPUs. Our objective is to model one compute node of a scratchpad-enhanced supercomputer.

4.1. Parallel Scratchpad Model

The parallel scratchpad model resembles the sequential model, except that there are p processors all occupying the same chip. Each processor has its *own* cache of size Z , and all p processors are connected to a *single* size- M scratchpad and to DRAM. In DRAM the blocks have size B and in the scratchpad ρB .

In any given *block-transfer step* of the computation, our model allows for up to p' processors to perform a block transfer, either from DRAM or from the scratchpad. Thus, although there are p processors, bandwidth limitations reduce the available parallelism to some smaller value p' . (In the sequential case, $p' = 1$, and in the fully parallel case, $p' = p$.)

4.2. Other Parallel External-Memory Models

The parallel external-memory (PEM) model [7] has p processors each with its own cache of size M . These caches are each able to access an arbitrarily large external memory (DRAM in our terminology) in blocks of size B . These block transfers are the only way that processors communicate. This model works with CRCW, EREW, and CREW parallelism, though most research in this model (i.e. [19, 20, 21]) uses CREW.

Another line of study has examined *resource-oblivious* algorithms, which work efficiently without knowledge of the number of processors [22, 23, 24]. This work uses a similar model to PEM, but concentrates on the algorithmic structure. Subsequent work extended this to the hierarchical multi-level multicore (HM) model, featuring a multi-level hierarchy [25].

The Parallel Disk Model [26] is another model of a two-level hierarchy with multiple processors. However, this model has the important distinction of having multiple disks (multiple DRAMs in our terminology). Multiple disks—rather than a single, external memory—lead to fundamentally different algorithmic challenges.

Other parallel external-memory models take more parameters into account, modeling communication time between processors [27], deep hierarchies [28], sets of sibling deep hierarchies [29], and multilevel cache hierarchies [30]. Our simple model targets the special case of sibling scratchpad and DRAM. We do not address issues such as asynchronous data transfer, as the primary aim of our theoretical modeling is asymptotic understanding of aggregate memory transfer. Asynchronous transfer of memory via double-buffering, for example, would speed up scratchpad algorithms in practice. However, is not part of our analysis or experimentation.

Figure 3: Phase 1 of NMsor.

4.3. Parallel Sorting

To obtain a parallel scratchpad sorting algorithm, we parallelize two subroutines from the sequential sort. In particular, we ingest blocks into the scratchpad in parallel, and we sort within the scratchpad using a parallel external-memory sort. These two enhancements reduce the bounds in Theorem 6 by a factor of p' .

We begin by analyzing the bucketizing procedure. We can randomly choose the elements of X and move them into the scratchpad in parallel.

To see how to sort in parallel, observe that when we have p processors, each with its own cache, but sharing the same scratchpad, we have the PEM model [7] and can apply PEM sorting algorithms.

Theorem 8 ([7]). *Sorting N numbers in the PEM model with p' processors, each with a cache of size Z and block size L , requires $\Theta((N/p'L) \log_{Z/L}(N/L))$ block-transfer steps.*

By applying Theorem 8, we obtain a bound for a parallel bucketizing scan, replacing the bound from Lemma 4.

Lemma 9. *Each parallel bucketizing scan costs $O(N/(p'B)) + O(N/(p'\rho B) \log_{Z/(\rho B)} M/(\rho B))$ block-transfer steps, of which $O(N/(p'B))$ are from DRAM and $O(N/(p'\rho B) \log_{Z/(\rho B)} M/(\rho B))$ are from the scratchpad.*

Proof. Sampling to obtain X and moving X to the scratchpad requires $O(m/p')$ block-transfer steps.

During the scan, we bring each group of $M - m$ elements into the scratchpad in parallel, using $O(M/Bp')$ block-transfer steps. Then we sort these elements within the scratchpad using $O(\frac{M}{p'\rho B} \log_{Z/\rho B} \frac{M}{\rho B})$ block-transfer steps via Theorem 8. Multiplying by the number of groups of $M - m$ elements, $\Theta(N/M)$, gives the final bounds. $\square$

Combining with Lemma 5, we obtain the total number of block transfers.

Theorem 10. *Sorting on a node that allows p' processors to make simultaneous block transfers requires $O(\frac{N}{p'B} \log_{M/B} \frac{N}{B} + \frac{N}{p'\rho B} \log_{Z/(\rho B)} \frac{N}{B})$ block-transfer steps with high probability.*

4.4. Practical Sorting in Parallel

In the algorithm described above, after each “chunk” of values gets sorted and bucketized in the scratchpad, the elements in the scratchpad that belong in bucket i , are appended to an array in DRAM that stores all bucket i ’s elements. Empirically, the number of elements destined for any given bucket might be small, so these appends can be inefficient.

As a practical improvement, we modify the algorithm to do fewer small memory transfers to DRAM. The practical algorithm, denoted Near-Memory sort or *NMsor*, operates in two phases, depicted respectively in Figures 3 and 4

Figure 4: The beginning of Phase 2 of NMsor.

and described in detail below. In Phase 1 we sort the chunks of elements and store metadata associated with bucket information rather than explicitly creating individual buckets. The bucket metadata enables us to avoid small memory transfers. Without this innovation, we were unable to exploit the scratchpad effectively.

We start Phase 1 by loading a chunk of $\Theta(M)$ elements from DRAM into the scratchpad and sorting it using a parallel algorithm (Figure 3(a)). That sorted chunk can immediately be scheduled for transfer back to DRAM (Figure 3(b)). This transfer could happen asynchronously as work proceeds, but our prototype implementation simply waits for the transfer to complete. Thus, it is likely that the simulation results we present later could be nontrivially improved.

After sorting and writing, we extract bucket boundaries in parallel from the sorted pivots array. Instead of populating individual buckets with chunk elements at that point, we simply record the bucket boundaries. Specifically, for each $\Theta(M)$ -sized chunk of values, we create an array of size $|X| = \Theta(M/B)$ where the i th element gives the index for the first element that belongs in bucket i . We then write out this auxiliary array, called *BucketPos* in Figure 3(c), to DRAM. We also keep global counts of the number of elements in each bucket in an array, called *BucketTot* in Figure 3(c). This array remains in scratchpad throughout both phases, and will hold the aggregate bucket counts from all sorted chunks.

We now consider the memory overhead (extra memory) of this method compared to storing bucket elements contiguously by writing to each bucket at each iteration. The chunks each have $\Theta(M)$ elements. The auxiliary array that gives bucket boundaries is of size $\Theta(M/B)$. If B (the cache line size) is 128, then the memory overhead is less than 1%, and larger cache lines reduce the relative overhead. Figure 3(d) shows the state of DRAM at the end of Phase 1.

After sorting all $\Theta(N/M)$ chunks, we create the final array in Phase 2. Using the array *BucketTot*, we determine how many buckets can fit into the scratchpad. Specifically, we find the largest k such that $\sum_{i=0}^k \text{BucketTot}[i] \leq M$ (Figure 4(a)). We then read in all elements from buckets 0 through k from each of the chunks in DRAM (Figure 4(b)). If i is the index of the first element of bucket $k+1$, then we read in all elements with index $0 \dots i$. We then use multi-way merge to combine the $\Theta(N/M)$ sorted strings into a single sorted string. This is the start of the final list, which we write to DRAM, as shown in Figure 4(c). We continue with the next set of buckets that (almost) fills the scratchpad until we are done. For the examples we used in our experiments, we batched thousands of buckets into one transfer. This consolidation was a prerequisite for the simulated time advantage we show in Section 5.

Figure 5: Simulation Architectural Setup.

5. Experiments

We implemented the NMsort algorithm described in Section 4.4. As no hardware systems with multi-level memory subsystems of the type targeted by NMsort are currently available, we used the Structural Simulation Toolkit (SST) [8] as a basis for predicting performance.

The implementation of NMsort bucketizes once and then sorts the buckets. Because inputs for sorting are random 64-bit integers, the buckets will generally be about the same size, and we do not expect any buckets to be large enough to require recursion. The nonrecursive algorithm is sufficient for our experiments. The bucketization is calculated using a multithreaded algorithm that determines bucket boundaries in a sorted list.

For a comparison to existing contemporary single-level memory sorting algorithms we use the GNU parallel C++ library’s multi-way merge sort (originally from the MCSTL [9]). The latter is the fastest CPU-based multithreaded sort from our benchmarking exercises. We use the GNU parallel sort as a subroutine call in NMsort for sorting integers once they are located in the scratchpad.

On current hardware, sorting may not be a memory-bandwidth-bound operation. However, we claim that it will become memory-bandwidth-bound in future high-performance-computing architectures as memory becomes more distant and the number of cores per NUMA region increases.

5.1. Experimental Design

Our analysis of memory-boundedness on the simulated system described in Section 3, indicates that sorting should be memory bound if the number of cores is 256. The transition to a compute-bound regime should occur when that number is reduced below 128 (reducing x in the calculations there). We give results for the plausible cases of 256 and 128 cores on-chip, as in Figure 2. In the future, it is expected that the number of cores per chip will grow well past 256.

To simulate this system, we use the Structural Simulation Toolkit (SST)[8], a parallel discrete event simulator that couples a number of architectural simulation components. The full processor is simulated by the Ariel model which models an arbitrary (in this case 256) number of cores subject to a single executing application thread per core. The Ariel processor connects to an application running on real hardware through a shared-memory region for high performance.

Each application instruction is trapped using the Pintool dynamic instrumentation framework [31]. These instructions, including executing thread ID, and special operation codes including any specialized memory allocation/free functions, are routed through the shared memory queues. The virtual Ariel processor cores then extract instructions from their executing thread queue and issue memory operations into the memory subsystem as required.

Figure 5 shows the simulation configuration including the local, private L1 data caches, shared L2 caches and the on-chip network (named “Merlin”) which routes memory requests at the flit-level to the various memory regions. Near and far memory timings use the DRAMSim2 simulation component for increased accuracy [32].

We consider sorting instances of 100 million random 64-bit integers. Each subproblem in Phase 1 of NMsort has size 10 million. Larger sizes would provide more parallelism for GNU parallel sort, but we were limited by SST simulation times. Each series of simulation runs took several days of computation time.

Figure 6: Simulated sorting times with varying scratchpad memory bandwidth.

5.2. Results

128-core runs	GNU Sort	NMsort (2X)	NMsort (4X)	NMsort (8X)	NMsort (16x)
Scratchpad Accesses	0	321933189	321916523	321705058	322558041
DRAM Accesses	286882568	79916549	82085507	81460951	82073575
256-core runs	GNU Sort	NMsort (2X)	NMsort (4X)	NMsort (8X)	NMsort (16x)
Scratchpad Accesses	0	295971731	295545258	295945545	296543427
DRAM Accesses	259867387	106895364	105754690	105864463	102741607

Table 1: SST Simulation Results for Various Scratchpad Near Memory Bandwidths. 128-core NMsort runs reduce far DRAM access counts by 70% compared to 128-core GNU parallel sort runs. The reduction is roughly 60% for 256-core runs. These reductions exceed the roughly 25% reduction predicted by our theory, as NMsort is a simple, non-recursive version of the algorithm analyzed in Section 3. The fairly uniform scratchpad and DRAM accesses for NMsort as ρ varies are expected.

Figure 6 contains simulation performance results for GNU parallel sort and NMsort on our instances of 100 million elements. Note that our theory does not directly predict wall clock performance. We do, however, see simulated performance gains of roughly 30% when NMsort is granted a scratchpad with 4-8X the bandwidth of far DRAM. Most of the NMsort computation time is devoted to sorting the scratchpad-sized chunks of data in Phase 1 of the algorithm. These chunks are quite small, at 10 million elements, and we note that GNU parallel sort does not exhibit strong scaling from 128 to 256 cores on instances of this size. We believe that is why the simulated wall clock time of 128 core runs is actually faster than that of 256 core runs. Furthermore, recall that 128 core runs are at the transition point between memory-bound and compute-bound regimes for this problem. Therefore, it is expected that the scaling line for these runs should flatten.

However, we believe that our 256 core runs are more representative of future runs on large data and real machines. Such runs will have plenty of parallelism and should exhibit strong scaling with increasing core count well into the memory-bound regime.

Next, we consider memory access counts in Table 1. Each NMsort column shows the bandwidth expansion factor (the ratio of the scratchpad to the DRAM bandwidth). This represents ρ from Section 2. Note that NMsort makes less than half as many DRAM accesses as GNU sort. All of the sorting comparisons done by NMsort involve the scratchpad; the DRAM is used only for storing the partial results of Phase 1.

These simulations did not use double buffering or direct memory access (DMA) transfers. These techniques would clearly improve performance of NMsort.

6. k -Means Clustering

Our second major algorithmic case study focuses on k -means clustering. In this section we formally describe the problem. We then give details of Lloyd's algorithm, the most popular k -means algorithm. Then we discuss how we can adapt this algorithm to the scratchpad architecture and bound its performance guarantees.

k -means clustering problem statement

Let A denote an input set of N points in $\mathbb{R}^d$ (d -dimensional Euclidean space). Let $v[i]$ denote the i th dimension of a point v . Let $D_{v,v'}$ denote the distance between two points v and v' .

The goal is to cluster A into k clusters, for input parameter k . Intuitively, points in the same cluster are more closely related to each other than to points in other clusters. More precisely, we can compute the variance of the pairwise distances between elements in the set of points assigned to the same cluster. This is a measure of how close the points are to each other as a group. We wish to minimize the total variance of the clustering, which is the sum of the variances over all clusters.

Say that the points u in A are partitioned into k clusters $C_1, C_2, \dots, C_k$. Then, one can define the **centroid** of the clusters. Intuitively, the centroid is the center of mass of the cluster. In Euclidean space this is the point that is the average of all of the other points in the set. Formally, the centroid v_i of a cluster C_i has j th dimension equal to $\frac{1}{|C_i|} \sum_{u \in C_i} u[j]$. The goal of the k -means problem is to choose clusters such that $\sum_{i=1}^k \sum_{v \in C_i} (D_{v_i, v})^2$ is minimized.

Alternatively, one can choose a set X of k points. This naturally defines a clustering where for point $v \in X$, contains all points in $v' \in A$ such that v is the closest point in X to v' . For a set X let $D_X(v) = \min_{v' \in X} D_{v,v'}$ be the minimum distance of a point v to a point in the set X . The goal is to choose a set X of size k such that $\sum_{v \in A} (D_X(v))^2$ is minimized.

Lloyd's algorithm

We now give details of Lloyd's algorithm. We assume that $k > B$ throughout the section.

1. The algorithm is initialized with a set X of k points. This initialization can be done arbitrarily, but is usually done using a sophisticated algorithm such as k -means++ [33, 34] or by selecting the points uniformly at random [35].
2. The algorithm then assigns each point in A to the closest of the k points in X . The points in A that are assigned to v form cluster C_v .
3. The algorithm then computes the centroid of C_v and replaces v in X with the centroid of C_v . Recall that the centroid of a set of points in Euclidean space is the point that is the average of all of the other points, that is, the centroid's j th dimension is $\frac{1}{|C_v|} \sum_{u \in C_v} u[j]$.
4. The algorithm iterates until the clustering does not improve, or does not improve beyond some threshold.

Optimizing Lloyd's algorithm for cache only

We begin by analyzing Lloyd's algorithm in an architecture without a scratchpad. Later when analyzing the scratchpad-optimized algorithm, we must consider if the set X fits in the cache. Here we give a single expression of the memory accesses irrespective whether X fits in cache or not.

Theorem 11. *An iteration of Lloyd's algorithm can be implemented in $O(\lceil kd/Z \rceil (Nd/B))$ memory accesses on a cache of size Z when the cache lines have size B .*

Proof. The cost to scan through all N points in A once is $\Theta(Nd/B)$. Because the formula for determining the centroid is linear, if we bring the first Z/d points into cache, scan through all the points, bring the next Z/d points, into cache, and so forth, then we can compute the centroid using $O(\lceil kd/Z \rceil)$, scans through all the points. □

If $kd < Z$, then the entire set X fits in cache and the only cost is scanning through all of the points in A .

Corollary 12. *An iteration of Lloyd's algorithm can be implemented using $O(Nd/B)$ memory accesses, if $kd < Z$.*

Implementing Lloyd's algorithm on the scratchpad

To take advantage of the scratchpad, Lloyd's algorithm performs the following steps in each iteration:

- Recall that the algorithm's goal is to compute new centroids. For each point $v \in X$ let c_v be initialized to 0^d . This will eventually store the new centroid for the cluster assigned to v .
- The algorithm fills the scratchpad half full of points A' from A , i.e., $O(M/d)$ points.
- The algorithm fills the remaining space with a set X' of as many points of X which can fit into the other half of the scratchpad. The algorithm will iteratively bring in chunks of X until scanning through all of the points in X . For each chunk X' , the algorithm determines for each point in A' its closest point X' . After scanning through all of X , the algorithm can determine which point in X' is closest to each point A' .
- Let C_v denote the points of A' which were closest to $v \in X$. The algorithm sets $c_v = c_v + \frac{1}{|C_v|} \sum_{v' \in C_v} v'$.
- The algorithm recurses on another set A' of points in A until it iterates through all points in A .

Now we bound the performance guarantees of this scratchpad optimized algorithm.

Theorem 13. *When there is a scratchpad of size M and a cache of size Z , an iteration of Lloyd's algorithm can be implemented with $O\left(\lceil \frac{kd}{M} \rceil \lceil \frac{Nd}{M} \rceil \frac{M}{B}\right)$ DRAM accesses and $O\left(\lceil \frac{kd}{M} \rceil \lceil \frac{Nd}{M} \rceil \lceil \frac{M}{Z} \rceil \left(\frac{M}{\rho B}\right)\right)$ scratchpad accesses.*

Proof. We first analyze the DRAM accesses. The scratchpad thus has to be loaded $O\left(\lceil \frac{kd}{M} \rceil \lceil \frac{Nd}{M} \rceil\right)$ times from DRAM and the cost for each scratchpad load is $O(M/B)$.

We next analyze the scratchpad accesses. For each of the scratchpad loads, the scratchpad runs what is effectively an iteration of Lloyd's algorithm. Using the same proof as in Theorem 11, since each iteration has $O(M/d)$ points and the block transfer size is ρB , the cost per iteration is $O\left(\lceil \frac{M}{Z} \rceil \left(\frac{M}{\rho B}\right)\right)$. $\square$

To understand the takeaway or "moral" of Theorem 13, we separate into three cases: (1) the k points in X fit in cache ($Z \geq kd$), (2) the k points in X fit in the scratchpad but not in cache ($Z < kd \leq M$), and (3) the k points in X do not fit in the scratchpad ($M < kd$).

The following three corollaries explain the benefit of the scratchpad for each of these three cases.

The first states that the scratchpad does not give any additional speed-up when all points in X fit into the cache.

Corollary 14. *If the k points in X fit in cache ($Z \geq kd$), then the scratchpad delivers no asymptotic benefit. With or without scratchpad, Lloyd's algorithm can be implemented so that the dominant memory-access cost comes from the DRAM accesses; it is $O(Nd/B)$, the cost of a linear scan.*

Now we consider the case where X is too large to fit into the cache, but fits into the scratchpad. In this case, we expect speed-up using the increased bandwidth of the scratchpad.

Corollary 15. *If the k points in X fit in scratchpad but not cache ($Z < kd \leq M$), then the scratchpad can accelerate Lloyd's algorithm by a factor of up to $\min\{\rho, kd/Z\}$. There is no more asymptotic benefit once $\rho \geq kd/Z$.*

Proof. By Theorem 13, with a scratchpad there are $O(Nd/B)$ DRAM accesses and by Theorem 11, without a scratchpad, there are $O((kd/Z)(Nd/B))$ DRAM accesses.

There are $O((kd/Z)(Nd/(\rho B)))$ scratchpad accesses, which means that the scratchpad accesses are not dominant if $\rho > kd/Z$, so increasing ρ does not help asymptotically. $\square$

Finally, we consider the case where X is too large to fit into the cache and also the scratchpad. In this case, we expect speed-up using the increased bandwidth of the scratchpad.

Corollary 16. *If the k points in X do not fit in the scratchpad ($kd > M$), then the scratchpad can accelerate Lloyd's algorithm by a factor of up to $\min\{\rho, M/Z\}$. There is no more asymptotic benefit once $\rho \geq M/Z$.*

Proof. Reason: by Theorem 13, with a scratchpad there are $O(kNd^2/(MB))$ DRAM accesses and by Theorem 11, without a scratchpad, there are $O(kNd^2/(ZB))$ DRAM accesses.

There are $O(kNd^2/(\rho ZB))$ scratchpad accesses, which is not the dominant term as long as $\rho > M/Z$, so increasing ρ beyond this value does not help. $\square$

Analyzing the algorithms with parallel processors

Now we generalize these theorems to multiple processor. Recall, that in the model considered there are p processors and up to $p' \leq p$ processors can simultaneously access the memory. Since Lloyd's algorithm linearly scans through the data, the process is fully parallelizable. In this case, the following theorem immediately follows from Theorem 11 for an architecture without a scratchpad.

Theorem 17. *An iteration of Lloyd's algorithm can be implemented in $O(\lceil kd/Z \rceil (Nd/(Bp')))$ memory accesses on a cache of size Z when the cache lines have size B and p processors can access p' memory locations in parallel.*

From Theorem 13, we have the following.

Theorem 18. *When there is a scratchpad of size M and a cache of size Z , and p processors can access p' memory locations in parallel, an iteration of Lloyd's algorithm can be implemented with $O\left(\lceil \frac{kd}{M} \rceil \lceil \frac{Nd}{M} \rceil \frac{M}{(p'B)}\right)$ DRAM accesses and $O\left(\lceil \frac{kd}{M} \rceil \lceil \frac{Nd}{M} \rceil \lceil \frac{M}{Z} \rceil \left(\frac{M}{p'\rho B}\right)\right)$ scratchpad accesses.*

These theorems give rise to the following corollary.

Corollary 19. *On a parallel process the speed-up produced by the scratchpad is the following.*

- If $kd \leq Z$ then the scratchpad delivers no asymptotic benefit.
- If $Z < kd \leq M$ the scratchpad can accelerate Lloyd's algorithm by a factor of $\min\{\rho, \frac{kd}{Z}\}$.
- If $M \leq kd$ the scratchpad can accelerate Lloyd's algorithm by a factor of $\min\{\rho, M/Z\}$.

7. Partitioning for k -Means Clustering

In this section we give the asymptotic analysis for a version of the partitioning algorithm [36, 37, 38] for solving the k -means clustering problem.

Partitioning algorithm

The partitioning algorithm works as long as $kd < M$. The high level idea behind the partitioning algorithm is to reduce the size of the input by clustering portions of it and then reducing each cluster into a single representative point. Then the algorithm clusters these representative points to get a clustering of the original input. This algorithm gives an additional benefit over the previous algorithm in memory accesses when the number of iterations of Lloyd's algorithm is large.

The partitioning algorithm divides the input A into $|A|d/M$ partitions containing M/d points of size M . For each partition, the algorithm clusters them into k clusters. Each of these clusters will be reduced to a single weighted point. For each of the constructed clusters, it creates a new point located at the centroid of the cluster and weights this points with a weight equal to the number of points in the cluster. Afterwards, the algorithm then takes each of these weighted points and clusters them.

We now describe the algorithm which is adapted from [36]. When we describe the algorithm, we note that any arbitrary clustering algorithm can be used when clustering the partitions and we will denote this algorithm by $\mathcal{A}$. We will later discuss the performance guarantees when $\mathcal{A}$ is Lloyd's.

1. The partitioning algorithm arbitrarily partitions the input set of points A into Nd/M sets, $S_1, S_2, \dots, S_{Nd/M}$ sets, each containing M/d points.
2. Let $\mathcal{A}$ denote any algorithm that clusters M/d data points inside the scratchpad. The algorithm sequentially brings each set S_i and clusters them separately using $\mathcal{A}$ to get k clusters $C_{i,1}, C_{i,2}, \dots, C_{i,k}$. For each cluster $C_{i,j}$, the algorithm computes the centroid $c_{i,j}$ and gives this centroid weight $w_{i,j} = |C_{i,j}|$.
3. At the end of the algorithm, there are $k \cdot Nd/M$ different centroids.
4. Finally the algorithm clusters the $k \cdot Nd/M$ centroids using the algorithms from Section 6 where the algorithm treats the weights as if there are $w_{i,j}$ points located at $c_{i,j}$. If there are too many centroids to fit into the scratchpad, the algorithm can recurse.

Using the analysis of [36], this algorithm has the following theoretical guarantee. We note that Lloyd’s has no approximation guarantee, but, intuitively, the following theorem states that the quality of the solution should not be too different between using Lloyd’s or the partitioning based algorithm.

Theorem 20 ([36]). *This algorithm is a $O(\alpha^2)$ -approximation algorithm where α is the approximation of the procedure $\mathcal{A}$.*

Let T denote the number of iterations of Lloyd’s algorithm if it were run on the entire dataset. Let T' be the maximum number of iterations Lloyd’s algorithm takes if it were used as the algorithm $\mathcal{A}$ in the partition based algorithm anytime it is invoked. If $\mathcal{A}$ is set to be Lloyd’s algorithm, the partitioning algorithm gives rise to the following performance guarantees when run on a single processor machine.

Theorem 21. *When $kd < M$, the total number of DRAM accesses by the partitioning algorithm where $\mathcal{A}$ is Lloyd’s algorithm (including all iterations of Lloyd’s) is $O\left(\lceil \frac{Nd}{M} \rceil \frac{M}{B}\right)$ and the total number of scratchpad accesses is*

$$O\left(T' \lceil \frac{Nd}{M} \rceil \lceil \frac{kd}{Z} \rceil \left(\frac{M}{\rho B}\right)\right).$$

Proof. First we consider the DRAM accesses. The dominate factor is clustering the partitions in the first step. Whenever we use Lloyd’s algorithm, the set of k centers always fits in the scratchpad since we have assumed $kd < M$ and therefore they do not need to be placed in the scratchpad as in the previous section. There are $\frac{Nd}{M}$ partitions to be clustered. We only load each partition once into the scratchpad and loading each partition requires M/B accesses from DRAM. Thus, the total number of accesses is $(Nd/M)(M/B)$.

Now we consider the scratchpad memory accesses. For any partition, Lloyd’s algorithm runs for at most T' iterations. Further, a single iteration of Lloyd’s algorithm scans through all the points in M for each set of $\lceil kd/Z \rceil$ centroids. There are $\lceil Nd/M \rceil$ partitions that are considered, so the total number of accesses is $O\left(T' \lceil \frac{Nd}{M} \rceil \lceil \frac{kd}{Z} \rceil \left(\frac{M}{\rho B}\right)\right)$. $\square$

We note that the previous two bounds can be scaled by p' if there are p processors that can access p' memory locations in parallel, since the algorithms can be fully parallelized as before.

The number of iterations of Lloyd’s algorithm cannot be bounded. However, when the partitioning algorithm is used, we have observed that usually $T' \leq T$. Assuming this, we have the following corollary when comparing against the standard Lloyd’s algorithm. This corollary shows that the speed-up scales with ρ up to $T'kd/Z$, allowing for a larger increase in speed-up by a factor of T' over just simulating Lloyd’s using the scratchpad when $kd < M$. Further, in this case we achieve speed-up even in the case that $kd < Z$, which was not possible for the scratchpad implementation of Lloyd’s algorithm.

Corollary 22. *The partitioning algorithm when $\mathcal{A}$ is set to be Lloyd’s algorithm, $Z < kd < M$ and $T' \leq T$ achieves a speed up of $\min\{\rho, T'kd/Z\}$. This algorithm achieves speed up up to a factor of $\min\{\rho, T'\}$ when $T' \leq T$ and $kd < Z$.*

8. Experiments to corroborate the theory

We now describe our experiments running the partitioned version of Lloyd’s algorithm described in Section 7. Recall that a major step in this algorithm is to run classical Lloyd’s algorithm from the scratchpad using a scratchpad-sized chunk of the larger input data. We discuss the conditions under which Corollary 15 of Theorem 13 applies. We show that the partitioned k -means algorithm for our data sets run on our specific machine satisfies each of the design properties introduced in Section 1.1 along with a k -means-specific property. We describe our experimental design and results.

Our experimental platform is an 8-core NUMA node with approximately 21MB of cache, 32GB of DRAM, a standard memory bandwidth of approximately 25GB/s, and a processing capability of roughly 166 GFLOPS. This is roughly representative of the Trinity nodes that motivate this work. The system has been modified to mimic two-level memory via a different socket at 12GB/s to mimic the far memory. The programmer controls which data is stored in near memory and which is stored in far memory.

We ran experiments on the “ElectricityLoadDiagrams20112014” dataset from the Machine Learning Repository at UC Irvine [39]. This dataset has 140,256 dimensions and 370 points. Our code was based on an open-source OpenMP code from Northwestern University [40]. We implemented the scratchpad-aware partition algorithm for k -means and used the code from Northwestern to implement the k -means algorithm for a subproblem in scratchpad. That is, the Northwestern code is Algorithm $\mathcal{A}$ in step 2 of the partition algorithm in Section 7.

8.1. Required Properties

1. (Memory boundedness)

Before implementing a scratchpad-aware algorithm we must ensure that the algorithm is memory bound if run with no scratchpad available.

Once more, let N be the number of points, each with d dimensions. Let k be the number of centers. Each iteration of Lloyd’s algorithm for k -means requires $O(Nkd)$ operations. In the code used, there are $3Nkd$ operations. The amount of memory transfer during a Lloyd’s iteration (not counting reuse) is $(N + k)d$. A simple calculation shows a regime where the computation will be memory bandwidth-bound.

$$\frac{3Nkd}{x} < \frac{(N + k)d}{y}$$

where, x is the processing rate in GFLOPS and y is the memory bandwidth in double precision values per second. Given our system parameters, we have:

$$\frac{3Nkd}{166} < \frac{(N + k)d}{1.5}.$$

We drop the $kd/1.5$ term on the right side since satisfying this inequality with a smaller righthand side makes memory-boundedness even more likely. Rearranging, we find that the N ’s and d ’s cancel, leaving us with an approximate upper bound on k , the number of centers: $k < 36$. Intuitively, we have just formalized the notion that k -means with large numbers of centers provides a large amount of computation per memory transfer, making it compute-bound.

Computing on scratchpad-resident data imposes yet a tighter constraint. Recall that ρ is the scratchpad bandwidth expansion parameter. In our case, $\rho = 2$. To remain memory-bandwidth bound during scratchpad computation (and hence to benefit from increases in ρ), we must have:

$$\frac{3Nkd}{x} < \frac{(N + k)d}{\rho y}$$

and hence $k < 18$. We note that this is a narrow window for performance gain.

2. (Scratchpad chunking is appropriate)

Our scratchpad-aware Lloyd’s iteration is an implementation of the partitioning algorithm of [36]. This breaks the problem into scratchpad-appropriate chunks that can be processed independently. Running Lloyd’s algorithm on the scratchpad-resident subproblem reduces a (large) scratchpad-sized chunk of data into a data set of size $O(kd)$. This is the size of the centers for the subproblem. These centers of the subproblem, taken over all subproblems, then form a final k -means problem that determines the solution. Thus the “reassembly” of the subproblems into a final result is efficient compared to the overall computation.

3. (Scratchpad chunk reuse)

Large k -means instances tend to require at least several passes before convergence. Thus, we expect to amortize the cost of copying a partition to scratchpad via sufficient numbers of passes through the data.

4. (Final check)

Chunking into cache-sized subproblems would mean that $kd < Z$. In that case, not only does Corollary 14 predict no advantage from scratchpad, but the notion of using partitioned Lloyd’s algorithm stops making sense. Recall that the fundamental computation of that heuristic reduces a subproblem down to size $O(kd)$. If the chunks fit in cache, that reduction is unnecessary and one would use unpartitioned Lloyd’s. That means enforcing the inequality $kd > 21\text{MB}$. Given our existing upper bound $k < 18$, and the fact that our instance uses double precision values, we must have $d > 152,917$. When we combine this requirement with the requirement that $k < 18$ for our simulated system, we see that algorithm engineering time spent on optimizing partitioned Lloyds for scratchpad may not be warranted.

8.2. Experimental Design

There are two fundamental problems with the constraints on k and d . The first is that our regime of expected performance boost invokes the “curse of dimensionality”[41], which consists of several properties that characterize high-dimensional Euclidean spaces and question the usability of distances in that context. However, recent work [42] suggests that this might not be a conclusive statement, and is at least an open research area. Variants of clustering the

Table 2: Predicted reductions in far DRAM access and observed runtime improvements

k	Speedup	Predicted Drop in Far Memory Accesses
10	1.33	1.02
14	1.34	1.42
18	1.36	2.0

full space may hold promise, and we predict that scratchpad-aware versions of them would experience performance boosts. For this reason, and to see whether our predictions are accurate, we proceed with an experiment.

The second and more immediate problem is that our lower bound on d exceeds the number of dimensions in our test problem, so we do not expect a benefit from scratchpad. Running the instance, we actually do see a small performance boost. However, in order to evaluate our theory, we have modified the problem by duplicating each dimension. The resulting 280, 512-dimensional problem allows the following experiments.

With our synthetically-doubled dataset, we find that cache will be flooded by roughly nine centers. Combining this with our bound $k < 18$, we can test an experimental range of $9 \leq k \leq 18$. Selecting $k = 10, 14, 18$, Corollary 15 leads us to predict relative speedups of 1.02, 1.42, and 2.0 for these experiments, assuming a direct relationship between reduced far-DRAM block transfers and runtime.

8.3. Experimental results

In our experiments, we find that the penalty in objective value is far less severe than that of the theoretical approximation bound.

Table 2 shows the results of our sweep through the center counts that both flood cache and keep the problem memory bound. We do observe increasing runtime speedups with increasing numbers of centers. However, we did not collect true DRAM access counts. Although the 30% runtime speedups do not match the predicted DRAM access reductions, we note that many runs with parameter settings outside of this regime of predicted benefit showed no runtime performance gain at all.

9. Architecture of the Scratch Pad

In this section we give a brief overview of the memory architecture that underlies our theoretical model. We start with motivation: why have architects introduced this new architecture? Then we discuss specifics of the design of the architecture.

9.1. Motivation for the Scratchpad

There are multiple technologies that are in development that may be a basis for a level of memory between a processor’s caches and conventional capacity memory. The WideI/O-2 standard, which is developed by JEDEC,² has memory parts with four ports that are 128-bits wide each. Such a memory thus has a bandwidth that is between two and four times higher than standard memory DIMMs. There are also several efforts underway that enable multiple memory chips to be “stacked” or vertically packaged on top of each other to reduce wire distances between components and improve performance. When such a part comes to market, it is expected that stacks of between four or eight dies will be economical. The result will be considerably higher bandwidth rates to these memory parts in relatively smaller spaces and lower power consumption than existing memory technologies.

However, the high pin density and electrical issues are likely to prevent stacked memory from becoming a complete replacement for existing memory DIMMs. Due to this limitation, the semiconductor industry is looking at mounting multiple die components either directly on top of CPU chips, or, more likely, as multiple dies that are mounted on the same substrate as the CPU. The result will be memory capacities of several gigabytes that can be implemented with higher bandwidths (“near memory”) but, in the short term at least, the greatest portion of memory capacity will continue to be built using lower-bandwidth “far” memories such as existing DDR DIMMS (Figure 7).

²JEDEC (jedec.org) is a consortium that develops standards for the microelectronics industry. DDR3 and DDR4 memory result from standards developed by JEDEC, which also does many other standardization efforts.

Figure 7: Two Level Memory Architecture.

Memory in the stacks described is likely to be organized differently than existing technologies where a single reference activates all dies on the referenced DIMM. In order to save power and provide greater parallelism, a reference will likely move data to/from a single memory die. With a sufficiently wide interface the memory will be able to provide increased bandwidth performance and somewhat reduced latency.

Our proposed scratchpad architecture bears some similarities to contemporary GPU memory architectures. The memory architecture of the GPU—like the scratchpad—has similar access times to main memory, but provides a higher bandwidth. Since the release of NVIDIA’s CUDA-6 SDK, a feature called Unified Virtual Memory (UVM) has allowed for automatic migration of memory between the host DDR and the local GPU-based GDDR memories. However, it is generally accepted in the community that the current UVM implementation provided by NVIDIA can be outperformed by handwritten data movement operations in the general case [43]. As we move into the future where more complex and elaborate memory hierarchies are likely, our model will become increasingly relevant, particularly in multi-level memory scenarios where automatic migration of pages is unlikely to be available for all memory levels.

9.2. The DRAM Scratchpad Design

Architecturally, the scratchpad memory is attached behind a directory controller after the last level cache, as a “peer” to the conventional main memory. Our experiments assume a level-2 cache with separate L1 caches for each core in the CPU (See Figure 8).

The scratchpad memory appears as a separate portion of the physical address space. L2 cache can contain both conventional memory and scratchpad references. References to scratchpad data from the cache are functionally the same as references to main memory from the cache on the basis of a fixed address range.

In this work, the conventional and scratchpad memories are accessed through standard load/store instructions. In future work, we will examine the use of DMA engines which can transfer data between the near and far memory in the background, allowing overlap of computation and communication.

9.2.1. Why Not Simply a Large Cache

A multi-gigabyte cache would have to devote a substantial area to memory tags. This brings size and performance issues that would raise cost and complexity. Additionally, DRAM has fairly high access latencies which could make tag comparisons costly. SRAM memories with small latencies could be added to hold the tags, but this would have a significant impact on area and power.

For example, a 4GB DRAM cache, consisting of 2^{23} 64-byte cache lines might require 17 bits of SRAM tag and flag information per data block, or about 272 MB. If SRAM is about 1/24th the density of DRAM (an updated estimate from what is given in [44]), this would be almost 80% of the area of the DRAM data arrays and would consume much more power. Of course, larger page sizes could reduce this overhead, but that invites its own complexities.

Figure 8: Architectural Setup.

By enabling an application to use an intermediate level of memory directly, the application can have a level of control such as prefetching data that is known to be needed and saving data that does not need further references. Another benefit is that the intermediate memory can be used to hold temporary values in calculations. This directly reduces the bandwidth needed to main memory, which will remain bandwidth limited.

9.2.2. Programmatic Interface

For this set of experiments it is assumed that the scratchpad is given a portion of memory address space, and that memory accesses (i.e., load/stores) treat each space identically.

We assume a modified `malloc()` call to allocate a portion of the scratchpad space and that the OS or runtime handles the virtual-to-physical mapping and contention between applications.

10. Conclusions and future work

This paper provides guidance for applications teams considering rewriting computational kernels of certain applications for new two-level memory architectures. Our approach requires programmer intervention rather than relying on vendor caching modes for scratchpad. To retain productivity, we believe that scratchpad-aware algorithms will have to be embedded in libraries that expose reasonable application programmer interfaces. We provide basic design properties and demonstrate them in two algorithmic case studies. Vendors have already committed to near-memory scratchpad memory stacks on-package. These are likely to be a feature of extreme-scale supercomputers.

Our sorting algorithms take advantage of scratchpad effectively. We show via simulation that sorting-based kernels can expect significant speedups with near-memory bandwidth expansion ratios up to 4X. Beyond that, there are diminishing returns, though we still observe simulated speedups through 16X.

Our k -means algorithms theoretically exploit scratchpad in certain regimes, but we find that these regimes are not practical. Clustering with k -means is a naturally compute-bound process, and we require very large dimensional problems to expect any advantage from scratchpad.

Our co-design efforts have been informed by theory, algorithm design, algorithm engineering, simulation, and computer architecture. We believe that our process should be applicable to a variety of computational kernels, though our experience suggests caution before committing to a redesign of any kernel. With real hardware scheduled to be available soon, we plan to validate our sorting results. We also plan to test more computational kernels for applicability to scratchpad architectures.

11. Acknowledgments

This work was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. Michael Bender was also supported by the following NSF grants: CNS 1408695, CCF 1439084, IIS 1247726, IIS 1251137, CCF 1217708, and CCF 1114809.

- [1] <http://nnsa.energy.gov/mediaroom/pressreleases/trinity>.
- [2] <http://www.hpcwire.com/2014/06/24/micron-intel-reveal-memory-slice-knights-landing/>.
- [3] S. Steinke, L. Wehmeyer, B. Lee, P. Marwedel, Assigning program and data objects to scratchpad for energy reduction, in: Proceedings of the Conference on Design, Automation and Test in Europe (DATE), 2002, pp. 409–415.
- [4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, P. Marwedel, Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems, in: Proceedings of the Tenth International Symposium on Hardware/Software Codesign (CODES), 2002, pp. 73–78. doi:10.1109/CODES.2002.1003604.
- [5] <http://insidehpc.com/2014/07/cray-wins-174-million-contract-trinity-supercomputer-based-knights-landing>.

- [6] A. Aggarwal, J. S. Vitter, The input/output complexity of sorting and related problems, *Communications of the ACM* 31 (9) (1988) 1116–1127.
- [7] L. Arge, M. T. Goodrich, M. Nelson, N. Sitchinava, Fundamental parallel algorithms for private-cache chip multiprocessors, in: *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2008, pp. 197–206.
- [8] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, B. Jacob, The Structural Simulation Toolkit, *SIGMETRICS Perform. Eval. Rev.* 38 (4) (2011) 37–42. doi:10.1145/1964218.1964225.
URL <http://doi.acm.org/10.1145/1964218.1964225>
- [9] J. Singler, P. Sanders, F. Putze, MCSTL: The multi-core standard template library, in: *Proceedings of the 13th International Euro-Par Conference on Parallel Processing (Euro-Par)*, 2007, pp. 682–694.
- [10] A. Rodrigues, R. Murphy, P. Kogge, K. Underwood, The Structural Simulation Toolkit: A Tool for Exploring Parallel Architectures and Applications, Tech. Rep. SAND2007-0044C, Sandia National Laboratories (2007).
- [11] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, S. McCauley, Cache-adaptive algorithms, in: *Proceedings of the Twenty-Fifth Symposium on Discrete Algorithms (SODA)*, 2014, pp. 116–128.
- [12] G. S. Brodal, E. D. Demaine, J. T. Fineman, J. Iacono, S. Langerman, J. I. Munro, Cache-oblivious dynamic dictionaries with update/query tradeoffs, in: *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010, pp. 1448–1456.
- [13] G. S. Brodal, R. Fagerberg, On the limits of cache-obliviousness, in: *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing (STOC)*, 2003, pp. 307–315.
- [14] M. Frigo, C. E. Leiserson, H. Prokop, S. Ramachandran, Cache-oblivious algorithms, *ACM Transactions on Algorithms* 8 (1) (2012) 4.
- [15] H. Prokop, Cache-oblivious algorithms, Master’s thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (Jun. 1999).
- [16] W. D. Frazer, A. C. McKellar, Samplesort: A sampling approach to minimal storage tree sorting, *Journal of the ACM* 17 (3) (1970) 496–507.
- [17] P. Carlini, P. Edwards, D. Gregor, B. Kosnik, D. Matani, J. Merrill, M. Mitchell, N. Myers, F. Natter, S. Olsson, S. Rus, J. Singler, A. Tavory, J. Wakely, *The GNU C++ Library Manual*, 2012.
- [18] R. Motwani, P. Raghavan, *Randomized algorithms*, Chapman & Hall/CRC, 2010.
- [19] L. Arge, M. T. Goodrich, N. Sitchinava, Parallel external memory graph algorithms, in: *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1–11.
- [20] N. Sitchinava, N. Zeh, A parallel buffer tree, in: *Proceedings of the Twenty-Fourth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012, pp. 214–223.
- [21] D. Ajwani, N. Sitchinava, N. Zeh, Geometric algorithms for private-cache chip multiprocessors, in: *Proceedings of the Eighteenth Annual European Symposium on Algorithms (ESA)*, 2010, pp. 75–86.
- [22] R. Cole, V. Ramachandran, Resource oblivious sorting on multicores, in: *Proceedings of the 37th International Colloquium Conference on Automata, Languages and Programming (ICALP)*, 2010, pp. 226–237.
- [23] R. Cole, V. Ramachandran, Efficient resource oblivious algorithms for multicores with false sharing, in: *Proceedings of the Twenty-Sixth International Parallel & Distributed Processing Symposium (IPDPS)*, 2012, pp. 201–214.
- [24] G. Bilardi, A. Pietracaprina, G. Pucci, M. Squizzato, F. Silvestri, Network-oblivious algorithms, *Journal of the ACM (JACM)* 63 (1) (2016) 3.

- [25] R. A. Chowdhury, V. Ramachandran, F. Silvestri, B. Blakeley, Oblivious algorithms for multicores and networks of processors, *Journal of Parallel and Distributed Computing* 73 (7) (2013) 911–925.
- [26] J. S. Vitter, E. A. Shriver, Algorithms for parallel memory, I: Two-level memories, *Algorithmica* 12 (2-3) (1994) 110–147.
- [27] L. G. Valiant, A bridging model for multi-core computing, *Journal of Computer and System Sciences* 77 (1) (2011) 154–166.
- [28] B. Alpern, L. Carter, E. Feig, Uniform memory hierarchies, in: *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*, IEEE, 1990, pp. 600–608.
- [29] M. H. Nodine, J. S. Vitter, Large-scale sorting in parallel memories, in: *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, ACM, 1991, pp. 29–39.
- [30] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, H. V. Simhadri, Scheduling irregular parallel computations on hierarchical caches, in: *Proceedings of the 23rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2011, pp. 355–366.
- [31] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, A. Karunanidhi, Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation, in: *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2004, pp. 81–92.
- [32] P. Rosenfeld, E. Cooper-Balis, B. Jacob, DRAMSim2: A cycle accurate memory system simulator, *IEEE Comput. Archit. Lett.* 10 (1) (2011) 16–19.
- [33] D. Arthur, S. Vassilvitskii, k-means++: the advantages of careful seeding, in: *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2007, pp. 1027–1035.
- [34] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, S. Vassilvitskii, Scalable k-means++, *PVLDB* 5 (7) (2012) 622–633.
- [35] S. Lloyd, Least squares quantization in PCM, *IEEE Trans. Inf. Theor.* 28 (2) (2006) 129–137. doi:10.1109/TIT.1982.1056489.
URL <http://dx.doi.org/10.1109/TIT.1982.1056489>
- [36] S. Guha, A. Meyerson, N. Mishra, R. Motwani, L. O’Callaghan, Clustering data streams: Theory and practice, *IEEE Trans. Knowl. Data Eng.* 15 (3) (2003) 515–528.
- [37] S. Guha, N. Mishra, R. Motwani, L. O’Callaghan, Clustering data streams, in: *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, 2000, pp. 359–366.
- [38] A. Ene, S. Im, B. Moseley, Fast clustering using mapreduce, in: *Proc. 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2011, pp. 681–689.
- [39] M. Lichman, UCI machine learning repository (2013).
URL <http://archive.ics.uci.edu/ml>
- [40] W. Liao, Parallel k-means data clustering, code (2011).
URL <http://users.eecs.northwestern.edu/~wkliao/Kmeans>
- [41] R. E. Bellman, *Dynamic Programming*, Princeton University Press, 1957.
- [42] A. Zimek, E. Schubert, H.-P. Kriegel, A survey on unsupervised outlier detection in high-dimensional numerical data, *Statistical Analysis and Data Mining: The ASA Data Science Journal* 5 (5) (2012) 363–387.
- [43] H. C. Edwards, C. Trot, D. Sunderland, Kokkos, a manycore device performance portability library for C++ HPC applications, presented at GPU Technology Conference (2014).

- [44] P. Kogge, Exascale computing: embedded style, slides from a talk given at the Fault-Tolerant Spaceborne Computing Employing New Technologies Workshop, Sandia National Laboratories (2009).
URL <http://www.cs.sandia.gov/CSRI/Workshops/2009/FaultTolerantSpaceborne/presentations/W-1900-Kogge-Notre-Dame-dinner-talk-exa-embedded-space-5-26-09.pdf>