An Energy Complexity Model for Algorithms

Swapnoneel Roy[*]
Department of CSE
University at Buffalo, SUNY
Buffalo, NY, USA
sroy7@buffalo.edu

Atri Rudra[†]
Department of CSE
University at Buffalo, SUNY
Buffalo, NY, USA
atri@buffalo.edu

Akshat Verma
IBM Research - India
New Delhi, India
akshatverma@in.ibm.com

ABSTRACT

Energy consumption has emerged as first class computing resource for both server systems and personal computing devices. The growing importance of energy has led to rethink in hardware design, hypervisors, operating systems and compilers. Algorithm design is still relatively untouched by the importance of energy and algorithmic complexity models do not capture the energy consumed by an algorithm.

In this paper, we propose a new complexity model to account for the energy used by an algorithm. Based on an abstract memory model (which was inspired by the popular DDR3 memory model and is similar to the parallel disk I/O model of Vitter and Shriver), we present a simple energy model that is a (weighted) sum of the time complexity of the algorithm and the number of "parallel" I/O accesses made by the algorithm. We derive this simple model from a more complicated model that better models the ground truth and present some experimental justification for our model. We believe that the simplicity (and applicability) of this energy model is the main contribution of the paper.

We present some sufficient conditions on algorithm behavior that allows us to bound the energy complexity of the algorithm in terms of its time complexity (in the RAM model) and its I/O complexity (in the I/O model). As corollaries, we obtain energy optimal algorithms for sorting (and its special cases like permutation), matrix transpose and (sparse) matrix vector multiplication.

[*]Supported in part by NSF grant CCF-0844796. Part of this work was done when the author was an intern at IBM Research - India.
[†]Supported in part by NSF grant CCF-0844796.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory, Algorithms

Keywords

Energy Efficient Algorithms, Parallel Disk I/O model

1. INTRODUCTION

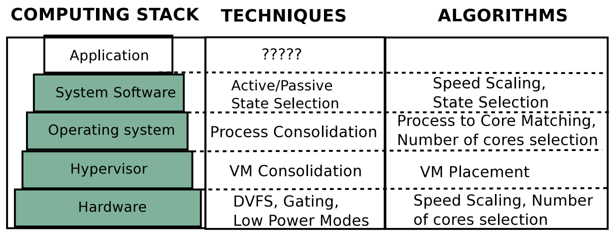
COMPUTING STACK
TECHNIQUES
ALGORITHMS
Application
?????
System Software
Active/Passive State Selection
Speed Scaling, State Selection
Operating system
Process Consolidation
Process to Core Matching, Number of cores selection
Hypervisor
VM Consolidation
VM Placement
Hardware
DVFS, Gating, Low Power Modes
Speed Scaling, Number of cores selection

Figure 1: Overview of energy management across the computing stack. Layers in green incorporate energy-aware design.

Energy has emerged as a first class computing resource in modern systems [37, 30]. Two trends have primarily led to the strong focus on reducing energy consumption. The first trend has been the ever-increasing energy consumption of data centers, which doubled between the years 2000 and 2006 [2]. Coupled with the growing awareness of the adverse impact on the environment due to data centers has led to a strong focus on energy management for server class systems. The second trend has been the explosive growth of personal computing devices like smartphones, handhelds, and notebooks, which run on batteries. Personal computing devices today perform a lot of computations and data transfer and energy conservation is a driving factor in the design of personal computing devices.

The focus on energy management has been cross-cutting across various computing disciplines including computer architecture (hardware design) [28], hypervisors [31], operating systems [27] and system software [11, 15]. Fig. 1 captures the various techniques developed to reduce energy consumption across the computing stack. One may note that energy management has not impacted application design, which includes the design of algorithms. Hardware and system engineers have treated the application as a black box and designed techniques to minimize energy without any modifications to this black box. This is not surprising since application or algorithmic design is not their area of expertise. Interestingly, the focus of the algorithmic community has been on designing energy management algorithms in support of the energy management techniques developed by other computing disciplines [36, 7, 16]. We believe that the algorithmic community can make a more significant contribution by introducing energy-awareness in algorithm design. The central question explored in this paper is the following:

Should we redesign software applications using energy-optimal algorithms?

Traditional algorithm design has mostly focused on minimizing time complexity (in the RAM model) and used polynomial time as a benchmark for efficient computation. With advances in computing technology, the algorithms community has refined the polynomial time model to better model these advances. The Input-Output (henceforth, I/O) model of Aggarwal and Vitter [5] was proposed to model the fact that problems of large size may not fit in main memory. This model has been refined over the years e.g., to include more realistic memory hierarchy in the form of cache oblivious algorithms [12] and parallel disks [34, 35]. Another popular recent model is that of data stream algorithms [26], which tries to model the fact that sequential access to disk memory is cheaper than random access to disk memory.

However, the energy consumed by a computational task is not covered by traditional algorithm design (nor by models mentioned above). There is a technological reason behind the relative lack of attention from algorithm community towards energy-aware algorithm design. Server models, developed as recently as 2004 (e.g., IBM Power5), were designed without energy awareness in mind and their power consumption was *work-oblivious*, i.e. they consumed the same power irrespective of the workload running on the server! It is only in the last decade that hardware architects have introduced techniques to ensure that circuitry not performing any operations at a given point in time consume minimal or no power.

In line with these change, ideal *work-proportional* energy models have been used recently for parallel algorithms, where a linear relationship is assumed between computational complexity and energy usage of an algorithm on a server running at a fixed frequency [21, 22]. This is diagonally opposite to work-oblivious energy models and do not model the complex memory circuitry. The second term in (1) represents this model.

We performed a number of experiments and observed that energy consumption is neither work-proportional nor work-oblivious. Fig. 2 shows the energy consumed by algorithms with *exactly* identical work and memory complexity (i.e. they are same even up to constants) on commodity desktops. The algorithms have been modified by changing their memory allocation on banks and the average power is measured to estimate the total energy consumed in completing a task. (See Appendix 6 for more details.) Energy estimated by work-proportional or work-oblivious models would be same for a given problem, i.e., max=min (represented by the flat line in the figure). We observe up to 225% higher energy usage for the same algorithm, highlighting a gap in existing complexity models. The platforms we used for our experiments had only 8 memory banks and do not support advanced fine-grained energy management techniques like per-core DVFS or per-bank refresh states [18]. We expect the difference between variants of algorithms to increase on larger memory systems or emerging platforms.

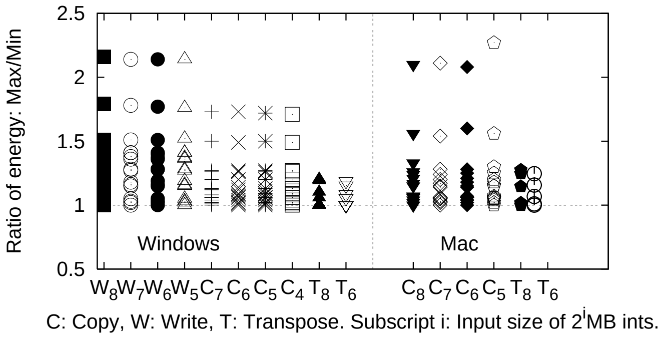

Figure 2: Impact of algorithm on energy. Each dot in the figure signifies the energy cost incurred by a given algorithm (among write, copy, and transpose) for a given input and access pattern.

A simple complexity model that captures the energy consumed by an algorithm can help design energy-efficient applications that can perform the same task, while consuming much less energy. An energy-aware design of applications will ensure that the entire computing stack from application to the hardware is energy efficient and has the potential to significantly reduce energy consumption of enterprise as well as personal computing devices. Designing such a model is tricky: it has to be simple enough so that it hides as much of the low level system detail to the algorithm design (so that the designed algorithms can be applied to various platforms) while on the other hand it should be a realistic enough to capture a lot of the ground truth.

System researchers have designed various models to estimate the power drawn by server systems. At high-level, they generalize the ideal work-proportional energy model of Korthikanti *et al.* [21, 22] by capturing memory activity as well as idle or leakage power. However, these models are typically based on system level parameters like CPU utilization, number of instructions dispatched, memory bandwidth etc. [9]. A recent study presented an application-aware model for estimating power drawn by server systems [20]. Even though the authors point out the requirement of capturing application behavior to accurately estimate power, the techniques treat an application as a black box and estimate the power based on empirical measurements. We believe that the lack of a simple yet realistic model of energy consumption at design time is one of the stumbling blocks for energy-aware algorithm design.

1.1 Our Results

Our main contribution is a simple energy complexity model for algorithms. Our model is based on the design of modern processors and memory elements. In particular, it turns out that a simple variation of the parallel disk I/O model [34, 35] (where the fast memory is not fully associative) models modern memory elements pretty well. Using this memory model, our final energy model for an algorithm $\mathcal{A}$ is a weighted linear combination of the work complexity of $\mathcal{A}$ (i.e. its time complexity in the traditional RAM model) and the number of "parallel" accesses to the memory. (We note that our memory model has some subtle differences from existing models such as the parallel disk I/O model and cache oblivious models: see the end of the section for a comparison.) To obtain our final energy model, we start with a fairly complex energy model and then use some simple reductions to reduce our energy complexity model to the two parameters above. We also present some simulation results to justify our energy model.

To the best of our knowledge, ours is the first work that naturally combines the work complexity and parallel I/O complexities into one. We also believe that the fact that our energy model is a simple combination of two well-studied models is a strength of our model as one can leverage existing techniques that have been developed for designing algorithms in these two models. Existing work in cache oblivious algorithms has focused on both of these parameters but existing literature in that model tries to optimize both these parameter *simultaneously*. By contrast, our model since it combines these two parameters, provides some flexibility. Indeed in our results, we use this freedom. For example, the problem of sorting N numbers has an optimal work complexity of $W(N) = \Theta(N \log N)$ (assuming a comparison model) and a parallel I/O complexity (where one has P parallel disks, each block has B items and the fast memory can hold P blocks) of $\mathcal{I}(N) = \Omega(\frac{N}{BP} \log_P N)$. Our energy model requires the algorithm to minimize the quantity $W(N) + BP \cdot \mathcal{I}(N)$. We briefly describe two examples that illustrate the two ends of the spectrum: one where $W(N)$ is the dominant term and the other where $\mathcal{I}(N)$ is the dominant term. First consider the sorting problem. Under the comparison model, for sorting we have $W(N) = \Theta(N \log N)$. On the other hand, (assuming the "k-way" mergesort algorithm can be fully parallelized), it is known that $\mathcal{I}(N)$ is $O(N/(PB) \log_k N)$. For optimal I/O complexity, one needs to pick $k = P$. However, in our case since $W(N)$ is the dominating term, even $k = 2$ (the "usual" mergesort algorithm) is also energy optimal. Second, consider the problem of permuting an array of elements under an implicitly specified permutation. This is a special case of the sorting problem and it is well known that for this problem $W(N) = \Theta(N)$. On the other hand, it is known that the I/O lower bound is the same as that of sorting. Thus, for the permuting problem, we have $\mathcal{I}(N) \geq \Omega(N/(PB) \log_P N)$. Thus, in this case to be energy optimal, we use the P-way mergesort algorithm (that is fully parallelized) to obtain the energy optimal algorithm for permuting. We are hopeful that this flexibility will enable the design of more algorithms that have the optimal energy complexity.

We also believe that our model is complementary to the existing work on energy efficient algorithms [6]. In particular, one could potentially use our model to design the algorithmic tasks and then using the existing work to schedule each of these tasks so as to leverage the benefits of speed scaling and power-down mechanisms.

Our second key contribution is simulations of broad classes of algorithms. In particular, we present three simulations that convert algorithm designed for RAM and/or I/O model into ones that are suitable for our model. Our simulations do not change the work complexities but mildly blowup the parallel I/O complexity parameter. The first simulation result is a trivial one where the original algorithm already has a nice parallelizable memory access pattern and one gets the same work and I/O complexity as the original algorithm. The next two classes of algorithms are based on their memory access patterns. Informally, the first subclass, which we call *striped* algorithms run in phases where in each phase, the algorithm accesses a fixed number of "runs" of memory locations sequentially (e.g. the merge sort algorithm). We use the power of two choices [25] to simulate this class of algorithms. We also consider a much more general class of algorithms we call *bounded*, which has the minimal restriction of knowing which the memory locations it needs to allocate upfront and one that accesses at least a constant fraction of these allocated locations during its execution. The parameters for bounded algorithms are weaker than those of striped algorithm. We then apply these simulation results to obtain energy efficient algorithms for sorting, matrix transpose, matrix multiplication and matrix vector multiplication.

1.2 Proof Techniques

Finally, we present an overview of some of the proof techniques used to prove the simulation results. The idea for the simulation is fairly simple: we randomly permute (twice in the case of striped algorithms) the memory locations so that we can "batch" memory accesses into few parallel I/O accesses. This involves bounding the maximum load in a balls and bins game. For bounded algorithms, the resulting balls and bins problem is the traditional setup. For striped algorithms, the setup is the so called "parallel" balls and bins with two choices [25]. In this setup it is know that if the two choices for each ball can be coordinated than the maximum load is $O(1)$. In our case we have to be a bit careful to make sure that these choices can be made (centrally) in linear time. (This latter result is probably folklore– we present a proof in Appendix D.)

The main issue is how one stores this random permutation (since the corresponding map needs to be computed every time a memory location is accessed). We present two solutions. The first solution uses a lot of randomness and affects the work and I/O complexity negligibly. The second solution uses little randomness but has a mild effect on the work complexity. The first solution just divides up the memory locations into appropriate chunks and applies independent random permutations on each such chunk. These permutations are themselves stored in memory. This potentially could lead to the problem of ensuring that the accesses to the table entries themselves are sufficiently parallelizable. We overcome this by exploiting a fact that has been used before: the size of the table is much smaller than the input. This allows us to repeat each entry multiple times, which in turn allows for sufficient parallelizability.

For the second solution, a natural option is to reduce the randomness in the first solution by using a standard derandomization tool. An inspection of the proof implies that one only needs (roughly) P-wise independent random sources to make the balls and bins argument go through. A natural step would then be to use P-wise independent random permutations. Unfortunately, good construction of such pseudorandom objects are not known: in particular, we need efficient algorithms that given the pure random bits can compute the resulting map. (In fact, the mere existence of such objects has only been proven recently [23].) To get around this predicament, we use the well-known P-wise independent random *strings* based on Reed-Solomon codes (which in turns are evaluations of low degree polynomials). Further, to make things easier, we first only map the locations to a random disk. To determine the location within a disk, we simply maintain a count of how many locations have already been mapped to the disk under consideration. (As before all this information is stored in a table.) To get a better work complexity, instead of using the naive polynomial evaluation algorithm, we use a recent efficient data structure for polynomial evaluation designed by Kedlaya and Umans [17].

1.3 Related Work

A lot of research has been carried out in the systems community to reduce energy consumption in diverse aspects of computing including design of server systems, voltage and frequency scaling, server consolidation, and storage energy management [11, 15, 27, 28, 31, 32]. These problems have also garnered a lot of attention in the theory community: see e.g. the recent survey by Albers [6], which gives an overview of the various threads of energy related research in the theory community. One of these techniques called speed scaling (which allows changing the speed of the processor in order to save energy) was defined about a decade and a half ago in the seminal paper of Yao et al. [36]. The last few years has seen a lot of activity in this area: see e.g. the paper by Bansal et al. [7]. Another approach is to dynamically power down a machine (or choose among multiple low power states) [16]. Both of these general approaches lead to scheduling problems in both the online and offline settings. However, in our opinion, the models proposed above for energy management though very nice, stop short of making energy minimization a primary goal of algorithm design like the I/O model made minimizing number of disk accesses a primary objective for algorithm design. The main impediment for pursuing this direction is a realistic model of energy consumption at the algorithm design. We believe that our model will complement this body of existing work.

Memory models with a small fast memory and a large slow memory have been studied for a long time. The I/O model [5] presents an external memory model, where accesses are made in blocks of size B to a fast memory of size M. Further, up to P parallel I/Os can be made simultaneously. Vitter and Shriver [34] extend this model with the restriction that the parallel I/Os can be made to P parallel disks only. This additional constraint introduces the aspect of data layout in the algorithm design, which ensures that blocks needed in parallel are written out to different disks. However, there is no associativity restriction and any block in slow memory can be mapped to any arbitrary block in the fast memory. Similar models have been proposed to model the number of cache misses in a 2-level cache-memory model with limited associativity [24]. A fundamental difference between our energy model and existing models is that in our model, we have a small value of M/B (equal to P) and an associativity of only 1 (every bank has exactly one sense amplifier and hence all blocks in slow memory have exactly one slot in fast memory that they can be mapped to).

Our model also has similarities with the cache oblivious model [12]. In this model, the goal is to minimize the number of cache misses (equivalent to the number of I/Os), while ensuring that the algorithm is also work optimal. Our model differs from this work in three ways: (i) The cache model assumes 'ideal cache' with high associativity whereas we have an associativity of 1 (ii) The tall cache assumption does not hold in our model due to small M/B and (iii) We aim to minimize a linear combination of work complexity and number of parallel I/Os whereas the cache oblivious model tries to minimize both the work complexity and the number of sequential I/Os without worrying about parallelism.

Parallel I/O algorithms naturally require a lookahead to read P blocks in parallel. Data oblivious algorithms are useful in this regard as they do not require a lookahead. We consider one data dependent problems in our work: sorting. For sorting, Goodrich recently presented an I/O optimal data oblivious algorithm [13]. However, it is non-trivial to ensure parallel access for the algorithm. Hence, in this work, we use prediction sequences [8] for ensuring lookahead as opposed to using data oblivious algorithms, which may be an alternate approach.

1.4 Extensions to our model

There are two obvious extensions to our energy model proposed in this work. First, since we wanted to follow the DDR3 memory architecture, each of the P disks has a cache that can only hold one block. A natural extension of this is to allow for the caches to be larger (this will mimic the "tall cache" assumption in the cache oblivious algorithms model). Our results easily extend to this more powerful model. We would like to point out that since our simulation results works for the much smaller caches (and the simulation only changes the disk memory accesses), they are more powerful and apply easily to the case when the cache sizes are larger. Another natural extension is to augment our energy model for multi-core machines. We leave this extremely interesting extension as future work.

2. DERIVATION OF AN ENERGY COMPLEXITY MODEL

The goal of our work is to define a model to capture the energy consumed by a modern server for executing an algorithm. In traditional data centers, servers perform the computation and I/O is handled separately by network attached storage. In this work, we focus only on server power, which includes the power drawn by processors and the server memory. Storage power is beyond the scope of this work.

2.1 Processor Power

The processor power drawn by a server can broadly be classified into the leakage power drawn by the processor clock and the power drawn by the processing elements. The processing power is determined by the number of operations performed by the processor. Hence, the computational energy consumed by a server for an algorithm $\mathcal{A}$ can broadly be captured as

$$E_{CPU}(\mathcal{A}) = P_{CLK}T(\mathcal{A}) + P_W W(\mathcal{A}) \tag{1}$$

where P_{CLK} is the leakage power drawn by the processor clock, $T(\mathcal{A})$ is the total time taken by the algorithm, and $W(\mathcal{A})$ is the total time taken by the non-I/O operations performed by the algorithm. (See (4) for the exact relation.) P_W is used to capture the power consumption per operation for the server.

2.2 Memory Elements

Modern memory is based on the DDR architecture [1]. Memory in this architecture is organized into a set of parallel banks. Each bank is arranged into a set of rows. Further, each bank has a sense amplifier in which exactly one row from the bank can be brought in. When a memory word needs to be read (or written), the row containing the word receives an ACT command. The ACT command activates the row and brings it into the sense amplifier. A column decoder is then used to identify the appropriate column of the row and a RD (or WR) command reads (or writes) the corresponding word from the sense amplifier. When the data is no longer needed, a PRE command writes back the sense amplifier into the row in the bank that the data belongs to. A memory clock is used to synchronize commands when the memory is activated.

A DDR memory may either be executing a command (e.g., ACT, PRE, RD, WR), be in power down mode, in a standby mode, or in an activated mode. A power down mode implies that no bank in the memory is being used and is not useful for algorithm design as it represents an input size of 0. When the memory is in the standby mode, each bank can individually be in an activated state (the sense amplifier has a row) or in a standby (precharged) state (the sense amplifier is empty). The time taken to activate a row in a bank is two orders of magnitude higher than the time taken to read a word. All banks can individually receive commands and execute these commands in parallel. Hence, memory access exhibits a parallel behavior (data in different banks can be parallely accessed) with a serial dependency (data in two rows of the bank can only be accessed in sequence).

We model the memory power as a sum of three components: (i) the first component P_{CKE} captures the leakage power drawn when the memory is in standby mode, but none of the banks are activated, (ii) the second component P_{STBY} captures the incremental cost over and above the leakage power for banks to be activated and waiting for commands, and (iii) the third component captures the incremental cost of various commands. The standby power needs to be computed for the duration that a bank is active and hence needs to be multiplied with the product of number of activations and the average activation time. Since ACT and PRE commands are always paired, we capture the energy cost of an ACT and a PRE command as E_{ACT}. Similarly, the cost of a RD or WR command is captured as $P_{RDWR}T_{RDWR}$. Hence, we model the memory power as

$$\begin{aligned} E_{MEM}(\mathcal{A}) =& P_{CKE}T(\mathcal{A}) + P_{STBY}T_{ACT}(\mathcal{A})ACT(\mathcal{A}) + \\ & E_{ACT}ACT(\mathcal{A}) + (RD(\mathcal{A}) + WR(\mathcal{A}))T_{RDWR}P_{RDWR}. \end{aligned} \tag{2}$$

where $ACT(\mathcal{A})$, $RD(\mathcal{A})$, $WR(\mathcal{A})$ denotes the number of activation cycles (ACT and PRE pair), the number of reads and writes respectively executed by an algorithm $\mathcal{A}$. $T_{ACT}(\mathcal{A})$ is the average time taken by one activation by $\mathcal{A}$.

2.3 Energy Complexity Model

We now present the energy complexity model, which is one of the key contributions of our work. (In what follows we will sometimes use equality instead of $\Theta(\cdot)$ to reduce clutter in some of the expressions.) We first combine the processor and memory power ((1) and (2) resp.) into a single model

$$\begin{aligned} E(\mathcal{A}) =& P_{CLK}(f)T(\mathcal{A}) + P_W W(\mathcal{A}) + P_{CKE}T(\mathcal{A}) + \\ & P_{STBY}T_{ACT}(\mathcal{A})ACT(\mathcal{A}) + E_{ACT}ACT(\mathcal{A}) + \\ & (RD(\mathcal{A}) + WR(\mathcal{A}))P_{RDWR}T_{RDWR}. \end{aligned} \tag{3}$$

We can represent the time element in the energy equation as

$$T(\mathcal{A}) = W(\mathcal{A}) + ACT(\mathcal{A})T_{ACT}(\mathcal{A}) + (RD(\mathcal{A}) + WR(\mathcal{A}))T_{RDWR} \tag{4}$$

where T_{RDWR} is the number of processor cycles needed to execute a RD or WR command. T_{RDWR} has decreased over the years and is now a small constant (between 2 and 10) and can be taken out. Further, since every read and write corresponds to some work done, we can “charge” this work to the corresponding computational operations and thus, we can subsume this term in the work term. Hence, the overall time taken can be represented as

$$T(\mathcal{A}) = \Theta(W(\mathcal{A}) + ACT(\mathcal{A})T_{ACT}(\mathcal{A})). \quad (5)$$

Using the above value of $T(\mathcal{A})$ in (3) and using $T_{RDWR} = \Theta(1)$, we get

$$\begin{aligned} E(\mathcal{A}) =& (P_{CLK} + P_{CKE})(W(\mathcal{A}) + ACT(\mathcal{A})T_{ACT}(\mathcal{A})) \\ &+ P_W W(\mathcal{A}) + P_{STBY}T_{ACT}(\mathcal{A})ACT(\mathcal{A}) \\ &+ E_{ACT}ACT(\mathcal{A}) + (RD(\mathcal{A}) + WR(\mathcal{A}))P_{RDWR}. \end{aligned} \quad (6)$$

Since number of reads and writes are less than $W(\mathcal{A})$ and $P_{RDWR} < P_{CLK}$ (Fig. 4 provides a summary of typical values for these parameters), we subsume the last term in the first term.

$$\begin{aligned} E(\mathcal{A}) =& (ACT(\mathcal{A})T_{ACT}(\mathcal{A}))P_{C_1} \\ &+ W(\mathcal{A})P_{C_2} + E_{ACT}ACT(\mathcal{A}) \end{aligned} \quad (7)$$

where P_{C_1} is $P_{CLK} + P_{CKE} + P_{STBY}$ and $P_{C_2} = P_{CLK} + P_{CKE} + P_W$. The average time taken for each activation $T_{ACT}(\mathcal{A})$ depends on the I/O parallelism of the algorithm. The activation cycle can be captured as a latency parameter (L) (e.g., in DDR3 typical value is 200 cycles) and can be amortized across all banks activated in parallel. For fully parallel access, the expected time to read turns out to be L/P. In modern processors with memory clocks approaching processor clock speeds, the best access to read PB blocks would be $O(PB)$ computations. Hence, $L = \Theta(PB)$ and $T_{ACT}(\mathcal{A}) = PB/k$, where k is the "average" I/O parallelism of the algorithm (or $ACT(\mathcal{A})/k$ is the number of parallel accesses made by the algorithm). Since $E_{ACT} < B \cdot P_{C_1}$ (Fig. 4 and B is typically 4096), we subsume the third term in the second term. Finally, for modern processors, $P_W < P_{CLK}$, which implies that $P_{C_1} = \Theta(P_{C_2})$, which in turn gives us our final model:

$$E(\mathcal{A}) = W(\mathcal{A}) + ACT(\mathcal{A})\frac{PB}{k} \quad (8)$$

Our energy model introduces a new term k or the average I/O parallelism for an algorithm. The other terms are either algorithm independent (P, B) or complexities that can be derived from known complexity models ($W(A)$ is work complexity and $ACT(A)$ can be derived from I/O complexity). The parameter k prevents direct application of known algorithms in other models and it is important to ascertain if the parameter is even needed in practice.

We conducted experiments with varying memory bank parallelism for some benchmarks, which are anecdotally known to be very frequent in applications. These include writing to a large vector, copying from one vector to another and matrix transpose. We also increase the work and I/O complexity by increasing the problem size for these operations and measured the energy. Each experiment is repeated 10 times and the means are reported in Fig. 3. More details of our experimental methodology as well as deviations from the average values is described in the appendix.

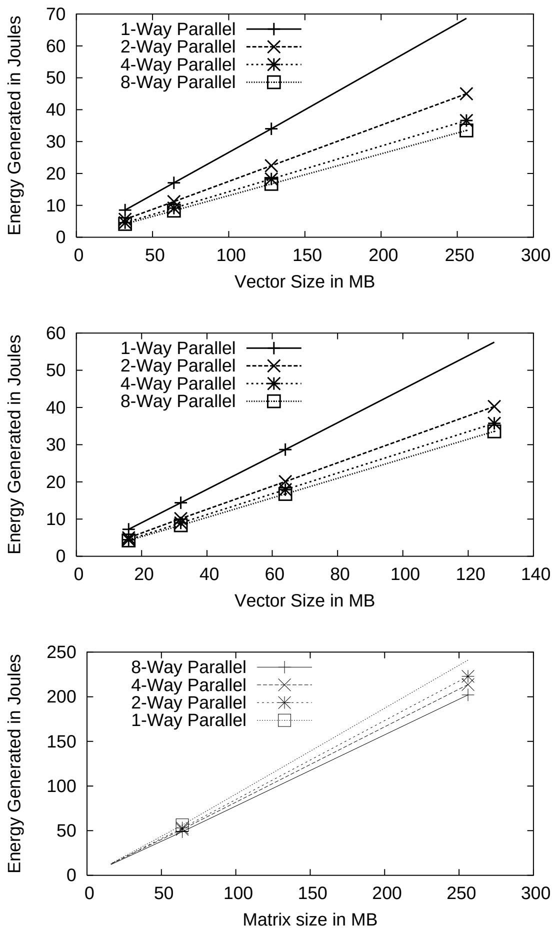

Figure 3: Energy consumed by (a) Write (b) Copy and (c) Matrix Transpose Workloads

As expected from (8), we observe that energy increases linearly with increase in the work and I/O complexity for all three benchmarks (for fixed k). We observe that a 1-parallel layout (i.e. $k = 1$) consumes more than $2X$ energy, as compared to a 4-parallel layout (i.e. $k = 4$) for both *copy* and *write* benchmarks. We would also like to note that our benchmark code had overheads to ensure that our memory allocation provided the parallelism we wanted. This book-keeping of memory allocation was more significant for matrices as opposed to vectors and as a consequence, the difference is moderate for matrix transpose. Finally, for the benchmarks we evaluated, work complexity started to dominate beyond 4-parallel layout. Hence, the performance improvement of using 8 banks in parallel is not significant as compared to using 4 banks in parallel. In fact, as we show later, the tradeoff between work complexity and I/O parallelism is a salient attribute of our model. Since our goal is to minimize a function of these two parameters, we can afford to not optimize any parameter that does not impact the overall objective function for a specific algorithm. We next align our energy model with existing parallel I/O models.

3. OUR ENERGY MODEL

In this section, we present our abstract energy model. In our model, the energy consumption is very closely tied with the memory layout, so we begin with our model of memory.

We will have two levels of memory. The second level of memory (which we will call *main memory*) is divided into P *banks*: we will denote the banks by $\mathcal{M}_1, \ldots, \mathcal{M}_P$. The unit of memory access will be a *block*. The locations of a block in a bank will be called a *bank row*. The collection of all the banks rows with the same (relative) position in each bank will be called a *row*. Each block will contain B items. Each of the P banks has its own *cache*. In particular, we will denote $\mathcal{M}_i$'s cache by $\mathcal{C}_i$. $\mathcal{C}_i$ can only hold *one* block. Further, for any $i \in [P]$, only a block form $\mathcal{M}_i$ can be read into $\mathcal{C}_i$ and only the block in $\mathcal{C}_i$ can be written back into $\mathcal{M}_i$. With these constraints, an algorithm can manipulate the items in the caches $\mathcal{C}_1, \ldots, \mathcal{C}_P$ however it wants. (In other words, as long as it does not need to either read from the main memory or write to the main memory, the algorithm can consider $\mathcal{C}_1, \ldots, \mathcal{C}_P$ as one big cache with PB items in it.) The algorithm is allowed a constant number of extra registers as scratch space.

We emphasize that in our model

$$P < B. \tag{9}$$

In particular, P can be a few orders smaller than B. This for example is different from the *tall cache* assumption in the cache oblivious model, where we would have $P \geq B$.

Given the memory model, we are now ready to start defining our abstract energy model. To begin with, let $\mathcal{A}$ be the algorithm under consideration and let x be an arbitrary input. We define $W(\mathcal{A}, x)$ to be the *work* that algorithm $\mathcal{A}$ performs on input x (i.e. the run time of $\mathcal{A}$ on x without accounting for the time to transfer blocks from/to main memory with each basic computational step taking $\Theta(1)$ work).

Next we account for the memory accesses. Let $\mathcal{I}(\mathcal{A}, x)$ be the sequence of indices of the banks corresponding to the actual memory access for $\mathcal{A}$ in input x.[1] Starting from the beginning of $\mathcal{I}(\mathcal{A}, x)$ we partition the sequence into maximal *batches*. We call a sequence of bank indices $b_1, \ldots, b_m$ to be a batch if (i) All the b_i's are distinct; and (ii) When $\mathcal{A}$ accesses the item corresponding to b_1, it also knows the location of the items corresponding to $b_2, \ldots, b_m$. The number m will be called the *lookahead* of $\mathcal{A}$ when it is accessing the item corresponding to b_1. Note that a batch can have size at most P. Let the resulting sequence of batches be defined as $\mathcal{T}(\mathcal{A}, x)$.

We would like to elaborate a bit on the second property: in general we allow $\mathcal{A}$ to have different lookahead for different item it accesses. We will also work with the following sub-class of algorithms

Definition 1. *Let $\ell \geq 1$ be an integer. An algorithm $\mathcal{A}$ is said to have a* lookahead *of ℓ, if for every memory location it accesses, it knows the location of the next $\ell - 1$ memory locations.*

As an example, consider the case for $P = 4$: $\mathcal{I}(\mathcal{A}, x) = 1, 2, 2, 2, 3,$
$4, 1, 3, 1, 2, 4$, then if the algorithm has a look ahead of 4, then we have $\mathcal{T}(\mathcal{A}, x) = (1, 2), (2), (2, 3, 4, 1), (3, 1, 2, 4)$, and if the algorithm has a lookahead of 2, then we have $\mathcal{T}(\mathcal{A}, x) = (1, 2), (2), (2, 3),$
$(4, 1), (3, 1), (2, 4)$.

Finally, the *batch complexity* of $\mathcal{A}$ on x is defined as $T(\mathcal{A}, x) = |\mathcal{T}(\mathcal{A}, x)|$.

We are now ready to define the *energy* consumed by $\mathcal{A}$ on x as $E(\mathcal{A}, x) = W(\mathcal{A}, x) + L \cdot T(\mathcal{A}, x)$, where L is a *latency* parameter with

$$L = \Theta(PB). \tag{10}$$

Given the size of the input N, we define the energy complexity of $\mathcal{A}$ as $E_{\mathcal{A}}(N) = \max_{x:|x|=N} E(\mathcal{A}, x)$, where we will drop $\mathcal{A}$ from the subscript if it is clear from the context. For a randomized algorithm, we consider its expected energy complexity.

4. SUFFICIENT CONDITIONS FOR EFFICIENT SIMULATION

In this section, we look at certain sub-class of algorithms for which we can do an efficient simulation in our model. To this end, we start with few definitions.

Definition 2 (Runs). *A sequence of memory locations is called a run if the locations can be partitioned into* stripes, *where each stripe consists of all the blocks in the same row in all the P memory banks.*

[1]Each memory access is defined by a bit that signifies whether it was a read or a write; the index of the bank that was accessed; and the bank row that was accessed. However, for this part of the model, we only need to consider the second component.

As an example, consider the case for $P = 4$ let $\mathcal{I}(\mathcal{A}, x)$ be the sequence of indices of the banks corresponding to the actual memory access for algorithm $\mathcal{A}$ in input x. Then $\mathcal{I}(\mathcal{A}, x) = 1, 2, 3, 4, 9, 10, 11, 12, 5, 6, 7, 8, 13, 14, 15, 16$ is a run with the stripes $\mathcal{I}(\mathcal{A}, x) = (1, 2, 3, 4), (9, 10, 11, 12), (5, 6, 7, 8), (13, 14, 15, 16)$.

Definition 3 (k-striped algorithm). *Let $k \geq 1$ be an integer. Call an algorithm* k-striped *if (1) One can divide the execution of the algorithm into phases, (2) In each phase the algorithm reads from at most k runs in memory and writes to one run*[2] *(and the runs are defined at the beginning of a phase); and (3) In each run it accesses the blocks in a sequential manner (i.e. row by row and within each row from the first bank to the last).*

As an example, consider the very simple algorithm of copying the contents of a vector v_1 into another vector v_2. Let v_1 be allocated the sequence of indexes $1, 2, 3, 4, 9, 10, 11, 12$ and v_2 be allocated the sequence of indexes $5, 6, 7, 8, 13, 14, 15, 16$ and $P = 4$. In this case we can have two phases in the algorithm. In the first phase we read the contents of run $(1, 2, 3, 4)$ and write it into run $(5, 6, 7, 8)$. In the second phase the contents of run $(9, 10, 11, 12)$ is written into run $(13, 14, 15, 16)$. Each run here consists of a single stripe and we read from exactly one run at each phase. Hence this algorithm in this case k-striped with $k = 1$.

Definition 4 (Bounded algorithm). *Call an algorithm* bounded *if (1) The algorithm only accesses blocks only within a run (and the run is defined/known at the beginning of the algorithm); and (2) The ratio of the size of the run to the number of memory access is upper bounded by a constant.*

Definition 5. *Let $\ell \geq 1$ be an integer. Then given an algorithm $\mathcal{A}$ and an input x, we call the sequence $\mathcal{I}(\mathcal{A}, x)$ to be* (ℓ, τ)-parallelizable *for some integer $1 \leq k \leq \ell$, if for every ℓ consecutive blocks in $\mathcal{I}(\mathcal{A}, x)$ has at most τ batches in it.*

We note that pretty much any reasonable algorithm will be a bounded algorithm. E.g. if an algorithm can earmark all its required space in one contiguous run and it does not "over-budget" by more than a constant factor, then it will be a bounded algorithm.

We first begin with a simple observation that any algorithm in the (parallel disk) I/O model can be easily modified to work in our model with a similar degree of parallelism and without incurring a penalty in the lookahead.

Theorem 4.1. *Let $\mathcal{A}$ be an algorithm with lookahead ℓ in the parallel disk I/O model with $P - 1$ disks and $M = (P - 1)B$. Then there exists an algorithm $\mathcal{A}'$ with the same functional behavior as $\mathcal{A}$ such that for every x, we have*

$$|\mathcal{I}(\mathcal{A}', x)| \leq O\left(|\mathcal{I}(\mathcal{A}, x)|\right)$$

and

$$W(\mathcal{A}', x) \leq W(\mathcal{A}, x) + O(|\mathcal{I}(\mathcal{A}, x)| \cdot B).$$

Further, $\mathcal{A}'$ is bounded (k-striped respectively) if $\mathcal{A}$ is a bounded (k-striped resp.) algorithm. Finally, if $\mathcal{A}$ has a lookahead of ℓ, then so does $\mathcal{A}'$.

Proof. Recall that the only way that the parallel disk I/O model differs from our memory model is that our P caches are associated with the P banks while the memory in the parallel I/O disk model is not associative. In our simulation, we will show that this is not a big restriction if we have one extra bank: the main idea is to use the cache of the extra bank as a temporary space. More precisely, we fix a map from the $(P - 1)B$ blocks in memory used by $\mathcal{A}'$ to the first $P - 1$ caches. (We also do the obvious map from the $P - 1$ disks to the $P - 1$ banks.) The only constraint on this map is that given any block's location for $\mathcal{A}$ we can immediately compute its location in the $P - 1$ caches. For the rest of the proof, we will assume that this mapping has already been done. Next we define $\mathcal{A}'$. We will show how an arbitrary I/O in $\mathcal{A}$ can be simulated by a memory access in our model and $O(B)$ extra work per I/O, which prove the desired result. Assume that $\mathcal{A}$ needs to transfer a block b from bank i to the cache j. If $i = j$, then this can be done easily. If not, we first move the contents of the ith cache to the last cache, bring in the block b to the ith cache, move b from the ith cache to the jth cache and finally move back the contents of the last cache to cache i. Similarly one can simulate a write from one of the caches to one of the banks.

Finally, since $\mathcal{A}'$ has the same access pattern in the banks as $\mathcal{A}$ does (in the corresponding disks), it preserves the lookahead as well as the property of being striped/bounded as $\mathcal{A}$. $\square$

We start with the following simple observation:

Lemma 4.2. *If a sequence $\mathcal{I}(\mathcal{A}, x)$ is (ℓ, τ) parallelizable, then $T(\mathcal{A}, x)) \leq \frac{|\mathcal{I}(\mathcal{A}, x)|}{\ell} \cdot \tau$.*

Proof. The sequence can be divided into $\frac{|\mathcal{I}(\mathcal{A}, x)|}{\ell}$ blocks with ℓ bank indices in them. Further, as $\mathcal{I}(\mathcal{A}, x)$ is (ℓ, τ)-parallelizable, each of these blocks have at most τ batches in them, which proves the claim. $\square$

The above implies the following simple observation:

Lemma 4.3. *Let $\mathcal{A}$ be an algorithm with lookahead $\ell = \Theta(P)$ such that for any input x, $\mathcal{I}(\mathcal{A}, x)$ is $(\ell, O(1))$-parallelizable. If $\mathcal{A}$ is worst-case work-optimal and I/O optimal, then $\mathcal{A}$ is also energy optimal.*

Proof. The main observation is simple: since there are P banks for any algorithm $\mathcal{A}'$ and input x, we have

$$T(\mathcal{A}', x) \geq \frac{|\mathcal{I}(\mathcal{A}', x)|}{P}.$$

Thus, Lemma 4.2 implies that for any given sequence, the number of batches is within a constant factor of the optimal. This implies that for any (large enough N), since the algorithm has worst-case optimal I/O complexity, $\max_{x:|x|=N} T(\mathcal{A}, x)$

[2]Our results can handle writing to a fixed constant many runs. However, it is easy to see that we can simulate such an algorithm with another one that just has one run with constant overhead in the parameters. Hence, we only explicitly consider the case have a single "write run."

is worst-case optimal. Further, it is given that the algorithm is worst-case work optimal. Thus, the claim follows by the definition of $E_{\mathcal{A}}(N)$ (and the fact that the delay parameter L is independent of N). $\square$

We are now ready to state and prove our first non-trivial sufficient condition:

THEOREM 4.4. *Let $1 \le k, \ell \le P$ be integers. Let $\mathcal{A}$ be a k-striped algorithm with lookahead ℓ. Then there exists a (randomized) algorithm $\mathcal{A}'$ with the same functional behavior as $\mathcal{A}$ (and uses up at most three times as much space as $\mathcal{A}$ does) such that for every x, we have $|\mathcal{I}(\mathcal{A}', x)| \le O(|\mathcal{I}(\mathcal{A}, x)|)$, and $\mathcal{I}(\mathcal{A}', x)$ is (ℓ, τ)-parallelizable, where*

$$\mathbb{E}[\tau] \le O(1). \tag{11}$$

In particular, we have

$$\mathbb{E}\left[T(\mathcal{A}', x)\right] \le O\left(\frac{|\mathcal{I}(\mathcal{A}, x)|}{\ell}\right). \tag{12}$$

Finally,

$$W(\mathcal{A}', x) \le O(W(\mathcal{A}, x)). \tag{13}$$

PROOF. We first argue that the number of batches in $\mathcal{A}'$ is bounded. (12) follows from (11), Lemma 4.2 and the linearity of expectation. Thus, for the rest of the proof, we focus on proving (11).

We will design $\mathcal{A}'$ from $\mathcal{A}$ by using the power of two choices in the balls and bins framework. The main idea is the following: at the start of each phase we make two copies of the run. We divide each of the two copies into disjoint groups and (independently) reorder each group with a random permutation. When we need to read a block, we choose one that minimizes the "load." In particular, since $\mathcal{A}$ has a lookahead of ℓ the idea is to divide each group of ℓ memory locations into small number of batches. It turns out that this number of batches (which will determine τ) is exactly the maximum load in a balls and bins problem induced by the above process (which corresponds to a balls and bins problem with two choices). We now present the details.

We will show how $\mathcal{A}'$ simulates any arbitrary phase of $\mathcal{A}$ with the required properties. The claimed result then just follows by applying the same procedure to all the phases. For the rest of the proof, we will assume that $\mathcal{A}$ only uses the first $P/2$ banks and does not access any block in the last $P/2$ banks. (In general, this assumption might not be true but since $\mathcal{A}'$ does a step by step simulation of $\mathcal{A}$, one can always modify $\mathcal{A}'$ so that $\mathcal{A}$ effectively only uses the first $P/2$ banks.)

Fix any phase of $\mathcal{A}$ and consider the time when $\mathcal{A}$ is about to start the phase. The simulation proceeds in three steps:

Pre-Processing.

At the beginning of the current phase, $\mathcal{A}$ knows the run. Let m be the size of the run. For now, let us assume that

$$m \le 2^{2^{B/P}} \tag{14}$$

$\mathcal{A}'$ will copy the run into fresh space (which is not used by $\mathcal{A}$). At the same time, for every block it stores the map between the old location and the new location. We now present the details.

Since $\mathcal{A}$ knows the run in the current phase, we'll assume w.l.o.g. that given the index of any block in the run, one can compute its actual location. Note that there are m/B blocks in the run (we'll assume that B divides m– this does not affect any of the asymptotic bounds later on). In particular, one can fit $g = \lfloor B/\log(m/B)\rfloor$ such indices into one bank row. (Note that (14) implies that $g \ge P/2$.) To compute the random map, we proceed as follows:

> Iterate the following $2(m/B)/(gP)$ times. In the $0 \le i < 2(m/B)/(gP)$ iteration, do the following *twice* (with independent randomness). Write the memory locations of the blocks indexed[3] by $i \cdot gP/2, \ldots, (i+1)\cdot gP/2 - 1$ into the last $P/2$ caches. Then using Knuth's Algorithm P (also known as the Fisher-Yates shuffle) [19] randomly permute the g contiguous groups of $P/2$ indices each in-place. (For different groups and iterations we use independent randomness. Note that we are permuting the $gP/2$ indices where relative positions of groups of sizes $P/2$ remains the same.) After the permutation, write all the $P/2$ cyclic shifts of the contents of the $P/2$ caches into $P/2$ rows in fresh memory in the last $P/2$ banks (allocated for storing the random map).

(Note that each block gets permuted randomly twice. We will refer to these two random maps as the first and second random map. We'll see later why the $P/2$ cyclic shifts will be useful.)

Next we copy the actual contents of the run: we access the run row by row and then using the random map that we stored above we copy $P/2$ blocks at a time. In particular, we load the random maps for the m/B blocks in chunks of size $gP/2$ in the last $P/2$ caches (each of the $gP/2$ locations have $P/2$ copies: pick any arbitrary one). We then divide the $gP/2$ into chunks of size $P/2$ (i.e. we consider the maps for the first $P/2$ of the m/B blocks and so on). For each chunk we copy the corresponding blocks in the first $P/2$ banks into their corresponding locations in a fresh memory space (that is not used by $\mathcal{A}$). We do this for both the first and the second random maps.

We now argue about the contribution of the pre-processing step to $\mathcal{I}(\mathcal{A}', x)$. We first upper bound the number of new accesses. During the creation of the random map, in each of the $2m/(gBP)$ iterations, we add $2\cdot P^2/4$ accesses, leading to a total of $mP/(gB) \le 2m/B$ extra accesses. Further, each of the $P/2$ blocks that are written to one row form a single batch (for a total of $m/(BP)$ batches). For the copying part, for loading the random maps into the last $P/2$ caches, the number of extra accesses and batches can again be upper bounded by m/B and $m/(PB)$ respectively. Finally, we consider the copying part: we copy over m/B blocks in chunks of size $P/2$, so the number of extra accesses is again at most

[3]Since everything in part of a run w.l.o.g., we will assume that these memory locations are also contiguous.

m/B. Further, while copying one chunk of $P/2$, note that by the assumption that these form a contiguous part of the run, when we randomly permute these $P/2$ elements, each such chunk forms exactly one batch.

Thus, overall, the preprocessing step adds $O(m/B)$ to accesses in $\mathcal{I}(\mathcal{A}', x)$. Further, these accesses are $(P/2, \tau)$ parallelizable, where τ satisfies (11). Recall that so far we have assumed that m satisfies (14). However, if this is not the case, we can just divide the run into consecutive chunks of size $2^{2B/P}$ and then apply the above procedure on each of these chunks. Note that doing so still preserves all the claimed bound (in terms of general value of m).

Since $\mathcal{A}$ is striped, it implies that m/B is $O(|\mathcal{I}(\mathcal{A}, x)|)$. Thus, overall this phase adds $O(|\mathcal{I}(\mathcal{A}, x)|)$ accesses with the required parallelizability property.

Simulation.

We now state how we simulate the actual computation steps of $\mathcal{A}$ in $\mathcal{A}'$: we will run the exact same steps as in $\mathcal{A}$ except when it does a memory read we access the "lighter loaded" of its two copies (according to the first and second random maps that we have stored). For memory writes since we have assumed that the algorithm writes to a single run in a sequential manner, we can easily simulate this step within the required resource parameters. Hence, for the rest of the description, we will focus on the simulation of the reads.

In particular, since $\mathcal{A}$ has a look-up of ℓ, $\mathcal{A}'$ will process the memory accesses of $\mathcal{A}$ in chunks of size $\ell \leftarrow \min(\ell, P/512)$ and perform the following two tasks:

- Load the two new locations of the ℓ blocks from the stored first and second random maps into the last $P/2$ caches.

- Think of the ℓ blocks as balls and the p banks as bins in the balls and bins framework in Section D. Then for each block pick the copy from either the first or second random map as dictated by the allocation algorithm in Section D.

- Using the location of the corresponding block from the previous step, one does the corresponding accesses into the ℓ blocks.

It is easy to see that each of the step above lead to at most ℓ memory accesses and thus, overall this step contributes $O(|\mathcal{I}(\mathcal{A}, x)|)$ number of accesses to $\mathcal{I}(\mathcal{A}', x)$. We now argue about the parallelizability of the accesses in this step. For the first step above, note that given the ℓ locations one can easily compute the corresponding indices into the random map. Given that each entry in random map is repeated $P/2$ times in all the last $P/2$ banks, implies that when we want to look up the new locations for the ℓ blocks, we pick the copies in such a way that all the ℓ accesses form a single block. For the second step above, Theorem D.2, implies that each of the chunk of ℓ accesses has $O(1)$ batches (in expectation). Thus, overall all the accesses in this step are (ℓ, τ)-parallelizable with τ as in (11), as desired.

Post-Processing.

In this step, we copy back blocks from the fresh memory to their original locations. To do this, we go over the old locations row by row, load the corresponding entries from (say the first) random map and then copy back the contents of the block in the new location to the original location.[4] It is easy to see that using the same arguments as in the pre-processing step, we add $O(|\mathcal{I}(\mathcal{A}, x)|)$ accesses with the required parallelizability property.

Finally, we argue (13). The inequality follows from the fact that the only ways $\mathcal{A}'$ differs from $\mathcal{A}$ are (i) Computing the random map– one needs $O(1)$ amount of work per entry in the random map (this includes the time taken to compute the random permutation– the Fischer-Yates shuffle can be implemented with $O(1)$ work per element); (ii) the copying of elements from the runs that are "active" in a phase to another portion in the memory using the random map– this just needs $O(1)$ work per elements in the run; (iii) while doing a memory access, one can has to compute its location from the random map, which again takes $O(1)$ work per element (here we use the fact that when computing the best of two choices we can compute the choices for the $\Theta(P)$ locations in time $O(P)$ due to Lemma D.1); and (iv) Undoing the random map for each block in the run, which again takes $O(1)$ work per element. The proof is complete by noting that the algorithm accesses each active element in each of the phases at least once. $\square$

We now ready to state and prove our second non-trivial sufficient condition:

Theorem 4.5. *Let $1 \le \ell \le P/2$ be an integer. Let $\mathcal{A}$ be a bounded algorithm with lookahead ℓ that uses space $S(x)$ for input x. Then there exists a (randomized) algorithm $\mathcal{A}'$ with the same functional behavior as $\mathcal{A}$ (and uses three times as much space as $\mathcal{A}$ does) such that for every x we have $|\mathcal{I}(\mathcal{A}', x)| \le O(|\mathcal{I}(\mathcal{A}, x)|)$, and $\mathcal{I}(\mathcal{A}', x)$ is (ℓ, τ)-parallelizable, where*

$$\mathbb{E}[\tau] \le O\left(\frac{\log P}{\log\log P}\right). \tag{15}$$

In particular, we have

$$\mathbb{E}\left[T(\mathcal{A}', x)\right] \le O\left(\frac{|\mathcal{I}(\mathcal{A}, x)| \cdot \log P}{\ell \cdot \log\log P}\right). \tag{16}$$

Finally,

$$W(\mathcal{A}', x) \le O(W(\mathcal{A}, x)). \tag{17}$$

Proof. The proof is similar to that of Theorem 4.4 except that we randomize the blocks used by $\mathcal{A}$ differently: we just randomly permute each group of size $P/2$ once and then apply the balls and bins result from Section C. For the sake of completeness, next we present the details.

[4]Since we can do both reads and writes in a phase, we first copy back the runs that were read and then copy back the run that was written to– this way we ensure that the original locations after the phase is done has the correct value.

Using a similar argument as in the proof of Theorem 4.4, w.l.o.g., we can assume that

$$|S(x)| \leq 2^{2B/P}. \tag{18}$$

Unlike before, $\mathcal{A}'$ does the random permuting before simulating any steps of $\mathcal{A}$. Thus, we will only need a pre-processing step (but no post-processing step). For the rest of the proof, we will assume that $\mathcal{A}$ only uses the first $P/2$ banks and does not access any block in the last $P/2$ banks. (In general, this assumption might not be true but since $\mathcal{A}'$ does a step by step simulation of $\mathcal{A}$, one can always modify $\mathcal{A}'$ so that $\mathcal{A}$ effectively only uses the first $P/2$ banks.)

Pre-Processing.

Note that since $\mathcal{A}$ is bounded, it knows the entire space that is going to use. For notational convenience, define $m = S(x)$. $\mathcal{A}'$ will copy this space into fresh space (which is not used by $\mathcal{A}$). At the same time, for every block it stores the map between the old location and the new location. We now present the details.

Since $\mathcal{A}$ knows the run it is going to use when executing itself on x, we'll assume w.l.o.g. that given the index of any block in the run, one can compute its actual location. Note that there are m/B blocks in the run (we'll assume that B divides m– this does not affect any of the asymptotic bounds later on). In particular, one can fit $g = \lfloor B/\log(m/B)\rfloor$ such indices into one bank row. (Note that (18) implies that $g \geq P/2$.) To compute the random map, we proceed as follows:

> Iterate the following $2(m/B)/(gP)$ times. In the $0 \leq i < 2(m/B)/(gP)$ iteration, write the indices $i \cdot gP/2, \ldots, (i+1)\cdot gP/2 - 1$ into the last $P/2$ caches. Then using Knuth's Algorithm P (also known as the Fisher-Yates shuffle) [19] randomly permute the $gP/2$ indices in-place. (For different iterations we use independent randomness.) After the permutation, write all the $P/2$ cyclic shifts of the contents of the $P/2$ caches into $P/2$ rows in fresh memory in the last $P/2$ banks (allocated for storing the random map).

(We'll see later why the $P/2$ cyclic shifts will be useful.)

Next we copy the actual contents of the run: we access the run row by row and then using the random map that we stored above we copy $P/2$ blocks at a time. In particular, we load the random maps for the m/B blocks in chunks of size $gP/2$ in the last $P/2$ caches (each of the $gP/2$ locations have $P/2$ copies: pick any arbitrary one). We then divide the $gP/2$ into chunks of size $P/2$ (i.e. we consider the maps for the first $P/2$ of the m/B blocks and so on). For each chunk we copy the corresponding blocks in the first $P/2$ banks into their corresponding locations in a fresh memory space (that is not used by $\mathcal{A}$).

We now argue about the contribution of the pre-processing step to $\mathcal{I}(\mathcal{A}', x)$. We first upper bound the number of new accesses. During the creation of the random map, in each of the $2m/(gBP)$ iterations, we add $P^2/4$ accesses, leading to a total of $mP/(2gB) \leq m/B$ extra accesses. Further, each of the $P/2$ blocks that are written to one row form a single batch (for a total of $m/(BP)$ batches). For the copying part, for loading the random maps into the last $P/2$ caches, the number of extra accesses and batches can again be upper bounded by m/B and $m/(PB)$ respectively. Finally, we consider the copying part: we copy over m/B blocks in chunks of size $P/2$, so the number of extra accesses is again at most m/B. Further, while copying one chunk of $P/2$, we note that we're in the situation of the balls and bins problem considered in Section C and thus, Theorem C.1, implies that each such chunk will have at most $O(\log P/\log\log P)$ batches in expectation.

Thus, overall, the preprocessing step adds $O(m/B)$ accesses in $\mathcal{I}(\mathcal{A}', x)$. Further, these accesses are $(P/2, \tau)$ parallelizable, where τ satisfies (16). Since $\mathcal{A}$ is bounded, it implies that m/B is $O(|\mathcal{I}(\mathcal{A}, x)|)$. Thus, overall this phase adds $O(|\mathcal{I}(\mathcal{A}, x)|)$ accesses with the required parallelizability property.

Simulation.

We now state how we simulate the actual computation steps of $\mathcal{A}$ in $\mathcal{A}'$: the idea is as before– we'll run the exact same steps as in $\mathcal{A}$ except when it does a memory access we access its corresponding copy (according to the random map that we have stored). In particular, since $\mathcal{A}$ has a look-up of ℓ, $\mathcal{A}'$ will process the memory accesses of $\mathcal{A}$ in chunks of size ℓ and perform the following two tasks:

- Load the new locations of the ℓ blocks from the stored random map into the last $P/2$ caches.
- Using the location of the corresponding block in the fresh space (stored within the last $P/2$ caches), one does the corresponding access into the ℓ blocks.

It is easy to see that each of the step above lead to at most ℓ memory accesses and thus, overall this step contributes $O(|\mathcal{I}(\mathcal{A}, x)|)$ number of accesses to $\mathcal{I}(\mathcal{A}', x)$. We now argue about the parallelizability of the accesses in this step. For the first step above, note that given the ℓ locations one can easily compute the corresponding indices into the random map. Given that each entry in random map is repeated $P/2$ times in all the last $P/2$ banks, implies that when we want to look up the new locations for the ℓ blocks, we pick the copies in such a way that all the ℓ accesses form a single block. For the second step above, Theorem C.1, implies that each of the chunk of ℓ accesses has $O(\log P/\log\log P)$ batches (in expectation). Thus, overall all the accesses in this step are (ℓ, τ)-parallelizable with τ as in (16), as desired.

Finally, we argue (17). The inequality follows from the fact that the only ways $\mathcal{A}'$ differs from $\mathcal{A}$ are (i) Computing the random map– one needs $O(1)$ amount of work per entry in the random map (this includes the time taken to compute the random permutation– the Fischer-Yates shuffle can be implemented with $O(1)$ work per element); (ii) the copying of elements from the run to another portion in the memory using the random map– this just needs $O(1)$ work per elements in the run; (iii) while doing a memory access, one can has to compute its location from the random map, which again takes $O(1)$ work per element.

□

Reducing the Amount of Randomness.

An inspection of the proofs of Theorems 4.4 and 4.5 reveals that $\mathcal{A}'$ needs a lot of random bits. In particular, if the algorithm uses S amounts of "active" space in a given run, then one needs $\Omega(S)$ space. In Appendix B, we show how using limited-wise independent sources based on Reed-Solomon codes, one can reduce the randomness to $O(P \log S)$ bits while pretty much maintaining the rest of the parameters.

5. COROLLARIES FOR SPECIFIC PROBLEMS

5.1 Inner Product.

We start the simple problem of computing the inner product of two vectors of length N. Assuming both the vectors are stored in the same order in memory, the naive algorithm of getting $PB/2$ consecutive entries from each of the vectors from memory to the P caches (and keeping track of the current partial value of the inner product in a register) has the optimal work complexity of $O(N)$ as well as the optimal I/O complexity of $O(N/B)$. Further, it is easy to see that the above algorithm is $(P/2, 1)$-parallelizable. Thus by Lemma 4.3, we have the following result:

Proposition 5.1. *The inner product problem can be solved with energy optimal complexity of* $\Theta(N)$.

We note that for this problem, having the tall cache assumption does not have any impact.

5.2 Sorting.

It is easy to see that the k-way merge sort is a k-striped algorithm. However, since the algorithm is data dependent, the "traditional" merge sort algorithm does not good lookahead. Barve et al. [8] showed how to use a prediction sequence to get around this predicament and achieve a lookahead of $\Theta(P)$ (while not asymptotically changing the work or I/O complexity). For the sake of completeness the details are in Appendix F. The (modified) P-way merge sort has a work complexity of $O(N \log N)$ and an I/O complexity of $O\left(\frac{N}{B}\log_P(N/B)\right)$. Thus, Theorem 4.4 implies the following:

Corollary 5.2. *Sorting of N numbers can be done with the optimal* $\Theta(N \log N)$ *energy complexity.*

We would like to point out that we do not need Theorem 4.4 to prove the above corollary. We note that 2-way merge sort trivially is $(P/2, 1)$-parallelizable. Further, it is known that 2-way mergesort has I/O complexity of $O\left(\frac{N}{B}\log(N/B)\right)$. Thus, Lemma 4.2 then also proves the above corollary. Further, note that in this case having a larger cache can only make the I/O complexity smaller, which makes the time complexity component of the energy complexity dominate even more.

Using the optimal algorithm for sparse matrix-dense vector multiplication from [10] and the result above, we can also obtain energy optimal result for the sparse matrix-dense vector multiplication problem. (The algorithm in [10] uses sorting as a routine and the rest of the algorithm can be performed with linear scans, which are both work and I/O optimal as well as $(P, 1)$-parallelizable.) Indeed sorting is a basic primitive that is used in many I/O algorithms (see e.g. the book by Vitter [33]) and thus Corollary 5.2 would be applicable in many other problems.

5.3 Permuting.

We now consider the problem of permuting a given vector of N items according to (an implicitly specified) permutation. Since this is a special case of sorting, we can use any sorting algorithm to solve this problem. However, one crucial way in which this problem differs from sorting is that the permuting problem has a work complexity of $\Theta(N)$. Further, it is known that the permuting problem has an I/O complexity of $\Omega\left(\min\left(N, \frac{N}{B}\log_M(N/B)\right)\right)$ [5]. This implies that the energy complexity of the permuting problem is $\Omega\left(N + \min\left(NB, N\log_M(N/B)\right)\right)$. Note that this implies that for this problem, the work complexity is the negligible part of the energy complexity (under the assumption that $N \geq MB$, which is reasonable even with the tall cache assumption). In other words, unlike the sorting problem, where even 2-way mergesort gave the optimal energy complexity algorithm, for the permuting problem, we need to use the (modified) P-way mergesort algorithm from Appendix F along with Theorem 4.4, to prove the following result:

Corollary 5.3. *Permuting of N numbers can be done with the optimal* $\Theta(N \log_P(N/B))$ *energy complexity.*

We remark that with larger cache size the logarithm term can be made smaller.

5.4 Matrix Transpose.

The naive $\sqrt{N} \times \sqrt{N}$ matrix transpose algorithm has linear $O(N)$ work complexity but is not I/O-efficient and may require up to N I/Os. Even if the I/O sequence is $(P, 1)$ parallelizable, the batch complexity of the algorithm would be N/P, which is a factor of B times the optimal complexity. Under the assumption of tall cache, the algorithm in [5] that transposes sub-matrices can achieve a batch complexity of $N/(PB)$. In the absence of tall cache, [5] present an algorithm that has a tight I/O bound of $\Omega\left(\frac{N}{B}\frac{\log(M)}{\log(1+M/B)}\right)$.

The recursive matrix transpose algorithm alluded to above is a bounded algorithm and thus, one can apply Theorem 4.5 to it. However, the simulation in Theorem 4.5 suffers an overhead of $O(\log P/\log\log P)$ in its parallelizability, which does not give us the optimal result. Instead we notice that given the special structure of the algorithm we can make the algorithm $(P/2, 1)$-parallelizable (and without using any randomness). This makes sure we still have the optimal parallelized I/O complexity, which leads to the following optimal energy result (even though its work complexity is *not* optimal):

THEOREM 5.4. *The recursive Matrix Transpose algorithm is* $(P/2, 1)$ *parallelizable. Thus, the matrix transpose problem has an optimal energy complexity of* $\Theta(N \log_{(1+P)}(B/P))$.

We note that for larger cache size, we can get rid of the log factor. The proof is in Appendix E. The above result also implies the that matrix multiplication can be done with energy complexity $O(N^3)$. (The algorithm to compute $A \times B$ would be to first transpose B and then perform the natural N^2 inner products using the energy optimal algorithm from earlier in this section.)

6. EXPERIMENTAL SETUP AND ADDITIONAL RESULTS

6.1 Energy Parameters

Parameter	Value
P_{RDWR}	$57mW$
P_{CLK}	$500mW$
P_{CKE}	$102mW$
E_{ACT}	$2.5nJ$
P_{STBY}	$16mW$

Figure 4: Energy Parameters for DDR3 RAM [1] and 1 Intel XScale CPU

Fig. 4 captures the values of various energy components for DDR3 memory. We use these values to derive the simple energy model used in this work.

6.2 Hardware Setup

We conducted our experiments on two different platforms. The first platform is a desktop server 2 Intel Xeon cores, each operating at a frequency of $2GHz$. The desktop has $2GB$ DDR2 memory with 8 banks and is running Windows 7. For running our experiments, we boot the desktop in safe mode and run only a minimum set of basic OS services. This ensures that the power measured by us can be attributed primarily to the application being tested. We measured the power drawn by an application using the Joulemeter tool [4].

Our second platform is a Mac note book with 4 Intel cores, each operating at a frequency of 2 GHz. The desktop has $4GB$ DDR3 memory with 8 banks and is running Mac OS X Lion 10.7.3. We repeated the copy and transpose experiments on this platform. We aborted most of the processes other than our programs while we executed it to ensure that the power measured by us can be attributed primarily to the application being tested. We measured the (total) power drawn using the Hardware Monitor tool [3].

6.3 Benchmark Algorithms

Our energy modeling experiment were driven using three benchmarks. Our first benchmark is Algorithm 1, which writes data into an integer vector along a specific access pattern. Our second benchmark (Algorithm 2) is inspired from the *dcopy* benchmark and copies a large integer vector into another. The third algorithm is a modified matrix transpose algorithm on an integer matrix.

Our goal in these experiments was to define an operating point with a fixed work compelxity and a fixed number of memory accesses. For this operating point, we would vary the allocation of data across the banks in such a way that the algorithm would have varying degrees of bank-parallelism (from 1 to 8). Since work complexity does not change due to data layout, the work complexity is preserved in our experiments by ensuring that we use identical code for all experiments. Having constant number of memory accesses across various data layouts is tricky. Further, memory controllers interleave memory across banks in a round robin fashion using a chunk size smaller than B called rs, complicating the problem further. In order to preserve memory complexity, we use a simple locality aware access pattern.

We elaborate with the example of vector copy. It is easy to see that the minimum number of I/Os needed to read a vector of size N is N/B. Ensuring that a 1-parallel access also uses N/B is trivial. We read one bank and ensure that we read one bank row (B bytes from the same row) before moving to another row in the bank. For a p-parallel access, we read data from the p banks in parallel in chunks of size r, where $r < B$ and is set to rs. To ensure that we do not need to make multiple memory accesses for multiple chunks in the same bank row, we maintain locality of access within one bank. All elements from an open row are read before any data is read from any other other row in the given bank. Elements across different banks do not preserve locality and can be interleaved by the algorithm. It is easy to note that this property ensures that the number of memory accesses are still N/B. The same idea easily extends to the write experiment. The matrix transpose algorithm is performed using multiple runs of the matrix. We again ensure this property for each run individually.

We next describe details of the individual algorithms.

```
Input: A large vector $\mathbf{V}$ of integers of length $N$,
       integers $G$, $C$, and $I$
$count = 0$;
while $count < G \times C$ do
    $i = 0, j = 0, k = 0$;
    for $i = 0 \to \frac{N}{G \times C} - 1$ do
        $j = count$;
        for $k = 0 \to C - 1$ do
            $V[j + k] = I$;
        end
        $j += G \times C$;
    end
    $count += C$;
end
```

Algorithm 1: Writing into a large vector of integers

In Algorithm 1 and 2 we assume C divides N, and G divides C. In Algorithm 1, we group the input vector $\mathbf{V}$ into $\frac{N}{G}$ groups each of size G. The access is then done in

```
Input: Two large vectors $\mathbf{V}_1$ and $\mathbf{V}_2$ of integers each
       of length $N$, integers $G$ and $C$
$count = 0$;
while $count < G \times C$ do
    $i = 0, j = 0, k = 0$;
    for $i = 0 \to \frac{N}{G \times C} - 1$ do
        $j = count$;
        for $k = 0 \to C - 1$ do
            $V_2[j + k] = V_1[j + k]$;
        end
        $j += G \times C$;
    end
    $count += C$;
end
```

Algorithm 2: Copying a large vector of integers into another

```
Input: An $N \times N$ matrix $\mathbf{M}$ of integers, integers $P$, $r$,
       and $s$
Output: Transposed matrix $(\phi_{r,P}(\mathbf{M}))^T$
Let $\mathbf{A} = \phi_{r,P}(\mathbf{M})$
Divide $\mathbf{A}$ into $\frac{N^2}{P^2}$ submatrices of size $P \times P$
foreach $P \times P$ submatrix do
    Transpose the submatrix naively in place;
end
\* At this time all the submatrices are
transposed *\
foreach $P \times P$ submatrix do
    Swap it with its corresponding submatrix within $\mathbf{A}$
    row-by-row;
end
```

Algorithm 3: Matrix transpose algorithm

the following way: At step i, we access the i^{th} chunk of size C of each group sequentially. The elements of the current chunk are accessed sequentially and set to the integer I. The idea behind the algorithm is to ensure that the access defines various degrees of parallelism over the banks of the memory. As an example, we have a 1-way parallel access if we set G to be the number of banks in the memory, and C to be the round robin chunk size rs, which is allocated per bank row at each allocation by the memory controller. In case the allocation is bank row wise across the banks, then C set to the bank row size (block size B), and G to the number of banks (P) gives us a 1-way parallel access. On the other hand setting $C = B$ and $G = 1$ leads to P-way parallel access. The work and I/O complexity of the algorithm does not change as we vary G, and C since we preserve the locality of access within each bank (elements within a bank are accessed sequentially ensuring that one row is written completely before we touch any other element on the same bank).

Algorithm 2 accesses two vectors defined in exactly the same way as in Algorithm 1. The difference here is we have two vectors on which this access pattern is defined, and data from the i^{th} element of $\mathbf{V}_1$ is copied into the i^{th} element of $\mathbf{V}_2$.

Algorithm 3 transposes an $N \times N$ matrix of integers. For this algorithm, we need to define a related map $\phi_{r,P}$ that maps an $N \times N$ matrix $\mathbf{M}$ into another matrix $\mathbf{A} \overset{\text{def}}{=} \phi_{r,P}(\mathbf{M})$ as follows. (The map ϕ depends on parameters P and r such that P divides r, and Pr divides N.) From $\mathbf{M}$, construct an $N \times N/r$ matrix $\mathbf{X}$ such that every r consecutive elements in any row in $\mathbf{M}$ constitute a single "mega"-element in the same row in $\mathbf{X}$. Now divide up $\mathbf{X}$ into $P \times P$ submatrices and transpose them in place to obtain the matrix $\mathbf{Y}$. (Note that by our choices of r, P and N, $\mathbf{Y}$ is also an $N \times N/r$ matrix.) The final matrix $\mathbf{A}$ is obtained from $\mathbf{Y}$ by simply "unpacking" each mega element in $\mathbf{Y}$ into r consecutive elements in the corresponding row in $\mathbf{A}$.

The central unit in this algorithm is a submatrix with P rows. The map vector ensures that this sub-matrix is hosted on p banks, where p can be varied from 1 to 8. The value of p dictates the degree of parallelism for the algorithm. We also process the sub-matrices in a row major fashion for the transpose step, which ensures the per-bank locality property required to preserve the number of memory accesses. For the swap step, the sub-matrices are picked in a row major fashion. This ensures locality property for the picked sub-matrices. The swap partners of the sub-matrices do not satisfy locality (only P elements out of B are read sequentially). However, P elements out of B are read across all the variants, leading to the same number of memory accesses for all the experiments.

6.4 Additional Results

6.4.1 Additional data for experiments on Windows

We present the variations in the energy readings for all the experiments of Figure 3 in Tables 1, 2, and 3.

Algorithm	Vector or Matrix size	Degree of Parallelism	Max	Min	Avg
Write	$256MB$	1-Way	69.0374	68.4705	68.6285
Write	$256MB$	2-Way	45.7844	44.3620	45.0138
Write	$256MB$	4-Way	37.2824	34.7816	36.6009
Write	$256MB$	8-Way	34.1156	33.1847	33.4776
Write	$128MB$	1-Way	34.8290	33.6623	34.0199
Write	$128MB$	2-Way	23.2194	22.2575	22.4373
Write	$128MB$	4-Way	19.1495	18.0503	18.2758
Write	$128MB$	8-Way	17.4870	16.0016	16.7338
Write	$64MB$	1-Way	17.3420	16.8986	17.0549
Write	$64MB$	2-Way	11.4637	11.1825	11.1737
Write	$64MB$	4-Way	9.6474	9.0376	9.1379
Write	$64MB$	8-Way	8.9030	7.9290	8.3259
Write	$32MB$	1-Way	8.5935	8.4816	8.5057
Write	$32MB$	2-Way	5.8870	5.5657	5.6086
Write	$32MB$	4-Way	4.8235	4.3250	4.5704
Write	$32MB$	8-Way	4.5240	4.0340	4.1615

Table 1: Maximum, minimum, and average energy consumption in Joules for the write algorithm on various input sizes, and various degree of parallelism on Windows.

6.4.2 *Results on Macintosh*

The copying and transpose experiments have been repeated on the Macintosh platform mentioned earlier in this section. We present the results in Figure 5.

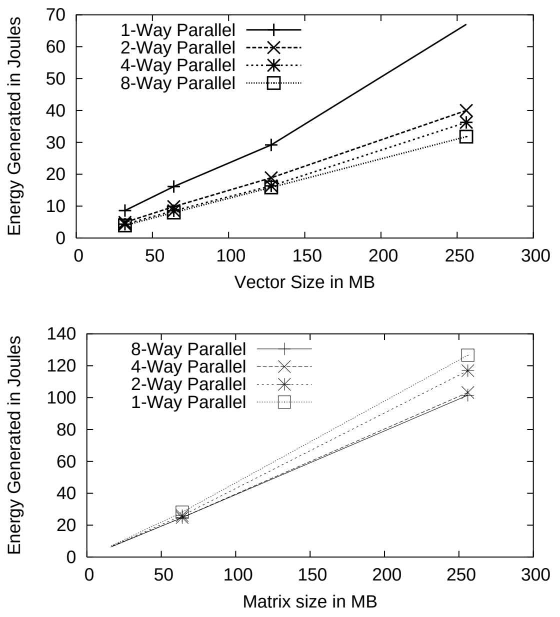

Figure 5: Energy consumed by (a) Copy and (b) Matrix Transpose Workloads on Macintosh

Acknowledgments

We thank an anonymous reviewers from an earlier conference who pointed out to us that the usual 2-way mergesort suffices to prove Corollary 5.2.

Algorithm	Vector or Matrix size	Degree of Parallelism	Max	Min	Avg
Integer copy	128MB	1-Way	58.9665	57.0488	57.5476
Integer copy	128MB	2-Way	41.3250	39.2921	40.2636
Integer copy	128MB	4-Way	36.8010	34.6318	35.7396
Integer copy	128MB	8-Way	34.5390	32.4379	33.5704
Integer copy	64MB	1-Way	29.2514	28.3649	28.681
Integer copy	64MB	2-Way	20.6575	19.6112	20.0854
Integer copy	64MB	4-Way	18.3955	17.3043	17.9162
Integer copy	64MB	8-Way	17.3420	16.2414	16.7388
Integer copy	32MB	1-Way	14.6255	14.2056	14.3898
Integer copy	32MB	2-Way	10.5264	9.8397	10.1326
Integer copy	32MB	4-Way	9.2025	8.6637	8.9552
Integer copy	32MB	8-Way	8.5935	8.0982	8.3288
Integer copy	16MB	1-Way	7.5400	7.1021	7.2384
Integer copy	16MB	2-Way	5.1234	4.9082	4.9764
Integer copy	16MB	4-Way	4.6785	4.4109	4.437
Integer copy	16MB	8-Way	4.3694	4.0716	4.2514

Table 2: Maximum, minimum, and average energy consumption in Joules for the copy algorithm on various input sizes, and various degree of parallelism on Windows

Algorithm	Vector or Matrix size	Degree of Parallelism	Max	Min	Avg
Transpose	256MB	1-Way	242.7729	235.7240	241.0828
Transpose	256MB	2-Way	222.9773	217.7890	222.8969
Transpose	256MB	4-Way	214.0016	209.0030	213.6227
Transpose	256MB	8-Way	203.2162	200.8830	202.0894
Transpose	64MB	1-Way	59.2514	58.3649	56.2803
Transpose	64MB	2-Way	53.3890	52.1630	53.0207
Transpose	64MB	4-Way	51.5910	51.5173	51.5272
Transpose	64MB	8-Way	48.8650	48.4010	48.8157
Transpose	16MB	1-Way	13.5400	13.1021	13.0929
Transpose	16MB	2-Way	12.9630	12.6730	12.8394
Transpose	16MB	4-Way	12.818	12.6730	12.6759
Transpose	16MB	8-Way	12.2380	11.7740	12.0524

Table 3: Maximum, minimum, and average energy consumption in Joules for the Transpose algorithm on various input sizes, and various degree of parallelism on Windows

7. REFERENCES

[1] Calculating memory system power for ddr3. http://download.micron.com/pdf/technotes/ddr3/TN41_01DDR3 Power.pdf.

[2] Data center energy consumption trends. http://www1.eere.energy.gov/femp/program/dc_-energy_consumption.html.

[3] Hardware monitor. http://www.bresink.de/osx/HardwareMonitor.html.

[4] Joulemeter: Computational energy measurement and optimization. http://research.microsoft.com/en-us/projects/joulemeter/.

[5] A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. In *Communications of the ACM. Volume 31, Number 9*, 1988.

[6] S. Albers. Energy-efficient algorithms. *Commun. ACM*, 53:86–96, May 2010.

[7] N. Bansal, T. Kimbrel, and K. Pruhs. Speed scaling to manage energy and temperature. *J. ACM*, 54(1), 2007.

[8] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4-5):601–631, 1997.

[9] F. Bellosa. The benefits of event-driven enery accounting in power-sensitive systems. In *Proc. SIGOPS European Workshop*, 2000.

[10] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the i/o-model. *Theory Comput. Syst.*, 47(4):934–962, 2010.

[11] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and

R. Doyle. Managing energy and server resources in hosting centers. In *Proc. ACM SOSP*, 2001.

[12] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–297, Washington, DC, USA, 1999. IEEE Computer Society.

[13] M. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *Proceedings of SPAA*, 2011.

[14] A. Gupta. Balls and bins and the power of 2 choices, Spring 2009. `http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15859m-s11/www/lectures/lect0202.pdf`.

[15] T. Horvath, T. Abdelzaher, K. Skadron, and X. Liu. Dynamic voltage scaling in multitier web servers with end-to-end delay control. *IEEE Trans. Comput.*, 56(4):444–458, 2007.

[16] S. Irani, G. Singh, S. K. Shukla, and R. K. Gupta. An overview of the competitive and adversarial approaches to designing dynamic power management strategies. *IEEE Trans. Very Large Scale Integr. Syst.*, 13:1349–1361, December 2005.

[17] K. S. Kedlaya and C. Umans. Fast polynomial factorization and modular composition. *SIAM Journal on Computing*, 40(6):1767–1802, 2011.

[18] W. Kim, M. S. Gupta, G.-Y. Wei, , and D. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *Proc. HPCA*, 2008.

[19] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms.* Addison-Wesley, 1969.

[20] R. Koller, A. Verma, and A. Neogi. Wattapp: An application aware power meter for shared data centers. In *ICAC*, 2010.

[21] V. A. Korthikanti and G. Agha. Towards optimizing energy costs of algorithms for shared memory architectures. In *Proc. SPAA*, 2010.

[22] V. A. Korthikanti, G. Agha, and M. Greenstreet. On the energy complexity of parallel algorithms. In *Proc. ICPP*, 2011.

[23] G. Kuperberg, S. Lovett, and R. Peled. Probabilistic existence of rigid combinatorial structures. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:144, 2011.

[24] K. Mehlhorn and P. Sanders. Scanning multiple sequences via cache memory. In *Algorithmica*, 2003.

[25] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results. In *in Handbook of Randomized Computing*, pages 255–312. Kluwer, 2000.

[26] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[27] R. Nathuji and K. Schwan. Virtualpower: coordinated power management in virtualized enterprise systems. In *Proc. ACM SOSP*, 2007.

[28] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level power management for dense blade servers. In *Proc. ISCA*, 2006.

[29] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *J. Comput. Syst. Sci.*, 69(3):330–353, 2004.

[30] A. Vahdat, A. Lebeck, and C. S. Ellis. Every joule is precious: the case for revisiting operating system design for energy efficiency. In *Proc. of ACM SIGOPS European workshop*, New York, NY, USA, 2000. ACM.

[31] A. Verma, G. Dasgupta, T. Nayak, P. De, and R. Kothari. Server workload analysis for power minimization using consolidation. In *Proc. Usenix ATC*, 2009.

[32] A. Verma, R. Koller, L. Useche, and R. Rangaswami. Srcmap: energy proportional storage using dynamic consolidation. In *Proc. of Usenix FAST*, 2010.

[33] J. S. Vitter. Algorithms and data structures for external memory, 2008.

[34] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, 1994.

[35] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2/3):148–169, 1994.

[36] F. F. Yao, A. J. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *FOCS*, pages 374–382, 1995.

[37] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *ASPLOS*, 2002.

APPENDIX

A. MORE ON THE ENERGY EQUATION

To recap, the equation for the energy consumed by any algorithm (Equation 8) is $E(\mathcal{A}) = W(\mathcal{A}) + ACT(\mathcal{A})\frac{PB}{k}$ where $W(\mathcal{A})$ and $ACT(\mathcal{A})$ are respectively the work and I/O complexities of algorithm $\mathcal{A}$, PB is the size of the cache, and k is the parallelization factor. Assuming k to be maximum (or constant) for $\mathcal{A}$, the following bounds for $W(\mathcal{A})$ and $ACT(\mathcal{A})$ hold:

$$W(\mathcal{A}) \leq 2^{O(PB)} \cdot ACT(\mathcal{A}) \tag{19}$$

$$ACT(\mathcal{A}) \leq W(\mathcal{A}) \tag{20}$$

Each of the bounds given in Equations 19, and 20 are tight. Now for any given problem $\mathcal{P}$, let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two algorithms. Further $\mathcal{A}_1$ and $\mathcal{A}_2$ are respectively work and I/O optimal algorithms for $\mathcal{P}$. One natural question is whether simulating both $\mathcal{A}_1$ and $\mathcal{A}_2$ in parallel gives us a constant factor approximation algorithm over the optimal energy consumed for $\mathcal{P}$. In other words if W_1 and W_2 and

ACT_1 and ACT_2 respectively be the work and I/O complexities of $\mathcal{A}_1$ and $\mathcal{A}_2$, is the following true? $min(W_1, W_2) + PB \cdot min(ACT_1, ACT_2) = \gamma \cdot E_{\mathcal{P}}$ where γ is a constant, and $E_{\mathcal{P}}$ is the energy consumed by an energy-optimal algorithm for $\mathcal{P}$.

We prove this to be in fact false. We show a problem for which we have two algorithms, one is work optimal, and the other I/O optimal. But simulating both in parallel does not give us a constant factor approximation algorithm for the total energy.

Problem ($\mathcal{P}$).

We have $L+1$ vectors and $y_1 = c+e$ where c is a code. The goal is to decode y_1 to c; $y_2, \cdots, y_{L+1}$ such that $\sum_i y_i = c$. Also length of each vector $n = O(PB)$.

Algorithm 1 ($\mathcal{A}_1$).

Use only y_1 and is an exponential time algorithm. So $W_1(\mathcal{P}) = 2^{O(PB)}$, and $ACT_1(\mathcal{P}) = 1$.

Algorithm 1 ($\mathcal{A}_2$).

Read all $L+1$ vectors, and compute $\sum_i y_i = c$. So $W_2(\mathcal{P}) == O(Ln)$, and $ACT_2(\mathcal{P}) = O(L)$.

Hence simulating $\mathcal{A}_1$ and $\mathcal{A}_2$ in parallel will not gives us a constant factor energy optimal algorithm since parameter L is not a constant.

B. REDUCING RANDOMNESS

Theorem B.1. *Let $1 \le \ell \le P/2$ be an integer and let $0 < \delta < 1$ be a constant. Let $\mathcal{A}$ be a striped (bounded resp.) algorithm with lookahead ℓ and uses space $S(x)$ in input x. Then there exists a (randomized) algorithm $\mathcal{A}'$ that uses $O(P \log S(x))$ bits of randomness with the same functional behavior as $\mathcal{A}$ (and uses three times as much space as $\mathcal{A}$ does) such that for every x, we can attain the same parameters for $\mathcal{A}'$ in Theorem 4.4 (Theorem 4.5 resp.) except*

$$W(\mathcal{A}', x) \le O(W(\mathcal{A}, x)) + \text{poly}\log P \log^{1+\delta} S(x) \cdot |\mathcal{I}(\mathcal{A}, x)| + O((P \log S(x))^{1+\delta}). \tag{21}$$

The proof of the theorem above is the same as that of Theorem 4.4 (Theorem 4.5 resp.) except the part where we compute the table that stores the random map of the memory location to their "fresh location." Thus, we will only concentrate on this aspect of the simulation– the rest of the simulation remains the same.

We first start with the observation that the the property required from the random map is the following: for any ℓ memory accesses we want that they are thrown randomly into the P banks. In other words, all we need is a map $f : [S(x)] \to [P]$ such that the map is P-wise independent. This ensures that the balls and bins analyses in both Sections C and D go through and one can prove that the resulting access pattern the required bound on τ. However, there are two issues that need to be resolved.

First, using the same arguments as in proofs of Theorem 4.4, we can assume an upper bound on m. In particular, here we will assume that

$$m \le 2^{O(B^{1-\delta}/P)}. \tag{22}$$

First issue is that since we are only mapping the memory locations to banks, we need to decide on the bank row within the bank. For this purpose we do the obvious thing: for each of the P banks, we keep a counter of how many memory locations have already been assigned to that bank, which can then be used to easily determine the bank row for a given memory location. Since we will need this information later on during the simulation, for any memory location b, we save both $f(b)$ as well as the bank row "counter." Because of condition (22), this implies that all the P counters can be stored in one cache.

Second, we need to figure out how to compute the function f. The most well-known construction of P-wise independent *bits* is one based on BCH codes. However, for our purpose since we need P-wise independent vectors over $[P]$, we use the other common construction based on Reed-Solomon codes. In what follows, for simplicity, let us assume that P is a power of 2. We pick q to be the smallest power of 2 larger than m and do an arbitrary assignment of $[m]$ to elements of the field $\mathbb{F}_q$. Define R to be a random univariate polynomial over $\mathbb{F}_q$ of degree at most $P-1$ (this is the only place we will need random bits and we need $P \log q = O(P \log m)$ many such bits as desired). Then $f(b)$ is defined to be the least $\log P$ significant bits of $R(b)$ (where we think of $b \in \mathbb{F}_q$). It is well-known that this map has the required P-wise independence. Finally, to compute the map for every memory location b, we compute $R(b)$. If we do this naively, we will need $O(P \log q)$ time to evaluate the polynomial R at each memory location b. It turns out we can do better. Kedlaya and Umans [17] recently presented a data structure that given the coefficients of R build a data structure in time (and hence space) $O((P \log q)^{1+\delta})$ (for any constant δ) such that given this data structure, $R(b)$ can be evaluated at any b in time poly $\log P(\log q)^{1+\delta}$. The bound in (22) implies that computing the data structure from the coefficients of $R(X)$ can be done within one cache itself. The bounds on the construction and query time for the polynomial evaluation data structure then implies (21).

C. THE FIRST BALLS AND BINS PROBLEM

In this section, we will consider the following balls and bins framework:

> n balls are thrown into P bins (numbered from $0, \ldots, P-1$) as follows. For a parameter $g \ge 1$, divide up the n balls into $n/(gP)$ groups of gP balls each. For each group, randomly permute the gP balls in place. Then for a ball in location $0 \le i \le n-1$, place it in bin $i \mod p$.

Bounding the Maximum Load..

We will prove the following bound on the maximum load on a bin (where we consider any arbitrary P balls):

Theorem C.1. *Consider the random process above with $g \geq 2$. Now consider any arbitrary subset $S \subseteq [n]$ with $|S| = P$. Then with probability at least $1 - 1/P$, every bin has at most $O(\log P/\log\log P)$ balls from S in it.*

Proof. Let ℓ be a parameter that we will fix later (to be $O(\log P/\log\log P)$). Fix an arbitrary bin $\mathcal{B}$. We will show that

$$\Pr[\mathcal{B} \text{ has at least } \ell \text{ balls from } S] \leq \frac{1}{P^2}. \tag{23}$$

Note that (23) along with the union bound proves the claimed bound.

Fix an arbitrary subset of ℓ balls from S. Further, let b be one of these ℓ balls. We claim that

$$\Pr[b \text{ lands in } \mathcal{B} | S \setminus \{b\}] \leq \frac{g}{gP - P} = \frac{1}{P} \cdot \frac{1}{1 - 1/g} \leq \frac{2}{P}, \tag{24}$$

where we use $S \setminus \{b\}$ as shorthand for where the balls in $S \setminus \{b\}$ land and the last inequality follows from the assumption that $g \geq 2$.

First, we use the argument for the standard balls and bins problem to use (24) to prove (23). Indeed, (24) along with the union bound over all $\binom{P}{\ell}$ choices for the ℓ balls implies that

$$\Pr[\mathcal{B} \text{ has at least } \ell \text{ balls from } S] \leq \binom{P}{\ell} \cdot \left(\frac{2}{P}\right)^{\ell} \leq \left(\frac{2e}{\ell}\right)^{\ell} \leq \frac{1}{P^2},$$

where in the above the second inequality follows by the bound $\binom{P}{\ell} \leq (eP/\ell)^{\ell}$ and the last inequality follows by choosing $\ell \leq O(\log P/\log\log P)$.

To complete the proof, we argue the first inequality in (24). Since the randomness used in different groups are independent, we can replace the conditional event by $S' \setminus \{b\}$, where $S' \subseteq S$ are the balls that lie in the same group as b. Note that in the random process, if b permuted to a location that is $\mathcal{B} \mod P$, then it lands in $\mathcal{B}$. In other words, within the random permutation of the group of b, this happens if it happens to land in one of the g locations that correspond to $\mathcal{B} \mod P$. As a mental exercise, think of the random permutation being done as follows: each of the at most $P - 1$ balls in $S' \setminus \{b\}$ are thrown one after the other into (distinct) random bins. Let $g - g'$ of the special bins be occupied by some ball in $S' \setminus \{b\}$. Now when b is about to be thrown, there are at least $gP - P + 1$ bins to choose randomly from and we're interested in the event that it lands in one of the g' special bins that are vacant. Thus, this probability is upper bounded by

$$\frac{g'}{gP - P + 1} \leq \frac{g}{gP - P},$$

as desired. $\square$

D. THE SECOND BALLS AND BINS PROBLEM

In this section, we will consider the following balls and bins framework:

n balls are thrown into P bins (numbered from $0, \ldots, P - 1$) as follows. For a parameter $g > 0$, divide up the n balls into $n/(gP)$ groups of gP balls each. For each group, randomly permute the gP balls twice. (The randomness is independent across groups and across the two permutations in each group.) For each ball $i \in [n]$, denote its two positions in the two permutations as i_1 and i_2.

Let $S \subseteq [n]$ be of size $|S| = \frac{P}{512}$. Consider a graph $G = ([P], E)$ as follows. For each ball $i \in S$, add the edge $(i_1 \mod P{+}1, i_2 \mod P{+}1)$ to E. Now we assign each ball in S to a bin in $[P]$ as follows. Let $K \geq 3$ be a parameter. Consider the following iterative algorithm:

```
1. $G' \leftarrow G$.
2. While $|V(G')| > K$ do
   (a) Let $T \subseteq V(G')$ be the set of all vertices with degree at most 5 in $G'$.
   (b) Assign to bins in $T$, all its incident edges (or balls). If an edge is completely contained inside $S$, then assign it one of its end point arbitrarily.
   (c) $V \leftarrow V(G') \setminus T$.
   (d) $G'$ is the induced sub-graph on $V$.
3. Assign edges in $G'$ arbitrarily to the bins in $V(G')$.
```

Before we proceed, we record a lemma on the time complexity of the algorithm above that will be useful later. The most obvious way to implement the above algorithm will be to use a priority queue to keep track of the degrees of the degree of vertices in G'. Using the best known implementation of priority queues (e.g. [29]) the above can be implemented in $O(P \log\log P)$ time. However, for our application we need $O(P)$ time and space. We are able to do so below by noting that we only care about vertices with degree at most 5 and not vertices with the minimum degree. Also the fact that the degrees are at most $P - 1$ also helps.

Lemma D.1. *The algorithm above can be implemented in $O(P)$ time and $O(P)$ space.*

Proof. We will assume that the original graph G is given to us in adjacency list format. In particular, given a vertex $v \in [P]$, we can iterate over all its neighbors in $O(d_v)$ time, where d_v is the degree of v. We will have an array of doubly linked list called DEG, where for any $0 \leq d \leq P - 1$, $\mathsf{DEG}[d]$ contains the list of all vertices with degree exactly d in G'.[5] Finally, we have an array of pointers PTR, such that for every vertex $v \in [P]$, $\mathsf{PTR}[v]$ points to actual location of v in the corresponding linked list in DEG (as well as keeping track of its current degree d_v). We will use the convention that $\mathsf{PTR}[v]$ is null if $v \notin V(G')$. It is easy to check that all these data structures use $O(P)$ space and can be initialized in $O(P)$ time. (Recall that G has $O(P)$ edges.) Next we

[5]It turns out that we only need one entry for all degrees between 0 and 5 but we ignore this improvement as it only changes the constants.

show we can implement the above algorithm in $O(P)$ time using these three data structures.

We will maintain a counter for $|V(G')|$ so the check in Step 2 can be computed in $O(1)$ time. Step 2(a) can be implemented in $O(|T|)$ time by just returning the linked lists $\mathsf{DEG}[j]$ for $0 \leq j \leq 5$. For Step 2(b), we do the following. For each $v \in T$, we go through all its neighbors $u \in [P]$. We can check if $u \in V(G')$ by checking if $\mathsf{PTR}[u]$ is null or not. If not, we assign the ball (v, u) to the bin v. Then using $\mathsf{PTR}[u]$, we remove u from its current linked list, decrement d_u by one, add u to the head to the linked list for (the updated value of) d_u and change $\mathsf{PTR}[u]$ to this new location. Since the linked lists are doubly linked, note that all this can be implemented in $O(1)$ time. To implement Step 2(c), for every $v \in T$, we set $\mathsf{PTR}[v]$ to null.

It can be checked that the above implementation is correct and does $O(1)$ work per edge and vertex, which leads to the claimed $O(P)$ overall run time, as desired. □

Bounding the Maximum Load..

We will prove the following bound on the maximum load on a bin.

THEOREM D.2. *Consider the random process above with $g \geq 1/4, K \geq 3$ and any arbitrary subset $S \subseteq [n]$ with $|S| = \frac{P}{512}$. Then with probability at least $1 - 1/P$, every bin has at most $3 \cdot \max(K, 5)$ balls from S in it.*

It is not a priori clear that the algorithm in the random process above even terminates. We first state a lemma that will help us guarantee that and also help us prove Theorem D.2.

The following lemma has been proven in the case when $n = P$, $g = 1$ and the balls are assigned two bins at random (instead of using a random permutation.) See e.g. [14]. Here we present a straightforward generalization of that proof to our problem.

LEMMA D.3. *Consider the graph G generated by S in the random process above. Then with probability at least $1 - \frac{1}{P^4}$, the following holds for every $U \subseteq V(G)$ with $|U| \geq K$. The average degree of the induced subgraph $G[U]$ is at most 5.*

We defer the proof of the above lemma for a bit and first prove Theorem D.2. Before that we prove two helpful lemmas.

LEMMA D.4. *Consider a fixed pair $(b_1, b_2) \in [P]^2$ with $b_1 \neq b_2$ and an arbitrary ball $i \in S$. The probability that the edge corresponding to i ends up at (b_1, b_2) is upper bounded by $\frac{20}{P^2}$ even when* conditioned *on where the remaining balls in $S \setminus \{i\}$ end up.*

PROOF. Assume that the choices for all balls in $S \setminus \{i\}$ have been made. Consider an arbitrary $j \in S \setminus \{i\}$. If j is in a different group than i (recall that $[n]$ is divided into groups of size gP), then the choice of edges for i and j are independent, which in turn implies that the location of such a j does not affect the probability that we want to bound. For the rest of the argument, we will only consider balls j that are in the same group as i.

For notational convenience define, $\bar{g} = \max(1, \lceil g \rceil)$. Now consider the case when $g_1 \leq \bar{g}$ (and $g_2 \leq \bar{g}$ resp.) balls j from the same group as i have $j_1 = b_1$ ($j_2 = b_2$ resp.). We consider the probability that $i_1 = b_1$ conditioned on the event above. Note that $i_1 = b_1$ can happen only if i happens to be one of the $\bar{g} - g_1$ locations in $[gP]$ that will lead to $i_1 = b_1$. Further, since everything in $S \setminus \{i\}$ has been assigned, there are $gP - |S| - 1$ locations in $[gP]$ that the first choice for i can land in. Thus, the probability that $i_1 = b_1$ (even conditioned on what happened with the balls in $S \setminus \{i\}$ is

$$\frac{\bar{g} - g_1}{gP - \frac{P}{512} - 1} \leq \frac{\bar{g}}{gP - \frac{2P}{512}} \leq \frac{1}{\frac{1}{4} - \frac{1}{256}} \cdot \frac{1}{P} \leq \frac{5}{P}.$$

One can make a similar argument to get the same bound as above for $i_2 = b_2$. Since the first and second choices for i are independent, we have that

$$\Pr[(i_1, i_2) = (b_1, b_2) | S \setminus \{i\}] \leq \frac{10}{P^2}.$$

Note that we also have to take into account the case that $(i_1, i_2) = (b_2, b_1)$, which again happens with probability at most $10/P^2$. The union bound completes the proof. □

LEMMA D.5. *With probability at least $1 - \frac{1}{2P}$, the graph G has at most 3 parallel edges for any vertex pair (b_1, b_2).*

PROOF. We will show that the probability that more than 3 balls $i \in S$ are assigned to a fixed bin pair $(b_1, b_2$ is at most $\frac{1}{2P^3}$. Taking union bound over the at most P^2 pairs completes the proof.

Now consider a fixed pair $(b_1, b_2) \in [P]^2$. Consider a fixed subset $T \subseteq S$ of size $|T| = 3$. By Lemma D.4, the probability that all balls in T are assigned (b_1, b_2) is at most $(20/P^2)^3$. Then taking the union bound over all possible choices of T, we get that the probability bound that we are after is upper bounded by

$$\binom{\frac{P}{512}}{3} \cdot \frac{20^3}{P^6} \leq \left(\frac{P}{512} \cdot \frac{20}{P^2}\right)^3 < \frac{1}{2P^3},$$

as desired. □

PROOF PROOF OF THEOREM D.2. Note that except for probability at most $\frac{1}{P^4} + \frac{1}{2P} < \frac{1}{P}$ (for $P \geq 2$), both Lemmas D.3 and D.5 are true. For the rest of proof, we will assume that this is indeed the case.

We will assume that each vertex pair in G has at most one edge. (By Lemma D.5, then whatever bound we prove can be safely multiplied by 3 to arrive at our final bound.)

W.l.o.g., we can assume that G has only one connected component. (If not, we can apply the subsequent argument to each component individually.)

Let $U \subseteq V(G)$ be the set of vertices (or bins) which form the connected component of G. If $|U| \leq K$, then by Step 3 of the assignment algorithm, we have that the maximum load in any bin in U is at most K (recall that because of Lemma D.5, we have assumed that each vertex pair has at most one edge between it). If $|U| > K$, then Lemma D.3 (and a Markov argument) implies that there is a bin $i \in [P]$ that has degree at most 5. Thus, Step 2(b) assigns a load of at most 5 to all such bins. We can argue the case for G' inductively. (Note since we remove assigned bins in Step 2(c), in all future iteration, this bin will not affect any of the future loads.) In this whole process we achieve a maximum load of $\max(K, 5)$. Lemma D.5 completes the proof. $\square$

We finish by proving Lemma D.3.

Proof Proof of Lemma D.3. For notational convenience define $m = \frac{P}{512}$. Note that the proof is complete if we show that $G[U]$ has at most $5|U|/2$ edges.

Fix $k \geq K$ and let U be an arbitrary subset of size k. Then Lemma D.4 and the union bound (over the at most k^2 pairs in $U \times U$) shows that a fixed ball i lands in $G[U]$ is upper bounded by $\frac{20k^2}{P^2}$ (even conditioned on what happens with the rest of the balls). Then Lemma D.4 implies that the probability that any fixed subset T of $5k/2$ balls land in $G[U]$ is upper bounded by $\left(\frac{20k^2}{P^2}\right)^{5k/2}$. Then a union bound over all choices of T, implies that

$$\Pr[G[U] \text{ has } > 5k/2 \text{ edges}] \leq \binom{m}{\frac{5k}{2}} \cdot \left(\frac{5k}{P}\right)^{5k}.$$

The rest of the argument is the same as in [14] (with some minor changes in the values of the constants). $\square$

E. MATRIX TRANSPOSE

We first present the standard I/O model algorithm for computing the transpose of a matrix, which has I/O complexity of $O(N/B \cdot \log_P(B/P))$ (and work complexity of $O(N \log_P(B/P))$. Then we will show how to modify this algorithm to also make it $(P/2, 1)$-parallelizable in a *deterministic* way.

The algorithm has three phases (where A is the $\sqrt{N} \times \sqrt{N}$ matrix, which is assumed to be stored in row major form in consecutive rows in the main memory). At the end of the first phase, all the $P \times P$ submatrices are correctly transposed. At the end of the second phase all the $B \times B$ submatrices are correctly transposed. The whole matrix is transposed after the end of the third phase. In particular,

- (Phase 1) Divide up A in N/P^2 submatrices of size $P \times P$, bring them into the cache and transpose them in place.
- (Phase 2) There are $\log_P(B/P) - 1$ many iterations. In the $2 \leq i \leq \log_P(B/P)$th such iteration, we correctly transpose the $P^i \times P^i$ submatrices within the $B \times B$ submatrices by swapping the $P^{i-1} \times P^{i-1}$ submatrices (within the same $P^i \times P^i$ submatrix) that are counterparts. Two $P^{i-1} \times P^{i-1}$ submatrices D and E are counterparts if they should be swapped to make the resulting $P^i \times P^i$ submatrix correctly transposed.
- (Phase 3) Swap the $B \times B$ submatrices with their counterparts (which are defined analogously to the counterparts from Phase 2).

It is not too hard to verify that the algorithm above is correct. We now quickly argue the claimed I/O and work complexities of the algorithm above. In particular, we will argue that Phase 1, Phase 3 and each iteration in Phase 2 takes $O(N/B)$ I/Os and $O(N)$ work, which immediately results in the claimed I/O and work complexity bounds.

- (Phase 1) Load the $P \times B$ submatrices of A into the P caches and transpose the B/P submatrices of size $P \times P$ in place and write them back. It is easy to see that overall this phase takes $O(N/B)$ I/Os and has a work complexity of $O(N)$.
- (ith iteration in Phase 2) Consider an arbitrary $B \times B$ submatrix of A. Now consider an arbitrary $P^i \times B$ submatrix (call it D) of this sub-matrix. Note that this has B/P^i many $P^i \times P^i$ submatrices within it. We arbitrarily fix such a $P^i \times P^i$ matrix (let us call it E) and state our operations. (One can easily verify that one can perform the same operations in parallel on the rest of the $P^i \times P^i$ submatrices of D.) Recall that we have to swap all the $P^{i-1} \times P^{i-1}$ submatrices in E with their counterparts. Divide up E into P submatrices (call them $F_1, \ldots, F_p$) of size $P^{i-1} \times P^i$. Note that when we swap the counterparts of size $P^{i-1} \times P^{i-1}$, we can swap them row by row. Further, for any fixed $j \in [P^{i-1}]$, note that all the jth rows of the $P^{i-1} \times P^{i-1}$ submatrices are themselves exactly contained in the jth rows of $F_1, \ldots, F_p$. Thus, we can implement this step, if we at the same time load the jth rows of $f_1, \ldots, F_p$ for every $j \in [P^{i-1}]$.

  It can be verified that the above can be implemented by reading and writing all the N/B blocks in A once and doing linear amount of work on them. Thus, this step needs $O(N/B)$ many I/Os and $O(N)$ work.
- (Phase 3) Consider all the pairs of $B \times B$ submatrices of A that are counterparts. Note that these pairs can be swapped by swapping the corresponding rows. Thus, this step can also be done with $O(N/B)$ many I/Os and $O(N)$ work.

We note that the algorithm as stated above does not have the required $(P/2, 1)$-parallelizability but we will show shortly that a few modification suffice. In particular, we will argue that for each iterations/phases a simple *deterministic* cyclic shift of each row in the main memory will allow for the above algorithm to have the required parallelizability. For simplicity, we state the permutation for each of Phase 1, Phase 3 and each iteration in Phase 2 separately. In the actual implementation of the algorithm, before each phase/iteration we perform the cyclic shifts and whenever we need to access a block we access the block from its new shifted position and once the iteration/phase has terminated, we undo the cyclic shift. These shifts are data independent and thus, do not need to be stored anywhere (and can be just computed from N, B, P and i). Thus this extra step for each phase/iteration will take $O(N/B)$ many I/Os and $O(N)$ work so asymptotically the claimed bounds on I/O and work complexity still hold (but now with a guaranteed $(P/2, 1)$-parallelizability). Next, we spell out the shifts:

- (PHASE 1) Consider any of the $P \times B$ submatrix that we consider. Note that if all the rows of this matrix are in different banks then the whole access will be $(P, 1)$-parallelizable. Thus, for this phase all we need to do is to make sure that this happens. Note that this can be easily computed given the location of the $P \times B$ submatrix within A and the different parameter values. (Further, these shifts for all the $P \times B$ submatrices can be applied in a consistent manner.)
- (ith iteration in PHASE 2) It can be verified that in this iteration, we need to swap corresponding rows between two $P^{i-1} \times B$ submatrices (let us call them X and Y). We note that if we can make sure that $P/2 \times B$ submatrices have all the $P/2$ rows in distinct banks, then all the P^{i-1} row swaps can be performed in a $(P/2, 1)$-parallelizable way. Again note that the shifts for X are completely determined by the location of X within A and other parameters. (Further, these shifts for the different $P^{i-1} \times B$ submatrices can be applied in a consistent manner.)
- (PHASE 3) This step is similar to that of PHASE 2. We must perform the shifts so that we can ensure that the rows of the $P/2 \times B$ submatrices are in distinct banks. Further, as before the shifts can be consistently applied and can be computed from the location of such submatrices in A and the parameters.

F. MERGE SORT

It is easy to see that the k-way merge sort is a k-striped algorithm. However, since the algorithm is data dependent, the "traditional" merge sort algorithm does not good lookahead. Next we see how to remedy this.

Achieving a $\Theta(P)$ lookahead..

We will define a prediction sequence [8] that will allow the k-way merge sort to have lookahead of k. In particular, note that as long as we can show to successfully define a prediction sequence for the operation of merging k runs, then we would be done. We do so next.

We begin with a few definitions. We start with a definition to handle the case when in the middle of the merge algorithm, we have only have the partial "unmerged" part of the first block in the run.

DEFINITION 6. *A* merge run *is a usual run except the first block might have fewer than B items. Further, we call such a merge run* sorted *if all the elements when written down in the order of appearance in the run are in sorted order.*

For the rest of the section, we will establish the convention that if two items have the same value, then we break ties in the favor of the merged run with the smaller index. If both items are in the same merged run, then we break ties in favor of the item appearing earlier in the run. With this convention, w.l.o.g., we will assume that all the items in the runs have *distinct* values.

Next, we define a compressed representation of a block that will be crucial in our result.

DEFINITION 7. *Let block $\mathcal{B}$ in a sorted merge run be preceded by the block $(i_1, \ldots, i_m)$ (for some $m \leq B$). Then we define the* label *of $\mathcal{B}$ to be i_m*

The following lemma is the crucial observation (and is well-known [8]):

LEMMA F.1. *Consider k sorted merge runs. Let T be blocks in the k merged runs where we drop the first block in each run. Let be the blocks in T with the $3k$ smallest labels. Then the smallest kB elements in T are contained within the blocks in . Further, all the blocks in that belong to the same run form a prefix of the run.*

PROOF. For national convenience let S denote the kB smallest elements in T. We first observe that S contained in at most $2k$ blocks. Note that there can be at most k blocks such that all of its B elements are in S. Further, since each of the merged runs are sorted, in each run only the last block can contribute less than B elements to S. Since there are k merged runs, this can contribute k more blocks, which leads to a total of at most $2k$ blocks that contain S.

Next we argue that if for any block $\mathcal{B}$ such that $\mathcal{B} \cap S \neq \emptyset$, then $\mathcal{B} \in$. For the sake of contradiction, assume that $\mathcal{B} \notin$. Note that for this to happen $\mathcal{B}$ has to be "displaced" by another block $\mathcal{B}' \in$ such that $\mathcal{B}' \cap S = \emptyset$. (Note that this has to happen as S is contained within $2k$ blocks and $|_e| = 3k$.) Now note that any block $\mathcal{B}''$ that comes after $\mathcal{B}'$ in the same run as $\mathcal{B}'$ cannot be in . Indeed, all elements in the block just before $\mathcal{B}$ are in S while none of the elements in $\mathcal{B}'$ are in S, which implies that the label of $\mathcal{B}$ has to be smaller than $\mathcal{B}''$. Thus, since $\mathcal{B}' \notin$, $\mathcal{B}'' \notin$. In particular, each merged run can have at most one block that can potentially displace some block $\mathcal{B}$ (such that $\mathcal{B} \cap S \neq \emptyset$). Thus, the total number of blocks that can potentially displace a legitimate block is at most $k - 1$. However, $|| = 3k$, which implies that the remaining $2k + 1$ blocks have to be filled in by blocks $\mathcal{B}$ such that $\mathcal{B} \cap S \neq \emptyset$ (recall we have argued that there are at most $2k$ such blocks).

The claim on all block in for one run forming a prefix, note that a "gap" implies that a block with a larger label was included in at the "expense" of a block with a smaller label, which is not possible. The proof is complete. □

LEMMA F.2. *Let $2 \leq k \leq P/6$. Given k runs each of which contain numbers in sorted order such that there are a total of N numbers among all the runs. Then there is an algorithm $\mathcal{A}$ that merges the k runs into one sorted run such that for any input k sorted runs (denoted by x):*

- *$\mathcal{A}$ is a k-striped algorithm with a lookahead of k.*
- *For any input x, $|\mathcal{I}(\mathcal{A}, x)| \leq O(N/B)$.*
- *For any input x, $W(\mathcal{A}, x) \leq O(N)$.*

PROOF. The algorithm $\mathcal{A}$ is the well known algorithm to merge k sorted lists (which is easy to see is k-striped) dovetailed with another algorithm that builds a prediction sequence (using Lemma F.1) that gives the final algorithm a lookahead of k. (It is not too hard to see that the "traditional" merge algorithm, which is data dependent, does not have any non-trivial lookahead in-built into it.)

We first assume that the input is present in the first $P/2$ banks: if not one can perform a scan of the input and transfer it to the first $P/2$ banks. It is easy to see that this transfer can be done in $O(N)$ time with $O(N/B)$ memory accesses and that this part has a lookahead of $P/2$ and is a 1-striped algorithm. We now present the algorithm in two steps. The first step will store the labels of all the blocks in the input, which would be used in the second step to do the actual merging.

We will use the last $P/6$ banks for enabling lookahead for $\mathcal{A}$. The remaining $P/3$ banks in the "middle" will be used to write out the merged output.

Pre-processing..

In this step, the algorithm makes a scan on the input (which recall is now in the first $P/2$ banks) and except for the first block in each of the k runs, stores the labels of all the blocks (in the same order as they appear in the input) in the last $P/6$ banks. It is easy to see that this phase needs $O(N/B)$ memory accesses. Further, this step is a k-striped algorithm with lookahead of $P/2 \geq k$.

Merging the runs..

At the beginning of this step we copy the head block of each run to k blocks in the middle $P/3$ caches.

Throughout this step, we will maintain the following invariants:

1. The first $P/2$ caches will contain $3k$ blocks from the input.
2. The last $P/6$ caches will maintain a table of size $k \times 6k$ of labels that will have enough information to predict the next $3k$ blocks that will be transferred to the first $P/2$ caches.
3. The middle $P/3$ caches will keep the "unused" part of the first block from each of the k runs as well function as a buffer for the next k blocks that need to be written to the output.

As mentioned above, we will maintain a table with one row for each of the k runs. Each row will always have at least $3k$ labels for the blocks from the corresponding run. (We initialize this table with the labels of the first $6k$ blocks in each run.)

This part of the algorithm will be divided into repetitions of a "reading" phase followed by "writing" phase. The writing phase is simple: once the buffer in the middle $P/3$ caches has accumulated the next k blocks in the sorted order, we write back those k blocks to the output.

The reading phase starts once a writing phase is over. Using the table stored in the last $P/6$ caches, we compute in $O(k)$ time, the top $3k$ blocks according to their labels. We then bring those $3k$ blocks from the first $P/2$ banks to the corresponding caches. Then we run the "traditional" merge algorithm on the k runs formed by the partial blocks stored in the middle $P/3$ caches followed by the k runs formed by these top $3k$ blocks (note that by Lemma F.1 these blocks form k sorted runs among themselves). We run the traditional merge algorithm till we have filled up the buffer for the next k blocks to be written to the output. Next, update the "head" blocks in each run in the middle $P/3$ caches if necessary. In the table in the last $P/6$ caches, for each run, we throw away the labels for the blocks that have been used up and been written to the output. If need be, we replenish the rows that have fewer than $3k$ entries and make sure we have $6k$ entries for the rows that we replenish.[6] At this point, the writing phase is triggered.

We first note that by Lemma F.1 that the above algorithm works: i.e., all the blocks that are needed to form the merged k output blocks for the writing phase are indeed accessed in the reading phase. Second, for every writing phase (which writes back k blocks), we access $3k$ blocks from the first $P/2$ banks. Further, we can in worst case access $6k^2$ labels for replenishing the table. Since each label is just one item, these are contained within

$$\frac{6k^2}{B} \leq \frac{6k^2}{P} \leq k$$

blocks (where the first inequality follows from (9) while the second inequality follows from the assumption that $k \leq P/6$). All of this implies that over all we access a total of $O(N/B)$ accesses for this step of the algorithm. By design, this step of $\mathcal{A}$ has a lookahead of k. Finally, it is easy to verify that this step is also k-striped.

Since both steps of $\mathcal{A}$ are k-striped and have lookahead of k, then so does $\mathcal{A}$, which proves the first two claimed properties. The final property follows by observing that both the steps have linear time complexity. $\square$

[6]We note that the above is wasteful in the sense that some blocks in the first $P/2$ banks might be accessed more than once but we keep the above description as it makes stating the algorithm slightly simpler.