Investigating power efficiency of mergesort

Naif Aljabri¹ · Muhammad Al-Hashimi¹ · Mostafa Saleh¹ · Osama Abulnaja¹

Published online: 11 April 2019
© The Author(s) 2019

Abstract

Excessive power consumption emerged as a major obstacle to achieving exascale performance in next-generation supercomputers, creating a need to explore new ways to reduce those requirements. In this study, we present a comprehensive empirical investigation of a power advantage anticipated in the mergesort method based on identifying a feature expected to be physically power efficient. We use a high-performance quicksort as a realistic baseline to compare. Results show a generic mergesort to have a distinct advantage over an optimized quicksort lending support to our expectation. They also help develop some insights toward power efficiency gains likely meaningful in a future exascale context where trading some of the abundant performance for much needed power savings in a ubiquitous computation may prove interesting.

Keywords Exascale computing · Mergesort · Barrel shifter · Power consumption

1 Introduction

Recent discoveries in science and technology were driven by a huge demand for computational capacity. Currently, some simulations are impossible using existing computers. In response to this demand, exascale supercomputers are needed to reach sustained performance in the order of exaops [9]. The first exascale supercomputer was expected to be available in approximately 2018–2020

✉ Naif Aljabri
naifkau@gmail.com; naljabry0001@stu.kau.edu.sa

Muhammad Al-Hashimi
mhashimi@kau.edu.sa

Mostafa Saleh
msherbini@kau.edu.sa

Osama Abulnaja
abulnaja@kau.edu.sa

¹ Faculty of Computing and Information Technology, King Abdulaziz University (KAU), Jeddah, Saudi Arabia

timeframe according to earlier estimates [23]. Recently, some experts say it will not be available before 2023 [24]. Excessive electric power consumption seems to be the primary obstacle for further scaling of high-performance computing (HPC) to achieve exascale level [39].

The most powerful supercomputer in the world (Sunway TaihuLight in China) consumes about 15 Megawatts of power [38]. Scaling this machine 100 times to achieve exascale performance would require about 1.5 Gigawatt of power. Providing this amount of power requires a good-sized nuclear power plant next-door [29]. The US Defense Advanced Research Projects Agency (DARPA) [11, 12] suggests a more reasonable peak of electrical power; they propose 20 Megawatts as the maximum power consumption for the exascale systems. Therefore, to operate the exascale systems within reasonable power consumption levels, tremendous efforts from both software and hardware researchers are needed to tackle this issue. New software techniques and hardware technologies are needed to reduce the high requirements of power to a more manageable and economically feasible level [35].

Software developers look for faster algorithms to increase their application speed and response time. With the rapid evolution of HPC, power consumption is becoming one of the foremost challenges to scaling these powerful machines. Prioritizing power-efficient code building blocks could save power if execution time penalty is acceptable. A minor power advantage can result in massive savings if accumulated suitably.

Zecena [42] estimated that 85% of HPC scientific applications in the world depend exclusively on sorting algorithms. Therefore, perhaps a significant amount of power can be saved by identifying power characteristics associated with these algorithms, including any power related advantages.

In this paper, we resolve the limitations of our previous work [4] and present a new comprehensive energy study of mergesort focusing on power characteristics. We resolve the drawbacks in the previous work by using a modern CPU, new reliable profiler, new environment configuration, and in-depth investigation of power consumption behavior of mergesort. Mergesort relies on repeatedly dividing lists in half. In modern CPUs, division by two is performed by Barrel shifter circuit that can shift a data word by a specified number of bits in one clock cycle [26]. In theory, it seems that we should observe some power advantage when compared to a sorting method that needs to work harder to make similar decisions in comparable time. Computing on the exascale is an example of a situation where that advantage is valuable. Another perhaps is mobile computing where a bigger battery could address the energy budget issue but does not help if the battery is drained too fast. We note however while mobile may share a power concern with exascale computing, it must deal with different hardware, workload, and economic considerations. A solution, therefore, may not offer comparable value in both situations.

The rest of this paper is organized as follows. In Sect. 2, we review relevant previous work. In Sect. 3, we develop the rationale motivating this work. In Sect. 4, we present the experimental setup and methodology. Results are

thoroughly discussed in Sect. 5. Finally, we sum up and identify interesting future investigations in Sect. 6.

2 Related work

Much of the literature on [6, 16, 34, 40, 41] focuses on energy consumption. Yuechuan et al. [41] analyzed and investigated energy consumption of bubble sort algorithm. They predicted the energy consumption for each atomic operation, and then, they validated their prediction using energy prediction algorithm. The prediction accuracy was more than 95%.

Poon et al. [34] used mesh connected computer with an optical pyramid layout to introduce a power-aware sorting algorithm. They reduced the energy usage by minimizing the time and total data movement.

Yildiz et al. [40] estimated the energy consumptions of different *I/O* management approaches. They suggest a model that selects the best *I/O* approach in terms of energy efficiency regardless of the systems architecture or application-specific parameters.

Cebrian et al. [16] used the micro-architectural techniques to match a power constraint while reducing the energy consumption of the processor. They use tokens to translate the dissipated CPU power at cycle level and basic block level to select between different power-saving micro-architectural techniques. The approach was up to 11% more energy efficient.

Hasib et al. [6] analyzed the energy efficiency effects of a motion estimation algorithm with applying data-reuse transformations to exploit temporal locality on a multicore processor. Their approach improves the energy efficiency up to 5.5 times.

Lastovetsky et al. [31] proposed novel model-based methods and algorithms to minimize the computations time and energy for data parallel applications executing on homogeneous multicore clusters (Intel E5-2670 Haswell Server). They also demonstrate significant average and maximum percentage improvements in performance and energy for the two applications compared to traditional load-balanced workload distribution. The result shows that the optimizing for performance alone can lead to a significant reduction in energy and the optimizing for energy alone can cause major degradation in performance.

The literature on [13–15] pays particular attention to mobile device energy consumption. They suggested an energy management component which dynamically selects and applies the optimal energy-efficient sorting algorithm on mobile platform. The results show that there is no significant impact on energy consumption from algorithmic performance, and the effort from software engineers can help to reduce energy consumption.

Gupta et al. [19] proposed a novel technique to allocate the optimal power consumption to the CPU and GPU in mobile platform, under a given power budget. They evaluated their work using both simulations and experiments. The experiments were performed on a state-of-the-art mobile platform running industrial benchmarks. The result shows a high throughput and effective utilization to the

available power slack. They successfully achieved their goal of allocating the optimal power consumption to the CPU and GPU, under a given power.

Abdel-Hafeez et al. [1] proposed a novel mathematical comparison-free sorting algorithm that sorts input data integer elements on-the-fly without any comparison operations. They evaluate their work using a CMOS custom application specified integrated circuits (90-nm Taiwan Semiconductor Manufacturing Company) with a 1 V power supply. The result shows a 50% of power saving compared to other optimized hardware-based hybrid sorting designs.

Aliaga et al. [7] performed a concurrent execution of ILUPACK on a multicore platform using energy-efficient runtime implementation. By reducing the idle time for the threads, their results showed a good saving in power consumption without performance degradation.

Basmadjian et al. [10] verified the assumption that the parallel computations power consumption on a multicore processor is equal to the total power of each of those active cores. They prove that for modern multicore processors, this assumption is not accurate. They suggested a new power estimation method that takes into account resource sharing and power-saving mechanisms, which provides an accuracy within a maximum error of 5%.

Kodaka et al. [28] proposed a method for multicore processors that predicted the number of cores needed to execute threads in the near future by using the control information of the scheduler. They reduced the power consumption without performance degradation.

Hamady et al. [22] used a limited number of cores for multicore processor to run different types of workloads to evaluate the power management. Their approach improved power efficiency by using one physical core with hyper-threading.

Azzam et al. [21] analyzed the performance of many kernels and the power consumption of various hardware components in terms of power capping. They use PAPI tool in power tuning to reduce overall energy consumption by 30% without loss in performance in many cases.

Kondo et al. [30] proposed a project called PomPP project; they developed many strategies to save power under a given power budget. They also introduced a power-aware resource manager, power-performance simulation and analysis framework for future supercomputer systems.

Chandra et al. [17] studied the effect of programming language on power consumption. They implement many sorting algorithms (bubble, selection, insertion and quicksort) using more than one programming language (java, visual basic and C#) to find the best power-efficient programming language. They find that Java is the most efficient programming language in terms of power efficiency, and the worst was Visual Basic programming language.

Ozer et al. [33] used the machine learning technology to predict the power and instruction retired metrics with the possible frequency settings by using the information collected at the runtime. They use the regression algorithm after generated pre-processed data and train the algorithm to predict optimum frequency settings.

Al-Hashimi et al. [3] investigated power and energy consumption of three control loops, which are for loop, while loop, and do-while loop. They gave predictions for the cost of control loop statements in terms of power and energy efficiency. The

main weakness of this study was the use of Sandy Bridge Intel chip which suffers from lack of accuracy [20].

Al-Hashimi et al. [5] investigated average peak power, average energy, and average kernel runtime of Bitonic Mergesort and compared it with Advanced Quicksort on NVIDIA Tesla K40 GPU. The researchers identified that a simple parallel Bitonic Mergesort outperformed a highly optimized parallel Advanced Quicksort both in terms of peak power and energy consumption in most cases. The authors use customized algorithms for NVIDIA environment, and it is possible that these results might have been affected by this customization.

Abulnaja et al. [2] investigated Bitonic Mergesort to identify the factors that result in its underlying power and energy efficiency advantage over Bitonic Mergesort. They analyze the power and energy efficiency of Bitonic Mergesort and Advanced Quicksort based on their performance evaluation on NVIDIA K40 GPU. The performance of both the algorithms is investigated using various experiments offered by NVIDIA Nsight Visual Studio. The work is an extension for the previous work in [5] and has the same limitation regarding the customized algorithms for NVIDIA environment.

Al-Hashimi et al. [4] investigated power and energy consumption of Mergesort and Quicksort on Intel Xeon E5-2640 CPU (SandyBridge). The researchers identified some power efficiency advantage for generic mergesort over a highly optimized 3-way quicksort in terms of power consumption. The results show a small, but statistically, significant difference according to a paired-sample *t* test. Moreover, they found that mergesort consumed less energy than quicksort. The main weakness of this study was the use of Sandy Bridge Intel chip which suffers from lack of accuracy [20].

Ikram et al. [25] presented an experimental methodology for measuring power and energy consumption of algorithms running on NVIDIA Kepler (Tesla K40c GPU). They applied the methodology on two customized algorithms for NVIDIA environment: Bitonic Mergesort and matrix multiplication. They measured the algorithms power, energy, and runtime.

3 Motivation

There is economy in performing multiplication and division with positional radix-based numerals when powers of the radix are involved. We move the decimal point when dividing by powers of ten instead of a costly long division. Similarly, in a modern computer where numbers are encoded in the binary system, multiplication and division by powers of two involve very little effort. This economy is reflected in arguably simple Barrel Shifters [26, 32], the classic hardware which implements those operations. In fact, less power consumption was reported [27] for a barrel shifter-based shift-add multiplication scenario in real-time signal processing context.

Moreover, economy in total energy budget may not relate well to how quickly that budget is expended. Workloads which consume similar amounts of energy but run faster will draw more power. In fact, an optimization that improves time efficiency

can become counterproductive if it ends up going through its energy budget too fast. In a power-focused environment, this could be an important concern. Similarly, an energy-focused optimization may end up being detrimental to power. So, what type of energy-related economy should be expected from a computation?

A basic mergesort simply divides a list by two to specify the next set of keys to consider, which does not involve performing a real division. A power advantage may emerge when compared to a sorting method that works harder to make essentially similar decisions in comparable time. A quicksort works harder to split lists to perform comparable work on average. Such behavior could reasonably be expected to consume energy at a higher rate to remain efficient time-wise. So, depending on the type of economy in effort, some procedures may reasonably be expected to do better when it comes to how fast they go through their energy budget. Based on how it splits lists, it seems that mergesort could have a natural advantage there, which a preliminary investigation seems to indicate [4].

Modern energy-conscious processors carefully regulate various energy-related aspects. They are designed to monitor those features accurately since they rely operationally on their own instrumentation, suggesting that an empirical approach to studying power characteristics of algorithms is now even more attractive than before. However, not unlike studying time efficiency empirically, this approach has the disadvantage of having to deal with interfering factors in a typical run environment. The main challenge in designing an experiment remains neutralizing the run environment to isolate the computational aspect of physical measurements.

Several energy-focused studies used empirical methods to develop or optimize algorithms for certain HPC scenarios, or to develop general energy models for those purposes, none of which is our concern here. Other studies focus on sorting performance and compare many methods under various HPC scenarios. This study, however, is not about energy performance of sorting in general, nor about optimizing sorting for specific scenarios. This study is about investigating an interesting algorithm based on features perceived to be desirable from power viewpoint.

In this study, we further investigate a power advantage of mergesort for a context where major system-wide gains could be obtained from small advantages in a ubiquitous workload. We compare a generic mergesort to a classically optimized quicksort. Apart from the obvious need to compare against a credible baseline, quicksort is the established choice for high-performance sorting, and therefore is the method to best show an advantage of value. Using a generic mergesort emphasizes characteristics that could be attributed to the method. More sorts will not serve an investigation focused on examining an advantage anticipated based on an observation about a specific operation expected to be physically power efficient.

In the following section, we describe the experimental setup. It is worth commenting that although it seems apparently reasonable to experiment with components common to HPC, we had other considerations. The choice of processor had more to do with extending previous work to a power-efficient architecture whose essential characteristics are likely to be utilized in future high-performance designs. Choice of Linux had more to do with finer control over processor

workload during the experiment. We henceforth use the abbreviations MS and QS to, respectively, refer to versions of mergesort and quicksort used in this study. The terms mergesort and quicksort are still used when we refer to the methods generically.

4 Experimental setup and methodology

The novelty of this work against our previous work [4] is the environment and the reliable instruments. In earlier experiments, we used Windows 7 OS. Windows OS features numerous graphics as well as other processes running in the background that needs a minimum amount of RAM which is greater than what Linux needs. Also, it is difficult to minimize the background processes that are always running, which make the CPU and RAM always busy and may also add some noise to the results during the sampling process. In this work, we used a lightweight version of Linux OS in our experimental environment to minimize the noise in the results. Another issue regarding Windows OS is that it is not easy to access the model-specific register (MSRs) to read the CPU energy and power consumption in Windows OS, and we need to write instructions in level 0 (kernel-mode or privileged mode) to read the MSRs, which is a very complicated programming work and takes a long time to develop. Furthermore, the OS protection blocks exploitation that attempts to access model-specific register (MSR) in Windows OS. In Linux OS, assessing the MSRs is very easy, using *perf tool* which is part of the OS kernel.

Furthermore, the lack of reliable instruments is particularly problematic for the previous work. In [4], we developed our own profiler that may produce some overhead during the sampling process and may affect our results. Furthermore, we run the experiments on Sandy Bridge Intel chip which suffers from lack of sensors accuracy [20]. In this work, we use a reliable and well-known profiler (*perf tool*) and we use the new Intel Haswell chip which gives reliable sensors readings [20].

The experiment design consists of four parts: the environment, the profiler used to collect the CPU metrics (such as power, energy, functions overhead and instructions overhead), the program files that represent the algorithms code, and the datasets generation. In this section, we briefly describe these parts.

Table 1 Machine Specifications

Processor	Intel Xeon CPU E5-2680 v3 2.50 GHz 12 Cores, 24 Logical Processors TDP limit = 120 W
Cache memory	L1 cache = 12 × 32 KB, 8-way set associative L2 cache = 12 × 256 KB, 8-way set associative L3 cache = 30 MB, shared cache
OS	Linux Ubuntu 16.04 LTS 64-bit
Physical memory (RAM)	8.00 GB

4.1 Evaluation platform and environment

The specifications of the machine used to run our experiments are given in Table 1. In order to capture the exact power and energy measurements of the algorithms executing on the CPU, we need to eliminate environment confounding factors as possible. To do this, the following configuration was made to the environment:

- Disable the CPU power management option and hyper-threading, which gives us access to many performance counters. Furthermore, hyper-threading would also require a more complex power model since the additional threads consume little power.
- Disable the Turbo Boost to make our experiments repeatable without being dependent on thermal conditions and to be sure that the CPU should not exceed its base frequency of 2.5 GHz.
- Terminate most of the operating system processes and services to run the operating system with minimum resources.
- Use Core number 0 to run the algorithms code and move the other processes to core number 2, 3, 4, 5 in order to avoid the context switching mechanism. We also keep core number 1 idle to eliminate the temperature effect on core number 0.
- Use one thread program for the algorithms to control the program running on a single core.
- Disable the compiler optimization options to avoid any optimizations to the code.
- To avoid the recording of datasets loading operations as a workload by the profiler (to avoid noise in power and energy readings), we embedded the datasets directly into the exe file and let the OS load all the data into memory before running. It is known as in-memory execution where the operating system loader is responsible for process initialization and data loading in memory. Therefore, in-memory execution technique will give our results more accuracy and more realistic values in terms of power and energy.

Since our main objective is to study the algorithm's power consumption, we run our code on a single core to ensure that the results are not affected by chip-level optimization or power-saving techniques. Furthermore, we run the code many times until averages converge in order to get a good estimation for our results. For our experiment environment, we found that the average convergence occurs after 230 runs, so we configured the profiler to run 300 times for each case. The profiler shows that our results of standard deviation are as follows $\pm 2\%$. The profiler shows the variations in results to give an indication about the percentage of the noise in the results. Less than 2% variations indicate a perfect environment with minimal noise in the results.

4.2 Measurement of power consumption and other statistics

We used *perf tool* to profile CPU power and energy consumption. We configured the tool to obtain an average of 300 runs. For each run, we use the following command:

```
perf stat -e power/energy-pkg ./algorithm
```

For cache, instructions, cycles and branches statistics, we use the following command:

```
perf stat ./algorithm
```

For functions and instructions overhead, we use the following command:

```
perf record ./algorithm  
perf report
```

4.3 Algorithm's executable files

Both in the case of MS and QS, the source code was taken from a well-known book in academic institutes [36] (The code for MS is listed on page 273 and QS code is listed on page 299). We write the algorithm's code using the C++ programming language. The code is compiled in release mode using the GCC 64-bit compiler. To avoid compiler optimizations, we disabled the compiler optimization options. Furthermore, to give a good estimation for the power and energy results while collecting the samples from the CPU sensors, we eliminate the CPU context switching by setting the affinity for the program and thread to CPU core number 0.

4.4 Datasets generation

Both algorithms was tested on 31 different datasets of integer random numbers that were generated using C++ function `rand()`. The following code was used to generate the datasets:

```
a[i] = rand() % RAND_MAX;
```

The features of the datasets generation are summarized as follows:

- The dataset element type is 4 bytes integer.
- Number of elements in the datasets are: 4–24 K, 48 K, 64 K, 128 K, 256 K, 512 K, 1 M, 2 M, 4 M, 6 M and 8 M. We exclude the datasets from 1 to 3 K,

where the standard deviation in the power and energy results was more than $\pm 2\%$.

- The number of different generated datasets for each size is 20 in order to make sure that the results are not affected by one specific set of random numbers.

The total generated datasets is $31 \times 20 = 620$ different datasets. Each of the generated set was run by each of the sorting algorithms 300 times. The first ten runs of each set were discarded, so the initial transient state of the system did not affect the results. Then, the average of the rest of the runs was obtained. Each efficiency metric reading of each set size was an average of over 290 runs of the generated 31 different sets of the same size. For each run, we obtained the average power (watts), average energy consumption (millijoules), average execution time (milliseconds), L1 Misses (Level 1 cache misses), L2 Misses (Level 2 cache misses) and L3 Misses (Level 3 cache misses), Instructions, Cycles, Branches, Branch Misses, functions overhead, and instructions overhead.

The datasets size was selected carefully to study the effect of cache misses among the three cache levels (L1 = 32 k, L2 = 256 k, and L3 = 30 MB). With 4 bytes integer size, the data stored will be (16 kB, 20 kB, 24 kB, 28 kB, 32 kB, 36 kB, 40 kB, 44 kB, 48 kB, 52 kB, 56 kB, 60 kB, 64 kB, 68 kB, 72 kB, 76 kB, 80 kB, 84 kB, 88 kB, 92 kB, 96 kB, 192 kB, 256 kB, 512 kB, 1 MB, 2 MB, 4 MB, 8 MB, 16 MB, 24 MB, 32 MB).

Fig. 1 Average power consumption in Watts for MS and QS, the numbers between the two curves are the variance in Watts between MS and QS (MS-QS), the trend reversed in favor of QS after dataset size 512 kB

Table 2 Average reported power, energy, and run times for a generic mergesort versus an optimized 3-way partitioning quicksort over 20 randomly generated datasets at each list size

Datasets size	Power (W)		Power differences		Percentage differences	Energy (mJ)		Energy differences		Time (ms)	
	MS	QS	MS-QS	MS power advantage		MS	QS	MS-QS	MS	QS	
16 kB	11.09	16.97	- 5.89	41.91%	40.03	40.05	- 0.01	3.61	2.36		
20 kB	12.92	14.96	- 2.05	14.63%	50.00	39.94	10.06	3.87	2.67		
24 kB	12.02	14.31	- 2.29	17.39%	50.00	40.07	9.94	4.16	2.80		
28 kB	12.19	13.69	- 1.50	11.59%	59.97	49.97	10.01	4.92	3.65		
32 kB	11.40	14.22	- 2.82	22.01%	59.96	60.01	- 0.04	5.26	4.22		
36 kB	10.49	12.30	- 1.81	15.88%	60.00	60.02	- 0.02	5.72	4.88		
40 kB	11.61	12.82	- 1.21	9.91%	70.01	70.00	0.01	6.03	5.46		
44 kB	10.39	12.24	- 1.85	16.35%	70.03	70.01	0.02	6.74	5.72		
48 kB	11.27	11.30	- 0.02	0.27%	80.02	70.06	9.96	7.10	6.20		
52 kB	11.73	12.02	- 0.29	2.44%	89.97	79.93	10.04	7.67	6.65		
56 kB	11.84	12.60	- 0.76	6.22%	100.05	89.96	10.08	8.45	7.14		
60 kB	11.70	12.43	- 0.73	6.05%	99.92	99.94	- 0.02	8.54	8.04		
64 kB	11.70	12.05	- 0.34	2.95%	109.98	100.02	9.96	9.40	8.30		
68 kB	11.68	13.09	- 1.40	11.38%	109.91	110.09	- 0.18	9.41	8.41		
72 kB	11.56	12.41	- 0.86	7.09%	110.05	109.95	0.10	9.52	8.86		
76 kB	11.94	13.32	- 1.38	10.93%	120.00	120.01	- 0.02	10.05	9.01		
80 kB	11.92	13.26	- 1.34	10.64%	130.05	120.00	10.04	10.91	9.05		
84 kB	11.87	12.78	- 0.92	7.38%	130.10	129.97	0.12	10.96	10.17		
88 kB	11.61	12.29	- 0.68	5.69%	140.02	130.03	9.99	12.06	10.58		
92 kB	12.09	12.80	- 0.72	5.71%	150.04	139.90	10.13	12.41	10.93		
96 kB	12.43	13.06	- 0.63	4.94%	159.97	150.06	9.91	12.87	11.49		
192 kB	11.87	12.23	- 0.36	2.99%	280.01	269.92	10.10	23.59	22.07		
256 kB	11.97	12.27	- 0.31	2.48%	380.17	359.88	20.29	31.76	29.33		
512 kB	12.08	12.10	- 0.01	0.17%	679.74	650.25	29.49	56.27	53.74		

Table 2 (continued)

Datasets size	Power (W)		Power differences		Percentage differences	Energy (mJ)		Energy differences		Time (ms)	
	MS	QS	MS-QS	MS power advantage		MS	QS	MS-QS	MS	QS	
1 MB	12.26	12.30	- 0.04	0.33%		1229.92	1140.21	89.71	100.32	92.70	
2 MB	12.94	13.14	- 0.21	1.53%		2330.75	2079.41	251.35	180.12	158.25	
4 MB	13.95	13.85	0.10	- 0.72%		4518.41	3969.41	548.99	323.90	286.60	
8 MB	14.78	14.34	0.44	- 3.02%		9102.12	7282.57	1819.55	615.84	507.85	
16 MB	16.66	15.51	1.15	- 7.15%		18,480.60	14,642.99	3837.61	1109.28	944.10	
24 MB	17.80	16.77	1.03	- 5.96%		28,882.99	22,480.86	6402.14	1622.64	1340.54	
32 MB	18.63	18.19	0.44	- 2.39%		38,229.13	29,988.22	8240.92	2052.02	1648.61	

5 Results and discussion

The average power consumption results for MS and QS obtained from the profiler are shown in Table 2 and Fig. 1. The content of Table 2 is indicated as follows: The first column is the dataset size from 16 kB to 32 MB. Power consumption averages in watt (W) are presented in the second and third columns for MS and QS, respectively. The fourth column shows the power differences between MS and QS in watt (W), and the fifth shows the power advantage for MS against QS in percentage differences. Furthermore, energy consumption averages for MS and QS in millijoule (mJ) are indicated in the sixth and seventh columns and the energy differences between the two algorithms are shown in the eighth column. Finally, the last two columns in the table show the averages of execution time in milliseconds (ms) for MS and QS. From Table 3, we estimate the cache boundaries for MS in each cache level based on the dataset size and the gradual increase from the numbers in the table (we include more details about our estimation in the below paragraphs in this section). We notice that there is an inverse association between the diminishing effect in the trend and the rise of miss rate among the cache levels. (Based on this finding, we decide to study the effect of miss rate on the power trend in the coming paragraphs.) There has been a steady decline in this trend when the dataset becomes larger. What stands out in Table 2 and Fig. 1 is the clear trend of power efficiency in favor of MS before the memory access. The average power advantage for MS is 41.91% in L1, 16.01% in L2 and 8.44% in L3. Moreover, in the memory access area at dataset size 512 kB, a drop in power efficiency for MS is marked and the trend reversed in favor of QS with average power advantage of 1.64% for QS. The interesting finding in these results is that the generic MS shows a clear power advantage against the optimized version of QS.

In contrast to our previous work in [4], the power advantage trend here for MS is significant, while in previous work we found some power advantage in some cases and we use the paired-samples *t* test to prove that MS has a power advantage over QS. We develop our own profiler which has some overhead. Also, in SandyBridge architecture (Intel Xeon CPU E5-2640) sensors gives an estimation-not actual-reading for power and energy [20]; moreover, we use windows 7 OS which has a large memory footprint.

In this work, we enhance the environment to give more reliable results. We eliminate the experiment confounding factors as possible as described in Sect. 4 by using small memory footprint OS (Linux Ubuntu), low overhead profiler, and accurate sensors readings in the new Intel Haswell CPU (Intel Xeon E5-2680). The Intel Haswell RAPL measurements [18] were validated in [20] and show an almost perfect correlation to the AC reference measurement in comparison with the previous Intel SandyBridge architecture.

In terms of energy, the average energy consumption results for MS and QS obtained from the profiler are shown in Fig. 2 and Table 2. The graph shows that there are no significant differences between MS and QS. We notice that QS beat MS in terms of energy in most cases. In some cases, they have similar energy consumption. This trend was expected due to the speed of QS. We know that the

Fig. 2 Average energy consumption in Millijoule (mJ) for MS and QS

energy = power $\times$ time. QS is always faster by 4–35% with an average of 14% and this the reason why QS is better in energy.

Power results show an advantage for MS ranging from 41.91 to 8.44% in upper memory hierarchy levels extending to main memory for datasets up to 3 MB roughly. Such savings may seem small but could be significant for a power-critical environment. Typical workloads in a future exascale supercomputer are expected to consist of massive computations collectively occupying a massive number of processors. Savings reasonably expected [42] in a major fraction of such substantial workloads may be interesting particularly if power draw is an obstacle. Mobile, also a power critical environment, may not benefit as much. Moreover, these are natural savings stemming from the original unoptimized method over a highly optimized realistic high-performance choice, therefore likely to carry over to a future platform. If power draw is an obstacle, then every bit of savings should be of interest. A power-focused concern may also note the marked difference in power draw in two methods otherwise virtually indistinguishable in terms of energy efficiency.

This behavior of power trend may partly be explained by collecting the number of cache misses in each level ($L1$, $L2$, and $L3$), instructions, cycles, branches, and branch misses. Furthermore, we also look inside the assembly code, functions

```

22:  89 c2          mov    %eax,%edx
24:  c1 ea 1f      shr    $0x1f,%edx
27:  01 d0          add    %edx,%eax
29:  d1 f8          sar    %eax
2b:  89 45 fc      mov    %eax,-0x4(%rbp)

```

Fig. 3 Barrel shifter instructions sar and shr in MS assembly code

overhead, and instructions overhead to validate that MS has low overhead against QS which result in low power consumption as we found in the above power results.

Moreover, these results provide further support for the hypothesis that MS has an efficient partitioning by exploiting the Barrel Shifter [26] digital circuit. In order to validate the hypothesis, we obtained further in-depth information from the profiler. Our hypothesis says that MS in its simplest form does fundamentally less work than QS when it comes to deciding the next set of keys to work on. MS simply divides a list by two, which does not involve performing a real division. In theory, it should not perform more comparisons than an efficient sort has to.

We look at the assembly code for MS, and we notice that MS exploits the Barrel Shifter circuit instructions (*shr* and *sar*) to divide the list by 2. Figure 3 shows these instructions in MS assembly code. These instructions provide a strong evidence that supports our hypotheses.

We also need more investigation about the impact of these instructions on MS overhead. To study this impact, we analyze each algorithm overhead based on functions. Both MS and QS are divide-and-conquer algorithms. They perform sort by two major steps, divide the list, then conquer and combined. After the main function, there are two functions in each sorting algorithm. In MS, we have:

- Partitioning function: divide the unsorted list into n sublists, each containing 1 element.
- Merge function: repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining,

while in QS we have:

- Partitioning function: determine the pivot and compare the other elements with this pivot.

Fig. 4 Functions overhead for MS

Fig. 5 Functions overhead for QS

- Swap function: move the elements so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it.

Figures 4 and 5 show the comparison of MS and QS for divide-and-conquer steps. We noticed that MS has 6.60% overhead on average when performing divide step. The reason for that appears in the assembly code in Fig. 3, as it exploits the barrel shifter circuit instructions (shr and sar) to divide the list by 2. On the other hand, divide overhead for QS was 42.90% on average. For conquer and combined step, MS and QS have an average of 93.40% and 57.10%, respectively.

With reference to the first paragraph in this section, we notice that there is an inverse association between the diminishing effect of the trend and the rise of miss rate among the cache levels. To study the effect of cache misses on power consumption, we obtained the number of cache misses in each cache level. The cache misses results are presented in Table 3.

The dataset sizes were selected carefully to study the effect of cache misses among the three cache levels ($L1 = 32$ k, $L2 = 256$ k, and $L3 = 30$ MB). With 4 bytes integer size, the data stored will be (16 kB–32 MB). Both MS and QS use a recursive function call, and we expect that each cache level may spill out earlier to the next level because it keeps data and stack frame in the same cache.

From Table 3, we estimate the cache boundaries for each algorithm in each cache level based on the dataset size and the gradual increase from the numbers in Table 3. We estimate that MS spill out of $L1$ after dataset size 16 kB, $L2$ after 24 kB, and $L3$ after 192 kB. For QS, its spill out of $L1$ after dataset size 28 kB, $L2$ after 44 kB, and $L3$ after 192 kB. We notice that MS spill out to the next level earlier than QS. This finding was expected because MS requires extra space proportional to N , for the auxiliary array for merging. Figures 6, 7, and 8 display more details about our estimation.

Based on the estimated cache boundaries in Table 3, MS has 54% more cache misses than QS in $L1$, 54% in $L2$, and 56% in $L3$. To study the effect of cache misses

Fig. 6 L1 cache boundaries for MS and QS

Fig. 7 L2 cache boundaries for MS and QS

on power consumption, we go back to the power trend in Fig. 1. We notice that MS power advantage exists when the data are located within the boundaries of on-chip memory (L1, L2, and L3). When MS access the off-chip memory (DRAM) after dataset size 192 kB, we notice a sharp drop in power and after dataset size 512 kB the power trend reversed in favor of QS. The literature on [37] has highlighted that the off-chip memory access has a major impact on power consumption, while there is a slight effect on power when accessing the on-chip memory. In our case, we have almost the same average cache misses in all cache levels (54%, 54%, and 56%), while the power advantage is only affected by the L3 misses where the data are located in the main memory. This is strong evidence that MS has an algorithmic

Fig. 8 L3 cache boundaries for MS and QS

power advantage over QS, and this advantage vanished into the large power consumption of the off-chip memory access.

We also obtained other metrics (number of instructions, number of cycles, number of branches, and number of branch prediction misses) as indicated in Table 4 for further investigations. One of the interesting findings in our results in Table 4 is the branch prediction misses. MS has about 25% more branch prediction misses than QS which may affect the power efficiency for MS, especially for large datasets (4–32 MB) where the branch prediction misses average was 37%.

To confirm prior findings and look for additional evidence, we analyze the executed instructions overhead. Table 5 shows the major instructions average overhead for each algorithm. We collect all the instructions in each run in terms of overhead and repetition, and then, we select the instructions that have a notable overhead and repetition. We discard the low overhead and low repetition instructions. Through the profiler trace feature, we can identify each instruction and its overhead. Some instructions appear in few numbers of runs. Others have a very low overhead during the execution. This depends on the pipeline scheduler, data-dependence constraints, resource hazards and branch prediction misses. Furthermore, super-scalar processors may waste a lot of clock cycles during the execution. For this reason, we selected the large repetition and the large overhead instructions, which exist in each run. These instructions represent the real behavior of the algorithm during the run.

Most of the overhead is reported for data movement and comparison instructions. `mov` instruction has the largest overhead. During the execution, `mov` instruction takes 80.51% for MS and 58.52% for QS. Also, `lea` instruction takes 2.82% for MS and 7.01% for QS. Furthermore, we found `pop` and `push` stack manipulation instructions in QS only and reported 1.88% average overhead for each. The `mov`, `lea`, `push` and `pop` instructions are classified as data movement instructions, and thus, the data movement in QS was less than MS. Furthermore, `cmp` instruction is an arithmetic instruction used to compare two values. The `cmp`

Table 3 The average cache misses for *L1*, *L2*, and *L3* for MS and QS

Dataset	MS <i>L1</i> misses	QS <i>L1</i> misses	MS <i>L2</i> misses	QS <i>L2</i> misses	MS <i>L3</i> misses	QS <i>L3</i> misses
16 kB	51,334	23,668	13,779	8518	23	18
20 kB	49,639	24,061	13,782	7623	17	19
24 kB	53,580	24,451	17,305	8304	19	24
28 kB	53,487	25,377	19,378	8690	16	15
32 Kb	54,695	26,391	20,026	11,368	19	14
36 kB	55,421	27,456	20,713	12,092	18	19
40 kB	57,952	28,844	21,321	12,104	17	16
44 kB	57,789	29,868	22,328	13,983	15	14
48 kB	59,629	30,607	30,765	14,784	19	13
52 kB	60,455	31,143	34,629	16,947	25	15
56 kB	63,676	30,821	43,743	21,836	23	16
60 kB	63,985	32,196	44,066	27,434	17	14
64 kB	61,827	31,319	51,606	28,442	18	20
68 kB	64,498	32,883	58,246	37,945	15	14
72 kB	65,501	33,347	79,788	38,288	22	15
76 kB	66,822	32,474	83,100	39,532	19	19
80 kB	68,268	35,158	83,829	41,359	16	14
84 kB	75,761	37,838	87,733	58,824	15	19
88 kB	77,822	38,454	87,455	62,754	14	16
92 kB	78,885	38,355	87,833	67,529	20	15
96 kB	80,004	38,733	108,363	71,586	16	14
192 kB	228,492	173,489	80,136	70,309	275	168
256 kB	290,961	178,480	110,935	87,883	315	115
512 kB	460,531	248,468	125,700	93,088	1464	405
1 MB	762,392	398,104	154,404	124,985	2322	768
2 MB	1,438,628	619,069	186,124	183,962	5976	2428
4 MB	2,643,181	1,001,925	251,912	258,583	5258	1540
8 MB	5,710,921	1,941,319	625,278	476,083	38,870	12,531
16 MB	12,335,511	4,145,939	795,298	834,113	166,899	84,203
24 MB	19,647,628	6,614,606	1,298,085	1,330,941	269,477	177,818
32 MB	26,587,788	8,968,179	1,467,491	1,179,411	373,966	235,896

instruction takes 4.31% for MS versus 8.44% for QS. So, MS has fewer comparisons than QS. The data movement and comparisons findings are consistent with [36].

Other instruction also existed and reported some overhead. For MS, we notice *jge* and *jb* branch instructions report 5.28% and 2.08% average overhead, respectively. In contrast, only one instruction *jge* reported in QS with average overhead 3.17%. Furthermore, we find that *add* and *subl* arithmetic instructions reported 8.26% and 2.68%, respectively, in QS and *addl* and instructions reported 2.6% and 0.9%,

Table 4 The average number of instructions, cycles, branches, and branch misses for MS and QS

Dataset	Instructions		Cycles		Branches		Branch misses	
	MS	QS	MS	QS	MS	QS	MS	QS
16 kB	6,231,726	4,806,317	6,337,325	4,601,794	950,864	585,301	37,292	22,335
20 kB	7,587,789	4,882,613	9,016,013	4,674,238	1,186,146	588,574	44,510	29,969
24 kB	7,802,359	6,267,190	7,265,486	5,729,672	1,137,474	740,298	51,437	35,030
28 kB	8,705,439	7,012,649	8,185,870	6,083,574	1,258,837	812,480	58,629	40,573
32 kB	9,387,835	8,314,646	8,236,599	6,567,943	1,322,979	932,545	65,949	46,308
36 kB	10,210,823	9,485,256	8,742,663	7,783,834	1,421,634	1,072,479	74,453	53,161
40 kB	11,308,272	10,197,179	10,192,576	7,823,904	1,583,937	1,123,276	81,419	59,341
44 kB	12,151,291	11,111,775	10,788,461	8,454,322	1,686,311	1,220,177	89,457	65,790
48 kB	13,244,053	12,027,867	11,269,328	8,720,285	1,837,184	1,294,894	96,915	73,329
52 kB	13,749,166	12,555,056	11,440,802	9,377,673	1,862,766	1,359,660	104,273	80,538
56 kB	14,756,893	13,956,992	12,602,304	11,466,921	2,004,096	1,556,071	110,926	85,446
60 kB	15,566,230	14,964,274	12,970,991	10,966,088	2,094,421	1,608,862	119,963	90,640
64 kB	16,228,845	16,120,953	16,228,845	11,986,556	2,148,645	1,737,824	127,195	98,847
68 kB	17,095,012	16,796,163	13,401,121	12,501,591	2,248,799	1,805,645	135,854	104,903
72 kB	18,040,192	18,989,754	14,285,085	13,603,813	2,368,105	2,025,491	145,050	109,239
76 kB	19,234,016	18,564,031	15,909,202	13,656,161	2,550,686	1,987,960	152,213	119,369
80 kB	20,029,213	19,540,631	16,076,198	14,032,791	2,631,984	2,077,076	159,071	126,270
84 kB	20,885,094	22,215,677	16,502,778	15,299,680	2,726,926	2,326,942	167,675	130,791
88 kB	22,093,135	22,151,043	18,252,194	16,466,613	2,910,736	2,372,283	176,970	136,565
92 kB	22,638,856	24,134,384	17,497,801	17,145,644	2,932,915	2,563,673	184,222	140,306
96 kB	23,605,276	24,747,548	18,275,160	17,663,391	3,053,441	2,620,140	191,129	174,003
192 kB	44,847,033	47,818,157	29,660,679	28,417,951	5,474,743	4,732,419	388,708	298,927
256 kB	60,112,692	64,193,014	39,733,839	38,459,520	7,288,052	6,334,236	525,586	410,413
512 kB	124,237,084	128,034,026	81,627,749	75,418,162	14,898,640	12,593,050	1,093,659	790,778

Table 4 (continued)

Dataset	Instructions		Cycles		Branches		Branch misses	
	MS	QS	MS	QS	MS	QS	MS	QS
1 MB	259,155,260	263,056,623	168,142,242	151,987,570	30,892,399	25,796,204	2,236,285	1,562,419
2 MB	542,189,660	540,142,132	345,914,621	304,061,111	64,394,209	52,847,885	4,539,432	2,968,745
4 MB	1,114,611,851	1,112,253,762	729,777,648	636,323,457	132,927,231	109,708,510	9,072,565	5,712,852
8 MB	2,321,403,476	2,089,022,494	1,466,204,261	1,218,627,643	275,208,537	205,901,957	18,210,652	11,383,346
16 MB	4,853,991,364	4,143,381,573	3,037,266,474	2,406,166,234	575,595,998	407,116,460	36,495,029	22,597,538
24 MB	7,488,860,957	6,348,333,521	4,661,299,734	3,681,673,421	888,812,280	623,609,551	54,705,159	34,627,091
32 MB	10,132,700,599	8,367,124,916	6,267,679,322	4,823,394,478	1,201,692,217	819,521,167	73,060,327	45,202,670

Table 5 Instructions overhead for MS and QS

MS		QS	
Instruction	Overhead %	Instruction	Overhead %
mov	80.51	mov	58.52
lea	2.82	lea	7.01
cmp	4.31	cmp	8.44
jge	5.28	jge	3.17
lg	2.08	pop	1.88
shl	1.5	push	1.88
addl	2.6	subl	2.68
add	0.9	add	8.26
		jmp	3.3
		retq	2.08
		callq	1.39
		cltq	1.39

respectively. Arithmetic instructions usually considered as CPU-intensive instructions and may increase the power consumption for the CPU.

One of the interesting findings in MS instructions was shl shift instruction and reported 1.5% in MS. It is used to multiply the array index by 4 to get byte offset. It is a very efficient multiply operation with less overhead on the CPU.

Furthermore, in QS, cltq loop exit instruction reported 1.39% average overhead and call functions callq and retq reported 1.39% and 2.08%, respectively, and these instructions were not reported in MS. Finally, jmp unconditional jump instruction reported 3.3% overhead in QS only.

The total number of instructions in the instructions overhead table was 12 instructions for QS versus 8 instructions only for MS. It seems that QS has more overhead on the architecture during the execution. These findings provide additional evidence for the power advantage for MS.

These findings collectively provide strong support for the hypothesis that MS is basically a power-efficient sorting algorithm.

6 Conclusion and future work

In this research, we investigated the power characteristics of mergesort under improved experimental conditions. Previous results showed evidence for some advantage under an older generation processor that better reflected a classic approach to energy management [4]. The energy-efficient processor used for the experiment reported here provided a major improvement in instrumentation, enabling us to link energy measurements to architectural features. In this study, we again compared generic mergesort (MS) with a classically optimized 3-way partitioning quicksort (QS), which is the method of choice for general purpose sorting. Both algorithms rely on dividing up their lists in each round. The basic

scheme of MS simply divides a list by two without performing a real hardware division. It does much less work under essentially similar energy budgets, confirmed by our results, in comparable time. We expected such behavior to result in some power advantage. Finally, we examined underlying machine instructions to confirm the use of shifts to split lists in MS, and to confirm a heavier instruction load corresponding to list partitioning in QS.

The main finding of this study is the clearly better power consumption of MS considering that the basic scheme was competing against a highly optimized QS. The signature fast average run time of QS resulted in consuming a comparable amount of energy but at a faster pace. Improved instrumentation helped uncover trends better, we believe. The fact that a mergesort which did not benefit from any optimization effort was able to outperform an optimized quicksort further highlights the power efficiency advantage under the investigated scenario. As expected eventually though, the advantage did indeed diminish as list sizes grew and MS, whose memory needs grow linearly with list size, started to progressively spill out of lower-level caches all the way to DRAM faster than QS until the trend was reversed. Results confirm our expectation of an inflection point as lists grow large enough relative to caches. Interestingly, we also found that MS generated more branch mispredictions than QS.

Results suggest that perhaps we should expect a suitably optimized in-place mergesort to keep the advantage longer. Moreover, results seem to argue for larger lower-level caches where the sort could be kept longer to take better advantage of the favorable part of the power trend. The lowest level cache though should likely remain small to continue to favor better cache hit time. Finally, while mergesort clearly works relatively less hard, relying on an efficient operation on simpler hardware, further investigation is needed to be able to confidently link the observed advantage to specific features of mergesort, as opposed to some compiler related effects for example.

Another obvious limitation is that results provide direct evidence on one architecture (Intel Haswell) running code from one compiler (64-bit GCC). However, if the power trend turns out to be due to an efficient frequent operation, such factors should not matter.

Notwithstanding these limitations, the findings confirm that mergesort should generally be of interest to applications where power is a concern. Another study involving a significantly different hardware architecture reports similar advantage with an architecture-appropriate optimized version of mergesort [5]. These results seem to lend further support to the expectation of an advantage likely to manifest in a future power-critical exascale environment. In such an environment, which is expected to contain a massive number of processors, even small power savings could add up to a major gain, particularly for a pervasive computation which is expected to occupy a significant portion of those processors. According to Amdahl's classic argument [8], the most significant factor in overall improvement is the proportion of time an enhancement is used.

A subtler consequence that the results seem to suggest is to rethink classic time-focused optimizations which could have an aggravating effect on power consumption. For some applications, sacrificing space or even increasing the energy budget

may be acceptable compromises. More importantly is to look at power as a separate concern from energy, opening the door for trading sometime in return for power savings. Such deliberate trade-offs could particularly be useful in power-critical situations, particularly an exascale environment where performance is expected to be abundant and less critical than power draw.

The observed diminishing returns due to linear growth of memory requirements in the basic scheme as dataset size increases encourages investigating in-place mergesort, less favored due to its complexity. A more sustained advantage may justify its implementation costs for application where running bigger problem sets is more important than timing in a power-conscious high-performance environment. Further examination of both nature and causes of the power advantage in mergesort could help uncover optimization opportunities for more savings or patterns beneficial to future algorithms. More generally, this investigation encourages looking into more algorithms, including those previously less favored on time efficiency basis, to explore their power characteristics.

The unexpected increase in mispredictions is another aspect worth looking into. Perhaps also how algorithms are expressed may be worth looking at (is a recursive algorithm more power efficient than its iterative counterpart, for example), or how they are translated into machine code by typical compilers.

Acknowledgements This work has been supported by the Deanship of Scientific Research (DSR), King Abdulaziz University (KAU) under Grant No. 1-611-1433/HiCi.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Abdel-Hafeez S, Gordon-Ross A (2017) An efficient $O(N)$ comparison-free sorting algorithm. *IEEE Trans Very Large Scale Integr (VLSI) Syst* 25(6):1930–1942
2. Abulnaja OA, Ikram MJ, Al-Hashimi MA, Saleh ME (2018) Analyzing power and energy efficiency of bitonic mergesort based on performance evaluation. *IEEE Access* 6:42757–42774
3. Al-Hashimi M, Saleh M, Abulnaja O, Aljabri N (2014) Evaluation of control loop statements power efficiency: an experimental study. In: 2014 9th International Conference on Informatics and Systems. IEEE, New York, pp PDC–45
4. Al-Hashimi M, Saleh M, Abulnaja O, Aljabri N (2017) On the power characteristics of mergesort: an empirical study. In: 2017 International Conference on Advanced Control Circuits Systems (ACCS) Systems 2017 Intl Conf on New Paradigms in Electronics Information Technology (PEIT), pp 172–178. <https://doi.org/10.1109/ACCS-PEIT.2017.8303038>
5. Al-Hashimi MA, Abulnaja OA, Saleh ME, Ikram MJ (2017) Evaluating power and energy efficiency of bitonic mergesort on graphics processing unit. *IEEE Access* 5:16429–16440
6. Al Hasib A, Kjeldsberg PG, Natvig L (2012) Performance and energy efficiency analysis of data reuse transformation methodology on multicore processor. In: European Conference on Parallel Processing. Springer, Berlin, pp 337–346
7. Aliaga JJ, Barreda M, Dolz MF, Martn AF, Mayo R, Quintana-Ort ES (2014) Assessing the impact of the CPU power-saving modes on the task-parallel solution of sparse linear systems. *Clust Comput* 17(4):1335–1348. <https://doi.org/10.1007/s10586-014-0402-z>

8. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18–20, 1967, Spring Joint Computer Conference, AFIPS'67 (Spring). ACM, New York, pp 483–485. <https://doi.org/10.1145/1465482.1465560>
9. Ashby S, Beckman P, Chen J, Colella P, Collins B, Crawford D, Dongarra J, Kothe D, Lusk R, Messina P (2010) The opportunities and challenges of exascale computing. Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, pp 1–77
10. Basmadjian R, de Meer H (2012) Evaluating and modeling power consumption of multi-core processors. In: Proceedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet, e-Energy'12. ACM, New York, pp 12:1–12:10. <https://doi.org/10.1145/2208828.2208840>
11. Beckman P (2011) On the road to exascale. *Sci Comput World* 116:26–28
12. Bergman K, Borkar S, Campbell D, Carlson W, Dally W, Denneau M, Franzon P, Harrod W, Hill K, Hiller J et al (2008) Exascale computing study: technology challenges in achieving exascale systems. Technical Report 15, Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), pp 1–297
13. Bunse C, Hpfner H, Mansour E, Roychoudhury S (2009) Exploring the energy consumption of data sorting algorithms in embedded and mobile environments. In: 2009 10th International Conference on Mobile Data Management: Systems, Services and Middleware, pp 600–607. <https://doi.org/10.1109/MDM.2009.103>
14. Bunse C, Hpfner H, Roychoudhury S, Mansour E (2009) Choosing the “best” sorting algorithm for optimal energy consumption. In: ICSoft, vol 2, pp 199–206
15. Bunse C, Hpfner H, Roychoudhury S, Mansour E (2011) Energy efficient data sorting using standard sorting algorithms. In: Cordeiro J, Ranchordas A, Shishkov B (eds) *Software and data technologies, communications in computer and information science*. Springer, Berlin, pp 247–260
16. Cebrin JM, Snchez D, Aragn JL, Kaxiras S (2014) Managing power constraints in a single-core scenario through power tokens. *J Supercomput* 68(1):414–442. <https://doi.org/10.1007/s11227-013-1044-2>
17. Chandra TB, Verma P, Dwivedi AK (2019) Impact of programming languages on energy consumption for sorting algorithms. In: Hoda MN, Chauhan N, Quadri SMK, Srivastava PR (eds) *Software engineering, advances in intelligent systems and computing*. Springer, Singapore, pp 93–101
18. David H, Gorbatoev E, Hanebutte UR, Khanna R, Le C (2010) RAPL: memory power estimation and capping. In: Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design. ACM, New York, pp 189–194
19. Gupta U, Ayoub R, Kishinevsky M, Kadjo D, Soundararajan N, Tursun U, Ogras UY (2018) Dynamic power budgeting for mobile systems running graphics workloads. *IEEE Trans Multiscale Comput Syst* 4(1):30–40
20. Hackenberg D, Schne R, Ilsche T, Molka D, Schuchart J, Geyer R (2015) An energy efficiency feature survey of the intel Haswell processor. In: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, pp 896–904. <https://doi.org/10.1109/IPDPSW.2015.70>
21. Haidar A, Jagode H, Vaccaro P, YarKhan A, Tomov S, Dongarra J (2019) Investigating power capping toward energy-efficient scientific applications. *Concurr Comput Pract Exp* 31(6):e4485. <https://doi.org/10.1002/cpe.4485>
22. Hamady F, Kayssi A, Chehab A, Mansour M (2013) Evaluation of low-power computing when operating on subsets of multicore processors. *J Signal Process Syst* 70(2):193–208. <https://doi.org/10.1007/s11265-012-0697-z>
23. Hemsoth N (2018) Details emerging on japans future exascale system. <https://www.hpcwire.com/2014/03/18/details-emerge-japans-future-exascale-system/>
24. Hsu J (2015) When will we have an exascale supercomputer? (news). *IEEE Spectr* 52(1):13–16
25. Ikram MJ, Abulnaja OA, Saleh ME, Al-Hashimi MA (2017) Measuring power and energy consumption of programs running on kepler GPUs. In: 2017 International Conference on Advanced Control Circuits Systems (ACCS) Systems and 2017 International Conference on New Paradigms in Electronics and Information Technology (PEIT). IEEE, New York, pp 18–25
26. Ito A (1989) Barrel shifter. <https://patents.google.com/patent/US4829460A/en>
27. Kamble MM, Ugale SP (2016) FPGA implementation and analysis of different multiplication algorithm. *Int J Comput Appl* 149(2):8887
28. Kodaka T, Takeda A, Sasaki S, Yokosawa A, Kizu T, Tokuyoshi T, Xu H, Sano T, Usui H, Tanabe J et al (2013) A near-future prediction method for low power consumption on a many-core processor.

- In: Proceedings of the Conference on Design, Automation and Test in Europe. EDA Consortium, pp 1058–1059
29. Kogge P (2011) Next-generation supercomputers. *IEEE Spectrum*, February
  30. Kondo M, Miyoshi I, Inoue K, Miwa S (2019) Power management framework for post-petascale supercomputers. In: Sato M (ed) *Advanced software technologies for post-peta scale computing: the Japanese post-peta CREST research project*. Springer, Singapore, pp 249–269. https://doi.org/10.1007/978-981-13-1924-2_13
  31. Lastovetsky A, Manumachu RR (2017) New model-based methods and algorithms for performance and energy optimization of data parallel applications on homogeneous multicore clusters. *IEEE Trans Parallel Distrib Syst* 28(4):1119–1133
  32. Mitra SK, Chowdhury AR (2015) Optimized logarithmic barrel shifter in reversible logic synthesis. In: 2015 28th International Conference on VLSI Design (VLSID). IEEE, New York, pp 441–446
  33. Ozer G, Garg S, Poerwawinata G, Davoudi N (2019) Energy-efficient runtime in HPC systems with machine learning. Technical University of Munich, Data Innovation Lab
  34. Poon P, Stout QF (2013) Time-power tradeoffs for sorting on a mesh-connected computer with optical connections. In: 2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum. IEEE, New York, pp 611–619
  35. Reed DA, Dongarra J (2015) Exascale computing and big data. *Commun ACM* 58(7):56–68. <https://doi.org/10.1145/2699414>
  36. Sedgewick R, Wayne K (2011) *Algorithms*, 4th edn. Addison-Wesley, Reading
  37. Theis TN, Wong HSP (2017) The end of moore's law: a new beginning for information technology. *Comput Sci Eng* 19(2):41–50
  38. TOP500 Supercomputer November (2017). <https://www.top500.org/lists/2017/11/>
  39. Villa O, Johnson DR, O'Connor M, Bolotin E, Nellans D, Luitjens J, Sakharlykh N, Wang P, Micikevicius P, Scudiero A et al (2014) Scaling the power wall: a path to exascale. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, New York, pp 830–841
  40. Yildiz O, Dorier M, Ibrahim S, Antoniu G (2014) A performance and energy analysis of *I/O* management approaches for exascale systems. In: *Proceedings of the 6th International Workshop on Data Intensive Distributed Computing, DIDC'14*. ACM, New York, pp 35–40. <https://doi.org/10.1145/2608020.2608026>
  41. Yuechuan Y, Guosun Z, Chunling D, Wei W (2013) Analysis method of energy for C source program and its application. In: 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing, pp 1397–1402. <https://doi.org/10.1109/GreenCom-iThings-CPSCom.2013.244>
  42. Zecena I (2013) Energy consumption analysis of parallel algorithms running on multicore systems and GPUS. Master's thesis, Texas State University-San Marcos

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.