Energy Consumption Analysis of Parallel Sorting Algorithms Running on Multicore Systems

Ivan Zecena [1], Ziliang Zong [2*], Rong Ge [3], Tongdan Jin [4] , Zizhong Chen [5], Meikang Qiu [6]

[1,2]*Computer Science Department, Texas State University*
[3]*Department of Mathematics, Statistics, and Computer Science, Marquette University*
[4]*Ingram School of Engineering, Texas State University*
[5]*Department of Electrical Engineering and Computer Science, Colorado School of Mines*
[6]*Department of Electrical and Computer Engineering, University of Kentucky*
[1,2,4]{iz1003, zz11, tj17}@txstate.edu
[3]rong.ge@marquette.edu [5]zchen@mines.edu [6]mqiu@engr.uky.edu

***Abstract*—Sorting algorithms have been ubiquitously used in numerous applications nowadays. As the data size scales up exponentially, energy-efficiency is gradually becoming equally important as performance for sorting algorithms that process large scale data. Unfortunately, the energy consumption behaviour of various sorting algorithms is not fully explored, although the performance of sorting algorithms have been well studied. Will the sorting algorithms consume more energy when they are parallelized and executed on multicore computers? How to improve the energy-efficient performance of sorting algorithms? In this paper, we comprehensively analyze the performance and energy consumption of three traditional sorting algorithms, Odd-Even Sort, ShellSort and QuickSort, on a multicore system. Our experimental results show that algorithms with better performance tend to conserve energy as well, when evaluated on a computer with eight AMD cores. In addition, for the same algorithm, we observe that more energy savings may be achieved when task granularity is properly selected.**

***Keywords*-Energy Efficiency; OpenMP; Multicore; Parallel Sorting;**

I. Introduction

Significant energy consumption has become a top concern for many data centers and high performance computing applications. It is estimated that in 2011 alone, servers and data centers in the United States consumed more than 100 billion kWh, which amounts to a $7.4 billion energy bill [1], [2]. At the same time, as the processing power increases and newer and larger scientific applications are developed, computer scientists and engineers are turning their attention to examining ways to reducing energy consumption in their applications.

Sorting is one of the fundamental algorithms in computer science and sorting algorithms have been widely used in a myriad of fields; from scientific applications that need to manipulate and make sense of their data, to search engines like Google, which uses sorting to implement its PageRanking system that ranks billions of pages as it crawls the Web [3]. In fact, it is estimated that 85% of scientific applications in the world depend exclusively on sorting algorithms [4]. Therefore, a massive amount of energy can be saved if we are able to improve the energy-efficiency of traditional soring algorithms. However, previous studies primary focused on reducing the time complexity and improving the performance of sorting algorithms. The energy consumption behaviour of sorting algorithms are not fully explored.

In this paper, we conduct an in-depth analysis on performance and energy consumption of three well known sorting algorithms: Odd-even sort, Shellsort, and Quicksort. These algorithms are carefully chosen to cover different implementation methods and time complexity categories. The Odd-even sort and Shellsort are implemented using iterative methods while the Quicksort is implemented recursively using OpenMP tasks. The time complexity of Odd-even sort, Shellsort, and Quicksort algorithms are O(n^2), O($n^{3/2}$) and O($nlogn$), respectively. We analyze both, the serial version and the parallel version of these three sorting algorithms, using a shared-memory system that contains two quad-core AMD 2380 Opteron processors. The size of data to be sorted scales from 10,000 to 1 billion elements.

The remaining parts of this paper are organized as follows. Section II provides the important background about sorting algorithms and parallel sorting. In Section III, we explain our algorithm design and implementation details. The experimental results are illustrated and evaluated in Section IV. Related work is discussed in Section V. And finally, we conclude our work and briefly discuss future research directions in Section VI.

II. Background

1) Sorting Algorithms: Odd-Even, Shellsort, and Quicksort are three well known and commonly used sorting algorithms today [5], [6]. Odd-Even is a simple algorithm with a O(n^2) time complexity that performs well on small data sets. Shellsort is a comparison-based sorting algorithm that improves upon Odd-Even sort in that it has an average time complexity of O($n^{3/2}$) and performs exceptionally well on medium-to-large data sets. Quicksort is known as one of the fastest sorting algorithms in practice today due to its O($nlogn$) time complexity and low memory requirements [6]. These three algorithms are selected to cover both iterative implementation (Odd-Even sort and Shellsort) and recursive

implementation(Quicksort). We analyze the performance and energy consumption of the serial code and parallel code of each algorithm as well. In Section III, we will provide more details on how to parallelize each algorithm.

2) Amdahl's Law: One of the guiding principles when improving and parallelizing applications is Amdahl's law. Amdahl's law is used as a guideline to find the maximum expected improvement in a program when only part of it can be improved. In essence, it gives us the theoretical maximum speedup when parallelizing a program across multiple cores [7]. Amdahl's law states that if P is the proportion of a program that can be made parallel, and (1 - P) is the proportion that remains serial, then the maximum speedup that can be achieved with N processors is given by $Speedup = \frac{1}{(1-P)+\frac{P}{N}}$. In other words, the speedup of parallel code will not linearly grow with the number of cores. It is also limited by the percentage of code that can be parallelized. For instance, if 95% of a program can be parallelized, the theoretical maximum speedup assuming an infinite number of cores would be 20x, no matter how many processors we use. A more realistic approach would be using 8 cores, which yields a theoretical maximum of 6x. Thus, the less serial code an application executes, the greater the speedup will be.

3) Task Decomposition and Task Granularity: In order to transform a sequential algorithm to a concurrent algorithm, the first step is to identify the code segment that can be executed in parallel. Task decompostion refers to the process of splitting a big task into smaller tasks that can be concurrently executed. Task granularity, accordingly, is defined as the amount of computation that a task need to complete. The more work a task will do before synchronization, the coarser the granularity will be. Since task decomposion and task management is not free, we must avoid the danger of creating over fine-grained tasks because they do not have sufficient work assigned to threads to overcome the overhead cost. We will use Quicksort as an example to explain the impact of task granularity on performance and energy consumption in Section IV.

III. Algorithms Design and Environment

In order to compare the performance and energy consumption of the serial code and parallel code, we developed a parallel version for each of the aforementioned sorting algorithms using OpenMP [4]. The OpenMP extension allows us to add parallelism to our algorithms on our shared-memory based testbed with two quad-core AMD 2380 Opteron processors. This is accomplished by issuing sets of *#pragmas* that indicate the compiler which sections of code to parallelize. All three algorithms utilize a dynamically-allocated Array structure to store the randomly-generated sample data. Our algorithms are implemented as follows.

4) Iterative Sorting: Both, Odd-even sort and Shellsort are implemented iteratively. Odd-even uses phases to compare all odd or even indexed pairs of adjacent elements in the list. Phases are incremented from 0 to n, and odd phases compare all odd-indexed elements and even phases compare all even-indexed elements. For each new phase, the list is partitioned

```
void Odd_even(long a[], long n)
{
   long phase;
   long tmp,i;

#  pragma omp parallel num_threads(thread_count) \
      default(none) shared(a, n) private(i, tmp, phase)
   for (phase = 0; phase < n; phase++)
   {
      if (phase % 2 == 0)
#        pragma omp for
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp for
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }//end for
}  // end Odd_even function
```

Fig. 1. Parallel Odd-Even sort algorithm.

in a block fashion and one chunk is assigned per core so that data dependencies are eliminated. Figure 1 contains a code snippet illustrating this algorithm.

Shellsort on the other hand, works by comparing elements that are separated by a *gap*. The first element is compared to the element located *gap* positions down the list, the next element is then located *gap* positions away, and so on. Each new sublist of elements to be compared is assigned to a core, so cores comparing and swapping their sublists of elements do not hava data dependencies with each other. At the beginning, the gap is roughly 1/3 of the array size. Then for each new iteration the gap is gradually reduced to 1. Figure 2 illustrates how the parallel Shellsort algorithm works.

5) Recursive Sorting: Quicksort is an recursive algorithm that selects a *pivot* and partitions the list into two disjoint groups. All elements in the left group are less than the *pivot* and elements in the right group are greater than the *pivot*. The partition is performed recursively for each sublist until all sublists have been sorted. The parallel Quicksort algorithm is implemented recursively as well using OpenMP Tasks, which are defined and supported in the OpenMP 3.0 Standard [8]. Each time a partition is called with a differnt list, a new task will be generated to complete the partition job. The parallel Quicksort algorithm is illustrated in the code snippet shown in Figure 3.

By using OpenMP tasks in our implementation, we parallelize this irregular and recursive algorithm by creating a pool of tasks from which available cores pull the tasks out. The number of tasks and the amount of work each task will do is determined by how *deep* we want the level of recursion to go.

```
void shell_sort_parallel(long sort_arr[], long size)
{
  long i,j,k;
  long gap=1;

   while(gap*3+1 < size)
     gap = gap*3+1;

    while (gap >= 1)
    {
      #pragma omp parallel for num_threads(thread_count) private(i,k,j)
        for (i=0 ; i<gap ; i++)
        {
            for (j=i; j<size-gap; j+=gap)
            {
              k = j;
              while(k >= i && sort_arr[k] > sort_arr[k+gap])
              {
                swap_shep( sort_arr, k, k+gap );
                k -= gap;
              }
            }
        }

        gap = gap / 3;
    }//end
```

Fig. 2. Parallel Shellsort algorithm.

```
void QuickSortOmpTask(long array[], const long left, const long right, const int deep)
{
    if(left < right){
        if( deep )
        {
            const long part = QsPartition(array, left, right);
            #pragma omp task
            QuickSortOmpTask(array,part + 1,right, deep - 1);
            #pragma omp task
            QuickSortOmpTask(array,left,part - 1, deep - 1);
        }
        else
        {
            const long part = QsPartition(array, left, right);
            QsSequential(array,part + 1,right);
            QsSequential(array,left,part - 1);
        }
    }
}

void QuickSortOmp(long array[], const long size)
{
    #pragma omp parallel
    {
        #pragma omp single nowait
        {
            QuickSortOmpTask(array, 0, size - 1 , 15);
        }
    }
}
```

Fig. 3. Parallel Quicksort algorithm.

The greater the depth, the less amount of work each task will complete and the finer the task granularity will become. As we decrease the depth, less tasks will be created and each task will contain more work [7]. We will tune the task granulity of parallel Quicksort algorithm in the experimental evaluation section by varying the depth and discuss the impact of task granularity on performance and energy.

IV. Experimental Evaluation

In this section, we comprehensively evaluate our parallel sorting algorithms using the performance, speedups, and energy efficiency metrics. Before we dive into the detailed discussion, we first describe our testbed system.

6) Experimental Environment: Our experimental environment consists of a computer node composed of two Quad-Core AMD Opteron(tm) 2380 Processors running at 2.5 GHz. Since two Quad-core processors are used per node, there are a total of 8 processing cores in this shared-memory setup. In addition, this node is connected to a meter node that runs in the background and measures the energy consumption of the node. In order to properly evaluate the performance and energy efficiency of our algorithms, the meter measures the node's energy consumption every second and all the readings are collected and saved in a file. For jobs that take less than 1 second to complete, the energy consumption is derivated from the run time and the node's base energy consumption. For all our evaluations the base energy consumption of the node when idle is 134.47 Joules per second.

7) General Performance and Energy Consumption Analysis: We tested our parallel Odd-even, Shellsort, and Quicksort algorithms using 1,2,4, and 8 cores respectively. For each case, we scaled the unsorted sample data from 10,000 up to 1 billion elements. The performance data and energy consumption data for each sorting algorithm is shown in Figure 4.

In Figure 4, we observe that the execution time of each algorithm decreases as we increase the number of cores, which is predictable. We also notice that total energy consumption reduced as the execution time gets shorter, even if the number of cores doing work is increased. The reason is that the energy consumption of the extra cores is negligible compared to the base power being consumed by the machine. The only exception to this behavior occurs when the sample data is so small that the overhead of parallelization and synchronization is significantly larger than the performance gained by using more cores. The exception can be observed when sorting 10,000 elements. In this case, both Shellsort and Quicksort have no performance gain or energy savings. It is in fact the case that Quicksort performs worse and consumes more energy when sorting 10,000 elements and the number of cores is increased from 4 to 8. However, for all remaining data sets, the larger the data set is, the more energy-efficient our parallel sorting algorithms become. Figure 5 and Figure 6 further detail the speedups and energy reductions obtained for each algorithm.

Again, as shown in these figures, the speedups and energy consumption percentages prove that in a shared-memory environment, using as many cores as possible to sort large data sets will yield a faster and more energy-efficient sorting process. In addition, as indicated by Amdahl's law, the maximum speedup is achieved when sorting the largest data set of 1 billion elements. Looking at Quicksort for example, a speedup of 6.50x was achieved compared to a 1.16x when sorting only 100,000 elements. Energy wise, sorting 1 billion elements consumed only 16.4% of the original energy compared to an 86.2% when sorting 100,000 elements.

8) Shellsort and Quicksort Comparison: Due to the fact that clusters and HPC systems usually deal with large amounts of data, we decided to further analyze the energy-efficiency of Shellsort and Quicksort when sorting 10 million elements.

Sorting Algorithms Performance (1 machine, 1-8 cores)									
Data	Algorithm	Time (s) 1 Core	Total Energy (J)	Time (s) 2 Cores	Total Energy (J)	Time (s) 4 Cores	Total Energy (J)	Time (s) 8 Cores	Total Energy (J)
10,000	Odd-Even	1.99	268.6J	1.00	134.8J	0.53	71.3J	0.29	39.0J
	Shellsort	0.01	1.3J	0.01	1.3J	0.01	1.3J	0.01	1.3J
	Quicksort	0.01	1.3J	0.01	1.3J	0.01	1.3J	0.04	5.4J
100,000	Odd-Even	199.57	27176.6J	99.26	13804.8J	53.05	7473.6J	27.31	4021.8J
	Shellsort	0.23	30.9J	0.14	18.8J	0.09	12.1J	0.07	9.4J
	Quicksort	0.07	9.4J	0.04	5.4J	0.03	4.0J	0.06	8.1J
500,000	Odd-Even	4968.73	677369.7J	2500.20	345726.8J	1302.90	184657.0J	675.19	100622.9J
	Shellsort	1.67	269.2J	0.93	134.9J	0.55	74.0J	0.37	49.8J
	Quicksort	0.37	49.8J	0.19	25.5J	0.11	14.8J	0.09	12.1J
1,000,000	Odd-Even	20091.48	2780763.0J	9998.62	1382570.4J	5232.57	743229.9J	2716.37	406571.3J
	Shellsort	3.69	539.6J	2.06	270.8J	1.16	135.7J	0.76	102.2J
	Quicksort	0.76	102.2J	0.40	53.8J	0.23	30.9J	0.15	20.2J
10,000,000	Shellsort	62.61	8591.2J	32.95	4563.7J	18.12	2713.2J	10.03	1497.1J
	Quicksort	9.30	1358.3J	4.79	684.6J	2.68	422.2J	1.64	282.6J
100,000,000	Shellsort	1115.57	152286.0J	575.33	79914.9J	298.02	42670.3J	157.18	23524.7J
	Quicksort	167.76	23024.2J	86.00	12067.0J	45.54	6541.4J	25.82	3783.8J

Fig. 4. Performance of sorting algorithms.

Speedups (over serial)					
Data	Algorithm	Serial	Parallel		
		1 Core	2 Cores	4 Cores	8 Cores
10,000	Odd-Even	1x	1.98x	3.75x	6.79x
	Shellsort	1x	1.55x	2.00x	2.33x
	Quicksort	1x	0.00x	0.85x	0.17x
100,000	Odd-Even	1x	2.01x	3.76x	7.31x
	Shellsort	1x	1.65x	2.47x	3.52x
	Quicksort	1x	1.81x	2.48x	1.16x
500,000	Odd-Even	1x	1.99x	3.81x	7.36x
	Shellsort	1x	1.79x	3.03x	4.16x
	Quicksort	1x	1.89x	3.30x	4.02x
1,000,000	Odd-Even	1x	2.01x	3.84x	7.40x
	Shellsort	1x	1.79x	3.18x	4.81x
	Quicksort	1x	1.90x	3.34x	5.21x
10,000,000	Shellsort	1x	1.90x	3.45x	6.24x
	Quicksort	1x	1.94x	3.47x	5.67x
100,000,000	Shellsort	1x	1.94x	3.74x	7.10x
	Quicksort	1x	1.95x	3.68x	6.50x

Fig. 5. Speedup of sorting algorithms.

Energy Consumption(compared to serial)					
Data	Algorithm	Serial	Parallel		
		1 Core	2 Cores	4 Cores	8 Cores
10,000	Odd-Even	100%	50.0%	26.5%	14.5%
	Shellsort	100%	100%	100%	100.0%
	Quicksort	100%	100%	100%	415.0%
100,000	Odd-Even	100%	50.0%	27.5%	14.7%
	Shellsort	100%	60.8%	39.2%	30.4%
	Quicksort	100%	57.4%	42.6%	86.2%
500,000	Odd-Even	100%	51.0%	27.3%	14.9%
	Shellsort	100%	50.1%	27.5%	18.5%
	Quicksort	100%	51.2%	29.7%	24.3%
1,000,000	Odd-Even	100%	49.7%	26.7%	14.6%
	Shellsort	100%	50.2%	25.1%	18.9%
	Quicksort	100%	52.6%	30.3%	19.8%
10,000,000	Shellsort	100%	53.1%	31.6%	17.4%
	Quicksort	100%	50.4%	31.1%	20.8%
100,000,000	Shellsort	100%	52.5%	28.0%	15.4%
	Quicksort	100%	52.4%	28.4%	16.4%

Fig. 6. Energy consumption of sorting algorithms.

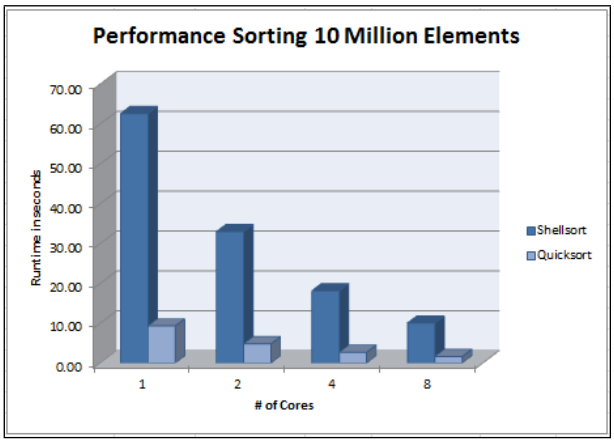

Fig. 7. Performance comparison of Shellsort and Qucksort.

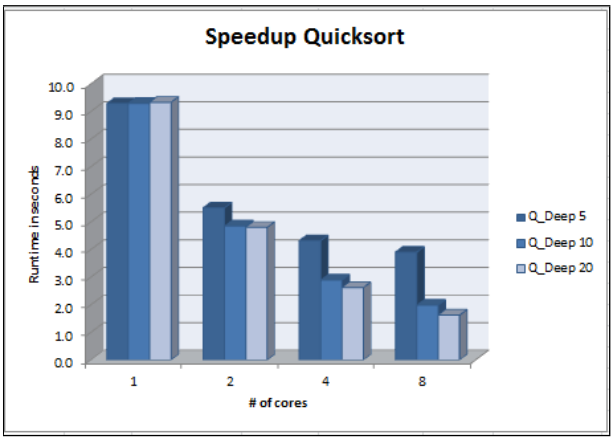

Fig. 9. Performance comparison of different Quicksort depths.

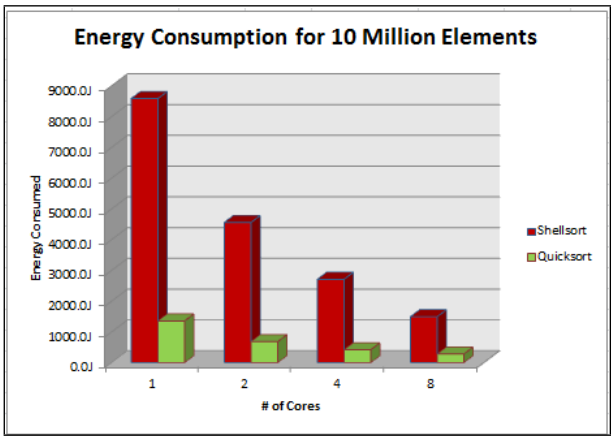

Fig. 8. Energy consumption of Shellsort and Quicksort.

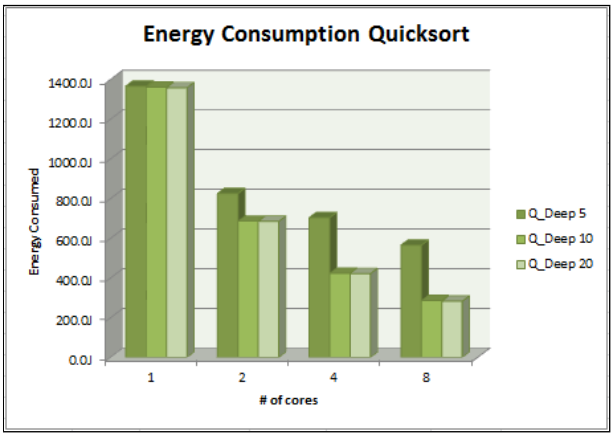

Fig. 10. Energy consumption of different Quicksort depths.

We ignored the results obtained for Odd-even sort because this algorithm takes too long to run for data sets at this scale. Figures 7 and 8 show the performance and energy consumption of both Shellsort and Quicksort.

Quicksort, as the conventional wisdom indicates, outperforms Shellsort [6]. Shellsort performs better than Quicksort for small data sets but Quicksort runs faster than Shellsort for large data sets such as 10 million. Our results also prove that the energy consumption of this algorithm is much lower than Shellsort, and the same behavior can be observed as we scale the work across many cores. As shown on Figures 4 and 6, running Quicksort on 2 cores consumes only 50.4% of the original energy, 31.11% on 4 cores, and 20.8% on 8 cores.

9) Impact of changing Quicksort's task granularity: Since Quicksort proves to be the most efficient algorithm among these three algorithms, we also analyze the impact of task granularity on performance and energy by varying the *depth* parameter, as explained in Section III. We ran Quicksort with recursion level depths of 5,10,15, and 20, and sorted the same 10 million elements as before. Figures 9 and 10 illustrate our results.

We can see that energy consumption of the algorithm is reduced the most when we reduce its recursion *depth* to 20. Using 8 cores and a *depth* of 20 levels produced an energy consumption of about 283.4J, compared to 565.5J when using a *depth* of 5 levels, this represents an energy savings of 50%. Although changing the *depth* to 10 also consumes about the same energy, the speedup increase with a *depth* of 20 significantly outperforms using a *depth* of 10 levels, as seen in Figure 9. In short, our results show that having a *depth* of 20 for the Quicksort algorithm increases performance, reduces energy consumption, and provides the best energy-efficient performance.

V. Related Work

Most of the previous research in the HPC field has focused on improving performance and maximizing speed [4], [5]. But as energy costs continue to rise, the need for energy-aware applications has become critical. As a result, recent research started to shift the focus to energy savings. For example, a number of recent work has proposed to optimize the energy consumption of processors [9], [10] and disks [11], [12] of high performance systems.

However, the research on investigating the energy consumption behaviour of different algorithms is still in its infancy. To the best of our knowledge, there are only a few work to study the energy consumption of different sorting algorithms. Zent et. al. proposed a hybrid sorting algorithm (Quicksort + Heapsort) that is able to conserve energy by taking advantage of Dynamic Voltage Scaling (DVS) technology [13]. Bunse et. al. published the energy consumption data of different sorting algorithms running on mobile devices [14]. However, their work only analyzed the energy consumption of sequential sorting algorithms. In our study, we analyze the energy consumption of both sequential and parallel sorting algorithms through detailed power profiling. In addition, we discuss the impact of different parallel task desgins and varying task granularities on the energy consumption of parallel Quicksort algorithm.

VI. Conclusion and Future Work

After analyzing the results generated by sorting large data sets with three parallel sorting algorithms, we can make the conclusion that the conventional wisdom of using more cores on a shared-memory system does in fact lead to more energy-efficient sorting algorithms. In this work, we also demonstrate that using a Quicksort algorithm on shared-memory systems can provide a significant increase in energy savings over other sorting algorithms. In addition, a fine-tuned Quicksort algorithm can also provide a 50% energy savings over another Quicksort algorithm that uses a different task granularity. As for future work, we plan to expand our energy consumption analysis of sorting algorithms to distributed memory systems as well. We plan to combine shared-memory with distributed-memory applications that make use of several machines in a cluster. We also plan to add component-level power measurement to our analysis, which will allow us to break down how much energy each hardware component consumes during the sorting process.

Acknowledgment

The authors sincerely appreciate the comments and feedback from the anonymous reviewers. Their valuable discussions and thoughts have tremendously helped in improving the quality of this paper. The work reported in this paper is supported by the U.S. National Science Foundation under Grants No. CNS-1118043, CNS-1116691, CNS-1118037, the U.S. Department of Agriculture under Grant No. 2011-38422-30803, and the Texas State University Library Research Grant.

References

[1] Environmental Protection Agency, "Report to Congress on Server and Data Center Energy Efficiency", *Public Law 109-431*, Public Law, 109, 2007.

[2] C. Lefurgy, et al., "Energy-Efficient Data Centers and Systems" *International Symposium on Workload Characterization*, Austin, Texas, November 2011.

[3] S. Brin, and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine", Stanford University, Stanford, CA. http://infolab.stanford.edu/ backrub/google.html.

[4] P. S. Pacheco, "Shared-Memory Programming With OpenMP", *An Introduction to Parallel Programming*, MA: Morgan Kaufmann Publishers, 2011. pp. 209-259.

[5] P. Kataria, "Parallel Quicksort Implementation Using MPI and Pthreads", http://www.winlab.rutgers.edu/ pkataria/pdc.pdf.

[6] H. W. Lang, F. H. Flensburg, "Sorting algorithms: Quicksort", http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/quick /quicken.htm.

[7] C. Breshears, "The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications", *California: O'Reilly Media*, pp. 169-180, Inc., 2009.

[8] A. Duran, "Tasking in OpenMP", *Barcelona Supercomputing Center*, June 3rd, 2009. https://iwomp.zih.tu-dresden.de/downloads/omp30-tasks.pdf.

[9] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced cpu energy". *In Proceedings of OSDI*, 1994.

[10] Z. L. Zong, A. Manzanares, X. J. Ruan, and X. Qin, "EAD and PEBD: Two Energy-Aware Duplication Scheduling Algorithms for Parallel Tasks on Homogeneous Clusters", *IEEE Transactions on Computers*, Vol. 60, Issue 3, pp. 360-374, Mar. 2011.

[11] F. Douglis, P. Krishnan, and B. N. Bershad, "Adaptive disk spin-down policies for mobile computers", *In Proceedings of MLICS*, 1995.

[12] Z. L. Zong, X. Qin, X. J. Ruan, and M. Nijim, "Heat-Based Dynamic Data Caching: A Load Balancing Strategy for Energy-Efficient Parallel Storage Systems with Buffer Disks," *In Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST 2011)*, May, 2011.

[13] G. Zeng, H. Tomiyama, H. Takada, "Analyzing and optimizing energy efficiency of algorithms on DVS systems A first step towards algorithmic energy minimization", *In Proceedings of the Asia and South Pacific Design Automation Conference*, Jan. 2009.

[14] C. Bunse, H. Hopfner, E. Mansour, S. Roychoudhury, "Exploring the Energy Consumption of Data Sorting Algorithms in Embedded and Mobile Environments", *In Proceedings of the 10th International Conference on Mobile Data Management: Systems, Services and Middleware*, 2009.