Power and Energy Footprint of OpenMP Programs Using OpenMP Runtime API

Anilkumar Nandamuri, Abid M. Malik, Ahmad Qawasmeh, Barbara M. Chapman
Department of Computer Science
University of Houston, Houston, TX, USA

{anandamuri, ammalik3, arqawasam, chapman}@uh.edu

***Abstract*—Power and energy have become dominant aspects of hardware and software design in the High Performance Computing (HPC). Recently, the Department of Defense (DOD) has put a constraint that applications and architectures need to attain 75 GFLOPS/Watt in order to support the future missions. This requires a significant research effort towards power and energy optimization. OpenMP programming model is an integral part of HPC. Comprehensive analysis of OpenMP programs for power and execution performance is an active research area. Work has been done to characterize OpenMP programs with respect to power performance at kernel level. However, no work has been done at the OpenMP event level. OpenMP Runtime API (ORA), proposed by the OpenMP standard committee, allow a performance tool to collect information at the OpenMP event level. In this paper, we present a comprehensive analysis of the OpenMP programs using ORA for power and execution performance. Using hardware counters in the Intel SandyBridge x86-64 and Running Average Power Limit (RAPL) energy sensors, we measure power and energy characteristics of OpenMP benchmarks. Our results show that the best execution performance does not always give the best energy usage. We also find out that the waiting time at the barriers and in queue are the main factors for high power consumption for a given OpenMP program. Our results also show that there are unique patterns at the fine level that can be used by the dynamic power management system to enhance the power performance. Our results show substantial variation in energy usage depending upon the runtime environment.**

***Keywords*-Power, Energy, OpenMP, Runtime API, Performance Analysis.**

I. Introduction

The High Performance Computing (HPC) community is heading toward an era of exascale computing. These machines will have high energy consumption. The average power consumption of the Top 10 systems in the TOP500 list is 2000 KW. The power consumption of the Jaguar system is nearly 7000 KW. The power performance of HPC system is expected to grow further to 100 MW in the next few years. High energy consumption results in many problems to HPC systems, such as low reliability, bad stability, and high costs. Thus, energy saving in parallel computing has become an important issue. For a large number of parallel scientific programs, parallel loops account for majority of the execution time. Energy is the product of power and time. Hence, reducing the energy of parallel loops is the key to reducing the energy of the whole program as well.

OpenMP is a popular shared-memory parallel programming model to explore parallelism in loops. With the increasing number of cores in a processor chip, the natural parallelism of OpenMP makes it more widely used in HPC systems, and makes it one of the most indispensable multi-threaded programming models. Thus, optimizing the energy consumption of OpenMP programs is important. It requires a comprehensive analysis of OpenMP programs that can help a user to characterize the programs with respect to energy usage. These characteristics can be used to build an efficient cost model for the dynamic power management system. Recently, work by Allan, et al. [17] tried to evaluate the characteristics of OpenMP programs. The work collects the information at the kernel level. No work has been done that characterizes OpenMP programs using the information collected at the fine level or the OpenMP event level.

OpenMP Runtime API (ORA), proposed by the OpenMP Standard Committee [10], permits a performance tool to gather information about an OpenMP program from the runtime system in such a manner that neither the performance tool nor the runtime system needs to know any details about each other. The APIs can be used to collect information at the OpenMP event level, e.g., fork event, join event, barrier event, task waiting event etc. ORA has been implemented in the OpenUH and GCC open-source compilers. Currently, the next version of the APIs, OpenMP Tracing Interface (OMPT) and OpenMP Debugging Interface (OMPD) [3], are under consideration with the OpenMP Standard Committee.

In this paper, we study several aspects of key OpenMP programs for performance and energy usage using ORA. Our results highlight the characteristics of performance, energy efficiencies and cache behavior of these programs by varying the data size, compiler optimization levels and OpenMP runtime parameters. We hope this work would help in understanding the true behaviour of OpenMP programs with respect to power consumption and energy usage. The main contributions of the work are:

1) Comprehensive analysis of OpenMP programs with respect to performance and energy usage.
2) Study of coarse and fine level characteristics of OpenMP programs for energy usage.
3) Study of patterns for best and worst energy performance for OpenMP programs.

4) Study of parameters and features responsible for best and worst OpenMP power and energy performance.

The rest of the paper is organized as: Section II gives the related work in this area. Section III discusses the detail of our framework. Section IV explains the experimentation. Section V gives results and analysis. Section VI gives conclusion and future work.

II. Related Work

Currently, there has been considerable research into power management for HPC systems and applications. In the past, power management was a big challenge for the embedded processors. The embedded community has responded to this challenge by making the system and the applications power-aware [4], [16]. However, embedded devices typically have stricter power constraints but less restrictive performance requirements compared to the HPC systems.

Power management on HPC systems has been mainly focused on using the available hardware mechanisms for controlling energy usage. The most common mechanism has been voltage and frequency scaling [6], [19]. The work by Ge, et al. [8] explored opportunities to save energy at fixed frequencies for memory bound applications. Freeh, et al. [6] used offline traces to manually divide the work into phases that are run at several frequencies do determine the most energy efficient choice. This approach is close to the runtime adaptation strategy. Tiwari, et al. [22] automates the process of finding phases and optimal frequencies using power models. A number of efforts use hardware performance counters to compute optimal off-line settings. Several projects estimate energy usage based on hardware counters with direct correlation including cache access [7], MIPS [11] and CPU stall cycles [12]. Li, et al. [14] address DVFS and dynamic concurrency throttling for hybrid, multi-node MPI+OpenMP applications. They statically determine the best concurrency level for each OpenMP phase and explore dynamic algorithms to reduce power utilization in nodes that are idle due to load imbalance. Osman, et al. [21] propose a software-based online resource management system that leverages hardware facilitated capability to constrain the power consumption of each node in order to optimally allocate power and nodes to a job. The scheme uses this hardware capability in conjunction with an adaptive runtime system that can dynamically change the resource configuration of a running job allowing a resource manager to re-optimize allocation decisions to running jobs as new jobs arrive, or a running job terminates. Oliver, et al. [15] and Ahmad, et al. [18] study different strategies for getting better execution performance using OpenMP programming model.

Recently an additional hardware mechanism, power clamping, has been introduced on Intel SandyBridge, along with similar mechanisms on IBM Power 6 and 7 (capping) and AMD Bulldozer (capping and thermal design power limits). Allan, et al. [17] uses RAPL to study the energy usage patterns under different environment variables. However, the work is at the coarse level. Yonghong, et al. [23] studies the performance, cache behavior, and energy efficiency of multiple parallel matrix multiplication algorithms on a multicore using RAPL. Rountree, et al. [20], [19] examine the effect of clamping for an HPC application (NAS MG) using RAPL. The work addresses processor performance variation as HPC moves from performance scheduling to power scheduling.

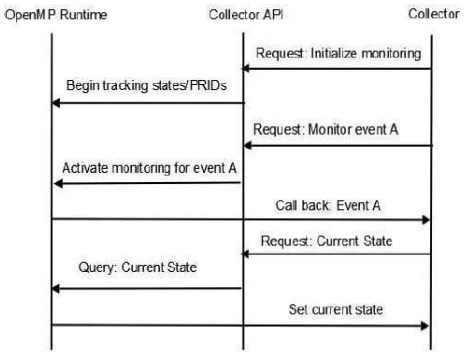

Fig. 1: Sequence of events in Collector API.

III. Framework

This section describes the main components of the framework used for the experimentation.

A. OpenUH Framework

We used the OpenUH compiler to compile the benchmark applications. The OpenUH Compiler is a branch of the open source Open64 compiler suite. It has support for a variety of architectures including x86-64, IA-32, IA-64, Opteron Linux ABI platform and PTX generation for NVIDIA GPUs. OpenUH is open source and it supports C, C++, Fortran 95/2003. The front end support is ported from the GNUC/C++ compiler, while back-end translation is implemented by the HPC Tools group [9] at the University of Houston in collaboration with Tsinghua University. The OpenUH framework supports OpenMP 3.0. For more detail about OpenUH, consult the work by Chapman, et al. [1].

B. ORA Interface

The ORA, also known as Collector API, [13] consists of a single routine that takes the form: *int __omp_collector_api (void *arg)*. The **arg** parameter is a pointer to a byte array that can be used by a performance or collector tool to pass one or more requests for information from the runtime. The collector requests notification of a specific event by passing the name of the event to monitor as well as a callback routine to be invoked by the OpenMP runtime library each time the event occurs. Figure 1 shows an example of a sequence of requests made by a collector to the OpenMP runtime library.

The OpenMP runtime and the collector tools are independent of each other. Therefore, it is essential that a mechanism be provided that allows each of them to interact while not compromising their ability to operate separately and individually. This is accomplished by having the OpenMP runtime

implement a single API function *__omp_ collector_ api* and export its symbol in the OpenMP runtime library. The collector may then query the dynamic linker to determine whether the symbol is present. If it is, then it may initiate communications with the runtime and begin to make queries and send requests to the OpenMP runtime using the interface. The OpenMP runtime will then start keeping track of states and triggering event notications.

When a collector tool sends a request to register any event through the ORA, the event type *OMP_COLLECTORAPI_REQUEST* and a callback function pointer is passed as an argument to the API call in the runtime. Race conditions might occur when multiple threads try to register the same event with multiple callbacks. The callback function pointer is stored in a table in which each entry has a lock associated with it to prevent race conditions. The table contains the event callbacks shared by all the threads. The frequency in which the events are registered relies on the nature of the collector tool. Two functions, *__ompc_event_callback(event)* and *__ompc_set_state(state)*, are inserted at different positions in the OpenMP runtime routines. These functions implement different events and states associated with the OpenMP execution model. The state values are stored in the OpenMP thread descriptor in the runtime. Once a thread reaches an event point, the function *__ompc_event_callback((OMP_COLLECTORAPI_EVENT) e)* is executed and the callback function, associated with this event, is invoked. The functionality of the callback is determined by a performance tool in order to collect the required performance measurements. Interested readers msy consult the work [13], [3] for more deatil on ORA.

C. Running Average Power Limit Sensors

Intel introduced the Running Average Power Limit (RAPL) feature with the Sandy Bridge microarchitecture. RAPL is available in newer versions of the Xeon server-level CPUs and provides sensors that allows measuring the power consumption of the CPU-level components listed in Table I. These available counters limit measurements to CPU and memory controller power consumption. It is impossible to measure energy consumption of I/O devices, but we hope that device vendors will follow Intel's lead and provide energy related information through their device specific interfaces. RAPL sensors can be configured and examined by reading Machine-Specific Registers (MSRs). On the Intel architecture this is only possible in privileged kernel mode. Hence, we require kernel-level support for energy readings.

RAPL_PKG	Whole CPU package
RAPL_PP0	Processor cores only
RAPL_PP1	A specific device in the unicore
RAPL_DRAM	Memory controller

TABLE I: List of available RAPL sensors.

D. PAPI (Performance API)

PAPI provides an interface to get information from underlying hardware counters. PAPI provides flexibility of using both low level and high level programming of the hardware counters to get the required information. The PAPI interface can be used as an abstraction for the underlying hardware counters. There are several countable events available in PAPI, which can be measured using several functions available within PAPI. There is a provision of measuring single events or multiple events at a time. When multiple events need to be measured then these hardware events should be formed as group. These groups are called Event Sets. Table II gives the list of interfaces used in our experimentation to access various hardware counters.

Event Code	**Event,Description**
PAPI_L1_DCM	Level 1 Data Cache Misses
PAPI_L1_TCM	Level 1 Cache Misses
PAPI_L2_DCM	Level 2 Data Cache Misses
PAPI_L2_TCM	Level 2 Cache Misses
PAPI_L2_DCH	Level 2 Data Cache Hits
PAPI_L2_DCA	Level 2 Data Cache Accesses
PAPI_L2_TCA	Level 2 Cache Accesses
PAPI_L3_TCM	Level 3 Cache Misses
PAPI_L3_TCA	Level 3 Cache Accesses
PAPI_TLB_DM	Data translation lookaside buffer misses
PAPI_TLB_IM	Instruction translation lookaside buffer misses
PAPI_flops	Number of floating point operations per second

TABLE II: PAPI interfaces to access hardware counters.

IV. Experimentation

This section discusses the experimental set up in detail. We used Intel Xeon for our work. Table III shows the detail of the processor.

CPU Type	Intel Xeon 2.5Ghz
Total Sockets	2
Cores per Socket	6
Threads per core	2
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	15360K

TABLE III: Detail of system configuration.

We used NAS Parallel Benchmarks (NPB) and Barcelona OpenMP Task Suite (BOTS) [2] for our work. The NAS Parallel Benchmarks (NPB) are a small set of programs designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications and consist of eight kernels given below:

1) IS - Integer Sort, random memory access.
2) EP - Embarrassingly Parallel.
3) CG - Conjugate Gradient, irregular memory access and communication.
4) MG - Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive.
5) FT - discrete 3D fast Fourier Transform, all-to-all communication three pseudo applications.
6) BT - Block Tri-diagonal solver.
7) SP - Scalar Penta-diagonal solver.
8) LU - Lower-Upper Gauss-Seidel solver.

BOTS benchmarks vary in terms of granularity, number of task directives, support for nested tasks, and computation structure. This variation is beneficial to comprise all possibilities. A brief description of these benchmarks is followed:

1) Fibonacci: Computes the n^{th} Fibonacci number using a recursive paralellization. It represents a deep tree composed of very fine grain tasks.
2) FFT: Is a divide and conquer algorithm that computes the one-dimensional Fast Fourier Transform of a vector of n complex values. In each of the divisions multiple tasks are generated.
3) Strassen: Uses hierarchical decomposition of a matrix for multiplication of large dense matrices. For each decomposition, which is dividing each dimension into two halves, a task is created.
4) Sort: Sorts a random permutation of n 32-bit numbers with a variation of sorting algorithms. Tasks are used for each split and merge.
5) Health: Uses multilevel lists where each element in the structure represents a village with a list of potential patients and one hospital. The hospital has several double-linked lists representing the possible status of a patient inside it. A task is created for each village.
6) Alignment: Aligns sequences of proteins using dynamic programming.
7) Nqueens: Finds solutions of the n-queens problem using backtrack search.
8) SparseLU: Computes the LU factorization of a sparse matrix.

To understand the complete behaviour of a given OpenMP program, one needs to select all the variables that control it's performance. These variables or features can be divided into six layers shown in Figure 2. In the Application Layer, we have two variables; type of application and input data size. Our benchmarks cover all types of OpenMP applications. Data sizes associated with each benchmark also give enough variation to study the characteristics of these applications under various data loads. In the Compiler Layer, we have different optimization levels and strategies that control the execution performance of a compiled code. In the OpenMP API layers, we have different OpenMP directives that allow a user to map parallelism in a code on a given hardware in different ways. Then we have different OpenMP runtime environment variables in the OpenMP Runtime Layer which are part of the OpenMP standards. The Hardware Layer can be treated as constant with no variables if one processor is being targeted. For our experimentation, we assume that the OS Layer also does not have any effect on the performance of a given code. To ensure this we performed our experimentation when no other applications were running on the machine.

Table IV gives the complete search space which include the variables that we used in our experimentation. Some variables are specific to the OpenUH OpenMP runtime library. In the following, we describe the OpenUH specific variables:

- $O64\_OMP\_TASK\_POOL$: Gives control over the organization of task queues; hence, it has an influence on efficiency, queue contention and work-stealing. Four different options can selected:
    1) DEFAULT: Each thread has two queues; One for tied tasks and another for untied tasks.
    2) SIMPLE: Each thread has one queue which holds any unscheduled tied or untied tasks that it creates.
    3) SIMPLE_2LEVEL: Each thread has a fixed-size queue for the tasks it creates. A global queue can be used when the private queue is filled.
    4) PUBLIC_PRIVATE: Each thread has a public and private queue. An environment variable, $O64\_OMP\_TASK\_POOL\_GREEDVAL$, is used to control how many tasks are placed in the private and public queues.
- $O64\_OMP\_TASK\_QUEUE$: Influences how tasks are scheduled. The types of queues used are:
    1) DEQUE: Puts task to the tail of the queue, gets task from the tail of the queue, steals task from the head of the queue, donates task to the head of the queue.
    2) FIFO: Puts and donates tasks to the tail of the queue, gets and steals tasks from the head of the queue.
    3) CFIFO: same as FIFO, except more efficient because it allows concurrent access to head and tail of a given queue.
    4) LIFO: Puts and donates tasks to the tail of the queue, gets and steals tasks from the tail of the queue.
    5) INV_DEQUE: Puts task to the tail of the queue, gets task from the head of the queue, steals task from the tail of the queue, donates task to the head of the queue.

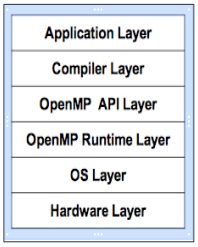

Fig. 2: OpenMP performance layers.

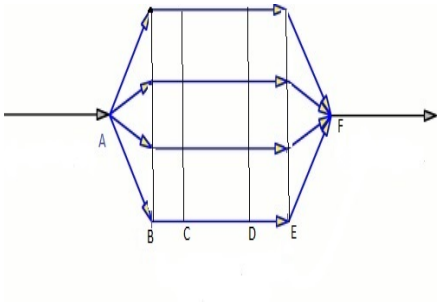

Fig. 3: OpenMP execution model. B, C, D and E are fine level OpenMP events. A and F are coarse level OpenMP events.

Figure 3 describes the OpenMP execution model. The previous work in this area study the power performance of an OpenMP kernel by using RAPL at Point A and F, i.e., at the fork and join events. We call it coarse level. This approach gives an average power performance of a given kernel. Using

	Environment Variables	Chunk Size	Optimization Strategy	No. of Threads
Work Sharing	OMP_SCHEDULE=STATIC	10,100	O0,O1,O2,O3	2,4,8,16,24
	OMP_SCHEDULE=DYNAMIC	1,10,100	O0,O1,O2,O3	2,4,8,16,24
	OMP_SCHEDULE=GUIDED	1,10,100	O0,O1,O2,O3	2,4,8,16,24
	OMP_DYNAMIC=TRUE	NA	O0,O1,O2,O3	2,4,8,16,24
	OMP_DYNAMIC=FALSE	NA	O0,O1,O2,O3	2,4,8,16,24
	OMP_WAIT_POLICY=PASSIVE	NA	O0,O1,O2,O3	2,4,8,16,24
	OMP_WAIT_POLICY=ACTIVE	NA	O0,O1,O2,O3	2,4,8,16,24
OpenMP Tasks	OMP_SCHEDULE=STATIC	NA	O0,O1,O2,O3	2,4,8,16,24
	OMP_SCHEDULE=DYNAMIC	1,10,100	O0,O1,O2,O3	2,4,8,16,24
	OMP_SCHEDULE=GUIDED	1,10,100	O0,O1,O2,O3	2,4,8,16,24
	O64_OMP_TASK_QUEUE	DEQUE,FIFO,LIFO,CFIFO,INV_DEQUE	O0,O1,O2,O3	2,4,8,16,24
	O64_OMP_TASK_POOL	DEFAULT,SIMPLE,SIMPLE_2LEVEL,PUBLIC_PRIVATE,GREEDVAL=1	O0,O1,O2,O3	2,4,8,16,24

TABLE IV: Search space for OpenMP performance analysis.

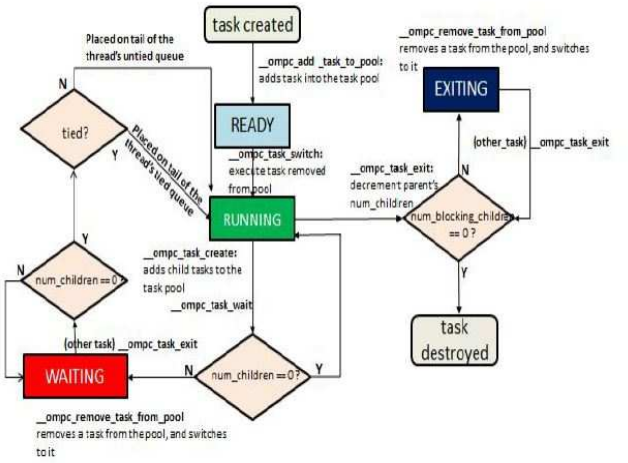

Fig. 4: Task Scheduling Model in the OpenUH RTL

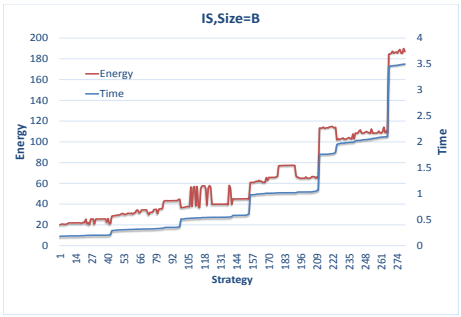

Fig. 5: IS benchmark application with data size=B.

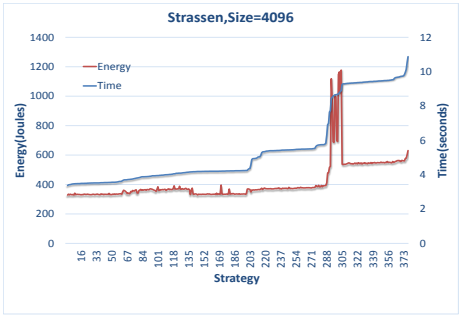

Fig. 6: Strassen benchmark application with data size=4096.

ORA, we are able to observe the power performance at the fine level, i.e., at Point B, C and D in Figure 3. These points can be any OpenMP events, e.g., events like task creation, start of a barrier, end of a barrier etc. Figure 4 gives the task execution model for the OpenUH compiler runtime library. It shows different states through which an OpenMP task goes during its life-cycle. It is important to know how these states effect the energy usage of a given kernel. What are the states that can be optimized in order to improve the power performance. Using ORA, we are able to calculate energy consumption associated with the OpenMP task stages.

V. Results and Analysis

In this section, we present our experimental results. We used 13 applications from NAS Parallel and BOTS benchmarks. However, due to space constraint we will present detail analysis of benchmark applications showing interesting results[1]. Figure 5, 6, 7 and 8 combine performance behaviour and energy usage behaviour of IS, EP, Fibonacci and Nqueens applications respectively. These figures give the reader an idea how the energy usage requirement changes with execution performance. The x-axes on these figures give different strategies, i.e., *different combinations of the variables from Table IV in Section IV*. During the experimentation, we ran each point three times. The average value was used to plot the information. The strategy points on the x-axes are sorted with respect to time. Hence, the first point gives the best execution performance and the last point gives the worst execution performance. The y-axes give the respective energy performance in Joules. Figure 5 to 8 give very interesting information regarding energy usage. One can see lots of spikes for energy performance. These spikes are very clear in Figure 6, 7 and 8. A clear deep dip in the energy usage can be seen in Figure 6. In order to ensure that we did not get this point accidently, we ran this specific point three times and got the same behaviour. Similar trend was observed for the other benchmark applications with respect to energy usage and execution time. This observation clearly shows that the best execution performance does not necessarily ensures the best energy performance.

Now the question arises; what features are responsible for the best and worst energy performance of a given program?

[1]Detail analysis will be presented in a journal paper.

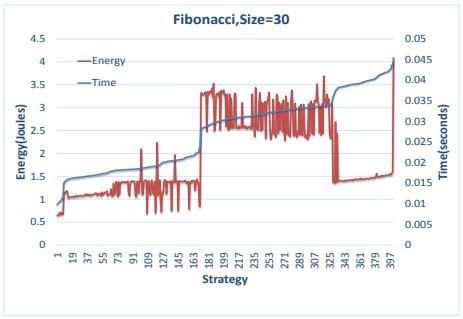

Fig. 7: Fibonacci benchmark application with size=30.

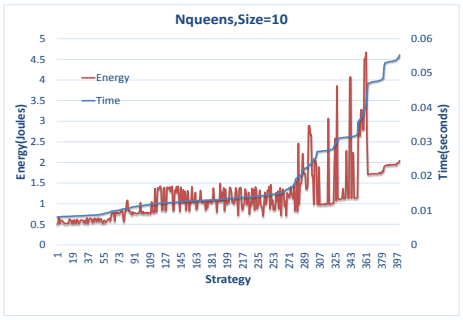

Fig. 8: Nqueens benchmark applications with size=30.

	Environment Variable	Ranking Percentage
IS	Static,100	20%
	Dynamic,1	15%
	Dynamic,False	15%
EP	Dynamic,True	14%
	Guided,1	7%
	Wait policy,active	7%
Fibonacci	TASK_POOL,SIMPLE	20%
	TASK_POOL_GREEDVAL=1	17%
	TASK_POOL,DEFAULT	9%
Nqueens	TASK POOL,PUBLIC PRIVATE	30%
	TASK_POOL,DEFAULT	23%
	Dynamic,100	12%
Strassen	TASK POOL,PUBLIC PRIVATE	50%
	TASK QUEUE,CFIFO	32%
	TASK_POOL,DEFAULT	5%
FloorPlan	TASK POOL,PUBLIC PRIVATE	74%
	TASK_POOL,SIMPLE	11%
	TASK_POOL,SIMPLE_2LEVEL	7%
Alignment	TASK_POOL,SIMPLE_2LEVEL	20%
	TASK POOL,PUBLIC PRIVATE	15%
	TASK_POOL,SIMPLE	15%

TABLE V: Top features for the best energy usage strategies.

	Environment Variable	Ranking Percentage
IS	Dynamic,True	40%
	Wait Policy=Passive	18%
	Dynamic,False	17%
EP	Dynamic	10%
	Dynamic,True	9%
	Guided,100	9%
Fibonacci	TASK QUEUE,CFIFO	22%
	TASK QUEUE,INV_DEQUE	16%
	TASK POOL,PUBLIC PRIVATE	8%
Nqueens	TASK QUEUE,INV_DEQUE	22%
	TASK QUEUE,DEQUE	15%
	TASK_POOL,DEFAULT	15%
Strassen	TASK POOL,PUBLIC PRIVATE	65%
	TASK QUEUE,FIFO	15%
	TASK QUEUE,LIFO	10%
Floorplan	TASK_POOL,SIMPLE_2LEVEL	38%
	TASK_POOL,SIMPLE	30%
	TASK QUEUE,FIFO	15%
Alignment	TASK POOL,PUBLIC PRIVATE	9%
	TASK QUEUE,CFIFO	7%
	TASK QUEUE,LIFO	7%

TABLE VI: Top features for the worst energy usage strategies.

The answer to this question needs a detail statistical analysis of the features that control the search space. However, we did a very simple analysis to answer this question. We sorted the points according to the best energy usage performance for each application and then calculated the frequency of each feature in the top 30% points. Table V presents this analysis. For IS application, the top feature is static scheduling strategy as it appears in 20% of the top 30% best points. Similarly, we sorted the points according to the worst energy usage performance and calculated the frequency of each feature in the top 30% points. Table VI gives this analysis. For IS application, the top feature is dynamic scheduling strategy as it appears in 40% of the worst energy usage points.

We also analyzed the energy usage profile at the fine level using ORA. We collected the energy at the core and package levels using RAPL senssors. For OpenMP work sharing, the following OpenMP events were used to collect the fine level information:

1) Fork event
2) Begin of Barrier Event
3) End of Barrier Event
4) Join Event

For OpenMP tasking, the following OpenMP events were used to collect the fine level information:

1) Fork Event
2) Task create Event
3) Begin task suspend Event
4) End task suspend Event
5) Join Event

Figure 9 gives power profile for the best and worst energy usage points for IS application. Bar A in Figure 9 shows the power consumption between fork and begin of implicit barrier events. Bar B gives the power consumption between the begin of implicit barrier and end of implicit barrier events. Bar C gives the power consumption between the end of implicit barrier and join event. One can see that the best energy usage point gives high power consumption while the worst point gives low power consumption. Also the average power consumption for the best energy usage point is more than the worst energy usage point. The reason; power is inversely propotional to time. Although the best point is giving less energy usage, however it has small execution time which makes the rate of energy usage, i.e. power, high.

Figure 10 gives power profile for the best and worst energy usage points for EP application. The profile is very much similar to IS application. We carefully analyzed the worst energy points of each NAS parallel benchmark applications. We found that these points have high barrier and queue waiting times. Figure 11 and Figure 12 show the memory profile for the best and wort energy usage points for EP and IS applications respectively. One can observe that the worst energy usage points have high cache miss rates which is also

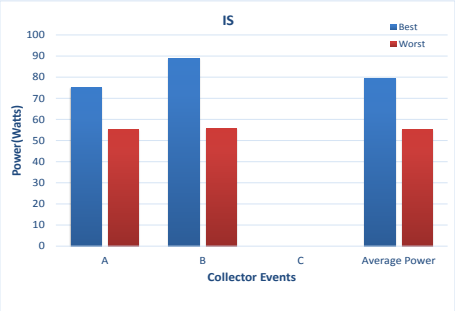

Fig. 9: Power profile for the best and worst energy usage points for IS.

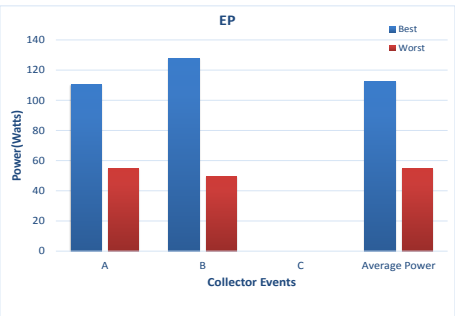

Fig. 10: Power profile for the best and worst energy usage points for EP.

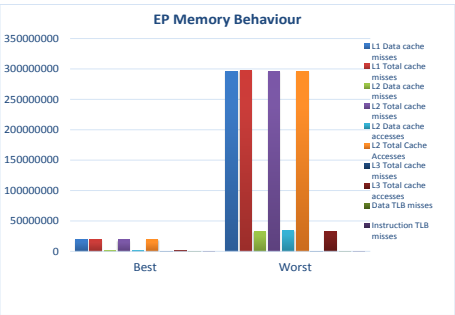

Fig. 11: Memory profile for the best and worst energy usage points for EP.

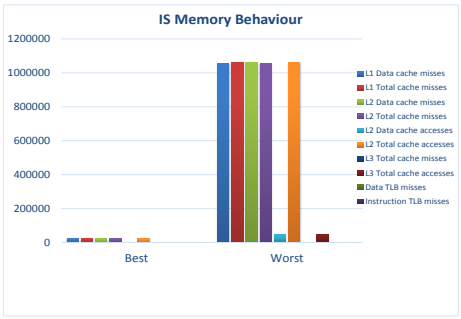

Fig. 12: Memory profile for the best and worst energy usage points for IS.

Fig. 13: Power profile for the best and worst energy usage points for Alignment.

responsible for the high waiting time.

Figure 13 gives power profile for the best and worst energy usage points for Alignment application from the BOTS benchmark. Bar A in Figure 13 shows the power consumption between fork and task creation events. Bar B gives the power consumption between task creation and start of task suspension events. Bar C shows the power consumption between start of task suspension and end of task suspension. Bar D gives the power consumption between end of task suspension and finishing of task. Bar E gives the power consumption between finishing of task and join event.

Figure 14, 15,16, 17 gives power profile for the best and worst energy usage points for Fibonacci, Nqueens, Floorplan and Strassen respectively. Figure 18 to Figure 21 give the memory profile for the best and wort energy usage points for the BOTS benchmark applications. One can observe that the worst energy usage points have high cache miss rates.

We also analyzed the scalability of the energy usage for the benchmark applications. For this, we concentrated on the best and worst energy usage points. For a given number of threads, we calculated the FLOPS/W for the best and worst energy usage points for each applications. Figure 22 gives this information for three benchmark applications. Interesting observation is that the performance decreases for the best energy usage points. However, for the worst energy usage points, it is almost constant.

During our analysis, we also tried to observe whether there exist a unique pattern of energy usage or power consumption for OpenMP kernels? If one studies the above figures associated with power profile for each benchmark application, one can see that this uniquness exists. We found that this

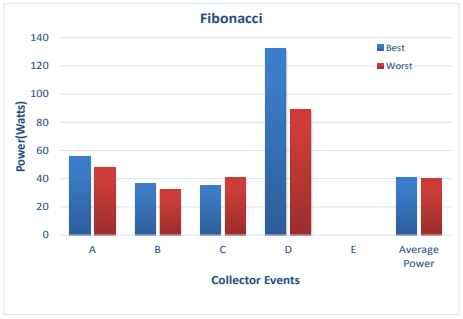

Fig. 14: Power profile for the best and worst energy usage points for Fibonacci.

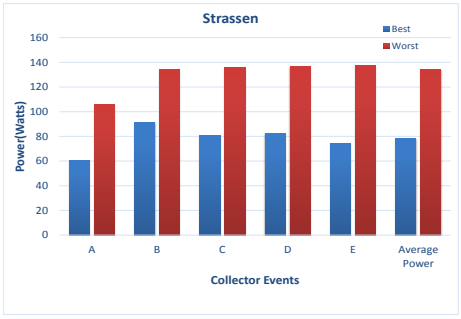

Fig. 17: Power profile for the best and worst energy usage points for Strassen.

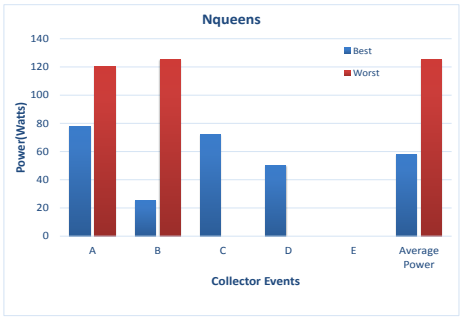

Fig. 15: Power profile for the best and worst energy usage points for Nqueens.

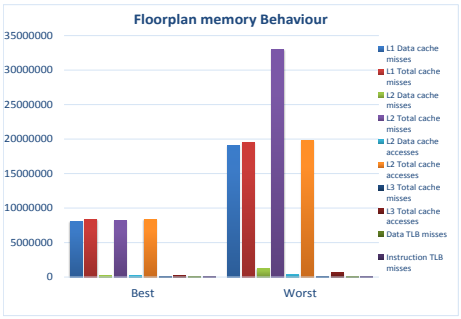

Fig. 18: Memory profile for the best and worst energy usage points for Floorplan.

Fig. 16: Power profile for the best and worst energy usage points for Floorplan.

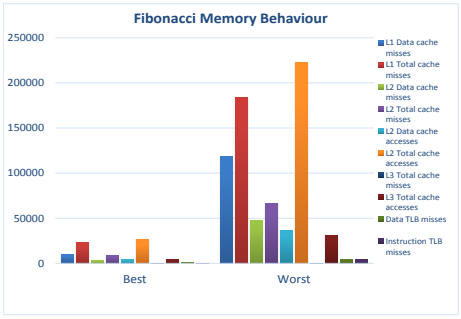

Fig. 19: Memory profile for the best and worst energy usage points for Fibonacci.

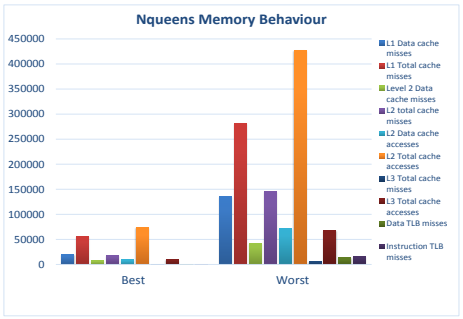

Fig. 20: Memory profile for the best and worst energy usage points for Nqueens.

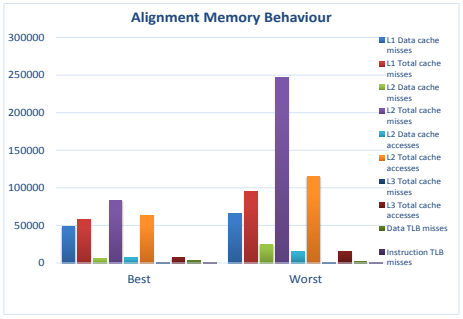

Fig. 21: Memory profile for the best and worst energy usage points for Alignment.

uniquness is weak for the NAS Parallel applications. However, strong patterns were found for the BOTS applications. Our observation is that these patterns define unique characteristcs of OpenMP applications. These patterns have a potential tobe used as features to build an efficient cost model for a dynamic power management system for thr future HPC systems.

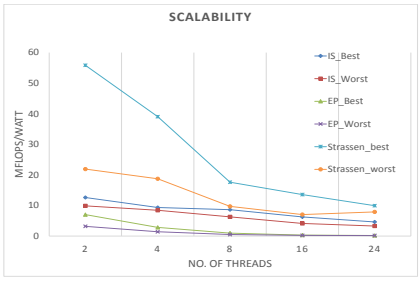

Fig. 22: MFlops/W performance for IS, EP and Strassen benchmark applications.

VI. Conclusion and Future Work

Buidling future generation supercomputers while constraining their power consumption is one of the biggest challenges faced by the HPC community. US Department of Energy has set a goal of 20 MW for an exascale supercomputer. To realize this goal, much of research is being done to revolutionize hardware design to build power efficient computers and network interconnects. From the software side, people are looking into characteristics of different parallel programming models. OpenMP programming model is an integral part of the future HPC and hence it is important to look into this model with respect to energy performance.

In this paper, we presented a comprehensive analysis of key OpenMP programs using ORA interfaces. We analyzed behaviour of the programs for performance and energy usage by varying data loads, compiler optimization levels and runtime environment variables. We found that the best performance does not ensure the best energy usage. Also, waiting time at the barriers and queues are the main factors for high power consumption for a given OpenMP program. Our results also showed that for a given OpenMP kernel, there exists a power pattern that can be used to build dynamic power management strategy.

For the future work, we are planning to do a detail performance analysis for a real time application. We are also aiming to build a prediction model for dynamic power management system. We are also planning to do power and energy analysis with respect to communication with-in cores and between cores and memory hierarchy.

Acknowledgements

The authors would like to thank their colleagues in the HPCTools group for their extensive collaboration to make this work a reality. This work is supported by the National Science Foundation under grant CCF-1148052.

References

[1] B. Chapman, D. Eachempati, O. Hernandez. *Experiences Developing the OpenUH Compiler and Runtime Infrastructure*. International Journal of Parallel Programming, Volume 41, Issue 6, pp. 825-854, August 2013.

[2] A. Duran, X. Teruel, R. Ferrer, X. Martorell, E. Ayguade. *Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp*. In Parallel Processing, 2009. ICPP09. International Conference on. IEEE, 2009, pp. 124131.

[3] Al. Eichenberger, J. Crummey, M. Schulz, N. Copty, J. DelSignore, R. Dietrichk, X. Liu,E. Loh,D. Lorenz. *OMPT and OMPD: OpenMP Tools Application Programming Interfaces for Performance Analysis and Debugging.*,2013.

[4] J. Flinn, M. Satyanarayanan. *Energy-aware adaptation for mobile applications*. In SOSP 1999: Proceedings of the 17^{th} ACM Symposium on Operating Systems Principles. ACM, 1999.

[5] V. W. Freeh, N. Kappiah, D. K. Lowenthal, T. K. Bletsch. *Just-intime dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs*. Journal of Parallel and Distributed Computing, vol. 68, no. 9, 2008.

[6] V. W. Freeh, D. K. Lowenthal. *Using multiple energy gears in MPI programs on a power-scalable cluster*. In PPoPP 2005: Proceedingds of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2005.

[7] R. Ge, X. Feng,K. W. Cameron. *Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters*. In SC'05: Proceedings of the 2005 ACM/IEEE Conference on High Performance Networking and Computing. IEEE Computer Society, 2005.

[8] R. Ge, X. Feng, W. Feng, K. Cameron. *CPU miser: A performance-directed, run-time system for power-aware clusters*. In ICPP 2007: 36^{th} International Conference on Parallel Processing. IEEE, 2007.

[9] http://web.cs.uh.edu/ hpctools/

[10] O. Hernandez, V. Bui, R. Kufrin, R. Nanjegowda, B. Chapman. *Open Source Software Support for the OpenMP Runtime API for Profiling*. In Proceedings of The Second International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2), pp.130-137, September, 2009, Vienna, Austria.

[11] C. Hsu, W. Feng. *Power-aware run-time system for highperformance computing*. In SC'05: Proceedings of the 2005 ACM/IEEE Conference on High Performance Networking and Computing. IEEE Computer Society, 2005.

[12] S. Huang,W. Feng. *Energy-efficient cluster computing via accurate workload characterization*. In CCGrid 2009: Proceedings of the 9^{th} IEEE/ACM International Symposium on Cluster Computing and the Grid. IEEE Computer Society, 2009.

[13] M. Itzkowitz, M. Mazurov, O. Copty, N. Lin, Y. Lin. *An OpenMP runtime API for profiling.* OpenMP ARB as an official ARB White Paper 314, 181190,2007.

[14] D. Li, B. R. de Supinski, M. Schulz, K. W. Cameron, D. S. Nikolopoulos. *Hybrid MPI/OpenMP power-aware computing*.In IPDPS 2010: Proceedings of the 24^{th} IEEE International Symposium on Parallel and Distributed Processing, 2010.

[15] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, J. F. Prins. *OpenMP task scheduling strategies for multicore NUMA systems*. International Journal of High Performance Computing Applications, vol. 26, no. 2, May 2012.

[16] P. Pillai, K. G. Shin. *Real-time dynamic voltage scaling for lowpower embedded operating systems*. In SOSP'01: Proceedings of the 18^{th} ACM Symposium on Operating Systems Principles, 2001.

[17] A. Porterfield, S. Olivier, S. Bhalachandra, J. Prins. *Power Measurement and Concurrency Throttling for Energy Reduction in OpenMP Programs*. In the Proceeding of 8^{th} IEEE Workshop on High-Performance, Power-Aware Computing (HP-PAC 2013). IEEE: Boston, MA, May 2013.

[18] A. Qawasmeh, A. Malik, B. Chapman. *OpenMP Task Scheduling Analysis via OpenMP Runtime API and Tool Visualization.* In the 28^{th} IEEE International Parallel and Distributed Processing Symposium: Programming Models, Languages and Compilers Workshop for Manycore and Heterogeneous Architectures (PLC2014), 2014.

[19] B. Rountree, D. H. Ahn, B. de Supinski, D. K. Lowenthal, M. Schulz. *Beyond DVFS: A first look at performance under a hardware-enforced power bound*. In HP-PAC 2012: Proceedings of the 8^{th} Workshop on High Performance, Power-Aware Computing, May 2012.

[20] B. Rountree, D. K. Lowenthal, B . R. de Supinski, M. Schulz, V. W. Freeh, T. K. Bletsch. *Adagio: Making DVS practical for complex HPC applications*. In ICS'09: Proceedings of the 23^{rd} International. Conference on Supercomputing, 2009.

[21] O. Sarood, A. Langer, A. Gupta, L. Kale. *Maximizing Throughput of Overprovisioned HPC Data Centers Under a Strict Power Budget*. Accepted at the Supercomputing SC14, 2014.

[22] A. Tiwari, M. Laurenzano, J. Peraza, L. Carrington, A. Snavely, *Green queue: Customized large-scale clock frequency scaling*. In CGC'12: Proceeings of the 2^{nd} International Conference on Cloud and Green Computing,Nov. 2012.

[23] Y. Yan, J. Kemp, X. Tian, A. M. Malik, B. Chapman. *Performance and Power Characteristics of Matrix Multiplication Algorithms on Multicore and Shared Memory Machines.* In Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA) in conjunction with International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12), 2012.