Abstract—Application level power budget allocation is one way to overcome the power constraint problem in future HPC systems. This technique mainly depends on finding an optimal number of compute nodes and power level for each node. However, utilizing that power at node level requires optimization of the underlying programming model. OpenMP is the de-facto standard for intra-node parallelism. In this paper, we investigate the impact of OpenMP runtime environment on the performance of OpenMP code at the different power level. We studied 28 OpenMP parallel regions from five NAS Parallel Benchmark (NPB) applications. Based on the study we show that for a given power level, a suitable selection of OpenMP runtime parameters can improve the execution time and energy consumption of a parallel region up to 67% and 72%, respectively. We also show that these fine grain improvements resulted in upto 26% execution time and 38% energy consumption improvement for a given OpenMP application.

I. INTRODUCTION

Power consumption has become a critical design factor for a large scale HPC system. It is going to be one of the biggest constraints for future systems if we are to reach exascale computing within 20-30 megawatts, a goal set by US Department of Energy. One way to design a power constrained system is through overprovisioning of hardware nodes [1]. In such systems, an application is allocated a strict power budget and a flexible node budget.

Recent advances in processor and memory hardware designs have allowed the user to control the power consumption of CPU and memory through software. For example, the power consumption of Intel Sandy Bridge family of processors can be controlled by the user through the Running Average Power Limit (RAPL) interface [2]. This ability to constrain the maximum power consumption of the subsystems allows a user to add more nodes while ensuring that the total power consumption of an application does not exceed its allocated power budget. It has been observed that in an overprovisioned power constrained system, most of the applications benefit from an overall performance improvement [1]. This improvement comes from the use of extra nodes within the same power budget. This fact led a lot of researchers [1]–[3] to focus on finding the best number of nodes and the best power level for each node for a given power budget. Performance improvement per node at various power levels did not get much attention. However, we believe that exploring maximum potential of an individual node will further improve the overall performance of an application in an overprovisioned system.

With hardware capability being constrained by power, it is important to adjust the execution environment of the underlying programming model to get the best performance out of an individual node. Since OpenMP is the de-facto programming model for intra-node parallelism, it is important to explore the execution environment behavior of OpenMP in a power constrained system. OpenMP provides certain performance adjustment runtime parameters that can be used to control the OpenMP execution environment [4]. Different OpenMP parallel regions have different execution behavior. Therefore, they require different runtime execution environment that gives the best performance [5]. If the OpenMP runtime parameters are not properly selected, one may see a severe performance degradation. Most of the application developers choose to use a default parameter setting provided by an OpenMP runtime library. As a result, one gets sub-optimal performance from a lot of the existing OpenMP applications. There are two main reasons behind this reluctance to change the OpenMP execution environment. First, the scarcity of quantitative study about the impact of the OpenMP runtime parameters on OpenMP performance in a power constrained system. Second, the complexity of choosing the best OpenMP runtime parameters for a given parallel region. This paper is focused on the first part.

In this paper, we provide a detailed analysis on the impact of the OpenMP runtime parameters on a given OpenMP parallel region’s execution time and energy consumption. We only consider loop based OpenMP parallel regions, as most of the existing OpenMP applications use loop-based parallelism. We show that a suitable runtime parameter selection at a particular power level can improve the execution time and energy consumption of a loop based parallel region significantly compared to the default OpenMP runtime environment provided by an OpenMP runtime library. It can go up to 67% in execution time improvement and 72% in energy consumption improvement. In this work, we consider three main OpenMP runtime parameters: Number of Threads, Scheduling Policy, and Chunk Size. We choose these three parameters because they can be changed per parallel region basis through OpenMP runtime routines.

The major contributions of this work are listed below:

- The work provides a detailed analysis of the OpenMP runtime configurations under various power levels which,
to the best of our knowledge is the first work in this area.
- The work provides a quantitative experimental study on the impact of OpenMP runtime environment on execution time and energy consumption.
- We answer several interesting questions which are pertinent to the HPC community interested in the power aware optimization using detailed analysis.
- We use two different OpenMP runtime libraries to validate our findings.

II. Motivation

The OpenMP programming model is an integral part of many important HPC legacy codes in the form of hybrid programming models (e.g. MPI + OpenMP). Therefore, tuning an OpenMP code to get a better per node performance for a given power budget is an important research problem. In this section, we motivate the reader about this problem by studying the performance behavior of an OpenMP code under various power levels.

We took a loop based OpenMP parallel region from the SP benchmark application of NPB. The parallel region belongs to the compute_rhs function, and has 11 different parallel loops, i.e., \#pragma omp for directives. We ran it at different power levels or power caps\(^1\) using a different number of threads, scheduling policies, and chunk sizes (148 different configurations).

![Fig. 1: Execution time comparison of different OpenMP runtime configurations at different power levels for the compute_rhs parallel region of SP.](image)

**Definition 1** (Optimal Configuration). An OpenMP runtime configuration within the configuration search space that gives the best execution time at a certain power level.

**Definition 2** (Default Strategy). An OpenMP runtime configuration used by an OpenMP runtime library by default.

Figure 1 shows the comparison of execution time using the optimal configuration (see Definition 1) and the default configuration (see Definition 2) at different power levels. We used the OpenUH OpenMP runtime library for this experimentation.

\(^1\)We use the two words synonymously in this paper.

For the default configuration\(^2\), the OpenUH runtime library uses the maximum number of available threads, static scheduling policy, and a chunk size that is calculated dynamically by dividing the total number of loop iterations by the number of threads. Figure 1 presents two key findings. **First**, the optimal configuration is different from the default configuration at all power levels. The optimal configuration improves the execution time of the parallel region up to 19% compared to the default configuration at the same power level. **Second**, the optimal configuration at a lower power level yields better execution time performance than the default configuration at the maximum power level or at Thermal Design Power (TDP). For example, the optimal configuration at 70\,W power cap improves execution time by 15% as compared to the default configuration at TDP (115\,W). These observations are useful for the improvement of OpenMP performance in a power constrained system.

Therefore, a detailed study of the OpenMP runtime parameters under different power levels for a power constrained system is necessary.

III. Configuring the Experimental Search Space

Here, we briefly describe our experimental parameters and how we performed our experimentation. Our experimentation mainly revolved around two main concepts, OpenMP runtime parameter configuration & Power capping/clamping level. First, we describe these two concepts and then we discuss how we worked with these concepts.

**A. OpenMP Runtime Configuration**

OpenMP is the de-facto parallel programming model for shared memory programming. In OpenMP, a certain region or block of a code is parallelized to run on multiple threads. These regions are called parallel regions. The main parameter of a parallel region is the number of threads that’s going to execute it. Now inside the parallel region, work can be divided among different threads in many different ways. These are called work sharing constructs. Although in OpenMP there are different work sharing constructs such as tasks for irregular parallelism (e.g. recursive algorithms), sections, OpenMP’s main impact comes from its ability to efficiently parallelize loops. OpenMP parallelizes a loop by effectively dividing the loop iterations among different threads. In this way, each thread executes only a portion of the total iterations. In this work, we mainly considered OpenMP’s loop level parallelism. Along with the number of threads, there are two more parameters that are used to tweak how a specific loop is going to be parallelized. They are Scheduling Policy & Chunk Size. These two parameters are used together. Combined, they describe how the loop iterations are divided among the threads. Scheduling policy describes a way to divide the work among the threads, while chunk size controls how many of the iterations are assigned to each thread at once. There are 5 existing scheduling policies as of the current OpenMP 4.5

\(^2\)Different OpenMP runtime libraries may have different default strategies.
standard [6]. Table I provides a brief overview of different types of scheduling policies. Chunk sizes are non-negative integer values which are less than or equal to the total number of loop iterations.

**TABLE I: OpenMP Loop Scheduling Policy Overview**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>Loop iterations are divided into pieces of size chunk and then statically assigned to threads. If a chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>Loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads. When a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.</td>
</tr>
<tr>
<td>GUIDED</td>
<td>Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread. The chunk parameter defines the minimum block size. The default chunk size is 1.</td>
</tr>
<tr>
<td>RUNTIME</td>
<td>The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE.</td>
</tr>
<tr>
<td>AUTO</td>
<td>The scheduling decision is delegated to the compiler and/or runtime system.</td>
</tr>
</tbody>
</table>

In our experimentation, we only STATIC, DYNAMIC, GUIDED and RUNTIME. We did not use AUTO, because AUTO delegates the scheduling decision to the compiler and/or runtime system. As a result, there is no way for us to know how a certain loop is scheduled. This stops us from analyzing and characterizing a loop behavior.

**B. Power Capping or Power Clamping**

Intel Sandybridge and subsequent processors have introduced a technique called power capping to limit the power consumption of different subsystems (CPU, memory, GPU) through RAPL [7] interface. This does not limit the power consumption in the strictest sense, rather it guarantees that it will not exceed the average power usage given a time window and power limit. Different processor versions offer power capping on different sub systems. Our experimental machine does not allow DRAM power capping, so we only capped the CPU power (Package Power). Also, each subsystem has a minimum cap size, as for our system 54W is the minimum power cap size and TDP of our system is 115W. We picked 5 power cap levels (55W, 70W, 85W, 100W and 115W) within this limit. Also, the minimum time window to cap the power for the system is 0.000977s; in all our experiments, we used this minimum limit to cap the power of the CPU sub system. We used libmsr [8], a library developed at Lawrence Livermore National Laboratory to cap the CPU power of our system.

**C. Creating the Search Space**

Our goal in this experiment was to create a comprehensive configuration search space of the above explained 4 parameters (Power Capping level, number of threads, Scheduling Policy & Chunk sizes). We tried to create the search space in a way that is manageable and still large enough to provide us key insight on the impact of these parameters on a specific OpenMP parallel region. We used a parallel region as the basic block for our measurement.

Although there are situations where multiple parallel loops can be part of a single parallel region, we assumed all the parallel loops inside a single parallel region conforms to a single optimal configuration. There are two main reasons for this. First, number of threads is a parallel region parameter rather than a parallel loop parameter. By that, we mean that designated number of threads are created before execution enters a specific parallel region and those threads are used during the execution of the parallel loop inside that parallel region. Second, it provides simplicity during experimentation and measurement. For example, a single loop may not be large enough to have a power cap impact (described later), but all the loops inside a parallel region combined may be large enough to have a power cap impact. It is also easier to collect information for a parallel region than a specific loop inside a parallel region.

Table II shows the different values of these 4 parameters we used during our experimentation. We used all possible combination of these given parameters. So, we used 735 (5*7*3*7) different configurations on each parallel region. We changed these configurations on the runtime.

**TABLE II: Configuration parameter details**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Used Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Cap Size (5)</td>
<td>55W, 70W, 85W, 100W &amp; 115W</td>
</tr>
<tr>
<td>Number of threads (7)</td>
<td>1, 2, 4, 8, 16, 24 &amp; 32</td>
</tr>
<tr>
<td>Scheduling Policy (3)</td>
<td>STATIC, DYNAMIC &amp; GUIDED</td>
</tr>
<tr>
<td>Chunk Sizes (7)</td>
<td>1, 8, 32, 64, 128, 256 &amp; 512</td>
</tr>
</tbody>
</table>

**D. Measurement Details**

We developed a tool to collect execution time and energy consumption information for each OpenMP parallel region. It used RAPL API. We chose 28 OpenMP parallel regions for our analysis. The selection was made based on execution time and load balancing behavior of these parallel regions. We did not consider parallel regions with execution time less than 10 milliseconds, because it is advised to have around 10 milliseconds time window between two RAPL calls for a reliable reading [9].

We accumulated the execution time of each parallel region every time it was called throughout the application. There were two reasons for this. First, it showed the parallel region behavior on that configuration throughout the application. It is important for an application with load balancing issue across different invocations of the parallel region. For example, a parallel region can have a small workload first time it was invoked but a larger workload in later invocations. Second, we used RAPL to enforce a new power cap at the beginning of an application. The RAPL framework takes up to 200 milliseconds to warm-up before imposing a new power cap. Therefore, if a parallel region is called within this period, it may not experience the effect of the selected power level. However, in our experimentation, the parallel regions were called tens to hundreds of times during a complete run of
an application. Therefore, even if a parallel region did not experience a new power cap in its first run, it experienced the new power cap in the later runs. Thus, the accumulated execution time provided a reliable reading. Besides that, we ran all the experiments three times and averaged it to get a more reliable reading.

In our experimentation, we capped the processor power, also known as the package power. The package consists of the cores, caches and other internal circuitry. We also reported the package energy\(^3\), energy consumed by the processor package.

IV. EXPERIMENTAL SETUP

In this section, we provide an overview of our experimental environment.

A. Machine Configuration

We used a dual socket machine with two 2.4 GHz quad-core Intel® Xeon® E5-2665 processors (based on the Intel Sandy Bridge architecture). It runs on OpenSUSE 13.1. The machine has a total of 16 cores (32 hyper-threaded threads) and 16 GB of memory. Each socket has a minimum and maximum (TDP) operating processor/package power of 54W and 115W respectively specified by the manufacturer.

B. Packages and Libraries

We used two compilers for our experimentation, OpenUH [10], version 3.0.33 and Intel compiler, version 16.0. OpenUH is an open source compiler developed at the University of Houston for Fortran/C/C++. It has been used to research OpenMP features and their implementation. Intel compiler is an industry standard compiler developed by Intel corporation. Both compilers use different OpenMP runtime libraries. We used –O3 optimization level of these compilers to compile the programs. We also used TAU (Tuning and Analysis Utilities), PAPI (Performance API), OMPT (OpenMP Tools Interface) and Intel RAPL (Running Average Power Limit) for our experimentation.

C. Benchmark Applications

We used five applications from the NAS Parallel Benchmark (NPB) suite [11]. Two of the applications from NPB; BT, and SP, are proxy applications. Three applications, CG, EP, and FT, are based on computational kernels. The benchmark applications can be run with six different data sets; S, A, B, C, D, and E. S is the smallest, and E is the largest data set. We used data set B for our experimentation. Table III provides a brief overview of these applications.

V. IMPACT OF OPTIMAL RUNTIME CONFIGURATION ON A POWER CAPPED NODE

In this section, we present our results and analysis using 28 parallel regions from the NPB applications. Due to the space constraint, we present the results primarily with OpenUH runtime. However, we use Intel runtime results to generalize the findings. We determined the optimal configuration of each parallel region based on its execution time. Some parallel regions showed the optimal performance with the default configuration, and some showed the optimal performance with a non-default configuration. Based on this observation, we divided the parallel regions into the following two groups:

- **Default parallel regions** are the parallel regions whose optimal configuration is the default configuration.
- **Non-Default parallel regions** are the parallel regions whose optimal configuration is different from the default configuration.

With OpenUH runtime, we found 35% of the total parallel regions as *Non-Default parallel regions* across all power levels. For example, at 85W power level, 10 out 28 parallel regions were *Non-Default parallel regions*. These *Non-Default parallel regions* showed significant execution time and energy consumption improvement with optimal configurations as compared to the default configuration\(^4\). Figure 2 shows the improvement in execution time and energy consumption for *Non-Default parallel regions* at 85W power level. The pink bar in the figure represents execution time improvement while the greenish bar represents energy consumption improvement. On average, we observed 26% execution time and 30% energy consumption improvement. The largest improvement is observed in SP_x_solve_1 parallel region from SP; where we achieve 67% in execution time and 72% in energy consumption improvement. We observed similar patterns at other power levels as well.

We witnessed a similar behavior with Intel runtime. On average, 30% of the total parallel regions were *Non-Default parallel regions*. Using the optimal configurations, these parallel regions showed an average execution time and energy consumption improvement around 29% and 32% respectively.

These results show that using the default configuration blindly could result in a sub-optimal performance for many parallel regions. This observation brings us to some interesting questions.

\(^3\)We used the words *package energy* and *energy* synonymously.

\(^4\)OpenUH and Intel runtimes have the same default configuration.

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>A simulated CFD computational kernel that uses an implicit algorithm to solve 3-dimensional (3-D) compressible Navier-Stokes equations. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the x, y and z dimensions.</td>
</tr>
<tr>
<td>SP</td>
<td>A simulated CFD application that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions.</td>
</tr>
<tr>
<td>CG</td>
<td>Uses a Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix</td>
</tr>
<tr>
<td>EP</td>
<td>An Embarrassingly Parallel benchmark. It generates pairs of Gaussian random deviates according to a specific scheme</td>
</tr>
<tr>
<td>FT</td>
<td>Contains the computational kernel of a 3-D fast Fourier Transform (FFT)-based spectral method</td>
</tr>
</tbody>
</table>

TABLE III: Used Benchmark Applications
Fig. 2: Execution time and energy consumption improvement using optimal configurations over the default configuration for Non-Default parallel regions. We used OpenUH runtime.

1) Does the performance improvement using the optimal configuration persist across all the power caps for Non-Default parallel regions? : Figure 3 shows the percentage improvement in execution time and energy consumption for Non-Default parallel regions using optimal configurations. We show the result for all the five power caps. Figure 3a shows the execution time improvement of the parallel regions. Each line in the figure represents execution time improvement for a specific power level. It is evident from the figure that almost all the lines coincide with each other. This result shows that the execution time improvement is comparable across all power caps. However, an optimal configuration for a parallel region may change across different power caps (Section V-2). Figure 3b shows the improvement in energy consumption across the different power caps. We observe a similar behavior in this case as well. We observed a similar trend in execution time and energy consumption improvement using Intel runtime as well.

Based on this analysis, we can conclude that for Non-Default parallel regions the optimal configuration performance improvement persists across all the power caps.

2) Is an optimal configuration for a parallel region uniform across all the power caps? : We analyzed the optimal configurations of the experimental parallel regions across different power caps for both runtimes (OpenUH and Intel). We observed that parallel regions which were Non-Default at the highest power level (115W) remained Non-Default at the lower power levels (100W, 85W, 70W, 55W). However, these Non-Default parallel regions frequently changed their optimal configurations across power caps. The parallel region discussed in Section II is an example of Non-Default parallel region. Figure 1 clearly shows that the parallel region has different optimal configurations at different power caps.

The question arises, why this behavior? The answer is; in Non-Default parallel regions, the optimal configuration does a variable work load distribution to different threads. The work distribution depends on the current state of cores and caches. With the change of power caps, the behavior of cores and caches changes. As a result, the work load distribution also needs to be changed to optimize the usage of cores and caches. To facilitate the changed work load distribution,

the optimal configuration changes with the change of power cap. We also noticed that an optimal configuration from a different power cap may degrade execution time significantly. Especially, if the difference between the power levels is high. For example, when SP_z_solve_1 parallel region from the SP application was run with the optimal configuration of the 100W power level at 55W, the execution time performance degraded about 15% as compared to the optimal configuration of 55W. Table IV shows the detailed results across all power caps.

TABLE IV: Optimal configuration for SP_z_solve_1 parallel region on different power caps & Performance penalty of using an optimal configuration of a certain power cap to another

<table>
<thead>
<tr>
<th>Power Cap</th>
<th>Configurations (Thrd Scl Chunk)</th>
<th>55W</th>
<th>70W</th>
<th>85W</th>
<th>100W</th>
<th>115W</th>
</tr>
</thead>
<tbody>
<tr>
<td>55W</td>
<td>32 GUIDED 8</td>
<td>0</td>
<td>2.78</td>
<td>2.99</td>
<td>1.94</td>
<td>3.37</td>
</tr>
<tr>
<td>70W</td>
<td>32 DYNAMIC 8</td>
<td>12.27</td>
<td>0</td>
<td>1.46</td>
<td>0.63</td>
<td>0.68</td>
</tr>
<tr>
<td>85W</td>
<td>32 GUIDED 8</td>
<td>11.10</td>
<td>0.66</td>
<td>0</td>
<td>0.19</td>
<td>0.01</td>
</tr>
<tr>
<td>100W</td>
<td>16 DYNAMIC 16</td>
<td>14.52</td>
<td>0.39</td>
<td>0.62</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>115W</td>
<td>16 GUIDED 16</td>
<td>14.52</td>
<td>0.39</td>
<td>0.62</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Based on this analysis, we can conclude that a Non-Default parallel region at higher power levels remain Non-Default parallel region at lower power levels and optimal configurations for Non-Default parallel regions are not uniform across all power caps.

3) Why does an optimal configuration improve execution time and energy consumption for Non-Default parallel regions? : An optimal configuration optimizes cache behavior and load balancing to improve performance. To verify this, we analyzed the hardware behavior of Non-Default parallel regions using both optimal and default configurations. We collected 19 dynamic hardware features for these parallel regions using PAPI. We used TAU to collect these features. The collected features included cache behavior, branch behavior, instruction mix and idle state behavior.

We observed that features related to cache and idle state of the processor played a decisive role in the performance gain. Features like $L_1$, $L_2$, $L_3$ cache miss rate, TLB (Transfer Look-Aside Buffer) miss rate represent cache behavior. The smaller is the miss rate, the better the performance. CPI (Cycles Per Instruction) and stall cycles per instruction capture the waiting/stall state behavior of the processor. The waiting-state behavior of the processor characterizes the load balancing behavior of the parallel region. The lesser the time the processor spends in the waiting state the better is the load balance. Figure 4 shows the dynamic feature comparison between the default and optimal configurations for SP_z_solve_1 parallel region at 85W power level using OpenUH runtime. This parallel region has 61% execution time improvement and 67% energy consumption improvement using the optimal configuration at 85W power cap. The figure
shows that the optimal configuration improves both cache and idle state behaviors. Similar behavior was observed with other power levels. For other Non-Default parallel regions, we also observed that optimal configuration improved cache and stall state behaviors. We saw similar results with Intel runtime.

Based on this analysis, we can conclude that the optimal configuration improves load balancing and cache behavior of a parallel region, and in the process, improves execution time and energy consumption.

4) Do we always get the best performance at the highest power cap (TDP)? A well-known concept in the HPC area is that more power provides more performance. So, if everything else (e.g. runtime configuration, data set size) remains constant, performance in higher power caps will be at least same if not better than performance in lower power caps. But when we compared the best execution time configurations of all the power caps, we observed that more than 85% parallel regions had absolute best (best among all power caps) configuration in lower power caps than TDP. These parallel regions had a configuration in lower power cap which outperformed the best configuration in TDP (115W in our case). Figure 5 shows the performance and energy improvement of these parallel regions in the absolute best configuration (in lower power cap) compared to the local best configuration in TDP. In the figure, we show the parallel regions divided into 4 different power caps. These power caps represent the power cap on which they had the absolute best configuration.

We see these results primarily because these parallel regions have a lower power profile. That means without power capping (running in TDP) these parallel regions consume less power than TDP. As a result, running in lower power level does not impact the natural progression of these parallel regions. But why the extra performance improvement at lower power level? There are two possible explanations.

First, in the lower power cap the CPU frequency decreases. As a result, CPU speed becomes more synced with the cache and memory speed. So, the cache and memory accesses become more effective and the number of cycles processors have to wait for data decreases. This, in turn, provides better performance.

The second one is, the configurations we selected for our experimentation are discrete. They are not comprehensive in the purest sense (e.g. we used 7 (1, 8, 32, 64, 128, 256 & 512) chunk size level out of possible 512 (1-512)). Now at a certain power cap we might have overlooked the best configuration. Based on this analysis, we can conclude that an optimal power level for a parallel region may not always be the highest power level. A parallel region may exhibit the best performance at a lower power level at a certain system.

5) How OpenMP parallel region level improvement affects the overall application execution time and energy consumption? We ran the NPB applications with each OpenMP parallel region using its specific optimal configuration and compared the application performance with the default configuration performance. Figure 6 shows the execution time and
energy consumption improvement for the SP application with OpenUH runtime. The figure shows 18 – 26% improvement in execution time and 22 – 38% improvement in energy consumption across different power levels. We observed similar performance behavior in other applications as well. We observed comparable performance behavior with Intel runtime.

Based on this analysis, we can conclude that parallel region level performance improvement has an impact on the overall application level performance.

VI. POTENTIAL IMPACT OF THESE FINDINGS

The findings we have presented in this paper can have a large and long lasting impact on future power aware HPC. Following is a list of possible scenarios that can be impacted from these findings.

- We have shown that an optimal runtime configuration is dependent on a specific parallel region. And choosing a suboptimal configuration may degrade the performance significantly. As most of the application has parallel regions with different behavior, one should focus on fine-grained parallel region level optimization to achieve the best performance.

- We have also shown that optimal configuration changes at different power levels. Now in the exascale era we may see situations where the power level may change during the program execution. This could happen due to various reasons such as power budget issue, node failure, node migration. In those scenarios predefined configuration may not be feasible. It would require a system that could adapt to the changing environment and choose runtime configurations accordingly.

- We have demonstrated that with careful configuration selection one can achieve better performance at lower power level depending on the parallel region behavior. This opens up a lot of possibilities. One such possibility is fine-grained power shifting across nodes. If we can classify the parallel regions based on their behavior, we can come up with appropriate power level for these parallel regions. Then based on the availability and necessity we can shift power across nodes when we have more power allocated than we need, or we can request more power from other nodes if necessary.

These are only some of the scenarios which can be benefited from this work but the possibilities are not limited to these. We believe this work would provide a guideline for future intra-node power aware work focusing on OpenMP.

VII. RELATED WORK

Bull et al. [4] is one of the first work which provides an insight into the impact of the number of threads, scheduling policy and synchronization on an OpenMP application’s performance. Suleman et al. [12] proposes a framework to dynamically control the number of threads using run-time information. However, none of these works consider power budget or energy consumption in their analysis. Their only consideration was execution time. But in our work, we consider both execution time and energy under a power budget.

As the number of threads and processor frequency have an enormous impact on performance and energy consumption of a given OpenMP application, many researchers have studied energy efficient performance prediction models for parallel applications. The work by Curtis-Maury et al. [13], [14] falls under this category. They employ dynamic voltage and frequency scaling (DVFS), dynamic concurrency throttling (DCT) and simultaneous multithreading (SMT) to implement various online and offline configuration selection strategies for OpenMP applications. They concentrate on decreasing energy consumption without degrading execution time. However, the work does not consider power budget. Peter Baily et al. [15] implements an adaptive configuration selection scheme for both homogeneous and heterogeneous power constrained systems. It considers only two parameters, the number of threads and processor frequency. Dong Li et al. [16], [17] uses DVFS and DCT to select energy efficient configuration for threads and operating frequency for MPI/OpenMP hybrid applications. They do not consider a power budget. Their main target is to save energy without the loss of execution time. The work by Wei et al. [18] shows the impact of optimal operating frequency on energy consumption improvement for parallel loops. It uses different operating frequency across different
loops using frequency modulation techniques. In contrast to these works, our’s concentrates on a complete set of runtime parameters on a strict power constrained system. Nowadays, power has become a limiting factor for large-scale HPC centers. As a result, work on overprovisioned systems with strict power budget is gaining popularity in the HPC community. Work by Rountree et al. [19] is one of the first to explore the impact of power capping. They investigate how different power levels impact the performance of different types of applications. Work by Patki et al. [2] explores the impact of hardware overprovisioning on a system level performance. The main contribution of their work is to select the number of nodes, the number of cores per node, and power cap per node. Work by Aniruddha et al. [20] and Bailey et al. [21] consider only two parameters, DVFS, and the number of threads, as configuration options. They focus on overall system level performance on an MPI/OpenMP hybrid application. Compared to these works, our work focuses on single node performance improvement at the OpenMP parallel region level using OpenMP runtime parameters.

VIII. CONCLUSION AND FUTURE WORK

Overprovisioned systems are becoming an attractive solution to handle the power challenge in the future HPC platforms. Previous work in this area primarily looks at the distributed level. However, node level performance at different power caps is also important. The OpenMP API is mostly used to exploit parallelism for shared memory processors. This paper provides a comprehensive performance analysis of OpenMP runtime configurations for power constrained systems using different power levels. We performed extensive experimentation using the NAS Parallel Benchmark applications. Based on the results, we showed that by selecting an efficient runtime configuration for a given power level, one can improve the execution time and energy consumption of a loop-based OpenMP parallel region up to 67% and 72% respectively. We also showed that the parallel region level improvement can, in turn, improve the execution time and energy consumption of an OpenMP application up to 26% and 38% respectively. In the analysis section, we answered some interesting questions which are pertinent to the HPC community interested in the power-aware optimization.

In the future, We plan to investigate the impact of memory power cap on OpenMP runtime settings. We aim to look into how the task based OpenMP applications fare in power constrained systems. Since OpenMP supports accelerators, we also aim to study the OpenMP runtime configurations in a heterogeneous environment.

ACKNOWLEDGMENT

This work is supported by the National Science Foundation under grant CCF-1148052.

REFERENCES