Mitigating Impact of Heterogeneity Across Power-constrained Nodes on Parallel Applications through Load Balancing

Sandeep Dasgupta
sdasgup3@illinois.edu

Karthik R. Gooli
gooli2@illinois.edu

Dhawal Seth
dseth3@illinois.edu

***Abstract*—Different processors across the nodes have different execution times for the same work-loads. This performance imbalance is seen only when the CPU power is capped to low values. This performance imbalance causes increased execution times of the parallel applications. We propose a power aware load balancer which minimizes the performance imbalance at the lower power caps by tackling this heterogeneity.**

***Keywords*-Energy minimization; Power capping; Load balancing**

I. Introduction

Today with the growing needs of power, one of the goals of the HPC community is to build larger systems given a strict power budget. The goal is not only to build larger systems but also to optimize the systems' performance under the power budget constraints. It is noted that running systems at their TDP [1] is a huge wastage since most of the times they are consuming lesser power than their TDP. Intel's Running Average Power Limit[1] toolkit is a feature that helps to constrain the power of its compute cores and DRAM and thereby enabling software controlled, optimized power allocation to the compute nodes based on the application running on them. It is noted that under a strict power budget and under certain circumstances, running applications with lower power limits on more number of nodes can be more efficient than running the same application with higher power on fewer nodes [9].

The motivation for this paper is drawn from the future guidelines of the work by Osman et al.[9] and [2]. It has been empirically observed that under the same power cap, different nodes yield different application performance. This can be due to several design factors: difference in chip designs, different in component assembly by the machine vendor, location of the node in the data center, difference in component design such as fans, etc. This difference in the design causes load imbalance across nodes despite same allocated power and equal compute load. Moreover we are going to experimentally show that this heterogeneity across the nodes is more prominent at lower power caps. Our goal is to minimize the load imbalance in the presence of such heterogeneity among the nodes using the over-decomposition and dynamic object migration features of Charm++[5].

Our work is a two-fold approach. Firstly we study the extent of heterogeneity under the lower power caps and based on this study we design and implement a power-aware load-balancer which fixes this heterogeneity.

[1]The thermal design power (TDP), sometimes called thermal design point, refers to the maximum amount of heat generated by the CPU, which the cooling system in a computer is required to dissipate in typical operation.

II. Heterogeneity Study

A. Heterogeneity metric

Our first contribution is to show that at lower power capping values, different nodes show prominent differences in their runtime performance. In this work, we use Intel's power_gov library[3] that in turn uses RAPL [1] to cap power of memory and package[2] subsystems.

We define heterogeneity at a given power cap in terms of idle waiting times of the cores at that power cap. We defined the idle waiting time of a core at a given power cap in the following two ways:

At a given power cap, let $t_i, 1 \leq i \leq C$ be the overall execution time of the core i for a particular application and T be the total execution time of the application.

Percentage Average idle waiting time, I_{av}

$$I_{av} = \frac{\sum_{i=0}^{P} (\max_{1 \leq j \leq C}(t_j) - t_i)}{C} \quad (1)$$

Percentage Max idle waiting time, I_m

$$I_m = \max_{1 \leq j \leq C}(t_j) - \min_{1 \leq i \leq C}(t_i) \quad (2)$$

Figure 1 shows that at lower power caps the idle waiting times (Equations (1) and (2)) are having higher values as compared to those at higher power caps.

[2]Package corresponds to the processor chip that hosts processing cores, caches and memory controller

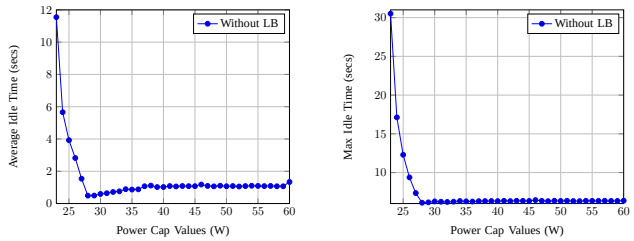


Figure 1: Behavior of idle waiting times at lower power caps

B. Testbed

Our testbed is a 60-node Dell PowerEdge R620 cluster installed at the Department of Computer Science, University of Illinois at Urbana-Champaign. Each node is an Intel Xeon E5-2620 Sandy-bridge server with 6 physical cores @ 2GHz, 2-way SMT with 16GB of DRAM. The Intel Sandy Bridge processor family supports on board power measurement and capping through the Running Average Power Limit (RAPL) interface [1]. The Sandy Bridge architecture has four power planes: Package (PKG), Power Plane 0 (PP0), Power Plane 1 (PP1) and DRAM. RAPL is implemented using a series of Machine Specifics Registers (MSRs) which can be accessed to get power readings for each power plane. RAPL supports power capping PKG, PP0 and DRAM power planes by writing into the relevant MSRs. The package power[3] for our testbed can be capped in the range 23W to 95W (73 integer power levels) while the memory power can be capped between 8W to 35W (28 integer power levels). The average base power per node for our cluster was 38 watts. The base power was measured using the in-built power meters on the Power Distribution Unit (PDU) that powers our cluster.

In our experiments we will **NOT** be capping the memory power as our work is focused on studying the heterogeneity that comes up at lower CPU power. The effect of lower memory power on heterogeneity is not discussed.

C. Applications used

We used two applications, namely, Jacobi2D & LeanMD[8] from the Charm++ test repository. We have manually modified these applications so that we get the precise timing measurements. These applications have different CPU and memory usage.

- **Jacobi2D:** A 5-point stencil **memory-bound** application that computes the transmission of heat over a discretized 2D grid. The global 2D grid is divided into smaller blocks that are processed in parallel. It is an iterative application where all processors synchronize at the end of each iteration. As is the case in a stencil computation, each grid point is the average of the neighboring 5 points. For example, the new value for element X is the current value of X plus the current values of its left, right, top, and bottom neighbors.

  T
  L X R
  B

  X' = (X + L + R + T + B) / 5.0

  Neighboring blocks communicate the ghost layers with each other so that averaging computations are done for all cells inside each block. This application is implemented in Charm++ using a 2D chare array.
- **LeanMD:** LeanMD [8] is a **computationally intensive** molecular dynamics application. This benchmark simulates atomic interaction based on Lennard-Jones potential.

  LeanMD is a molecular dynamics simulation application written in Charm++ for PetaFLOPs class supercomputers. It is being developed as the next generation of NAMD [6], one of the parallel applications winning the Gordon Bell Award in SC2002. NAMD, as a state-of-the-art parallel molecular dynamics application that is also written in Charm++, has already been proven to be able to scale to 3000 processors. However, it is not ready for next generation parallel machines with hundreds of thousands, or even millions, of processors due to the limited parallelism exploited in the application. Clearly, it requires a new parallelization strategy that can break up the problem in a more fine-grained manner to effectively distribute work across the extremely large number of processors. With that outlook in mind, LeanMD is being developed as an experimental code.

  LeanMD computes the interaction forces based on Lennard-Jones forces amongst particles in a 3D space. It does not include any long range force calculation. The object decomposition is achieved using a scheme similar to NAMD. The 3D space is divided into hyper-rectangles, called cells or patches in NAMDs nomenclature, each containing a subset of particles. A compute object is responsible for the force calculations between each pair of cells. In each computation of the application, each cell sends its particle data to all computes objects attached to it and receives the updates from those computes objects. This mini-application is implemented using Charm++ where the set of cells and compute objects are represented by chare arrays.

D. Charm++ & Load Balancing

For this research, we used Charm++ programming paradigm which supports dynamic object migration to improve performance of a parallel application[7]. It relies on techniques such as processor virtualization and over-decomposition (having more work units than the number of cores) to improve performance via adaptive overlap of computation and communication and data-driven execution.

[3]Package corresponds to the processor chip that hosts processing cores, caches and memory controller

Charm++ gives the freedom to the programmer to define program into multiple grain size objects which can be migrated across the cores. The programmer need not make the application core aware. This multiple objects defined by the programmer is moved around during program execution by adaptive runtime system not only for load balancing purposes but also for communication optimization and fault tolerance. Load balancer keeps the statistics of all the migratable objects for effective load balancing act[4]. The runtime system provides load balancing strategies that can account for different application characteristics. Application programmers can provide their own implementation of load balancers based on the characteristics of the application and the ecosystem under which it is run.

E. Observations

Figure2(a) shows how the execution times varied under different power caps. For this experiment we have not used any existing load-balancers. As we can see, execution time of the application increases as we move from higher power to lower power caps. We see an exponential increase of execution time as we move to the lower power values. This is partly due to reduction in frequency as the power decreases and hence increased execution times. But we also have heterogeneity factor contributing to the increased execution times. We will next see how the heterogeneity factor contributes to this increased execution times.

Figure2(b) shows how Max Idle Time metric varies under different power values. As we can see under lower power regions the max idle times increases exponentially. This means the difference in execution times vary a lot under lower power regions. This suggests that there is a big margin between the processors which finish its load quickly and the processor which finished the last. In the next part lets see how average idle times varies under lower power capping regions. This metric gives a better understanding of how the idle time varies across the nodes and not just the difference between the best and the worst performer unlike this graph.

Figure 2(c) how Average Idle Time metric varies under different power values. As we can see the average idle time varies exponentially under lower power capping regions. This shows that most or all processors exhibit different execution times and there by having different idle times and hence we see increased average idle times under lower power regions. This further establishes the fact that heterogeneity is present across the nodes and not just only with few nodes. We wanted to establish the extent of heterogeneity among the processors. We study that next.

Figure 2(d) show how the execution time has varied across the processors at 23W. As we can see we have huge difference in execution times among the processors. The difference in these execution times as we have seen is best captured by max idle time and average idle time graphs. The execution time of the processors within a node is more or less a constant. The execution times vary only among the PEs across the nodes. This shows that heterogeneity holds good across the nodes and not within the same node.

III. Design Of Power Aware Load Balancer

Our second contribution is to minimize the above heterogeneity using the Charm++ load balancers. We first observed the working of existing load balancers like *RefineLB* & *GreedyLB* to evaluate the its effect in minimizing the heterogeneity.

A. Existing Load-balancers

Load balancing is a technique of distributing computational and communication load evenly across processors of a parallel machine so that no single processor is overloaded. Charm++ implements a generic, measurement-based load balancing framework which automatically instruments all Charm++ objects, collects computation load and communication structure during execution and stores them into a load balancing database. Charm++ then provides a collection of load balancing strategies whose job it is to decide on a new mapping of objects to processors based on the information from the database. These strategies work under the assumption that objects in a Charm++ application tend to exhibit temporal correlation in their computation and communication patterns, i.e. future can be predicted to some extent using the collected data, allowing effective measurement-based load balancing without application-specific knowledge. Following are the two widely used load balancing strategies:

1) RefineLB: The objective of this strategy is to move objects away from the most overloaded processors to reach an average. Following is the pseudo-algorithm of this strategy:

Data: V_t:(the set of objects; V_p (the set of processors)
Result: Map:$V_t \rightarrow V_p$ (An object mapping)
// build heap ;
ProcessorHeap heavyProcs(V_p);
Set *lightProcs;
while *!done* **do**
  donor = heavyProcessors→deleteMax();
  while *ligthProcs* **do**
    (obj, lightProc) ← BestObjFromDonor(donor);
    if *obj.load + lightProc.load > avg_load* **then**
      continue;
    end
    if *obj_obtained* **then**
      break;
    end
    deAssign(obj, donor);
    assign(obj, lightProc);
  end
end

Algorithm 1: RefineLB Pseudocode

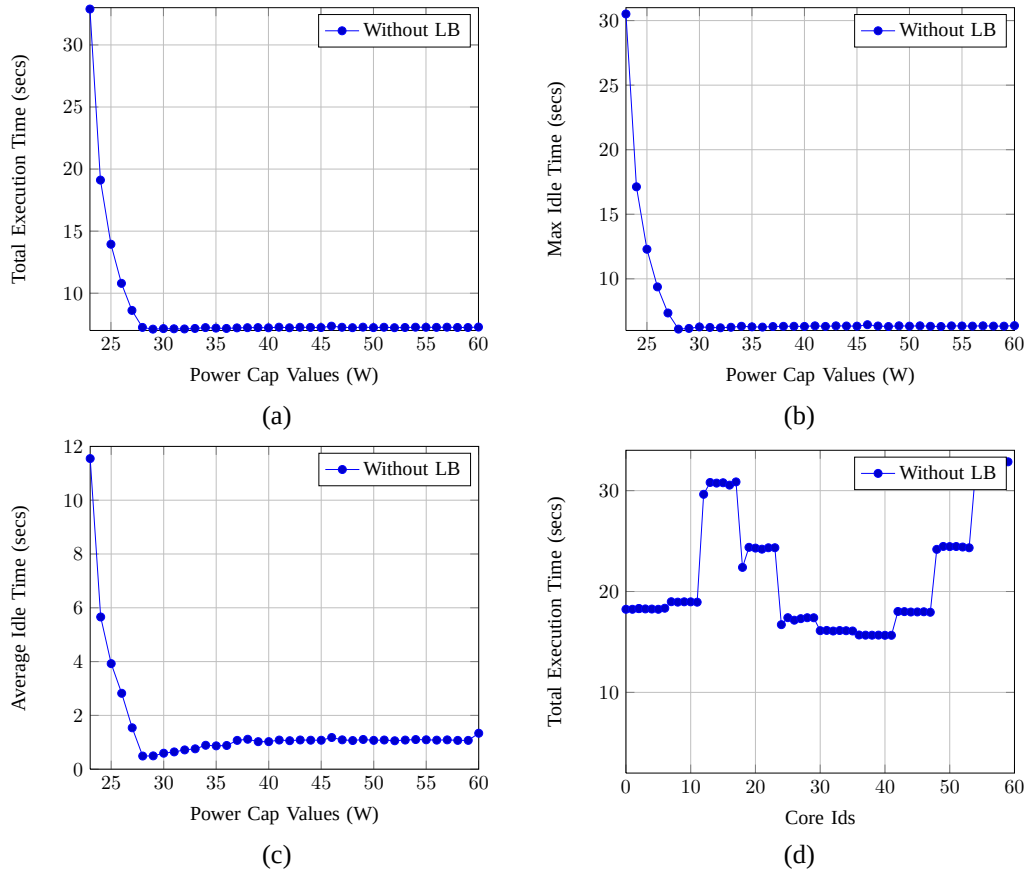


Figure 2: (a) Total Execution Time vs Power (b) Max Idle Time vs Power (c) Average Idle Time vs Power (d) Heterogeneity at 23W

As evident from the above algorithm that it tries to migrate objects based on a global average which is computed as the average of each processors's load. As a result it restricts the number of migration which is otherwise possible. This restriction on object migration will be more pronounced at lower power caps as the opportunity for object migration is high at lower power caps due to the increased heterogeneity.

2) GreedyLB: This uses a greedy algorithm that always assigns the heaviest object to the least loaded processor.

The assumption for object migration is that the time taken by the object to execute on a processor will remain the same both before and after the migration. This assumption is valid at higher power caps because (1) all the objects are of same size and (2) at higher power capping values the processors are running at nearly the same frequencies and hence the time taken by a chare to run any of them is nearly equal. But this assumptions fall apart at lower power caps because of the heterogeneity introduced between the processors and as a result it may happen that the object time will differ after the migration. Following is the pseudo-algorithm:

Data: V_t:(the set of chare objects;
V_p (the set of processors;
G_p (the background load of processors) // due to non-migratable objects, etc.)
Result: Map:$V_t \rightarrow V_p$ (An object mapping)
// build heap of size equal to the number of objects ;
ObjectHeap objHeap($|V_t|$);
// insert each element of Vt in objHeap;
$V_t \rightarrow$objHeap ;
MinHeap cpuHeap(P);
//Initially processors are empty with only background load;
cpuHeap$\leftarrow G_p$;
for *i$\leftarrow$1 to nmigobj* **do**
    o$\leftarrow$ objHeap.deleteMax();
    donor$\leftarrow$cpuHeap.deleteMin();
    Assign c to donor and record it in Map;
    donor.load += c.load // add object load of c to the donor;
    cpuHeap.insert(donor) ;
end

Algorithm 2: GreedyLB Pseudocode

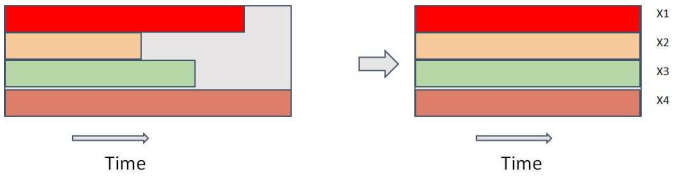
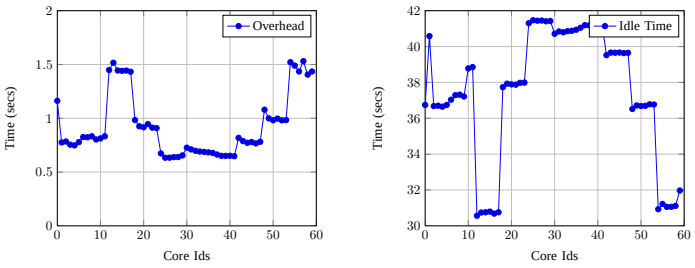


Figure 3: Expected Result of Power Aware Load Balancer



(a) (b)

Figure 4: Idle and Overhad time over the cores at 24W power cap

B. Design: Power Aware Load Balancer

We have established the fact that current load-balancers in charm++ are not power aware. Current Load Balancers do not load balance based on the individual capacity of the processor. We have also established that there is a definite scope for improvement in execution time of the application by minimizing the maximum idle and the average idle times of the processor. Our motive behind the design was to make all the processors finish the execution at the same point of time. This approach takes relative speeds of the processors into consideration for balancing the load. We use the object times of the chares to measure the relative speeds of the processors. We explain in the next section, the reason to choose object time for the load balancing. In this approach we try to average out the execution time by assigning the load in such a way that each of them is assigned exactly that much amount of load which helps all the PEs finish at the same time. As it can be seen in the Figure 3 different PEs finish execution at different points of time. As per the motive of the design we make all the processors to finish at the same time. The following two equations explains our design.

$$(w_1 x_1) = (w_2 x_2) = (w_3 x_3) = \cdots = (w_n x_n) \quad (3)$$

$$x_1 + x_2 + x_3 + \cdots + x_n = N \quad (4)$$

where, w_1, w_2,..., w_n are the weights in terms of times for executing x_1, x_2,..., x_n assigned objects by PE_1, PE_2,. . ., PE_n and N is the total number of objects. Before the load balancing, x_1, x_2, ..., x_n are all the same. All the objects have the same load/weight in terms of computation. Each PE executes the same number of objects before load balancing. Each PE before load balancing has the same total load since each object has the same weight. By using the above equation we get the proportionality based on which the number of objects will be assigned for the iterations after load balancing. For instance, PE1 could be 2 times faster than PE_2 and P_3 could be 0.5 times faster than PE_2. In which case we will have the following proportionality: PE_1 : PE_1 : PE_3 = 2 : 1 : 0.5. Now based on this proportionality each PE is assigned such a load that everyone can finish their execution at the same point of time.

We use the second equation (4) to determine x_1, x_1, ..., x_n. We determine x_1, x_1, ..., x_n values by solving for each x_1 up to x_nn by using both the equations. Now the determined x_1, x_2 . . . x_n values is the new load for its respective PEs that help finish all of them at the same time. The newly assigned load for each of the PEs is based on the relative speeds of the processors. Solving the above two equations and assigning the new loads to the processors will now help all the processors finish at the same time.

C. Selection of metric "w"

We need to have a suitable metric for out parameter "w". The selected metric should demonstrate heterogeneity at the lower power caps. We have considered processor idletime, processor overhead and processor object time[4] as candidates for our selection. We measure these values for each processor id at a power cap of 24W. Figures 4 and 5 show the experimental plots. The metric which has the maximum variance depicts the heterogeneity in the best possible way. Percentage change between the maximum and minimum idle time was 0.35, and the same for background time[5] was 1.41 and that of processor object time was 4.09 (This was partially due to a misbehavior of node0:core0 as seen in the Figure 5) . Using the above observation we chose processor's object time as a metric for "w" because this shows the maximum variance at lower power caps.

IV. Results

We used the existing charm++ implementation of Jacobi and executed the application at different power caps ranging from 23 to $60W$. Each execution of the Jacobi application (at a particular power cap) is done with the following parameters:

1) Grid size = 36000×36000
2) Chare Size(or block size) = 600×600
3) Number of Iterations = 20

[4]Object time is the time taken by a charm++ object to run on a processor. This is also know as object walltime. Processor's object time is the sum of walltimes of all the objects executing on that processor

[5]Processor Background time amounts to the overhead due to non migratable objects on that processor

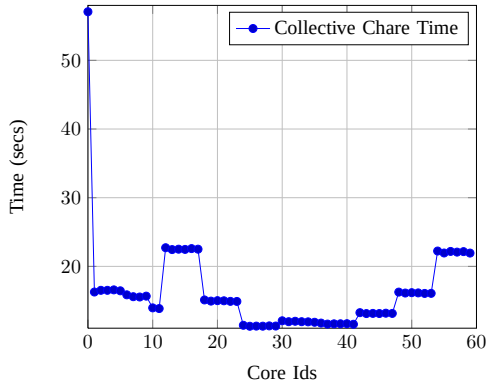


Figure 5: Collective chare time on each processor at 24W power cap.

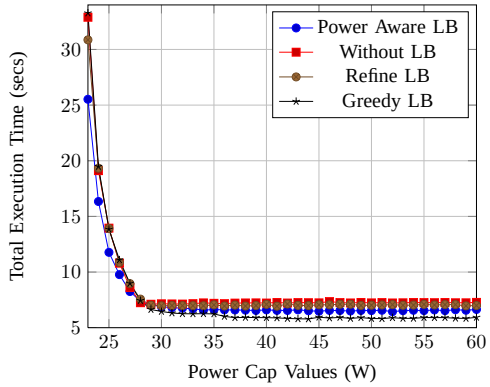


Figure 6: Total Execution Time Vs Power Comparison between LBs

A. Performance Evaluation Of Power Aware Load Balancers

To analyze the performance of our power-aware load balancer, the metrics of heterogeneity are compared. This comparison is done with the performance of Jacobi2d execution in the absence and presence of existing load balancers like RefineLB and GreedyLB. Power aware LB has proved to be more efficient than other LBs. The main reason is the frequency awareness our load balancer takes into account while trying to minimize the load among the nodes. RefineLB and GreedyLB are not aware of the differences in frequency that exists at lower power caps. This makes them power unaware. They only consider the workload in terms of object wall time while balancing the load and try to converge to that point without taking into the account individual capacity of the processor. When migrating Chare objects from one node to other, our load balancer works takes in to account the existing workload and the time taken to complete the execution for that amount of workload, at both sending and receiving ends. This makes it perform well even in the presence of heterogeneity at lower power caps.

As shown in Figure 7 a maximum speedup of 1.3× was seen when compared to GreedyLB. Maximum speedup of 1.2× w.r.t to RefineLB and 1.28× w.r.t without LB. The total execution time when run without any load balancer at 23W power cap is 34sec. This is somewhat reduced in case of RefineLB and GreedyLB. But the power aware LB is able to decrease is to 25sec as shown in figure 6. The percentage decrease with the Power Aware LB is more.

Power	WithoutLB	RefineLB	GreedyLB
27	1.04	1.08	1.08
26	1.10	1.11	1.13
25	1.18	1.18	1.17
24	1.16	1.18	1.19
23	1.28	1.20	1.30

Figure 7: Speedups

There is little or no speed up in cases of higher power caps when using the power aware load balancer. The reason behind this is the absence of heterogeneity at higher power caps. The product of object wall time w and the workload x comes to be almost the same, and so it leaves power aware LB no scope of further balancing the workload based on frequency. Load imbalance is observed only at lower power caps. Hence, there is very little migration that power aware load balancer does at higher power caps. This makes GreedyLB perform better at higher power caps, as it takes the greedy approach of freshly loading the heaviest object with lightest loaded processor.

GreedyLB and RefineLB are unaware of the differences in frequencies of different nodes that exist at lower power caps. There is equal weightage given to each node in GreedyLB and RefineLB. These do not take into account the frequency of each node and assume that all the nodes in the cluster have identical performance capability.

Performance of power aware LB is further analyzed with respect to other LBs in terms of heterogeneity metrics average idle time and max idle time, as shown in Figure 10. Power aware LB has out performed the other existing power unaware LBs in terms of average and max idle times. The average idle time is reduced using the power aware LB by as high as 60% when compared to the average idle time of the run without any LB. If compared with RefineLB and GreedyLB, the reduction is as high as 54% and 63% respectively. The average idle time at 23W comes to be 12,10,14 and 5sec for runs without LB, with Refine, Greedy and power aware LBs respectively as shown in figure 8.

The reduction in maximum idle time is as high as 25% in the presence of power aware LB. It is 30sec in the absence of any load balancer and it comes down to 23sec when run with power aware LB at a low power cap of 23W as shown in figure 9. We could have further decreased the maximum idle time. But due to an off node in the cluster

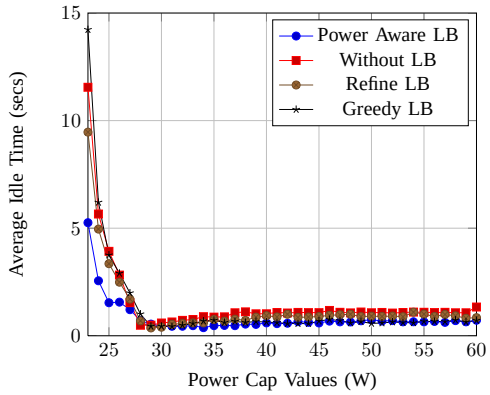


Figure 8: Average Idle Time vs Power Comparison between LBs

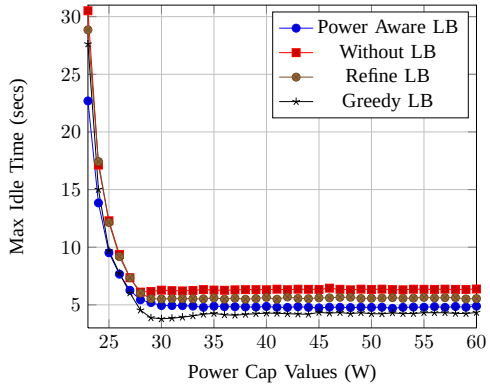


Figure 9: Max Idle Time vs Power Comparison between LBs

with an unexpected behavior led to an increased max idle time. However average idle time is a better metric to measure the performance of our load balancer than the max idle time since its taking average idle time into account and the misbehavior by a one off node doesnt pull the average down to a great extent unlike the max idle time metric.

Power aware LB performs equally well in cases of higher power caps when there is little or no heterogeneity among the nodes. One positive aspect here is that power aware load balancer does not try to do unnecessary balancing at higher power caps that could have led to an increase in the average or max idle leading to an increase in the total execution time.

Another observation of the total execution time is done

	% AVG IDLE TIME REDUCTION			% MAX IDLE TIME REDUCTION		
Power	WOLB	RLB	GLB	WOLB	RLB	GLB
27	21.16	28.78	39.05	14.87	14.85	-3.76
26	44.58	37.21	46.11	18.27	16.57	1.005
25	60.86	54.16	59.05	22.55	21.61	1.78
24	54.81	48.43	58.77	19.17	20.58	7.79
23	54.48	44.46	63.05	25.63	21.36	17.87

Figure 10: % Reduction in Idle times

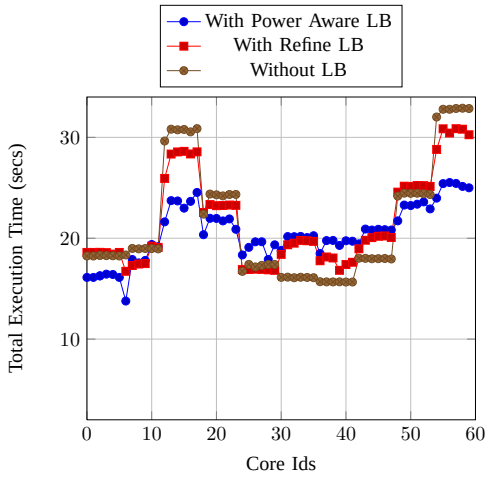


Figure 11: Heterogeneity comparison between LBs at 23W

on a per core basis at 23W. Figure11 depicts the amount of heterogeneity among the nodes. Further from the same figure we can note that there is a lot of difference in the total execution times by each node in the absence of any load balancer. The total execution time ranges from 15-35 sec for the run without any load balancer. RefineLB does not help in minimizing the heterogeneity by a large extent. The range of total execution times remains close to 15-30 sec. The power aware load balancer proves to redistribute the workload in a better manner compared to existing Load Balancers. As depicted in figure 11 the range of total execution time has decreased to 15-25 sec with most of the nodes closing to the average. The range could be further reduced by making use of node specific information or hard coding based on performance profiling of individual nodes. This profiling information is not incorporated as they make the load balancer usable only for specific applications running on known clusters.

V. Conclusions & Future Work

We observed that the execution time increases with decrease in CPU power, but the rate at which it increases was seen to be different with some nodes performing better than others, leading to performance heterogeneity. This leads to some nodes exhibiting higher performance than others, and thus such nodes tend to have higher idle-times waiting for other slower ones to complete their iteration, before the next iteration could be started. Heterogeneity is measured in terms of average idle times of the nodes in the cluster. This heterogeneity becomes significant when the power is pulled down to the allowed minimum. This load imbalance leads to more wait times among the nodes and thus there is scope to minimize the imbalance by having a power aware load balancer that gathers information of the initial few iterations and then based on the collected information about

the frequency and workload of different nodes, tries to bring down the idle times.

To help mitigate this performance imbalance, we developed a power-aware load balancer that helps in minimizing the heterogeneity. Our load balancer performed better than the existing power-unaware load balancers in the Charm++ framework. It helped reduced the existing amount of heterogeneity and achieve a maximum speed up of 1.3x with respect to other load balancers when used with Jacobi2d application.

The current limitation with this load balancer is that it does not take into account the size of the workload on a particular node. This limits its usage in the applications where the object size varies with time. One such applications that we studied for heterogeneity at lower power regions was LeanMD, where there is particle migration happening as the application executes leading to different object sizes at different times.

Our future work aims at making our power-aware load balancer aware of the changing size of the workload, and thus incorporating this parameter while balancing the workload among the various nodes. This work also involves periodic invocation of the load balancer in order to keep the idle times minimized. This has to be done carefully, keeping the overall load balancing workload to the minimum.

ACKNOWLEDGMENT

We would like to thank Prof. Josep Torrellas for helping us procure the resources required to establish the TestBed. The comments for the mid-term review was very encouraging which motivated us further. We would also like to thank Prof. Tarek Abdelzaher for giving us the permission for using the physical machines which had the capability of power capping. This project would not have been possible otherwise. We would like to thank the class of CS-533 itself for providing the opportunity to pursue this project. Finally, we would like to thank Akhil and Osman, the PhD students of Prof. Laxmikant V. Kale, for sharing their knowledge and inputs throughout the course of the project.

REFERENCES

[1] Intel, Intel-64 and IA-32 Architectures Software Developers Manual , Volume 3A and 3B: System Programming Guide, 2011..

[2] Osman Sarood, PhD Thesis 2013, Optimizing Performance Under Thermal and Power Constraints for HPC Data Centers.

[3] Intel, Intel Power Governor. http://software.intel.com/en-us/articles/intel-power-governor.

[4] R. K. BRUNNER AND L. V. KALÉ, *Handling application-induced load imbalance using parallel objects*, in Parallel and Distributed Computing for Symbolic and Irregular Applications, World Scientific Publishing, 2000, pp. 167–181.

[5] L. KALE, *The Chare Kernel parallel programming language and system*, in Proceedings of the International Conference on Parallel Processing, vol. II, Aug. 1990, pp. 17–25.

[6] L. V. KALE, A. BHATELE, E. J. BOHM, AND J. C. PHILLIPS, *NAnoscale Molecular Dynamics (NAMD)*, in Encyclopedia of Parallel Computing (to appear), D. Padua, ed., Springer Verlag, 2011.

[7] L. V. KALE AND S. KRISHNAN, *Charm++: A portable concurrent object oriented system based on c++*, in Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '93, New York, NY, USA, 1993, ACM, pp. 91–108.

[8] V. MEHTA, *LeanMD: A Charm++ framework for high performance molecular dynamics simulation on large parallel machines*, Master's thesis, University of Illinois at Urbana-Champaign, 2004.

[9] O. SAROOD, A. LANGER, L. V. KALE, B. ROUNTREE, AND B. DE SUPINSKI, *Optimizing power allocation to cpu and memory subsystems in overprovisioned hpc systems*, in Proceedings of IEEE Cluster 2013, Indianapolis, IN, USA, September 2013.