Mitigating Processor Variation through Dynamic Load Balancing

Bilge Acun, Laxmikant V. Kale
University of Illinois at Urbana-Champaign, Department of Computer Science
{acun2, kale}@illinois.edu

***Abstract*—There can be performance variation among same-model processors in large scale clusters, and supercomputers that are caused by power, and temperature variations among the processors. These variations manifest itself as frequency difference of the processors under dynamic overclocking, such as Turbo Boost. Different-model processors also create an inherent variation when used in same cluster. For some tightly coupled HPC applications even one slow processor in the critical path can slow down the whole application therefore this variation is an important problem. To mitigate the performance variation among processors, we propose a speed-aware dynamic load balancing strategy which works on both homogeneous and non-homogeneous hardware. Our main idea is to provide an estimation of the task completion time based when moving a task from one processor to another on the processor speed. We show up to 30% performance improvement using our speed-aware load balancer compared to the no load balancing case. We also show that our speed-aware balancer performs 5% better than non-speed aware counterpart.**

I. Introduction

Placement of cores, differences in node assembly, and other manufacturing factors can cause same-model processors to have different temperature and power consumption in a cluster. This difference can be evident from performance when the processors are under dynamic overclocking. Power, heat and cost prevent processors from running at maximum frequency constantly. Hence, dynamic overclocking technique is used in many modern processors to improve performance by increasing the frequency of the processors opportunistically. Intels Turbo Boost Technology is an example of this. When Turbo Boost is enabled, the frequency of the processor depends on: type of workload, number of active cores, estimated current consumption, estimated power consumption, processor temperature [1].

Turbo Boost improves the clock speed and therefore the application performance [2]. However, it can also cause performance variation among processors. We observe that there exists up to 30% execution time difference among same-model processors under Turbo Boost running the same local computational kernel, as shown in Figure 1. Such variations can lead to performance degradation, especially for tightly coupled HPC applications. A slow processor in the critical path, can slow down the whole application.

To understand the cause of this performance variation, we look into the frequency and temperature of the processors. Figure 3 shows the frequency and temperature trends of tree selected same-model processors with Turbo Boost turned on in a cluster. Node 42, 48, and 70 demonstrate 3 distinct behaviors. Node-42 is a typical fast node. During the whole experiment, the temperature of Node-42 remains under 80 °C, and its frequency stays at the maximum Turbo Boost frequency – 2.3GHz. On the other hand, Node-48 has a higher starting temperature and its temperature quickly reaches the thermal limit – 91 °C – after the execution started, and its frequency dropped till 1.5GHz as a result. Node 70 displayed a more interesting behavior as its temperature rose slowly. Even though their starting temperature was almost the same as Node-42, Node-70 reaches the threshold temperature and the frequency starts to get throttled. Figure 3 right-bottom plot shows the correlation between the frequency and temperature of all participating 20 nodes in the experiment. In summary, we observe a dynamic behavior in speed of the processors that can change over time.

Another type of frequency variation can happen when a cluster is composed of two different type of processor models. We propose a speed-aware dynamic load balancing

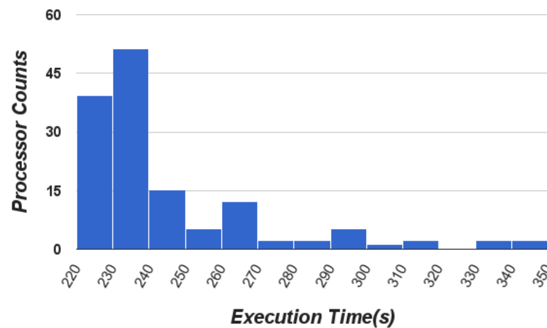


Figure 1: Histogram of execution time same-model for processors to run a matrix multiplication kernel. The variation is caused by frequency difference of the processors under Turbo Boost.

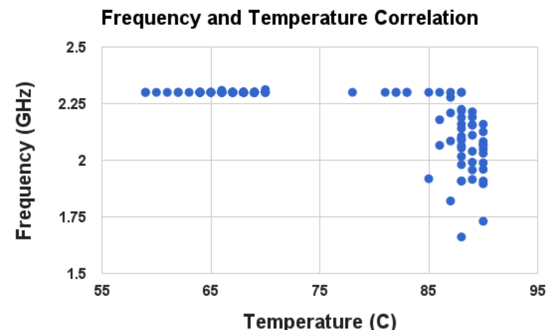


Figure 2: Bottom-right plot shows frequency is inversely correlated by the temperature of the processor.

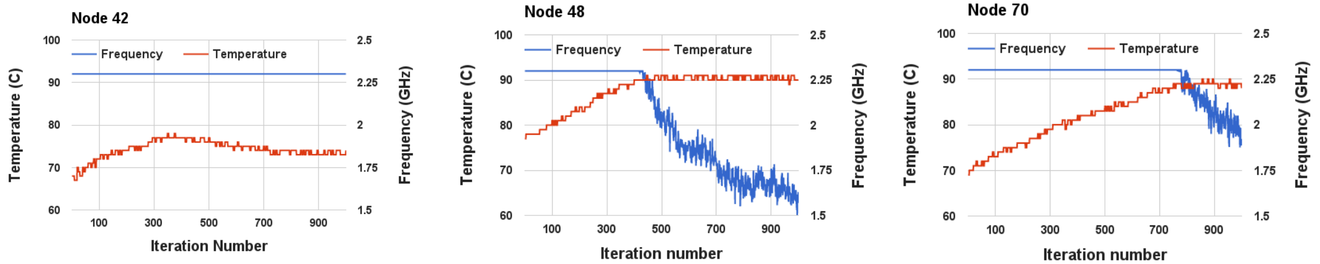


Figure 3: Turbo Boost causes different frequency behavior on same types of processors; node 42, node 48, node 70, even though they are running the same local computation kernel.

algorithm to resolve both types of the variation problem. We instrument application runtime to track the speed of the processors and balance the workload proportionally to the speed. We implement our intelligent load balancing algorithm in CHARM++ framework, and show the effectiveness of our load balancer with through evaluations on applications with various problem size and different cluster configurations; heterogeneous processors and homogeneous processors.

II. Experimental Setup

A. Platform

We use an 80 node local cluster at UIUC. The cluster is composed of two different processor types as shown in Table I. Some of our experiments use only one type processors to show the variation among same-model processors – *homogeneous setup*. Some experiments use both processor types at the same time – *heterogeneous setup*. We use model specific registers (MSR) to calculate the frequency of the processors. We read the temperature data of the cores through the $coretemp$ kernel driver.

Table I: Platform hardware details

Processor	Intel Xeon E5-2620	Intel Xeon X3430
Clock Speed	2.0 GHz	2.4 GHz
Max Turbo Speed	2.5 GHz	2.8 GHz
Cores	6	4
Cache size(L3)	15MB	8MB

B. Applications

Matrix Multiplication: This is a basic double precision dense matrix multiplication kernel. We run the sequential kernel in a loop on each core with data size fitting in the last level cache (L3). Specifically, we use three 248x248 double-precision matrices, which requires around 1.5MB data per core, which is in total smaller than the cache size of the processors. This eliminates the effect of memory access related performance variation in our timings.

Jacobi-2D: This is a 5-point stencil application on a 2D grid. The application uses CHARM++ parallel programming framework for parallelization. The grid is divided into multiple small blocks, each is represented as an object. We experiment with multiple different grid and block sizes.

III. Speed-Aware Dynamic Load Balancing

In this section, we propose solution based on dynamic load balancing to mitigate this variation problem. Task based parallel programming models is a good fit for solving this problem since it gives opportunity to redistribute the tasks among processors. For this purpose, we use CHARM++ parallel programming runtime system, where each task is represented as a C++ object that can migrate from one to another processor. The runtime collects statistics of how much time each object spends on executing the work, background load time etc. and use those statistics to balance work among processors. The load of the processor or object is the time it takes to execute. CHARM++ has a load balancing framework that provides various load balancing algorithms. In this paper, we look into refinement load balancing algorithm, *RefineLB* [3], and make it speed-aware.

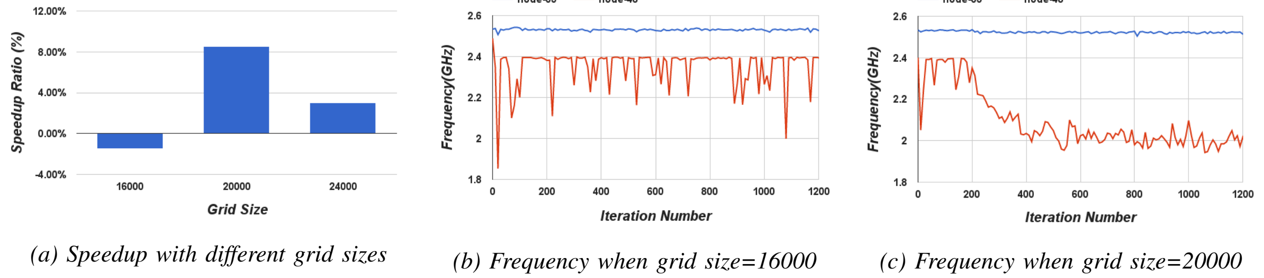
RefineLB moves heaviest objects from heavy processors to light processors until the heavy processor's load becomes average [3]. However, it does not estimate the load correctly when moving objects between processors if the speed of the processors are different. Because the load of an object can change when moved from a slow to a fast processor since the speed of the processors are different. A simple way to make this algorithm speed-aware is to scale the load with the processors speeds, i.e. multiply with the donor speed and divide by the recipient speed. To facilitate this, we modified CHARM++ framework to track the speeds of the processors dynamically. At the time of making the load balancing decision, i.e. whether to move an object or not, the load of object will be scaled by the tracked speeds of processors to give a more precise load estimation.

IV. Evaluation

In this section, we evaluate our speed-aware load balancing technique under different system configurations.

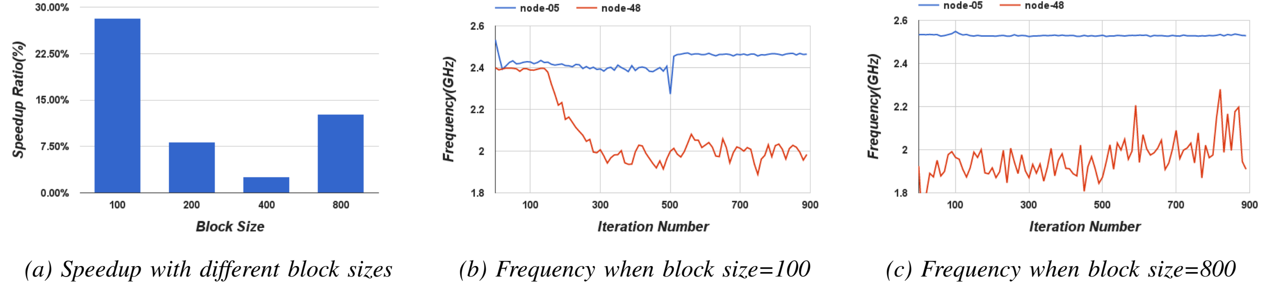
A. Heterogeneous System

In our first set of experiments, we use a heterogeneous system configuration where the processor types are different; i.e. half of them have 2.0 GHz base clock speed and the other half has 2.4 GHz. In Figure 6, we evaluate *RefineLB* and *Speed-aware RefineLB* performance against no load



(a) Speedup with different grid sizes *(b) Frequency when grid size=16000* *(c) Frequency when grid size=20000*

Figure 4: Speed-aware RefineLB performance on Jacobi-2D with different grid sizes.



(a) Speedup with different block sizes *(b) Frequency when block size=100* *(c) Frequency when block size=800*

Figure 5: Speed-aware RefineLB performance on Jacobi-2D with varying block size. Load balancer is triggered at iteration 800.

balancing performance. We use Jacobi-2D application with grid size as 20000 and block size as 200.

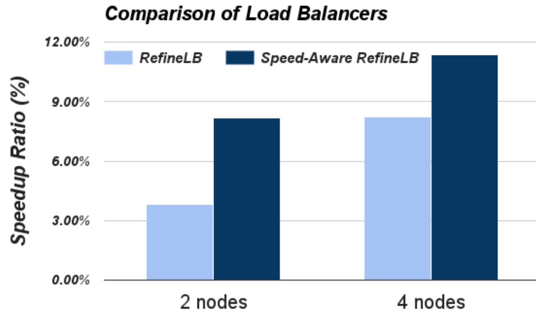


Figure 6: RefineLB and Speed-aware RefineLB performance compared to without load balancing case

RefineLB improves the performance by 4% and 11% on 2 and 4 nodes(processors) respectively. Speed-aware version performs around 5% and 3% better compared to *RefineLB* . By taking processor clock speed into consideration and combining it with history workload information, *Speed-aware RefineLB* has a more precise estimation of resulted workload of target processor after migration. Therefore, it can move objects more aggressively than non-speed-aware version. *Speed-aware RefineLB* migrates 204 whereas *RefineLB* migrates 95 objects out of 10000 objects from the fast processors to the slow processors.

Problem Size Effect: Different problem size of Jacobi-2D application varies the intensity of computation, communication and memory accesses. In the two sets of experiments below, we show that depending on the grid and block size, the benefit of load balancing changes.

Grid size is the total workload size of the array before divided into smaller blocks. We fix block size at 200 and vary grid size from 16000 to 24000 which is around 1.9GB to 4.2GB of data in total. Figure 4a shows effect of grid size with our speed-aware load balancer. We show the processor frequency behavior over iterations in Figure 4b and 4c. When the grid size is 16000, it takes 1.47% longer to finish execution, load balancer has negative effect on performance and both processors run close to their peak speed. Smaller grid size does not require much computation between two synchronizations (iterations), and processors turn into communication stage before heat up. The reason of the slowdown could be the higher communication overhead when the objects are moved from slow to fast processor. When grid size is set at 20000 and 24000, we have 8.51% and 3.03% performance gain respectively. The frequency of fast node does not change over time, while slow node's frequency drops after around 180 iterations from 2.4G GHz to 2.0 GHz. Larger grid size introduces more workload for both processors and leads to frequency throttling. On one hand, fast node only needs to spend shorter time to finish the work, then it gains some time to cool down before synchronization. On the other hand, slow node has no time to only cool down. When grid size increases to 24000 load balancer works less effectively because of the increased memory access latency.

Block size determines the workload of each migratable object. We show speedup ratios in Figure 5a with grid size fixed at 20000 and block size varying from 100 to 800. Processor frequency behavior with block size 100 and 800 are shown in Figure 5b and Figure 5c respectively. The highest performance improvement is 28% when block size is set as 100. Performance gains decreases to 8.2% and 2.5% when

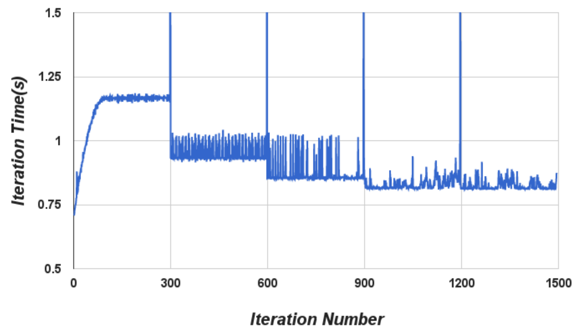


Figure 7: Homogeneous Processors under Turbo-Boost using Speed-aware RefineLB

block size is increased to 200 and 400. This can be explained as the following. Firstly, system creates more objects with smaller block size, resulting in heavier background load. Secondly, smaller block size is more flexible to migrate. In our load balancer algorithm, we conservatively compute workload of target processor after object migration. Larger objects tend to make target processor overloaded, and less likely to be moved. Moreover, when block size is 800, we see 12.7% performance improvement and we see objects moved from fast node to slow node. This inverse behavior could be caused by different cache size of the processors. Overall, load balancing still has significant performance benefit.

B. Homogeneous System

In Figure 7, we show our speed-aware load balancer can play an effective role in homogeneous system with a frequency variation. Here, we run Jacobi-2D with 24000 grid size and 400 block size on 2 same-model processor nodes. One of the nodes keeps running at 2.4 GHz, while the other node slowly throttles down from 2.4 GHz to 1.8 GHz due to it's high temperature. During every iteration, fast node has finishes computation earlier and has more time to cool down till the barrier synchronization after each iteration. Whereas slow node is overloaded and has no time to cool down. Load balancer is triggered every 300 steps. Since the slow processors get throttled down more and more over time, one load balancing is not enough to balance the load. Only after first three migrations, performance stabilizes. Compared to execution time before the first load balancer, we get 30% improvement after last migration.

Overhead of the load balancing does not exceed 2 seconds in all experiments shown. It is a small overhead which is compensated after a few iterations. For larger node-counts, more scalable load balancing algorithms can be made speed-aware in a similar way we made *RefineLB* speed-aware.

V. Related Work

Earlier work shows the existence of temperature variation in large scale supercomputers [4], performance and power variation under power capping [5]. Sarood et al. uses dynamic voltage and frequency scaling (DVFS) technique to remove hot-spots/reduce power and energy consumption [6]. Our goal is not to reduce power consumption or increase energy efficiency, but to get better performance by mitigating the process variation. Hammouda et al. propose noise tolerant stencil algorithms to mitigate the performance variations caused by including dynamic power management, OS jitter etc. with a static mapping approach [7]. Choi el al. proposes thermal-aware task scheduling [8] and Langer et al. proposes variation-aware task scheduling among cores of many-core processors [9]. Our approach is dynamic, and with not just among cores but also among processors.

VI. Conclusion

In this paper, we first show the variation among the same-model processors and how this variation can degrade the performance of the parallel application that runs on those. The variation among processors can increase as the scale gets bigger in future supercomputing systems, therefore it is important to find ways to reduce the negative performance effects of the variation. We suggest that dynamic load balancing techniques can be used to mitigate the variation on homogeneous and heterogeneous platforms. Our primary results show up to 30% performance benefit using our speed-aware load balancer compared to the no load balancing case.

References

[1] Intel Corporation, "Intel Turbo-Boost Technology." [Online]. Available: http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

[2] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova, "Evaluation of the intel® core i7 turbo boost feature," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 188–197.

[3] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction and load balancing," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[4] K. Zhang, S. Ogrenci-Memik, G. Memik, K. Yoshii, R. Sankaran, and P. Beckman, "Minimizing thermal variation across system components," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 1139–1148.

[5] Y. Inadomi, T. Patki, K. Inoue, M. Aoyagi, B. Rountree, M. Schulz, D. Lowenthal, Y. Wada, K. Fukazawa, M. Ueda *et al.*, "Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing," in *SC'15*. ACM, 2015, p. 78.

[6] O. Sarood and L. V. Kalé, "Efficient'cool down'of parallel applications," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. IEEE, 2012, pp. 222–231.

[7] A. Hammouda, A. R. Siegel, and S. F. Siegel, "Noise-tolerant explicit stencil computations for nonuniform process execution rates," *ACM Transactions on Parallel Computing*, vol. 2, no. 1, p. 7, 2015.

[8] J. Choi, C.-Y. Cher, H. Franke, H. Hamann, A. Weger, and P. Bose, "Thermal-aware task scheduling at the system software level," ser. ISLPED '07. ACM, 2007, pp. 213–218.

[9] A. Langer, E. Totoni, U. S. Palekar, and L. V. Kalé, "Energy-efficient computing for HPC workloads on heterogeneous manycore chips," in *Proceedings of Programming Models and Applications on Multicores and Manycores*. ACM, 2015.