LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-739564

PANN: Power Allocation via Neural Networks - Dynamic Bounded-Power Allocation in High Performance Computing

W. Whiteside, S. Funk, A. Marathe, B. Rountree

October 6, 2017

Energy-Efficient SuperComputing (E2SC) Workshop at
SuperComputing 2017 Conference
Denver, CO, United States
November 13, 2017 through November 13, 2017

Disclaimer

PANN: Power Allocation via Neural Networks

Dynamic Bounded–Power Allocation in High Performance Computing

Will Whiteside
Shelby Funk
wew@cs.uga.edu
shelby@cs.uga.edu
Department of Computer Science
University of Georgia

Aniruddha Marathe
Barry Rountree
marathe1@llnl.gov
rountree4@llnl.gov
Lawrence Livermore National Laboratory
Department of Energy

ABSTRACT

Exascale architecture computers will be limited not only by hardware but also by power consumption. In these bounded power situations, a system can deliver better results by overprovisioning – having more hardware than can be fully powered. Overprovisioned systems require power to be an integral part of any scheduling algorithm. This paper introduces a system called PANN that uses neural networks to dynamically allocate power in overprovisioned systems. Traces of applications are used to train a neural network power controller, which is then used as an online power allocation system. Simulation results were obtained on traces of ParaDiS and work is continuing on more applications. We found in simulations PANN completes jobs up to 24% faster than static allocation. For tightly constrained systems PANN performs 6% to 11% better than Conductor. A runtime system has been constructed, but it is not yet performing as expected, reasons for this are explored.

ACM Reference Format:
Will Whiteside, Shelby Funk, Aniruddha Marathe, and Barry Rountree. 2017. PANN: Power Allocation via Neural Networks: Dynamic Bounded–Power Allocation in High Performance Computing. In *E2SC'17: E2SC'17: Energy Efficient Supercomputing, November 12–17, 2017, Denver, CO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3149412.3149420

1 MOTIVATION

As high–performance computing reaches towards exascale systems, power is becoming a critical constraint. An exascale computer built using modern hardware would consume levels of power that are both physically and fiscally impractical[8]. Future high–performance computers will have a power constraint, and optimizing performance under this constraint will be necessary. This adds an additional requirement for scheduling high–performance computing systems – one where automated systems can provide better results than manual systems. Thus this problem should be handled by an automated system that requires as little input from the users as possible.

This paper presents PANN, Power Allocation via Neural Networks, a distributed runtime power allocation system using artificial intelligence. The system consists of a neural network[1] controller on each rank to craft a request for power. The neural network is trained using a genetic algorithm and traces of previous similar jobs. The idea of training neural networks by genetic algorithm was introduced by Montana et al.[7]. A copy of the neural network is run on each rank at the end of each execution phase. The power requests are collected and distributed to the ranks in such a way that each rank can determine its own power bound for the next time step. PANN adheres to two constraints: (1) the power used must never go above the power bound, and (2) overhead must scale linearly with the number of processors. Simulation results demonstrate that this system is capable of completing simulated jobs up to 24% faster than static power allocation and up to 11.5% faster than Conductor.

This paper deals with allocating power on high performance computing jobs that follow a particular set of constraints:

- A job splits the work across many MPI ranks
- The work is further split into many smaller tasks
- Each task has an execution phase followed by a communication phase
- Communication is done in an all–to–all manner
- The next execution phase cannot start until the all–to–all communication phase completes
- A similar job on similar architecture will have similar work distribution properties

These constraints are satisfied by many real world applications. Tests were run with ParaDiS[4], a "large scale dislocation dynamics simulation code to study the fundamental mechanisms of plasticity[3]." It partitions the work across many ranks and simulates each rank independently for a timestep, but, due to the highly interconnected nature of this problem, it still uses a global synchronization after each timestep to maintain a global state of the model. This is a not uncommon application model, and is a good fit with the goals of this project.

1.1 Motivating Example

Figure 1 is a diagram of a synthetic job consisting of four ranks executing ten tasks each, with and without a power controller. The color of the rectangle shows the power allocation to that rank. The power controller determines that rank 2 can complete on time with a lower power allocation and allocating more power to rank 3 reduces the execution time of the critical path. Using a power controller that makes this change (as well as a bevy of other, less obvious changes), this job shows a 16% speedup under the power bound. It

[1] For an introduction to neural networks and genetic algorithms see [9].

E2SC'17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5132-4/17/11…$15.00
https://doi.org/10.1145/3149412.3149420

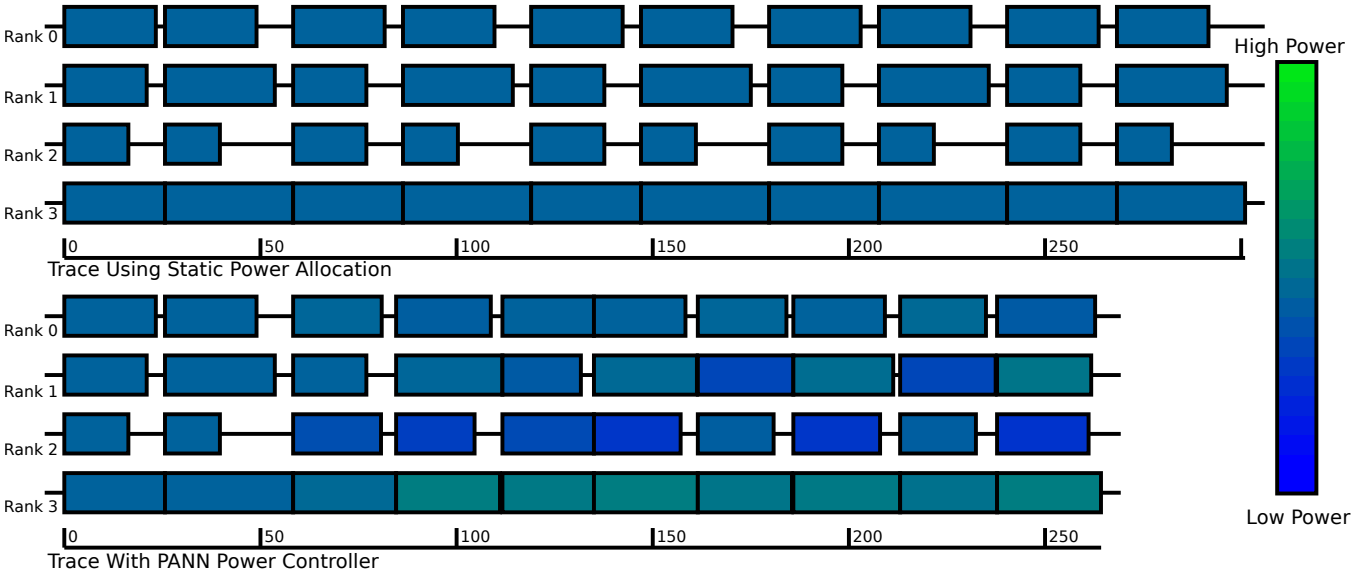

Figure 1: Diagram of a synthetic job showing of ten tasks each on four ranks. The boxes represent execution phases and the lines represent communication phases. Note that the scales are the same and the trace with the power controller finishes approximately 16% sooner.

also decreases the internal wait times for communication phases to finish, so less power is wasted waiting for MPI calls, however the goal of the controller remains to finish a job as quickly as possible, and any power savings are merely incidental.

2 A DISTRIBUTED APPROACH

A common metric used in predicting power usage is utilization – the proportion of time spent in the execution phase divided by combined time of the execution and communication phases. This metric is particularly useful in predicting power usage – low utilization means that the rank can be run at a lower power and not affect overall runtime, and high utilization indicates that apportioning more power to this rank would speed up the system.

Much of the current work on power management in high–performance computing has been focused on job–independent online scheduling. For example the systems Conductor[6] and POWsched[5] use utilization history to choose the correct run configuration and power bound. These systems are hampered by making no attempt to predict how power needs will change in the next execution step. There is room for improvement over these systems by using a system that can use patterns in execution times to predict future power needs and by reducing overhead, both in communication and power controller execution.

Since each rank is in a good position to predict its future power consumption needs, a distributed system can be a good fit. Each rank tracks its power and utilization history, and separately computes power requests. Minimizing interprocess communication increases scalability at the cost of less precise decisions on power allocation.

2.1 Global Synchronization

Communication in many applications surveyed [1][2][3] follow an all–to–all communication network, where all the ranks communicate with each other. This strategy, while not ideal for peak computing performance, is in heavy use, and systems designed to improve throughput on this strategy can have a large impact on real world applications. Since the different runtimes result in different ranks reaching the all–to–all communication phase at different times, most of this idle time is spent in the first all–to–all MPI call of a communication phase. In some applications over 90% of the time is spent waiting for MPI calls[1], and a system that balances runtimes by adjusting power allocated to ranks would result in much faster performance.

This all–to–all communication assumption is made at the MPI level, and an application that uses hybrid MPI/OpenMP parallelism would be a good fit for the PANN controller if most of the MPI communication is global. In this case PANN would control power allocation among the MPI ranks and the OpenMP parallelism would be handled by a different system. Conductor [6] demonstrates the efficacy of a system to maximize OpenMP performance under a processor level power bound while using an MPI level power allocation system to determine these power bounds.

3 THE PANN SYSTEM

PANN uses data from previous runs of an application to produce a power controller to speed up future runs of that same application. First, several runs of an application are run with tracing software to extract relevant data. From these runs, the execution and communication phases are collected, as well as statistics about communication networks.

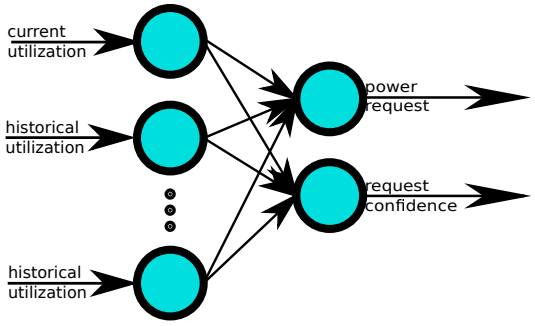

Figure 2: Inputs and outputs of the neural network controller

These traces are then used as a training set for a genetic algorithm to train a neural network to produce the minimum root mean square runtime on the traces. The genetic algorithm produces a great many candidate neural network power controllers. These controllers are selected for fitness by running a simulation of the system, using the trace data. Formally, the genotype of the genetic algorithm is the weights of the neural network, the fitness function is the root mean square runtime of the training set of traces using the neural network controller, and the crossover and mutation operations are standard neural network/genetic algorithm operations[9]. Once the neural network is trained, it is assessed by simulating a previously unseen trace and measuring the speedup. Experimentation with mutation operators changing the size of the network or the connection graph and incorporating backwards propagation into the training did not substantively change the results, but did increase the training time, and thus are no longer included.

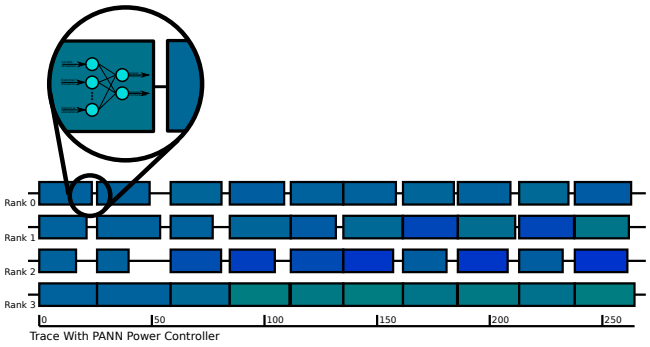

Figure 3: A detail showing where the neural network is used in PANN

As a power controller, PANN works by having a trained neural network craft a request at the end of each execution phase. Figure 2 shows the request consists of a power need and a confidence score. This combination allows a rank to make many different statements; a rank that needs more power (is at a high utilization) can make a high request with high confidence, likewise a rank that has too much power (is at a low utilization) can make a low request with high confidence. If a rank is at some middle utilization it can declare a low confidence in its request, and that will be taken into account. The confidence score is used as a cumulative vote on the amount of power to put up for distribution. This allows the system to trend toward a stable fastest allocation (if one exists) by allowing ranks that appear to be near their stable values to give low confidence and ranks far from their stable values to give high confidence, then the power that is voted to be up for allocation is distributed according to the requests made. Since this is an artificial intelligence system, the definitions of high, low, and middle are learned by the model and do not need to be specified by the user. As a note, the request scores are used as percentage of total requests, and are thus unitless (the exact values do not correspond to the wattages for the power bound) and the confidence scores are interpreted as on a largely 0 to 1 scale, but the system does not actually constrain them.

Additionally, the power currently allocated to this node is appended to its request. This is used in the calculations to correct rounding errors that would otherwise accumulate and cause the system to not fully utilize the power allocation.

These requests are summed with three all–reduce calls, one for requests ($r_{t,i}$) one for confidence ($c_{t,i}$) one for power ($p_{t,i}$). These sums are then be used independently by each rank to determine the amount of power it has been allocated for the next execution phase using the algorithm given below. Since the only global calculation is summing the requests, there is no single pinch point and all the calculation can be completely distributed. The CPU level power bound is changed before the beginning of the next execution phase. When rounding is performed, the system always rounds down, to keep the power bound strict. We have found the rounding error to be negligible – with the correction factor, it remains within a single unit of power allocation per processor for the entire runtime, and can be made as small as desired by using finer resolution power allocations.

For ease of expression, let $C_t = \frac{1}{n}\sum_{j=0}^{n-1} c_{t,j}$ and $R_t = \sum_{j=0}^{n-1} r_{t,j}$ and $P_t = \sum_{j=0}^{n-1} p_{t,j}$.

The algorithm to apportion power uses three cases based on the sum of the confidence scores. In the case that $C_t \leq 0$ the algorithm transfers no power at all, leaving the power in the previous configuration. On the other hand, when $C_t \geq \frac{1}{2}$ the power is apportioned based directly on the requests $p_{t+1,i} = \frac{r_{t,i}}{R_t} P_{bound}$. Below is the formula used when $0 < C_t < \frac{1}{2}$.

$$p_{t+1,i} = p_{t,i}\left(1 - C_t\right) + \left(\frac{r_{t,i}}{R_t} C_t\right)\left(2P_{bound} - P_t\right) \quad (1)$$

Claim: This algorithm will never allow the total power to go above the allocated power. More formally:

$$\sum_{j=0}^{n-1} p_{t,j} \leq P_{bound}\ \forall t \quad (2)$$

PROOF. The algorithm begins by setting $p_{0,i} = \frac{P_{bound}}{n}$.

Case 1: $C_t \leq 0$

In this case the algorithm transfers no power at all, so $p_{t+1,i} = p_{t,i}$.

Case 2: $C_t \geq \frac{1}{2}$

In this case the algorithm apportions power according to this formula:

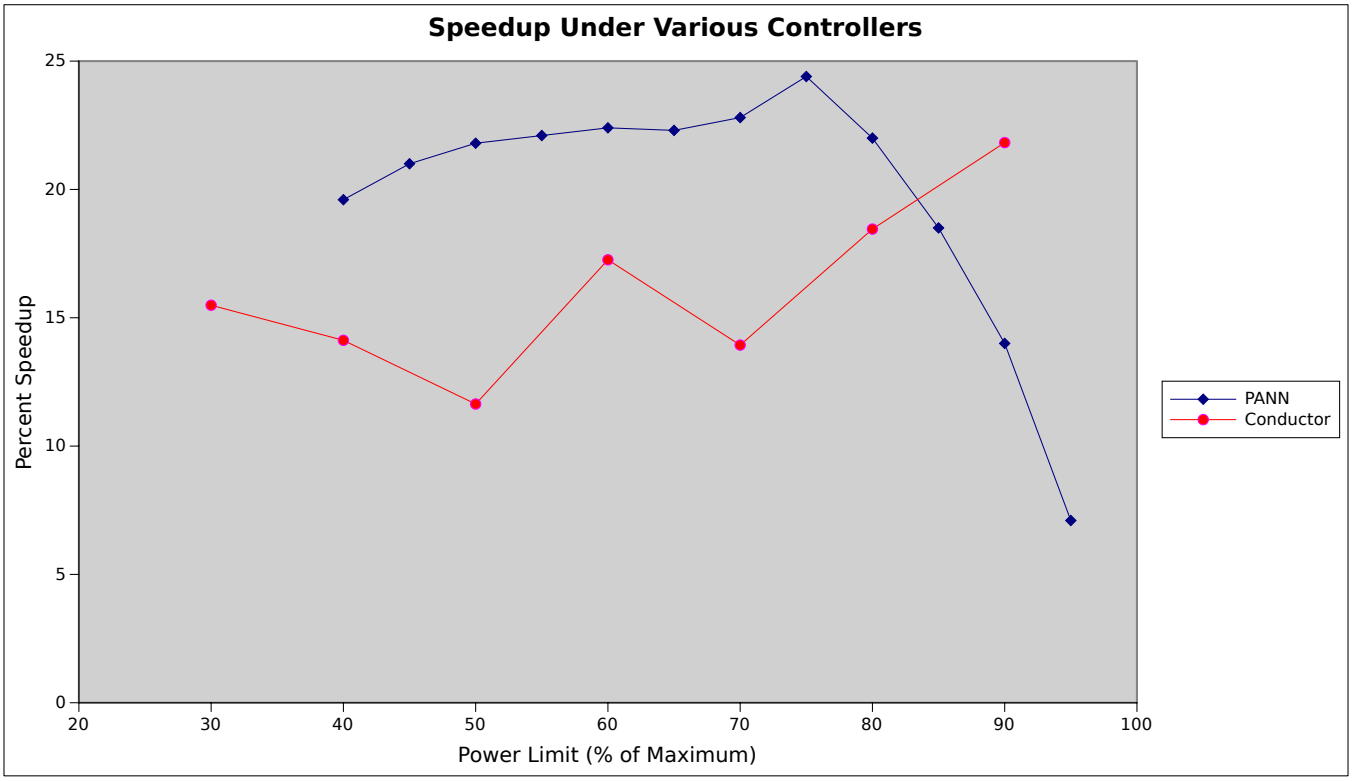

Figure 4: Speedup comparison between simulated PANN and Conductor

$$p_{t+1,i} = \frac{r_{t,i}}{R_t} P_{bound} \tag{3}$$

This sums to P_{bound} and, with rounding down, is guaranteed to be at or below the power bound.

Case 3: $0 < C_t < \frac{1}{2}$

Then the algorithm will use the formula given above:

$$p_{t+1,i} = p_{t,i}\,(1 - C_t) + \left(\frac{r_{t,i}}{R_t} C_t\right)(2P_{bound} - P_t) \tag{4}$$

Summing all the indices i gets:

$$\sum_{i=0}^{n-1} \left(p_{t,i}\,(1 - C_t) + \left(\frac{r_{t,i}}{\sum_{j=0}^{n-1} r_{t,j}} C_t\right)(2P_{bound} - P_t) \right) \tag{5}$$

Noting that some of the variables do not depend on i gives us:

$$\sum_{i=0}^{n-1} p_{t,i}\,(1 - C_t) + \left(\frac{\sum_{i=0}^{n-1} r_{t,i}}{R_t} C_t\right)(2P_{bound} - P_t) \tag{6}$$

Canceling some common terms gives:

$$\sum_{i=0}^{n-1} p_{t,i}\,(1 - C_t) + (C_t)\,(2P_{bound} - P_t) \tag{7}$$

Distributing gives:

$$P_t + 2C_t P_{bound} - 2C_t P_t \tag{8}$$

Factoring gives:

$$(1 - 2C_t)P_t + 2C_t P_{bound} \tag{9}$$

which since by induction $P_t \leq P_{bound}$ and $1 - 2C_t \geq 0$ means:

$$\begin{aligned}(1 - 2C_t)P_t + 2C_t P_{bound} &\leq (1 - 2C_t)P_{bound} + 2C_t P_{bound} \\ &\leq P_{bound}\end{aligned} \tag{10}$$

so:

$$P_{t+1} \leq P_{bound} \tag{11}$$

This middle confidence level allows the controller to make small adjustments to the allocation. This is used for all the confidence levels between the two cases above and tends to be the most common action in the experiments.

3.1 Handling Different Granularity MPI Node and Power Configuration

On the systems used for testing PANN, the configuration presented a particular problem. The MPI ranks were assigned at the core

level, whereas the RAPL could only change the power level at the socket level. This means that there are a group of MPI ranks that all must be assigned the same power bound. There are several potential strategies to deal with this problem. For consistency between this restriction and systems without these restrictions – e.g. new Intel processors that allow a per-core or even more granular power bound – PANN uses an averaging system for setting the power bound on a core. For a cores on socket J set the power level $\rho_{t+1,J}$ to:

$$\rho_{t+1,J} = \frac{\sum_{j \in J} p_{t+1,j}}{N} \tag{12}$$

This satisfies the two requirements of making the power bound strict and giving the ability to change power bounds on runtime systems. This does reduce the ability to choose power levels at small granularity, and thus reduces the ability of the system to produce speedups.

3.2 Comparison to Conductor

Conductor[6] is an online dynamic power management system. It begins by characterizing performance levels with various configurations of OpenMP threads at various power levels at the beginning of a job. It then apportions power according to previous time steps and choosing the configuration that performs best at that power bound. This works well on small to medium jobs. On large jobs it requires a good deal of communication overhead to accumulate the history onto a single node and then this node requires a rather long execution time to apportion the power optimally.

Because PANN has very few all–reduce calls and the computation of power allocation is distributed across all of the ranks, it has significantly less overhead than comparable systems. The schedules produced by this system have fewer guarantees, and, being designed by an artificial intelligence system, do not have an easily expressible goal for the method used to allocate power. While these design parameters seem to indicate that the power allocation of this system must be inferior to that of Conductor, simulated experiments shows that is not the case. The most likely reason for this is that the artificial intelligence training causes the individual ranks to work in concert, making requests that are as accurate as possible, while not being greedy. Another possible reason that this system usually performs better than Conductor is much simpler; the application and computing system is noisy and an exact solution to the previous time step is an inexact solution to future time steps.

4 RESULTS

To test PANN both a simulation and a runtime system were created. Both of these are necessary components, as the simulation is used both as a training device and as a validation tool. The runtime system is not yet performing as expected, it currently behaves similar to the static power allocation, and requires further tuning. Positive experiments on the simulation lead to expectation that the runtime system will be effective.

4.1 Training

For this paper, three traces were used as a training set. These traces were extracted from runs of ParaDiS at full power. These traces need to be similar to the applications that the power controller will be used on, and the greater the similarity, the better the controller will perform. However, some testing has been done using small traces, with few ranks and few timesteps, to produce controllers for larger traces running for longer, and good results were found.

While the traces must be extracted from the target system, or a similar system, the training can be performed on any system. This means that the training can be taken offline and performed on whatever system is convenient. The training runs were allowed to run for about 12 hours on a single core of a Xeon E5 processor, and used less than one gigabyte of RAM, so are comparable to a consumer desktop. This shows that cheaper processor time can be used to save more expensive processor time, and if the trained system is used several times – or even just once on a large run – the total power used to run a program can be reduced.

Once a PANN network is trained, it can be deployed either in the simulation for validation or in the runtime system.

4.2 Simulation Results

Experiments were run on Lawrence Livermore National Laboratory's Cab and Catalyst, both Intel Xeon parallel compute clusters. Three sample runs of ParaDiS on eight MPI ranks, using test inputs from the ParaDiS test set (Copper[1-3]), were run on Catalyst to produce the training set. The genetic algorithms and its simulations were run on a single thread on Catalyst to train the model. This training phase can be as long or as short as the user desires. Simulation results for PANN were created by simulating with a new test trace (using Copper4) that was not a part of the training set.

Tests were run with sample inputs to ParaDiS producing several execution traces. These traces were used as a training set by PANN to produce a neural network controller for each power bound. We compared both PANN and Conductor to runs using static power allocation where all ranks were allocated the same power level. This gives a fair comparison of the speedups due to each controller and controls for any differences between computer systems.

These neural network controllers were run on the test trace extracted from ParaDiS and the speedups were calculated. Speedup here is defined as the percent reduction in time from simulating the trace with static power versus the simulated time of the trace with either PANN or Conductor. These speedups are graphed in figure 4. The highest speedup was found running the simulated processors at an average of 75% of their maximum performance power, with the speedup decreasing as the power allocation moves away from this value. It is expected that there would be no speedup at 100% power – there is nothing to do but allow the processors to work at their maximum speed. The most notable feature of this graph is the relatively consistent 22% to 25% speedups in the 40% to 75% power bound range. This indicates that PANN can perform well over a range of power levels. This is an important result because the goal of PANN is to be a part of a larger scheduling system that manages both time and power allocations, and having a large range of power allocations under which an application performs well increases the usability of the scheduler, allowing the computer to change its effective characteristics based on the users' needs at the time.

Conductor was tested with ParaDiS on the same input file at a range of power bounds. The Conductor speedups are also graphed

in figure 4. This shows that PANN usually performs better than Conductor at the same power bound, but that at higher power bounds Conductor does better. We speculate that this is due to PANN's dynamic nature – when large changes need to be made, the neural network in PANN does a better job of predicting and making those changes, but at high power allocations there is not much room for changes. In these cases the deterministic and exact Conductor performs better. Conductor's relative decrease in performance as the power allocation decreases indicates that Conductor does not perform as well as PANN over as wide of range of power bounds. Additionally, this test was over a small set of processors, so any communication overheads were minimized and the calculation overhead is virtually non-existent.

4.3 Runtime Results

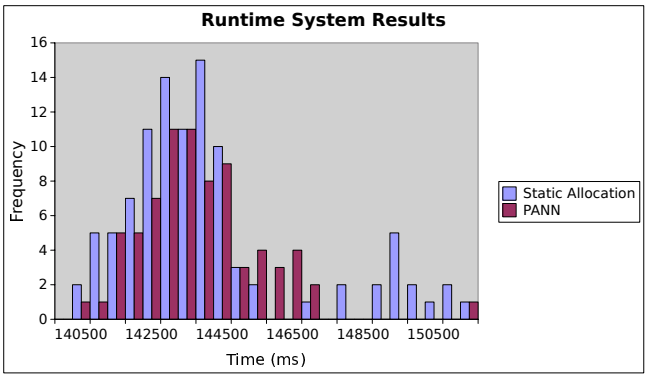

Figure 5: PANN comparison with static power allocation on test set.

We created a runtime system, which we tested on Catalyst, the results of which are shown in figure 5. The median runtime of the PANN test was 143,634ms, while the median runtime of the static power allocation was 143,247ms. This represents a slowdown of 0.27%, statistically indistinguishable from zero. Unfortunately due to time constraints we were unable to do the level of tuning and optimizing we would like to improve the runtime system's performance.

There are several potential reasons the runtime system is not giving results on par with the simulations. The most significant reason appears to be inter-run variation. Though the system was trained on several runs of the program, the results show that the distribution of runs was insufficient for training purposes, and more runs, or more variation in the training runs, are required to produce better results.

ParaDiS runtimes are highly stochastic, varying upwards of 10% on total runtime and even higher on individual task runtimes. This randomness Figure 6 shows the variation measured in the controllerless runtimes. This randomness may be causing additional issues in the real system as opposed to the simulations.

Additionally, the implementation shows that the neural network needs more accurate training. This may need to be accomplished with more training traces or with traces more similar to the objective trace. Obtaining these similar traces would not be prohibitive in the real world, as often ParaDiS is run for a period, the results are

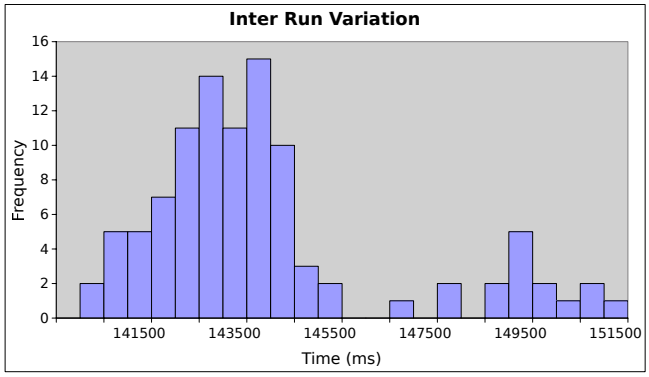

Figure 6: Variation in runtime in ParaDiS. All runs used exactly the same inputs and were performed on the same cluster.

recorded, and then the run is restarted from that point with some updated parameters, usually re-meshing. These earlier runs should be a good source of training data for the later runs.

Another factor that would improve the runtime system is better tuning of the genetic algorithm.

However some useful information can be gleaned from this implementation. Disabling the power bound setting part of the controller allows a real-world analysis of the overhead. For this test a ParaDiS with a mean runtime of 143.90 seconds was run many times with and without the controller attached. The runs with the controller calculating but not changing power resulted in a mean runtime of 144.05 seconds, for a difference of means of 149.15 ms, or 0.1%. This accounts for most of the potential sources of slow downs, the MPI communication calls and the calculation of the requests and the power limits. The only overhead unaccounted for is the RAPL method for changing the power bound. This is largely negligible, as Intel's RAPL system has very fast methods for changing the power bounds as well as the fact that this cost would be incurred by any system that balances load by allocating power. This indicates that the real world overhead is insignificant.

4.4 Additional Testing

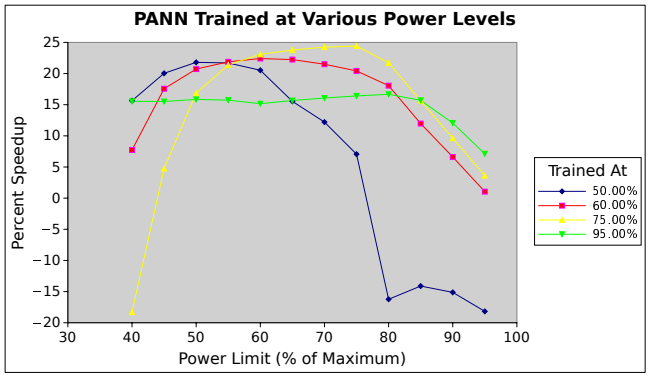

Figure 7: Simulated speedup of various PANN controllers at various power levels

As an additional test, the neural network controllers trained at various power levels were simulated at the whole sweep of power levels to judge how well a controller would perform outside of the training parameters. Figure 7 shows the results of this test. This shows that some controllers performed well when they were close to their training level and poorly when the power bound was significantly different. Most notable is the controller from the 75% power bound, it performed the best on the largest sweep (from 60% to 80%). However, it performed by far the worst on the lowest power bound, actually slowing the completion down by 18%. Similarly, the controller from 50% did well in the low power range and poorly in the high power range.

Conversely, the controller from 95% performed evenly across the whole sweep, and actually performed worst on the power bound it was trained on. This seems to show that it has learned some of the pattern, but did not learn as much because at the 95% power bound it did not have as much wiggle room to learn. Comparing the 50% and 95% bounds appears to be a near textbook example of overfitting, the 50% fits very well at 50%, but does very poorly at everything ¿ 80%, whereas the 95% one does about the same on all the power levels all the way down to 40%. Strangely, the controller trained at 60% did not perform as well at the 60% power bound as the controller trained at 75%. This is an odd result, but it is likely due to the 60% controller getting stuck in a local minimum in the learning phase. Notably missing from this chart are controllers trained at 40% and 45%. These controllers were excluded from the chart because they performed so poorly on power bounds from 60% to 95% that it broke the scale, at some points almost doubling the runtimes from the static power allocation. This illustrates one of the weaknesses of artificial intelligence based approaches, particular care must be used to guarantee the maximized fitness function corresponds to the actual goal of the user. In this case the goals and fitness functions may seem to be fairly close – maximize speedup at 40% versus maximize speedup over a range of values – but it turns out that those are different enough to produce negative results[9].

The results gleaned from this analysis are that if at all possible, the controller should be trained at the power allocation that the process is going to be run at. However, it seems that the performance does not degrade too much as long as the power bound is relatively close to the training bound. If the allocation and training bound are too different, the controller can actually degrade performance, so care should be taken to choose a controller appropriate to the power allocation in use.

5 CONCLUSION AND FUTURE WORK

This paper presents PANN, a novel online distributed power allocation controller for power–bounded high–performance computing. Using artificial intelligence increases applicability and performance. The controller adds minimal execution and communication overhead and produces speedups when operating under a power bound. It simulates up to a 24% speedup over static power allocation on traces of real jobs and 6% to 11% speedups compared to Conductor.

PANN works by collecting traces of an application. These traces are used as a training set of a genetic algorithm, which uses the traces to train a neural network to allocate power during those traces dynamically. Once this neural network is trained it can be deployed to dynamically allocate power on running programs.

Currently, the training phase requires logging previous runs of an application, but future versions will have the ability to learn automatically during the run of an application. Using the online training version will allow PANN to be used on a wide range of platforms and programs with no work from the application developer.

In the opposite direction, we plan to add functionality to PANN that allows developers to give hints about the power needs and utilize that information to better allocate power. These hints will be used as part of the learning process, so the controllers could learn exactly what the developers mean by their hints, without requiring developers to understand the underlying power demands of their applications.

Currently this research only considers CPU power. Future versions could take memory and cooling power into account, as well as using actual input power, measured at the power supply. This also has great promise in being used to control cooling systems, allowing a fan to be spun up before the processor starts to warm, maintaining efficiency and reducing overall cooling costs. We also plan to have a similar system that is is designed to improve performance on applications with more localized communication patterns.

6 ACKNOWLEDGEMENTS

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE–AC52–07NA27344 (LLNL-CONF-739564). Also, we would like to thank Walt Whiteside and Brandon Posey.

REFERENCES

[1] AMG. https://codesign.llnl.gov/amg2013.php.
[2] CoMD. https://github.com/exmatex/CoMD.
[3] ParaDiS. paradis.stanford.edu.
[4] BULATOV, V. E. A. Scalable line dynamics in paradis. Supercomputing (2004).
[5] ELLSWORTH, D., MALONY, A., ROUNTREE, B., AND SCHULZ, M. Dynamic power sharing for higher job throughput. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (2015), 80:1–80:11.
[6] MARATHE, A., BAILEY, P., LOWENTHAL, D., ROUNTREE, B., SCHULZ, M., AND DE SUPINSKI, B. A run-time system for power-constrained hpc applications. High Performance Computing (2015), 394–408.
[7] MONTANA, D., AND DAVIS, L. Training feedforward neural networks using genetic algorithms. In International Joint Conference on Artificial Intelligence (1989), vol. 89, pp. 762–767.
[8] PAWLOWSKI, S. Exascale science: the next frontier in high performance computing. International Conference on Supercomputing (2010).
[9] RUSSELL, S., AND NORVIG, P. Artificial Intelligence: a Modern Approach. Pearson, 2003.