2015-01-01

ISO-Power-Efficiency: An Approach To Scaling Application Codes With A Power Budget

Rogelio Long

ISO-POWER-EFFICIENCY: AN APPROACH TO SCALING APPLICATION CODES WITH A POWER BUDGET

ROGELIO LONG

Master's Program in Computational Science Program

APPROVED:

Shirley Moore, Ph.D., Chair

Barry Rountree, Ph.D., Co-chair

Vladik Kreinovich, Ph.D.

Rodrigo Romero, Ph.D.

Ramon Ravelo, Ph.D.

Charles Ambler, Ph.D.
Dean of the Graduate School

Dedication

To my daughter Rayne.

ISO-POWER-EFFICIENCY: AN APPROACH TO SCALING APPLICATION CODES WITH A POWER BUDGET

by

ROGELIO LONG, BS Mathematics

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Program in Computational Science

THE UNIVERSITY OF TEXAS AT EL PASO

May 2015

Acknowledgements

I am very grateful for my supervisor, Dr. Shirley Moore, whom never gave up on me and whose guidance and support helped to achieve this goal.

I cannot express enough thanks to my committee: Dr. Shirley Moore, my committee chair; Dr. Barry Rountree, my committee co-chair; Dr. Vladik Kreinovich; Dr. Rodrigo Romero; and Dr. Ramon Ravelo. I offer my sincere appreciation for the learning opportunities provided by my committee.

I am highly indebted to my parents. Without your support I would have most likely not made it through graduate school. Thank you for being the best parents anyone could ask for.

I am grateful for all my friends for always throwing ideas around and for providing me an outlet for relieving stress.

Finally, to my caring, loving, and supportive soon to be wife, Marisol: my upmost gratitude. Your encouragement is what got me through all the hard times. You are truly the best.

Abstract

For many applications, speedup saturates and parallel efficiency decreases if the problem size is held fixed while increasing the number of processors. For some problems, it is possible to maintain a fixed parallel efficiency by increasing both the problem size and the number of processing elements. The rate at which the problem size must increase to maintain constant efficiency for a given rate of increase of the number of processors is given by the iso-efficiency function. We have developed a new scalability function called *iso-power-efficiency* that determines the rate at which the problem size must increase to maintain constant efficiency for a given rate of increase in the application's power budget. For a given power budget, an application can choose to use a larger number of processors running at lower power. We show that such overprovisioning can lead to better scaling behavior.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

For many applications, speedup saturates and parallel efficiency decreases if the problem size is held fixed while increasing the number of processors (the form of scaling known as strong scaling). For some problems, it is possible to maintain a fixed parallel efficiency by increasing both the problem size and the number of processing elements. The rate at which the problem size must increase to maintain constant efficiency for a given rate of increase of the number of processors is given by the iso-efficiency function [1]. We have developed a new scalability function called *iso-power-efficiency* that determines the rate at which the problem size must increase to maintain constant efficiency for a given rate of increase of the application's power budget. For a given power budget, an application can choose to use a larger number of processors running at lower power. As shown in [2], speedup can often be obtained within a given power budget by such overprovisioning. Deriving the iso-power-efficiency function for a given problem involves 1) determining optimal configurations for problem instance/power budget pairs, and 2) expressing the parallel overhead as a function of problem size and power budget. We hypothesize that the rate of growth required for problem size can be lower with iso-power-efficiency than with iso-efficiency, thus yielding better scalability. Our approach is to use a regression modeling methodology, similar to the focused regression modeling described in [3], to fit observed execution data to an iso-power-efficiency function.

Users of large shared parallel computing systems are currently charged according to the number of processing elements they use for however long they use them. We expect that this will change to charging for the amount of energy used -- that is, for the number of processing elements times the average power per processor times the runtime. With future batch queueing systems, users will request a given power budget in addition to the number of processors and the estimated runtime, and they will be required to stay within that power budget. Our research on iso-power-efficiency will help users to scale their applications efficiently under this future scenario.

1.1 Background

As we move towards exascale computing, we are presented with many of challenges. The U.S. Department of Energy's goal is for the first exaflop machine to consume no more than 20MW. Exascale refers to both this and larger machines which may have higher power bounds. Because of this power bound, future supercomputers will most likely be limited by the amount of power that they can consume rather than the physical hardware available. From a monetary standpoint, a megawatt of power costs about one million dollars each year. Knowing this, we find it prudent to make better use of the energy these computers use.

Chapter 2: Problem Description

Iso-efficiency is a parallel performance metric that measures scalability. Given a particular algorithm/architecture combination, the iso-efficiency function can represent the characteristics in a single expression. Using this expression, we are able to predict the best combination of problem size and number of processors rather than brute forcing our way through every combination. Parallel overhead is given by $T_o = c \cdot T_c - T_1$ where c is the number of processing elements (e.g., cores), T_c is the parallel runtime, and T_1 is the runtime of the best known sequential implementation. The iso-efficiency function [1] tells us how we need to scale up the problem with increasing number of processing elements to maintain the same efficiency:

$$W = \frac{E}{1-E} T_o(W,c)$$

where E is the efficiency to be obtained, and $W = T_1$. For example, the iso-efficiency function for matrix-vector multiplication with 1-D block data decomposition is $W = K \cdot c \log c$, meaning we need to scale up the problem size not just proportionally to the increase in the number of processors, but with an additional factor of log c.

For iso-power-efficiency, we modify the parallel overhead function $T_o(W,c) = c \cdot T_c - W$ [1] to the following:

$$T_o(W,b) = \frac{b}{b_1} \cdot T_b - W$$

where b_1 is the smallest overall power budget for all the cores under which we can run the application, $W = T_1$ where T_1 is the runtime for the best sequential version, and T_b is the runtime for the best-performing configuration under overall power budget b. A way of interpreting this function is to say that we are now scaling our problem with respect to the normalized power budget $\frac{b}{b_1}$, rather than only with respect to the number of cores. Iso-power-efficiency essentially encompasses iso-efficiency. Working off the assumption that iso-efficiency runs all its cores at max power, we can represent b as a number of cores running at max power and b_1 as one core running at max power. By doing this, we can see that that $T_o(W,c)$ is just a special case of $T_o(W,b)$. Our goal is to show that by scaling with respect to power budget,

instead of just the number of cores, we can obtain lower overhead T_o and require less power to maintain a given efficiency E.

If we have a load-imbalanced problem and if we have can vary the power per node, or even per core, then we can achieve a lower overhead in terms of power than in terms of cores. To see this, consider a task dependency graph in which not all paths have the same length. In this case, we can lower the power setting on nodes off the critical path without increasing the parallel runtime. Then $T_o(W,b)=T_o(W,c,p_c,p_1)$, where p_c is the average power cap per core and p_1 is the max power cap per core using the minimum core configuration which is needed for the application to run, will be less than $T_o(W,c)$ for the same c.

For example, consider a task dependency graph that has two paths each of length l. Suppose we can scale up the problem by parallelizing each of these paths. Assume doing so for one path introduces no extra communication between the cores, but parallelizing the other path introduces communication proportional to the logarithm to the base 2 of the number of cores for that path – that is, $\lg \frac{c}{2}$. Assume the communication is the only parallel overhead. Then the critical path length and the parallel execution time go from l to $l + \lg \frac{c}{2} = l + \lg c - 1$. Suppose that the execution time is inversely proportional to the power supplied to a node and that we scale back (i.e., cap) the power to the nodes not on the critical path by a factor equal to $\frac{l}{l + \lg c - 1}$ so that the execution time for these nodes is also $l + \lg c - 1$. Now the average power cap per node is given by averaging the power of the two nodes: $p_c = \frac{p_1}{2}\left(1 + \frac{l}{l + \lg c - 1}\right) = \frac{p_1}{2}\left(\frac{2l + \lg c - 1}{l + \lg c - 1}\right)$, where p_1 is the original power per node. Without the power capping, the overhead function is given by $T_o(W,c) = \frac{c}{2} \cdot (l + \lg c - 1) - l \approx \frac{c}{2} \lg c$ if $\lg c >> l$. With power capping of the nodes that are off the critical path, the overhead function is given by $T_o(W,b) = \frac{1}{4}\left(\frac{2l + \lg c - 1}{l + \lg c - 1}\right) \cdot c \cdot (l + \lg c - 1) - l$. If $\lg c >> l$, then for large enough c, $T_o \to \frac{c}{4} \lg c$, and thus we can scale up the problem size at approximately half the rate as without power capping.

Whether or not we can obtain different asymptotic complexities for overhead functions for a realistic problem with and without power capping remains to be seen.

Chapter 3: Related Work

3.1 Overprovisioning

The idea of overprovisioning with respect to power is the basis of the research described in [2]. Overprovisioning means that more cores exist than can simultaneously run at the highest CPU clock frequency, and equivalently, at the highest power setting. The goal is to find the best configuration for an application so that the maximum performance is achieved without exceeding a given power bound. An application's scalability characteristics determine whether to use few nodes at higher power or more nodes at lower power. The investigation of overprovisioning in [2] assumes uniform power allocation per node and that the applications are perfectly load balanced. The authors use exhaustive search to find optimal configurations for a number of applications with fixed problem sizes, including SP-MZ, LU-MZ, and SPhot, under various system power bounds. For future work, they plan to develop a model to predict the optimal configuration, including allocation of non-uniform inter-node power based on critical path and load imbalance of the application (as for the example we gave above). Our work assumes such a model and we build our iso-power-efficiency model on top of it.

3.2 Iso-energy-efficiency

The concept of iso-energy-efficiency is described in [4, 5]. The energy overhead for parallel execution is given as $E_o = E_p - E_1$, where E_1 is the energy consumption for the sequential execution and E_p is the total energy consumption for the parallel execution. Iso-energy-efficiency is defined as $EE = \frac{E_1}{E_p} = \frac{1}{1+\frac{E_o}{E_1}}$. This is analogous to the time-based efficiency metric $E = \frac{1}{1+\frac{T_o}{T_1}}$ rather than to iso-efficiency. The energy efficiency factor $EEF = \frac{E_o}{E_1}$ is modeled by sets of machine-dependent and application-dependent parameters that can be measured by small-scale executions so that the EEF can be estimated for larger-scale executions. The goal is to determine what parameter settings, such as frequency, problem size, and number of processors, will minimize the energy overhead and thus maximize the energy efficiency. In contrast, our work assumes that some method exists to find the configuration under a given

power budget that yield the minimum runtime, and determines how to scale up the problem to maintain the same power efficiency with increasing power budget.

Chapter 4: Methodology

In order to determine how to scale a problem to obtain iso-power-efficiency, we need a model that predicts the overhead function $T_o(W,c,p_c,p_1)$ for the application, where W is the workload, c is the number of cores, p_c is the power cap per core, and p_1 is full power (i.e., uncapped power) for a core. Our methodology is to sample the space of workload/power budget pairs. For a given workload and power budget, we do an exhaustive search to determine the best configuration and use the resulting parallel runtime to compute the overhead. We use each such sampled result (inputs: W, b; output: T_o) as a training input to our regression model for T_o.

We hypothesize that even with uniform inter-node power, we may still be able to achieve better scaling with a power budget. Our reasoning is that as we scale up a problem, its strong scalability characteristics change. For example, with more nodes, the communication proportion of an application may increase. As the communication fraction increases, we may achieve faster runtime by running cores at lower power. We used some of the communication-intensive benchmarks from [2], modified to have different problem sizes, to test this hypothesis.

Chapter 5: Experiments

All of our experiments were run on the Cab cluster at Lawrence Livermore National Laboratory. The CAB cluster consists of 1296 nodes. 256 of these nodes were available to us. Each node has 2 sockets, each of which holds an Intel Xeon E5-2670 (2.6GHz, 8 cores) from the Sandy Bridge architecture family. For more information Refer to **Cluster** in the appendix.

5.1 RAPL/LIBMSR

The Sandy Bridge family allows us to use Intel's Running Average Power Limit (RAPL). RAPL provides mechanisms to enforce power consumption limits so as to stay under a given power bound and retrieve power measurements. This is done by controlling several model-specific registers (MSRs).

We can gather information that our PKG supports a lowest power domain, which is 51W, the *thermal specification power*, which is rated at 115W with a maximum power of 180W, and the largest time window, which is 0.0459 seconds. The hardware guarantees that the power set will, on average over the time window, not go over this power bound. The Sandy Bridge server processor we are using can perform power clamping in units of .125 watts, .0000152 joules and .000977 seconds [6]. All experiments for the iso-power-efficiency were done with the minimum .000977 second time window. Power capping is done by adjusting the processor's P- and T- states and using other undocumented techniques. The PKG domain offers two separate clamping windows, each of which has its own maximum average-power bound and time window. Both of these windows may be set and are abided by simultaneously. Ideally we would be able to set a high power bound with a small time window and a low power bound with a large time window to allow for greater application flexibility in instantaneous power used.

We used the MSR-safe kernel module rather than the default MSR kernel module. MSR-safe allows us the same capabilities as the default, but some of the more critical risks with full access to MSRs are avoided [7].

To manipulate the power bounds for our experiments, we used the LIBMSR library. The LIBMSR library provides access to Intel RAPL MSRs, allowing the user to easily adjust the power bounds and time

intervals desired. The version of LIBMSR we used allowed us to simply use environment variables to set power bounds and time intervals. LIBMSR can also be used to retrieve power consumption at a given time. LIBMSR can be found at https://github.com/scalability-llnl/libmsr [7].

Both MSR-safe and LIBMSR were developed at Lawrence Livermore National Laboratory by Dr. Barry Rountree's group.

Other power such as consumed by DRAM, NIC and the motherboard are beyond the immediate scope of this work. Future work will investigate these using the BLC cluster at LLNL [6] [8] [2].

5.2 Experiment

We began our experiment by generating many test runs for each of our benchmarks. Each run consists of a combination of processors, cap on the power per socket, and problem size. The number of processors is in the range of 16 to at most 1024 in multiples of 16. The cap on the power per socket is selected from a set of discrete values 51, 65, 80, 95, and 115 watts. The problem size is dependent on the benchmark. Every Class for every benchmark contains a different problem size. Every combination of processors, cap on the power per socket, and problem size was run multiple times. For EP and SP-MZ we attempted to run each of these combinations 10 times each and LU 5 times each. This was done to reduce the amount of noise and to provide a better runtime sample over all the nodes available on the system. The reason that LU has less combinations is that we ran experiments for LU late in the experiment. Some of these combinations were never ran. This is because the cluster I was using was being used by many people and I currently only have student priority in the job queue. So many of the combinations that have a long runtime or that require a lot of processors were not ran the 10 or 5 times each. Refer to **Parsed benchmark runs** in the appendix.

With these runs, we created a model for each benchmark that estimates the runtime given the number of processors c, average power per socket P_s and problem size S. For simplicity, problem size S was used in place of workload W. This is fine because W is a function of S with all other parameters of W kept constant. These models are second order linear regression models with logarithmic transformation using log-log form. Each of these models is given in more detail for each benchmark below.

From here we attempt to find the best combination of processors and average power per socket that minimizes the runtime for a given power budget b such that $b = \frac{c}{8} * p_s$. This is done for many power budgets ranging from some minimum given by the benchmark up to the maximum watts sampled for each benchmark.

After finding these best configurations, we used these data to create two more models that just use a power budget and a problem size to estimate run time T_b and overhead time T_o. T_b and T_o are used to estimate different efficiencies E.

R was used to create all statistical modeling [9]. R is a programming language that specializes in statistical computing and graphics. We used R for its ability to create linear models and produce informative plots.

5.3 Benchmarks

The NAS Parallel Benchmarks [10] have input sets for different problem classes. Some of the classes below were created and/or the values used from pre-existing classes that may have been altered from the default provided in the NPB benchmark suite.

5.3.1 EP

EP stands for Embarrassingly Parallel. EP is a pseudo number generator that generates independent Gaussian random variates using the Marsaglia polar method.

We had a total of 1477 test runs for EP.

Class D: m = 36

Class Q: m = 37

The computational complexity of EP is 2^m.

5.3.2 SP-MZ

SP stands for Scalar Pentadiagonal and MZ stands for multi-zone. SP-MZ solves a synthetic system of nonlinear PDEs using Scalar Pentadiagonal.

We had a total of 3144 tests runs for SP-MZ.

Class D: gx_size = 1632; gy_size=1216; gz_size=34; x_zones = y_zones = 32; dt = "0.00030d0"; niter = 500;

Class H: gx_size = 2448; gy_size=1824; gz_size=51; x_zones = y_zones = 32; dt = "0.00030d0"; niter = 500; //Class H's gx, gy & gz is class D+.5*D

comp

5.3.3 LU

LU stands for Lower-Upper symmetric Gauss-Seidel solver. LU solves a synthetic system of nonlinear PDEs using Lower-Upper symmetric Gauss-Seidel.

We had a total of 523 tests runs for BT-MZ.

Class D: problem_size = 408; dt_default = "0.5d0"; itmax = 300;

Class H: problem_size = 714; dt_default = "0.5d0"; itmax = 300;

Class E: problem_size = 1020; dt_default = "0.5d0"; itmax = 300;

The computational complexity of LU is k*N^2 where k is the number of iterations and N is the problem size.

5.3.4 Benchmark Models

The following is the second degree regression model used to estimate T_o.

Regression model:

$T_0=2^{(B0 + B_1*(\log_2(Ps)) + B_2*(\log_2(c)) + B_3*(\log_2(S)) + B_4*(\log_2(Ps)^2) + B_5*(\log_2(c)^2) + B_6*(\log_2(S)^2) + B7*(\log_2(Ps)*\log_2(c)) + B_8*(\log_2(Ps)*\log_2(S)) + B_9*(\log_2(c)*\log_2(S)))}$

Table 5.3.4.1: Values of Betas for each Benchmark

	EP	**SP-MZ**	**LU**
B_0	-117.1593	-5.945419	26.40012
B_1	-0.9618277	-2.676733	-3.416239
B_2	-0.8215667	-0.9458221	-0.6984303
B_3	26.03894	1.090646	-2.991218
B_4	0.06804725	0.169925	0.05842392

B_5	0.0002239609	0.02139848	0.06581095
B_6	0	0	0.2564773
B_7	-0.0119897	-0.02056776	-0.03969327
B_8	-0.0945159	0.009142483	0.2752029
B_9	-0.01992771	-0.002133025	-0.09628437

Table 5.3.4.2: Statistical summaries of each model

	EP	**SP-MZ**	**LU**
Residual Standard Error	0.01892 on 1468 DF	0.1878 on 3135 DF	0.1426 on 513 DF
Multiple R-squared	0.9999	0.9819	0.9904
Adjusted R-squared	0.9999	0.9819	0.9902
Predicted R-squared	0.9999006	0.9818247	0.9898612
F-Statistic	1.866e+06 on 8 and 1468 DF	2.131e+04 on 8 and 3135 DF	5862 on 9 and 513 DF
p-value	< 2.2e-16	< 2.2e-16	< 2.2e-16

We can see that given the high values for each of the three R-squared values, we can safely assume that not only are the sample points fitted well but that future points will be predicted accurately.

Chapter 6: Results

6.1 Overhead

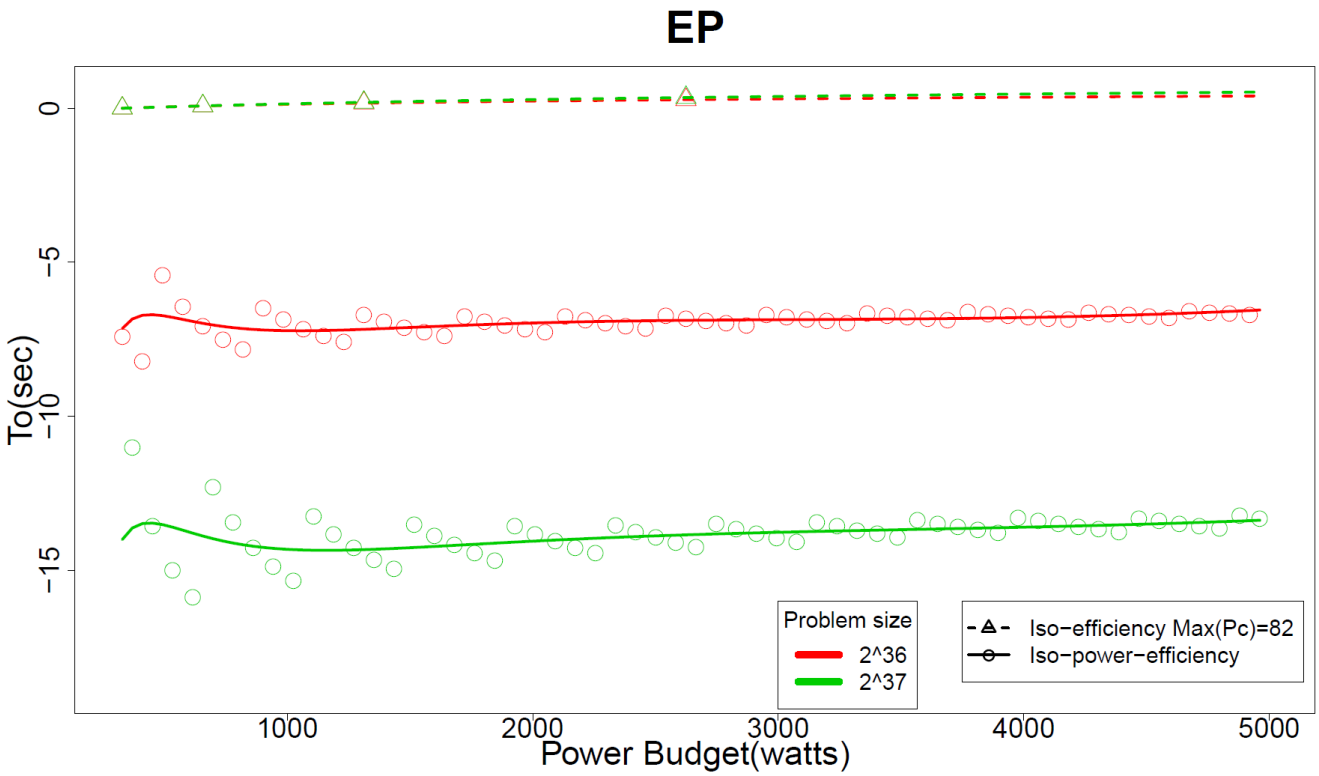

Figure 6.1.1: Overhead (To) given a Power Budget for EP with problem size 2^36(red) and 2^37(green).

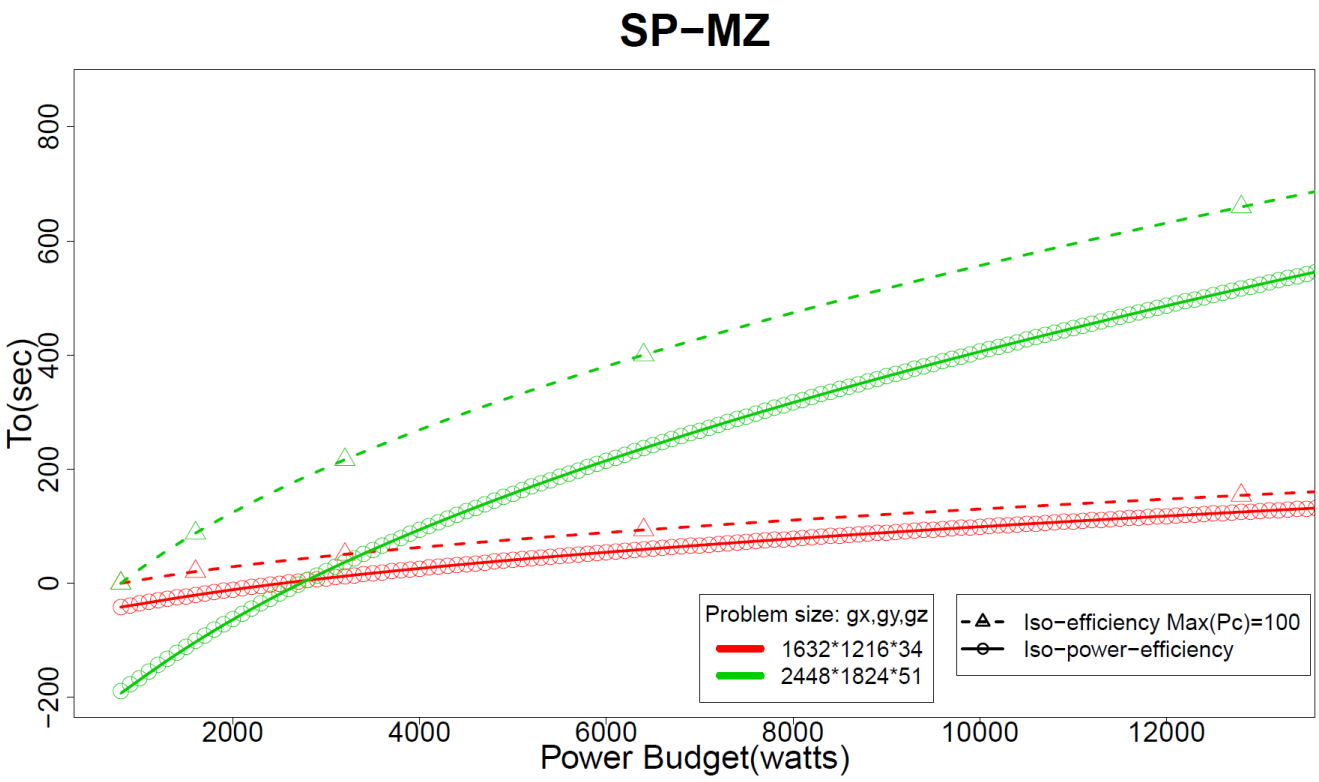

Figure 6.1.2: Overhead (To) given a Power Budget for SP-MZ with problem size 1632*1216*34(red) and 2448*1824*51(green).

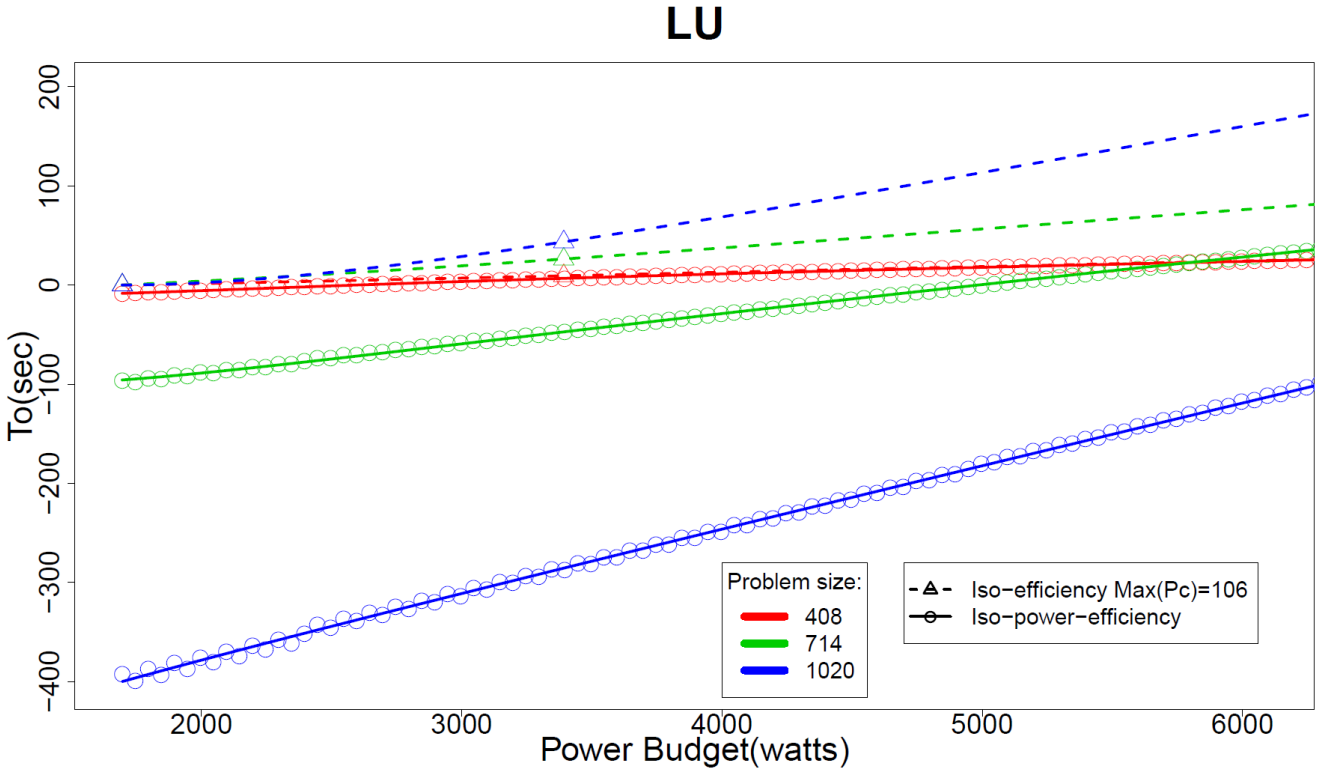

Figure 6.1.3: Overhead (To) given a Power Budget for LU with problem size 408(red), 714(green), and 1020(blue).

Figures 6.1.1, 6.1.2, and 6.1.3 shows the amount of overhead T_o using iso-efficiency and iso-power-efficiency. We can see that at all times the value of T_o is lower for iso-power-efficiency but as the power budget grows we see that the values for T_o begins to converge to the same value. By increasing the problem size the overhead gap increases. This allows for an increase in the longevity of the iso-power-efficiency advantage.

The sharp spikes in Figure 7.1 are attributed to the average power per socket increasing to stay at the power budget but the number of cores staying the same. The spike goes up when the cores stay the same and down when the number of cores increases but the average power per socket decreases to stay at the power bound. The average power per socket tends to our minimum allowed power per socket 51 watts. When the best configuration would be a combination that calls for less than 51 watts, we must choose a lesser combination which generally means fewer cores at higher power.

6.2 Efficiency

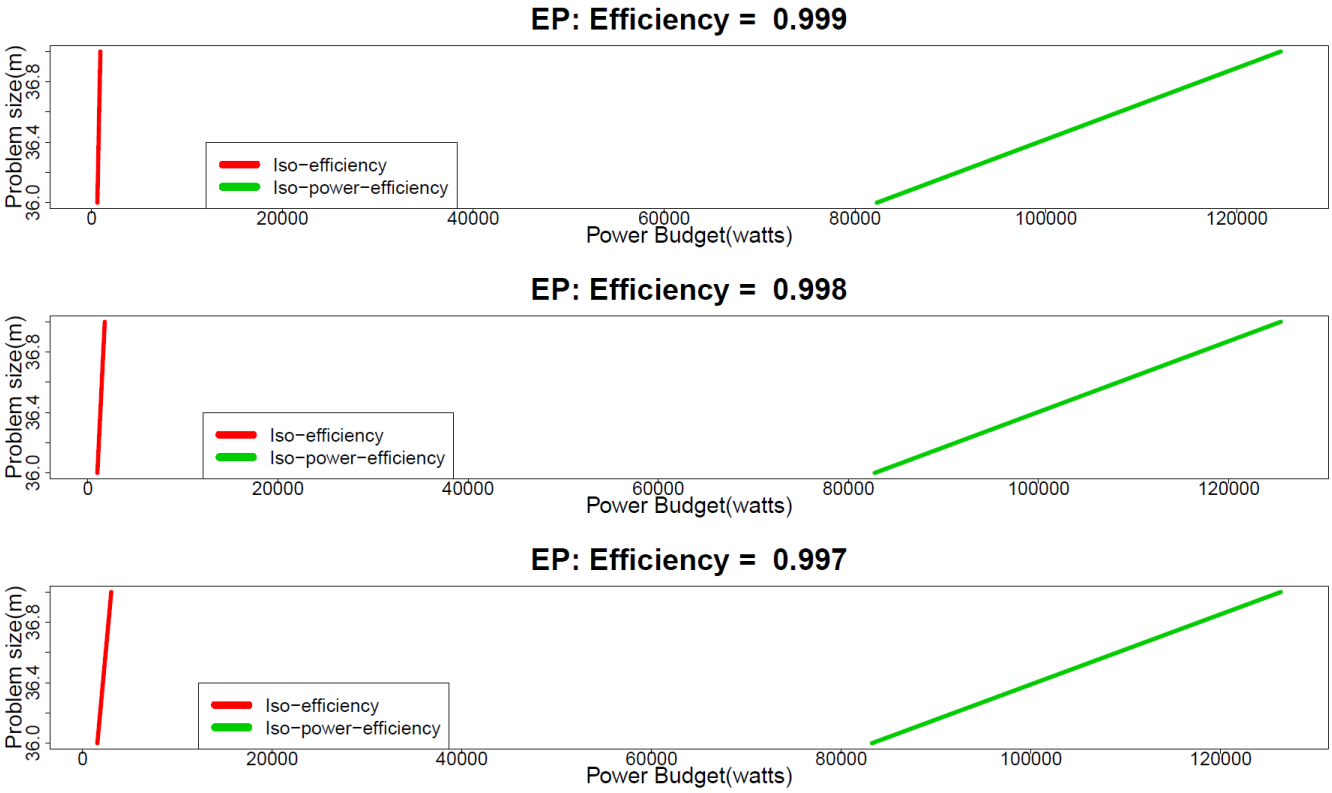

Figure 6.2.1: Problem size to keep efficiency (.999, .998, and .997) given a Power budget for EP.

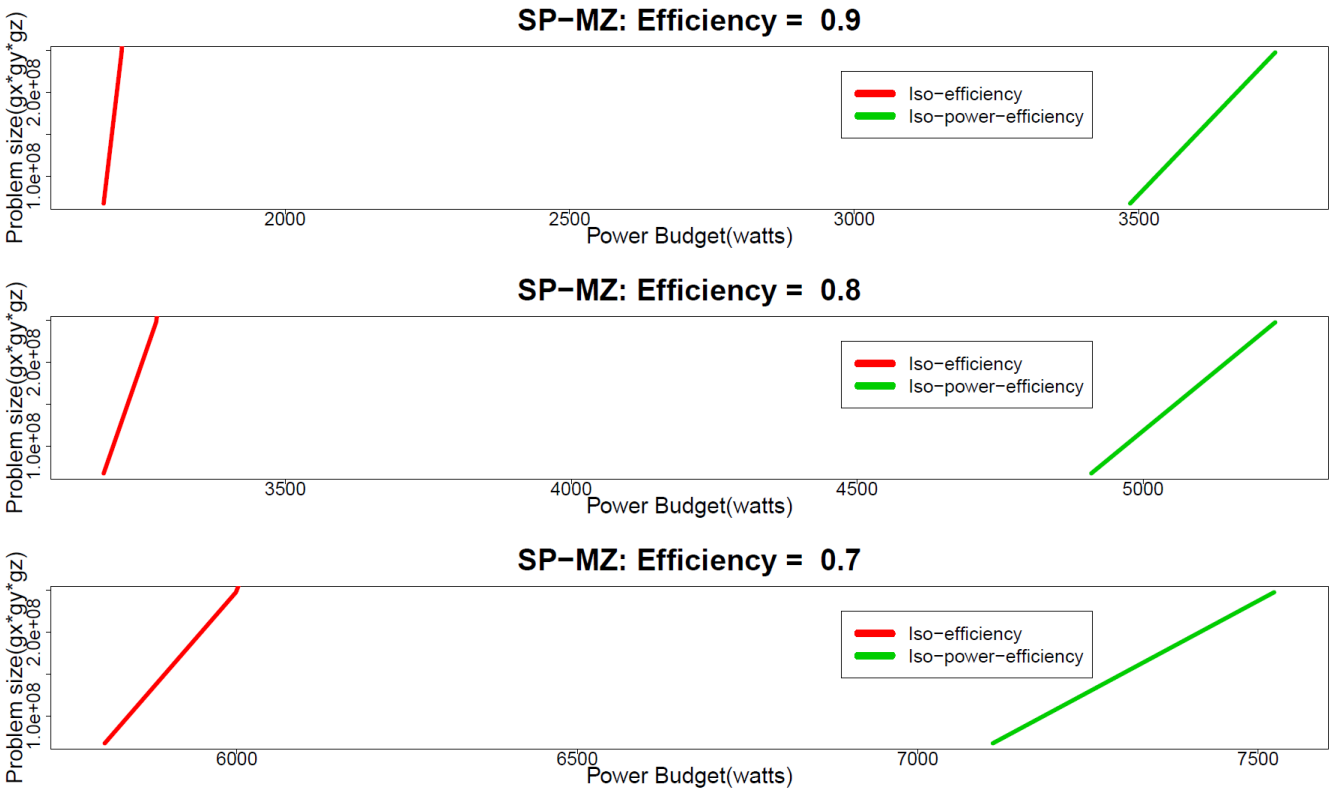

Figure 6.2.2: Problem size to keep efficiency (.9, .8, and .7) given a Power budget for SP-MZ.

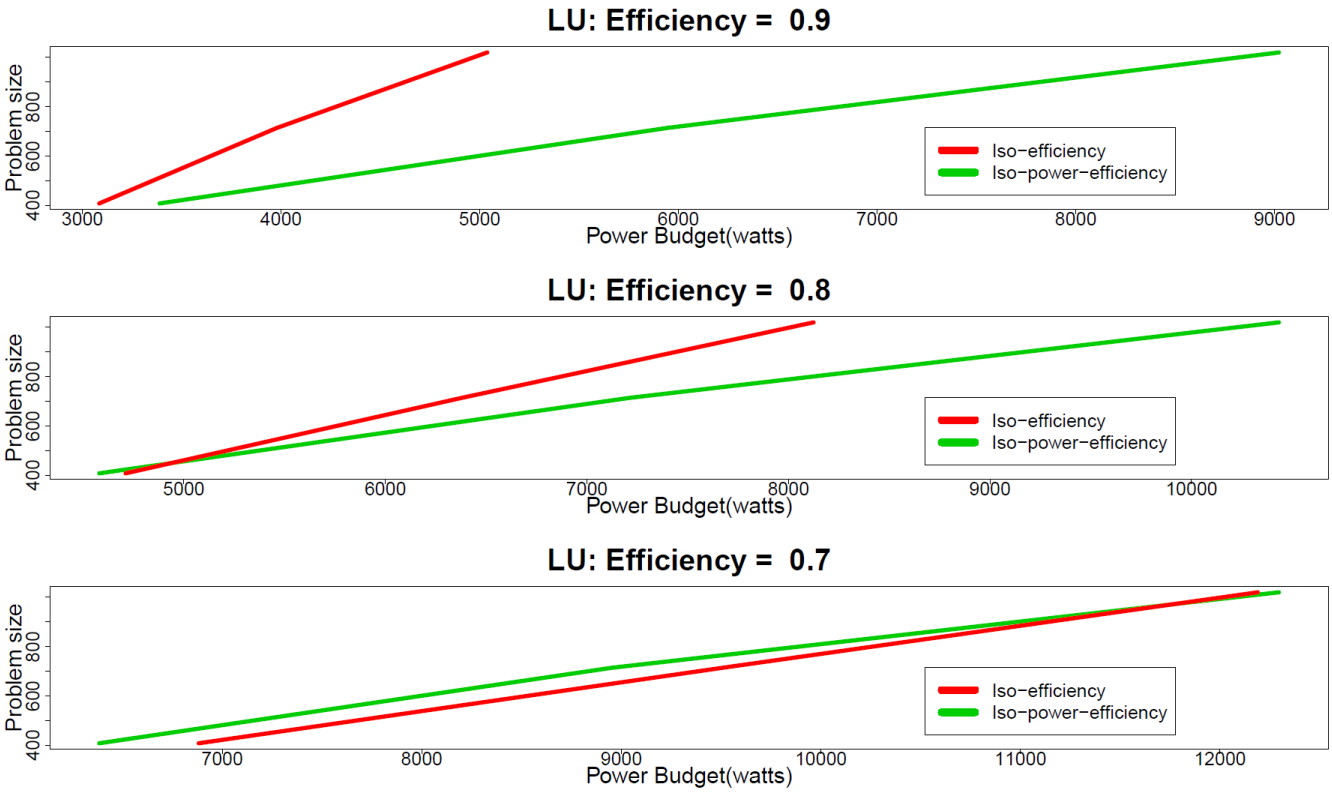

Figure 6.2.3: Problem size to keep efficiency (.9, .8, and .7) given a Power budget for LU.

In Figures 6.2.1, 6.2.2 and 6.2.3, we can see the necessary power budget needed to maintain a given efficiency at some problem size. Each of these tables, with the exception of Figure 6.2.1, shows efficiencies .7, .8, and .9. We are forced to use very high values of efficiency, .997, .998, and .999 for EP in Figure 7.2.1 because of how EP scales. For each of these figures we used problem size rather than workload for convenience. Using workload rather than problem size would just alter the values of the y-axis but would not change what we are ultimately trying to show. We can see that for each of these benchmarks, as the power budget increases, the required problem size grows more slowly using our iso-power-efficiency function compared to iso-efficiency.

Table 6.2.1: EP efficiency as a numerical representation of Figure 6.2.1

EP		Iso-power-efficiency		Iso-efficiency		
Efficiency	Problem size	power bound	slope	power bound	slope	slope ratio
0.999	36	82283.02	1623795.418	607.4905	219687769.7	0.007391378
	37	124603.3		920.2957		
0.998	36	82763.67	1609147.029	1006.126	88522830.66	0.018177763
	37	125469.2		1782.417		
0.997	36	83244.32	1594760.567	1559.258	46973672.63	0.033950093
	37	126335.1		3022.194		

Table 6.2.2: SP-MZ efficiency as a numerical representation of Figure 6.2.2

SP-MZ		Iso-power-efficiency		Iso-efficiency		
Efficiency	Problem size	power bound	slope	power bound	slope	slope ratio
0.9	67473408	3484.726	706626.7972	1681.328	5717298.084	0.123594535
	247402496	3739.357		1712.799		
0.8	67473408	4909.515	559848.3084	3182.411	1965299.75	0.284866626
	247402496	5230.904		3273.964		
0.7	67473408	7110.596	435725.8979	5805.969	934007.5788	0.466512165
	247402496	7523.537		5998.611		

Table 6.2.3: LU efficiency as a numerical representation of Figure 6.2.3

LU		Iso-power-efficiency		Iso-efficiency		
Efficiency	problem size	power bound	slope	power bound	slope	slope ratio
0.9	67917312	3487.587	162204.3934	3246.307	1326747.768	0.122257144
	363994344	5312.92	417284.0654	3469.467	3058920.607	0.136415461

	1061208000	6983.757		3697.395		
0.8	67917312	4758.835	181131.4048	4899.979	503175.5235	0.359976581
	363994344	6393.433	420403.5578	5488.396	2042129.537	0.205865275
	1061208000	8051.872		5829.811		
0.7	67917312	6663.322	246984.2247	7173.538	343428.2294	0.719172751
	363994344	7862.091	441741.0898	8035.66	1216442.335	0.363141825
	1061208000	9440.422		8608.818		

In Tables 6.2.1, 6.2.2, and 6.2.3, we can see a numerical representation of the Figures 6.2.1, 6.2.2 and 6.2.3 along with slope, which is the amount needed to increase the problem size for each given increase in power. With these tables it is obvious to see that using iso-power-efficiency we require less of an increase in input for a unit increase in watts. In each of these tables we can see the ratio of the slope of iso-efficiency and iso-power-efficiency. For a high efficiency this ratio is very low. As the efficiency decreases this ratio tends to 1. This is because as the power budget increases, the optimal combination for iso-power-efficiency tends to the same combination as iso–efficiency. Generally this is because the amount of communication overhead outweighs the advantage of using more cores. Generally this can usually be alleviated by increasing the problem size.

6.3 Runtime vs Joules

Figures 6.3.1, 6.3.2 and 6.3.3 show the amount of Joules used for a particular run with a given runtime. These runtime are obtained by finding the iso-efficiency and iso-power-efficiency runtimes for different power budgets. It can be seen that for any runtime, iso-power-efficiency always uses less Joules than iso-efficiency.

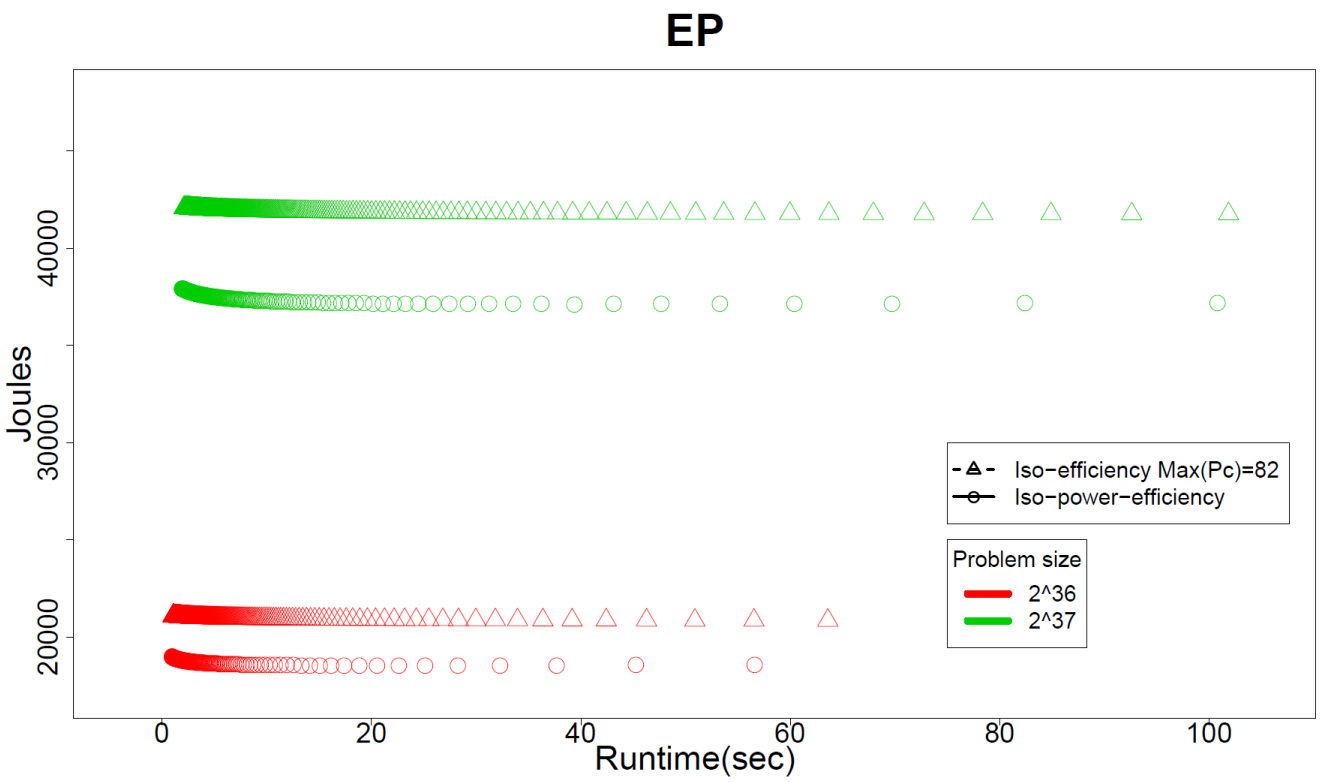

Figure 6.3.1: Comparison of Joules used for a given runtime for iso-power-efficiency and iso-efficiency for EP using problem sizes 2^36(red) and 2^37(green).

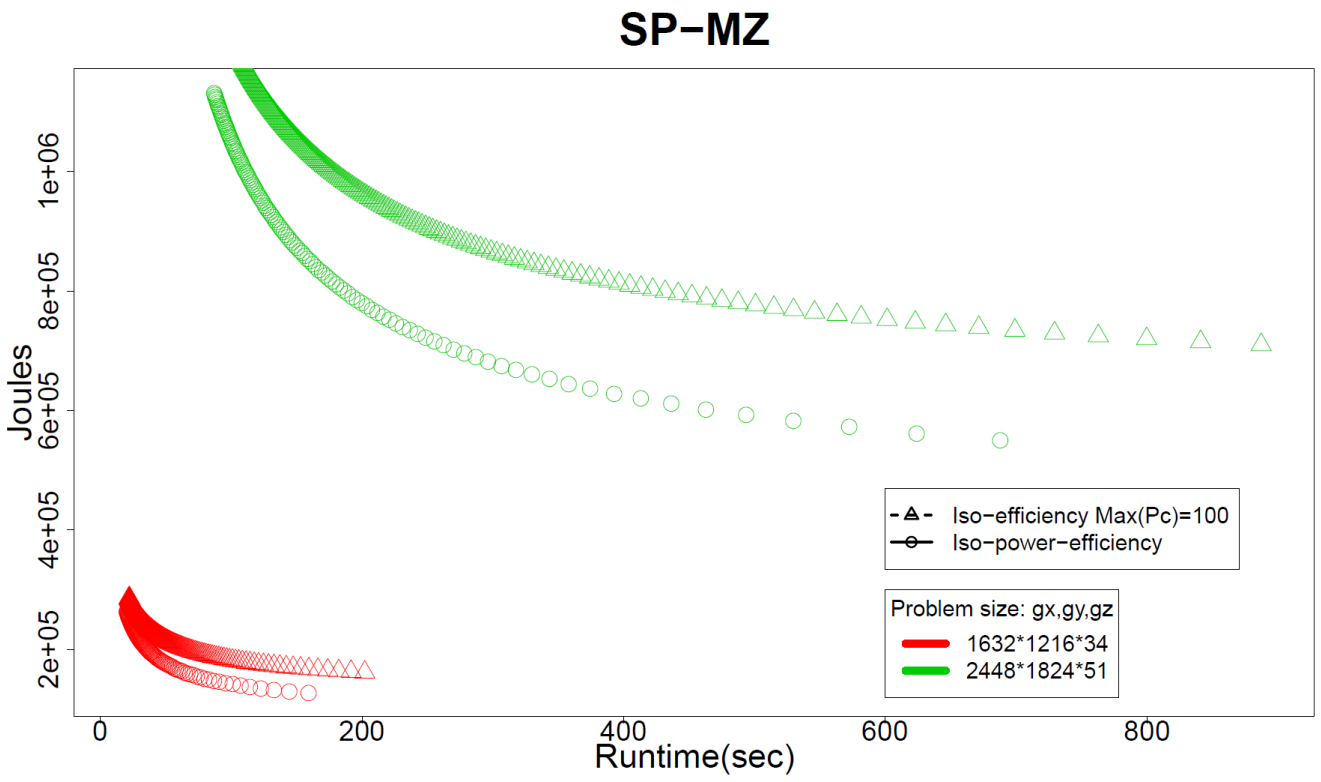

Figure 6.3.2: Comparison of Joules used for a given runtime for iso-power-efficiency and iso-efficiency for SP-MZ using problem sizes 1632*1216*34(red) and 2448*1824*51(green).

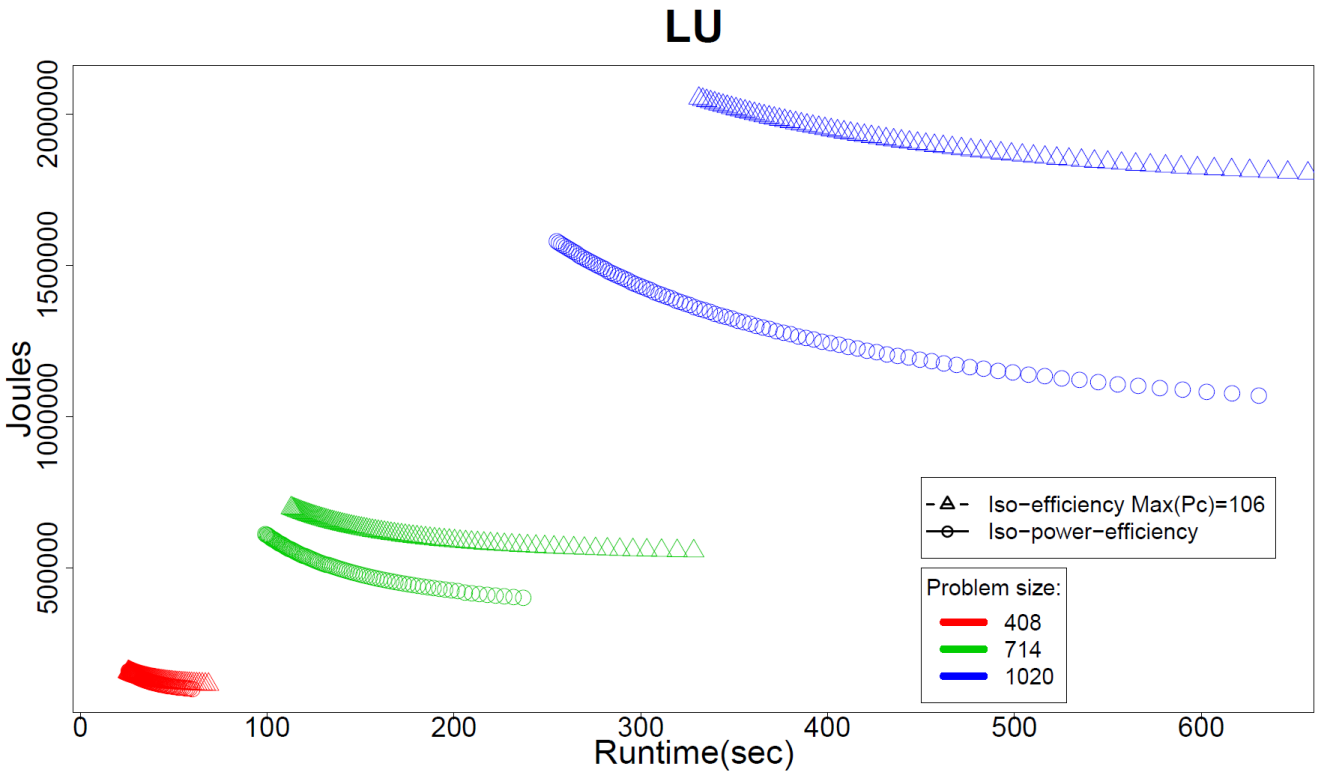

Figure 6.3.3: Comparison of Joules used for a given runtime for iso-power-efficiency and iso-efficiency for LU using problem sizes 408(red), 714(green), and 1020(blue).

Chapter 7: Conclusions

We have shown that the use of overprovisioning can result in lower parallel overhead, thus enabling better iso-efficiency scaling. As we move towards exascale computing, we will find ourselves more attentive to our energy consumption as we run HPC applications. With the true cost of computing becoming energy consumption, a function that allows applications to be scaled with better power efficiency as its priority will become invaluable. This is good to keep in mind as we predict that future queuing systems will require jobs to submit a power budget. In general, we have shown such a function that, with the benchmarks provided, showed better efficiency scaling than the iso-efficiency function which is based on using maximum power per core.

Chapter 8: Future/Proposed work

8.1 Shock physics benchmarks

We have currently only run our iso-power-efficiency test on the NPB. We plan on expanding on the benchmarks we test. We will be studying iso-power-efficiency scaling of shock physics benchmarks such as LULESH and CoMD.

8.2 Predicting critical paths

We are working on a way to predict critical path candidates in a given application. When we can do this, we should be able to allocate power accordingly throughout the nodes that will be running this application. The idea is that nodes that are on the critical path will be given more power while nodes off the critical path will be given less. We expect that by doing this the results will have better iso-power-efficiency scaling.

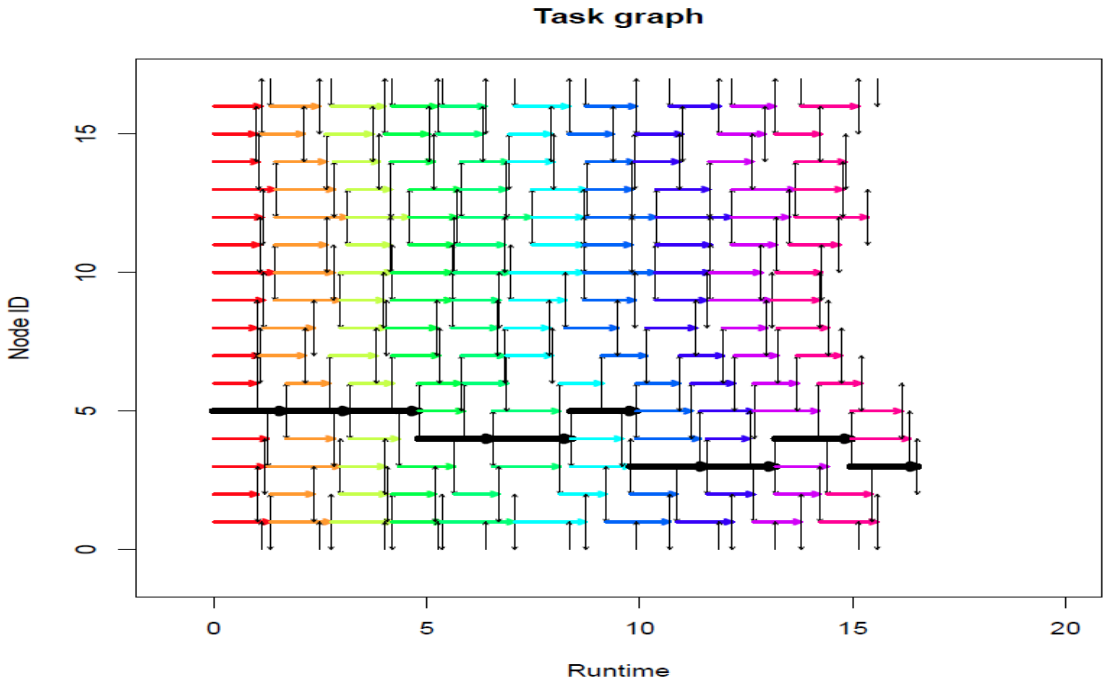

Figure 8.2.1: Task graph with critical path

In Figure 8.2.1 we can see a simulation of some application in which every node is given the same amount of work, but at each iteration some “noise” is produced that increases the runtime of each of these

iterations. The node may only continue its next iteration when its nearest neighbors communicate some necessary information. We are considering a torus connection, so node 1 sends information to node n-1 and vice versa. As the number of iterations increases, large noise bursts produced by a node propagate though its neighbors. This ultimately affects the final termination time of the application. The blackened arrows illustrate the critical path at a given iteration.

8.3 Two power bounds/time intervals using RAPL

We are working on learning more about what we can do with RAPL. Currently we are only setting one power bound with a time window. RAPL has the capability of having two power bounds set with different time windows. The idea is that we could set one bound with a high power bound and small time interval and the second power bound lower with a larger time interval. We are hoping that by doing this the larger power bound could be used to take advantage of larger computation spikes and the lower power bound to keep the overall power consumption low. We have begun running tests to check if there is an advantage to having two power bounds.

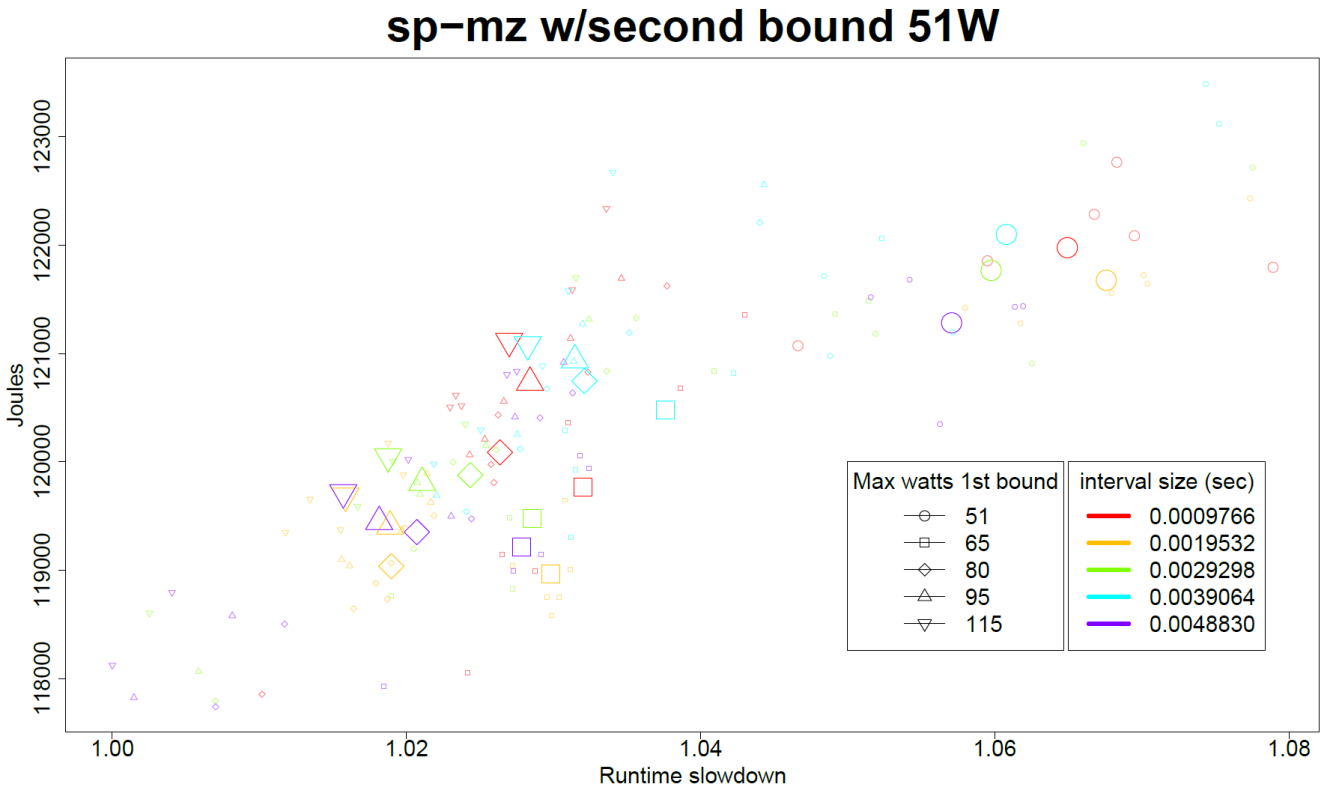

Figure 8.3.1: SP-MZ using two power bounds and time intervals. The first with interval size .0009766 secs and the second with power bound 51W.

In Figure 8.3.1 we can see some initial results from setting two power bounds and time intervals. This figure shows runtimes vs Joules. The second power bound was kept at a constant 51watts and the time interval for this second bound varies from .0009766 seconds to .0048830 seconds in increments of .0009766 seconds. The first power bound varies its power bound from 51, 65, 80, 95 and 115 watts but keeps a constant time interval of .0009766 seconds. We can see that even though we keep an overall power bounds of 51 watts, the runtime varies up to 8%.

8.4 Optimal configuration

Although we were able to estimate T given a power bound and problem size for a particular benchmark, we have yet to come up with a way of finding the optimal configuration that would result in this T. Our goal is to model the problem of finding the optimal node/power cap configuration (c, cpn, p_c) for a given power budget B for the job, where c is the number of cores, cpn is the number of cores per node, and p_c is the power cap per core, as an optimization problem and solve the optimization problem. By optimal configuration, we mean the configuration that gives the fastest runtime while staying under the power budget. Overprovisioning means using a larger number of nodes with a lower power cap. Previous work has shown that overprovisioning can lead to faster runtimes. The previous work found the optimal configuration by means of exhaustive search.

Initially we are attempting to solve the simpler optimization problem of finding the optimal configuration $(n, \Delta\pi)$ where n is the number of sockets and $\Delta\pi$ is the power cap per socket. We use the formalism developed in [Choi 2014]. Notation is as follows:

Machine Parameters		**Application Parameters**	
τ_{flop}	time per flop	W	workload
τ_{mem}	time per byte	Q	bytes moved
ε_{flop}	energy per flop	$I = W/Q$	operational intensity
ε_{mem}	energy per byte		
$\pi_{flop} = \varepsilon_{flop}/\tau_{flop}$	power per flop		

$\pi_{mem} = \varepsilon_{mem} / \tau_{mem}$	power per byte
π_1	idle power
$\Delta\pi$	usable power
$\beta_\tau = \tau_{mem} / \tau_{flop}$	machine time balance
$\beta_\varepsilon = \varepsilon_{mem} / \varepsilon_{flop}$	machine energy balance

Here W is the number of floating point operations for a scientific application, and Q is the number of bytes moved from slow (i.e., main) to fast (i.e., cache) memory.

We determine the machine parameters using regression modeling of data collected from microbenchmarks. We use the microbenchmarks from [11]. The intensity microbenchmark allows us to vary the operational intensity I at will. We can estimate the parameters ε_{flop}, ε_{mem}, and π_1 by measuring the energy consumption E and runtime T for different values of (W, Q) and fitting the data to the following equation for E: $E = W\varepsilon_{flop} + Q\varepsilon_{mem} + \pi_1 T$. To estimate τ_{flop}, we can configure our intensity microbenchmark to be compute bound and fit the data to $T = W\tau_{flop}$. To estimate τ_{mem}, we can configure our intensity microbenchmark to be memory bound and fit the data to $T = Q\tau_{mem}$.

We are investigating how to determine the application parameters by using hardware counters.

Given these parameters, the optimization problem is formulated as follows:

minimize: $T = T(W, I) = W \cdot \tau_{flop} \cdot \max\left\{1, \frac{\beta_\tau}{I}, \frac{\pi_{flop}}{\Delta\pi}\left(1 + \frac{\beta_\varepsilon}{I}\right)\right\}$

subject to: $n\overline{P} \le B, \quad n \ge 1, \quad \Delta\pi_{\min} \le \Delta\pi \le \Delta\pi_{\max}$

where B = power budget, n = number of sockets, $\overline{P}$ =power consumption per socket. We have

$$\overline{P} = \overline{P}(I) = \pi_1 + \left\{ \begin{array}{l} \pi_{flop} + \pi_{mem} \dfrac{\beta_\tau}{I} \; if \; I \geq \beta_\tau^+ \\ \pi_{flop} \dfrac{I}{\beta_\tau} + \pi_{mem} \; if \; I \leq \beta_\tau^- \\ \Delta\pi \quad otherwise \end{array} \right\}$$

where $\beta_\tau^+ = \beta_\tau \max\left(1, \dfrac{\pi_{mem}}{\Delta\pi - \pi_{flop}}\right)$

$\beta_\tau^- = \beta_\tau \min\left(1, \dfrac{\Delta\pi - \pi_{mem}}{\pi_{flop}}\right)$

The above model is a simplification of the real problem, as all models are. None of the parameters are actually constants. Part of our research is to determine if expected values can be used as constants without making the model produce incorrect results. That is, how sensitive is the model to small changes in each of the parameters? One application parameter that is definitely not a constant is the operational intensity I, since and Q $I = W/Q$ depends on the number of sockets n (and also possibly on the number of cores being used per socket). So for the minimization problem, we really need $T = T(W,n) = W \cdot \tau_{flop} \cdot \max\left\{1, \dfrac{\beta_\tau}{I(n)}, \dfrac{\pi_{flop}}{\Delta\pi}\left(1 + \dfrac{\beta_\varepsilon}{I(n)}\right)\right\}$

Then we need the function $I(n)$ which we can try to get by sampling I for different values of n and fitting the data with a regression model.

Another simplification of the above model is that the possible power caps are not continuous between $\Delta\pi_{\min}$ and $\Delta\pi_{\max}$, but rather take on discrete values. We are also ignoring communication costs between nodes.

References

[1] Ananth Y. Grama and Anshul Gupta and Vipin Kumar. Isoefficiency: measuring the scalability of parallel algorithms and architectures. IEEE Parallel and Distributed Technology 1(3):12-21, 1993.

[2] Tapasya Patki, David K. Lowenthal, Barry Rountree, Martin Schulz, Bronis R. de Supinski: Exploring hardware overprovisioning in power-constrained, high performance computing. *ICS 2013*: 173-182.

[3] Brad Barnes, Jennifer Garzen, David K. Lowenthal, Jaxk Reeves, Bronis R. de Supinski, Martin Schulz, and Barry Rountree. Using focused regression for accurate time-constrained scaling of scientific applications. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS),* 2010.

[4] Shuaiwen Song, Chun-Yi Su, Rong Ge, Abhinav Vishnu, Kirk W. Cameron: Iso-Energy-Efficiency: An Approach to Power-Constrained Parallel Computation. *IPDPS 2011*: 128-139.

[5] Shuaiwen Song, Matthew Grove, Kirk W. Cameron: An ISO-Energy-Efficient Approach to Scalable System Power-Performance Optimization. *CLUSTER 2011*: 262-271.

[6] Intel. Intel-64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2, June 2013

[7] Kathleen Shoga, Barry Rountree, Martin Schulz, Jeff Shafer. Whitelisting MSRs with msr-safe. Presented at CS14, 2014.

[8] Barry Rountree, Dong H. Ahn, Bronis R. de Supinski, David K. Lowenthal, Martin Schulz: Beyond DVFS: A First Look at Performance under a Hardware-Enforced Power Bound. *IPDPS Workshops* 2012: 947-953.

[9] R version 3.1.2 http://www.r-project.org

[10] NASA Advanced Supercomputing Division, NAS Parallel Benchmarks Suite v3.3.1

[11] Jee Choi, Marat Dukhan, Xing Liu, Richard W. Vuduc: Algorithmic Time, Energy, and Power on Candidate HPC Compute Building Blocks. IPDPS 2014: 447-457.

Appendix

BENCHMARKS:

NAS Parallel Benchmarks (NPB):
EP – Embarrassingly Parallel
SP-MZ – Multi-zone Scalar Penta-diagonal solver
LU- Lower-Upper symmetric Gauss-Seidel

CLUSTER:

Information on the Cab cluster used for our experiments can be found at the following link.
https://computing.llnl.gov/?set=resources&page=OCF_resources

CODES:

If any of my readers would like any of the following code, feel free to email me with a request.

Parsed benchmark runs:

all_Data_ep.txt
all_Data_sp-mz.txt
all_Data_lu.txt

Script that launches jobs:

run.sh
run_MZ.sh
run_lu.sh

Parsing script:

Bring_it_all_together.sh
Bring_it_all_together_sp_mz.sh
Bring_it_all_together_lu.sh

Statistical modeling codes:

Reg_EP_full.r
Plotting_EP.r
Plotting_EP_RTvsJ.r
Reg_sp-mz_full.r
Plotting_sp-mz.r
Plotting_sp-mz_RTvsJ.r
Reg_lu_full.r
Plotting_lu.r
plotting_lu_RTvsJ.r

Vita

Rogelio Long was born in El Paso, TX. Over his life he has lived in over half a dozen cities. He started his college career at Del Mar community college where he received an Associates in Business administration. He then transferred to UTEP where he received his Bachelors in Mathematics and later a Masters in Computational Science. Rogelio conducted research as an intern at Lawrence Livermore National Laboratory summer of 2014 under Dr. Barry Rountree and as a research assistant under Dr. Shirley Moore since Fall 2012 to current (summer 2015). I summer of 2015 Rogelio got his first paper "Iso-Power-Efficiency: An approach to scaling application codes with a power budget" published at the HPPAC2015.

Contact information: rlong021@gmail.com

Permanent address: 9531 Sims Dr. Apt 96

El Paso, Tx 79925

This thesis was written by Rogelio Long.