Autonomic Green Computing

Nikhil Gahlot
nikhilsgahlot@gmail.com

Neil Philip
nphilip33@yahoo.com

Liam Stewart
gset@liamstewart.com

Kristian Wu
kristian.wu@yahoo.com

Abstract

Thanks to revolutions in hardware architecture, computers can analyze large sets of data at high speeds, but do so at a high-energy cost. This key issue plagues a world making its first steps into the age of big data. This project aims to address this problem by creating a scheduling algorithm that minimizes the execution time of a program given a specific power budget. Different size programs were run on three models of parallelization to test which model is most efficient for which size of data. Through research, it was determined that a hybrid approach combining all three is the most effective for parallel computing, as it can handle and adapt too many scenarios. A scheduling algorithm was created to effectively delegate tasks to the resource that will achieve a minimum execution time given a power budget. Knowing that power consumption is a contradictory goal of performance of the system, and hence rightfully NP-hard, the situation was reduced to a system of linear inequalities. This led to the conclusion that effective green computing is possible via efficiently computable scheduling algorithms.

1. Introduction

By 2017, more than half of the global population will be connected to the Internet[1]. Given this increase in connectivity amongst people, corporations and governments alike will have to be able to efficiently and securely handle massive amounts of data. Faster computation speeds will create problems regarding the scarcity of physical resources such as power, and the scarcity of virtual resources such as memory.

Currently, many computers are capable of computing large amounts of complex codes at high speeds, but they do this at a high-energy cost. For every two watts of power taken in, one watt is wasted. This two to one ratio, or the power usage equivalent, is very inefficient and, ideally, should be reduced to a 1.2 to 1 ratio[5]. The current top system in the world, Tianhe-2 (MilkyWay-2), follows the pattern of a two to one power usage equivalent[3].

As a result of these inefficient energy practices, the architecture of supercomputers must continue to move

towards multiple-core processors. Many current systems, including the CAPER system created at Rutgers University, rely on multi-core processors in order to divide tasks among various processors. At the forefront of these processors is the Intel Xeon-Phi processor, which is three times faster than the highest rated commercial processor[4].

Evidently, the current state of computing is poor in regards to power efficiency. To help the problem, algorithms that can help allocate codes to different resources in order to alleviate current problems. Such an algorithm must improve upon time and energy efficiency[7].

2. Background

2.1 Speed Up

The speed up value allows for a comparison between different computational models by creating a ratio between the execution time of single processor computing to parallel computing. The larger the number, the more efficiency parallel computing is in the given situation. It determines the time complexity of a parallel computation and is calculated by:

$$S(p) = \frac{\text{Execution time using one processor (best sequential algorithm)}}{\text{Execution time using a multiprocessor with } p \text{ processors}} = \frac{t_s}{t_p}$$

Speed up demonstrates the potential benefits of using multiple threads or cores, while also displaying the marginal increase of efficiency of additional processors and threads[2]. Power efficient machines will have an increase in speed up as more threads or cores are utilized. Once speeds up stops increasing as more resources are used, the efficiency of the machine decreases.

2.2 Parallelization

Parallelization is the process by which tasks are divided amongst different processors and cores in order to maximize efficiency. As seen in FIGURE, the parallelization allow for the master thread to process many different threads at the same time. The green regions represent these threads that run at the same time. It breaks a problem down into discrete parts, or parallel regions. Parallel regions can be computed simultaneously on different threads. Parallelization is particularly meaningful when discussing efficiency because different methods of parallelization can result in computations that are either faster and more power intensive, or in computations that are slower and less power intensive[8].

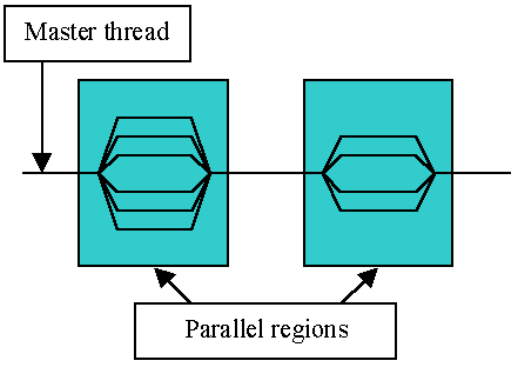


Figure 1: Parallelization Diagram

2.3 CAPER08

CAPER is the many-core system that Rutgers University created for research purposes. It consists of eight cores with 16 threads each and within each core, there are eight many-integrated cores (MICs) with 244 threads each[10].

Figure 2: Many-Core Computing

2.4 Multi-Core Computing

A multi-core processor is a single machine that contains two or more CPUs (cores). The multiple cores read and execute code and allow for improved performance, reduced power consumption, and parallelized processing.

2.5 MIC

A MIC is an example of multi-core computing[9]. Each MIC contains one Intel Xeon Phi processor and 244 threads. Programs can be run natively on a MIC.

Offloading is another means of utilizing MICs to run code in a parallel configuration. When offloading, the core transfers computations to different MICs or threads simultaneously[2]. This allows for several parts to be computed at once and combined later as opposed to just one part being computed at a time.

2.6 Benchmarks

A benchmark is a means of analyzing and testing the performance of computing models.

2.6.1 Rodinia

Rodinia is a package of applications and kernels designed at the University of Virginia that target CPU's, GPU's, and other multi-core computing systems in order to test and analyze their performance. Rodinia records the amount of power, memory, and time the computer took to complete the program. Rodinia computes large amounts of data in order to test the limits of computers and computational models. For this specific experiment, the LavaMD^2 benchmark was utilized, which measures the movements of a group of celestial bodies in a 3-D space, and then allocates various cluster nodes and particles into the space[12].

2.6.2 Matrix Multiplication

Matrix multiplication (Matmul) is a benchmark used to help determine the power and time consumption of specific computations. It can be changed to involve small or large sets of information[2].

2.7 The Power Capping Problem

The power capping problem deals with the creation of a scheduling algorithm that can minimize execution time while operating under a power budget (or power cap). A scheduling algorithm is an algorithm that delegates resources based on instructions from the operating system. The decision-problem (language) form of the power capping problem belongs to a complexity class known as NP (Non-deterministic Polynomial Time)[13]. A language in the NP complexity class has solutions that can be verified in polynomial time (and thus can be efficiently computed), but not necessarily found in polynomial time (or else the language would belong to the P complexity class). Additionally, the power capping problem is NP-complete, meaning any problem in NP can be reduced to the power capping problem in polynomial time.

2.8 The Scheduling Algorithm

A scheduling algorithm accomplishes the goal of minimizing execution time of

programs run on the CAPER cores. This algorithm is concerned only with execution and completion time.

2.9 The Power Capping Problem

The power capping problem is the problem of creating a power capping algorithm. A power capping algorithm attempts to minimize execution time while working under a specific power budget or power restraint. Power consumption and execution time are inversely related, leading to multi-objective optimization scenarios[7].

3. Experimental Design

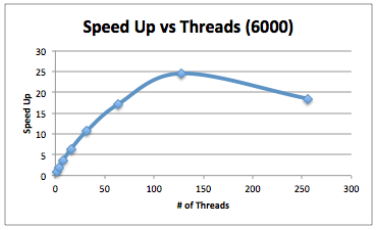
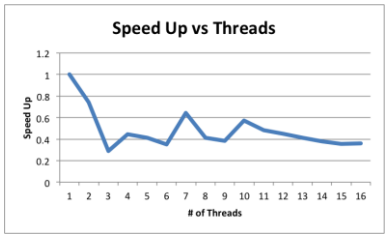
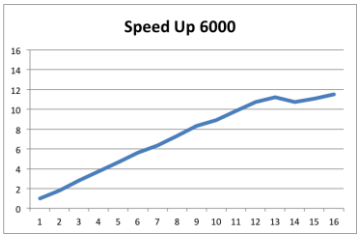
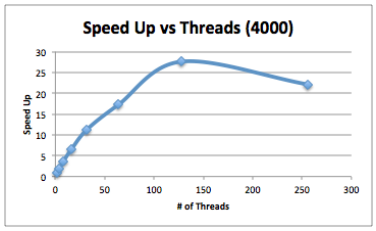
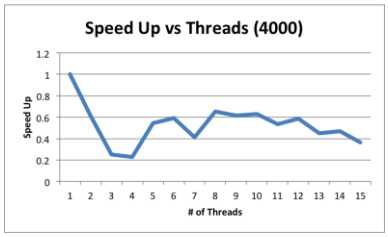
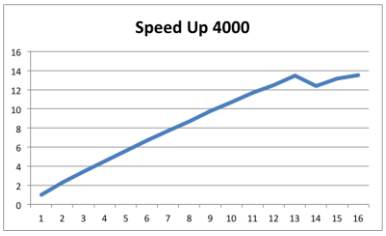
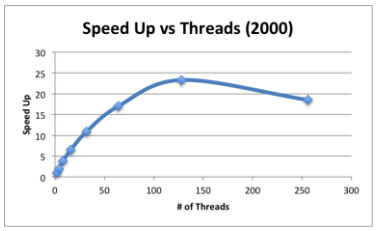
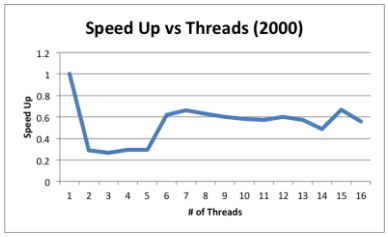
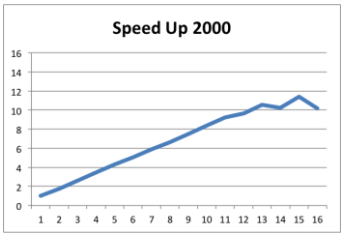
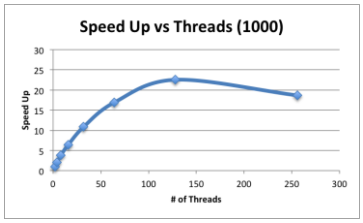
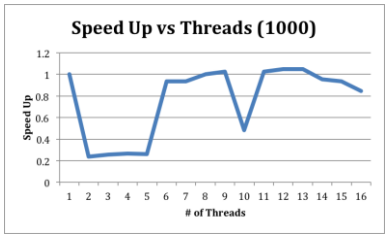
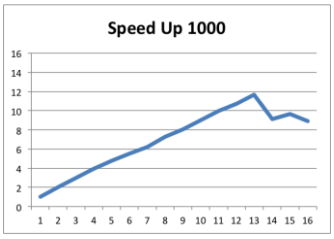
In order to improve the efficiency of modern servers, the effectiveness of three different models of computation, offloading, native core threading, and native distribution on Xeon-Phi, was studied[10]. After generating baseline power consumption, various benchmarks were run to actively monitor time and memory efficiency, as well as power consumption.

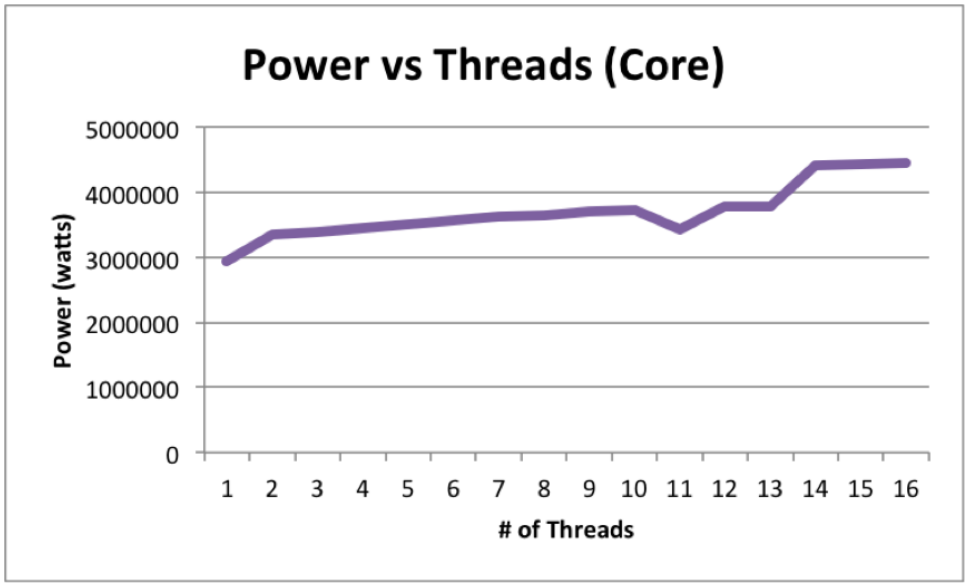
The Rodinia Benchmarks, and Matmul were designed to perform computations of large amounts of data. With Matmul and Rodinia Benchmarks, OpenMP was utilized, a framework that provides parallelization capabilities, in order to distribute the workload onto various cores, threads, and MICs. Arrays of size 1000, 2000, 4000, and 6000 while testing Matmul were used. The different sizes helped determine efficiencies based on the amount of data and computation. While these programs ran, the power consumption and time spent were measured and reported through Python and shell scripts. Given this data, a was computation. After collecting and comparing the data, two scheduling algorithms were created to handle resource allocation.

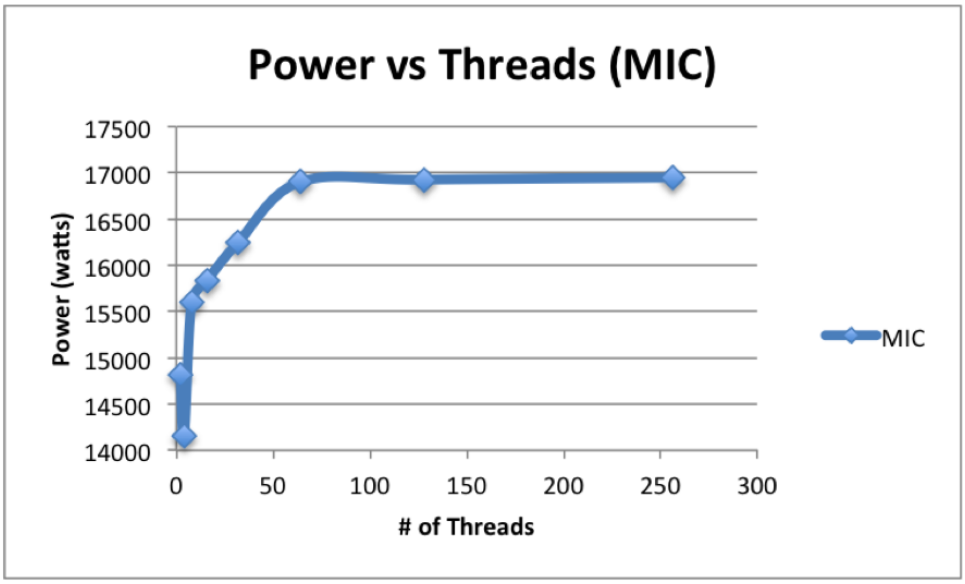
4. Results and Discussion

4.1 Matmul Results

While running the Matmul benchmarks, the time spent for the various computations processes - MIC, offloading, and Native Core Threading - were measured. When running Matmul natively on the MICs, there was a polynomial relationship between speed up and the number of threads. For each scenario, the maximum speed up was reached when 128 threads were used, and the speed up values began to decrease afterwards. Similarly, when recording power, the amount of power used reached an asymptote at 17,000 watts after 128 threads were used. For Matmul of size 6000, the maximum speed up was 25.43. These results were comparable to those of the native core threading model. The linear relationship between speed up and the number of threads occurred until the 14th thread, at which point the amount speed up began to decrease. The maximum speed up achieved was nearly 12. However, unlike MICs, the power consumption of native core threading consistently increased as the number of threads increased. Lastly, with offloading, the speed up values were maximized with one thread, and fluctuated after that. The power values of offloading mimic the power values of MICs because offloading utilizes MICs. The results of these tests can be seen through the following diagrams.

	MIC	Offloading	Core
6000			
4000			
2000			
1000			


Power vs Threads (Core)
Power (watts)
5000000
4000000
3000000
2000000
1000000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
# of Threads


Power vs Threads (MIC)
Power (watts)
17500
17000
16500
16000
15500
15000
14500
14000
0 50 100 150 200 250 300
# of Threads
MIC

4.2 Minimum Execution Time Algorithm

The minimum execution time algorithm allocates resources to achieve the minimum execution time by utilizing the primary allocation protocol that relies on the execution time of a given task on the various resources that are available. See appendix.

4.3 Power Capping Algorithm

The Simplex-Based Algorithm is as follows:

Power_Cap_Algorithm.cpp

To be implemented in C++ with OpenMP API

Specifications:

Define final double value that sets power cap.

Calculates power consumption versus threading choices based on experimental power data.

Reduces power consumption versus execution time relationships to system of linear equations.

Passes system of linear equations as parameter into Simplex Algorithm method, as well as goal of minimizing execution time variable.

Using simplex method, finds the minimum allowable execution time based on constraints. Calls modified Minimum_Execution_Time algorithm to generate a critical path that can be used to attain a suboptimal, yet reasonably efficient solution to the power capping problem.

4.4 Discussion

The results of the experiments demonstrate the limits of utilizing MIC threading offloading, and native core threading. With MICs, after 128 threads, the speed up values began to decrease, showing that adding infinitely many cores will not result in an increase of productivity. From 128 threads to 256 threads, the speed up values decreased from 25 to 20. Similarly, while running on a native thread, there was a linear trend until the fourteenth thread, at which point the speed up values decreased from 12 to 10. This shows that there is a limit to efficiency. Then, with offloading, there was evidence that using 1 thread was the fastest in terms of speed up, and every subsequent number of threads resulted in a decreasing and varied speed-up value. This illustrates how offloading can be effective for small calculations, in terms of time and power efficiency. In addition to this information about the speed information of various computational models, information about the power consumption of the models was also gathered. For MICs, the amount of power usage in watts increased from 14,000 watts to 17,500 watts as the number of threads increased in an exponential fashion, thus illustrating how power must be taken into consideration when dealing with overall speed up levels. Similarly, while running on a native thread, there was a slight increase in power as the number of threads increased, but the change was not as profound as with the MICs.

4.4.1 Minimum Time Algorithm

Based on the data, it can be ascertained that for small work-loads, native core threading is the most effective, as there is less memory contention, overhead, and bus transferal when compared to MIC threading and offloading.

4.4.2 Power Capping Algorithm

As described earlier, there currently exists no algorithm to solve the Power Capping Problem, as it is an NP complete problem. As a result, the project focus changed to the Power Capping Problem. The power objectives can be reduced to a linear constraint because the power-variable is capped. Thus, the entire scenario can be expressed as a system of linear inequalities. The linearity eradicates the need for a computationally complex Nelder-Mead algorithm[13]**.** Based on the data, it was determined that a Simplex Method is the most appropriate and effective algorithm for optimization in this situation. The CAPER scenario reduces to a simple system of linear inequalities, consisting of the power cap constraint, power consumption - performance relationships derived from the experimentation, and one dynamic variable. Under these circumstances, the Simplex Method, provides consistently faster runtimes[14]

5. Conclusion

The algorithms provide evidence to the concept that there is no superior method of parallel computation, whether it be native core threading, MIC threading, or offloading (as displayed by the trends in the Results section). As the algorithms show, power consumption has a significant effect on the execution time of an algorithm, but it does not stand in the way of efficient computing. In relation to the MICs, as the amount of power increased past the 14th thread, the overall speed up values decreased because of the increase in power consumption to 17,500 watts. A hybrid approach based on Minimum Completion Times can effectively use all three models of parallel computing to minimize execution time. Linear Algebra algorithms (such as the Simplex-Based one in the Results section) provide the most effective solution to computing under a power budget and can lead to vastly reduced energy consumption of server systems and reduced costs for IT firms. Efficient green computing is possible via polynomial-time scheduling algorithms, as demonstrated by the algorithms developed on the data collected. Further improvements upon this design can be made through more exhaustive testing of different computational models such as sectional-offloading. By doing so, more specialized approaches can be crated for problems that require different types of resources, such as time and memory. In addition, further research is needed for the implementation of this algorithm in order to determine the computational efficiency of the algorithm itself in relation to the servers it is managing.

Acknowledgments

This study of Autonomic Green Computing required the work of many others in addition to our own. Many people from Rutgers University generously dedicated many hours to help us overcome the various obstacles that we faced throughout the research process. We would like to specifically thank Dr. Javier Diaz-Montes and Dr. Ivan Rodero for their guidance. Without their help and extensive knowledge of the subject, this project group would not have been able to undertake this experiment. In addition, we would like to thank Daniel Hillman, our project RTA, for helping us organize and achieve the goals we set for ourselves in relation to our project and our papers. We would also like to thank our Math Behind the Machine instructor, Dr. Andrew Drucker, and our Software

Engineering with Android instructor Christine Hung. Next, we would like to thank the New Jersey Governor's School of Engineering and Technology and its director, Jean Patrick Antoine for the opportunity to perform this study. Lastly, we would like to thank all of the sponsors of the program: The State of New Jersey, Morgan Stanley, Lockheed Martin, Silverline Windows, South Jersey Industries, Inc., The Provident Bank Foundation, and Novo Nordisk.

References

1. World Internet Users Statistics Usage and World Population Stats. (n.d.). World Internet Users Statistics Usage and World Population Stats. Retrieved July 19, 2014, from http://www.internetworldstats.com/stats.htm

2. Rodero, I. (Fall 2014). Parallel and Distributed Computing. Lecture conducted from Rutgers University, Piscataway.

3. Top 500 Supercomputers - Home | TOP500 Supercomputer Sites. (n.d.). . Retrieved July 22, 2014, from http://www.top500.org/

4. Intel® Xeon Phi™ Coprocessor DEVELOPER'S QUICK START GUIDE . (2014, January 1). . Retrieved July 19, 2014, from https://software.intel.com/sites/default/files/article/335818/intel-xeon-phi-coprocessor-quick-start-developers-guide.pdf

5. Metrics & Measurements. (n.d.). The Green Grid Data Center Power Efficiency Metrics: PUE and DCiE. Retrieved July 19, 2014, from http://www.thegreengrid.org/Global/Content/white-papers/The-Green-Grid-Data-Center-Power-Efficiency-Metrics-PUE-and-DCiE

6. Home | TOP500 Supercomputer Sites. (n.d.). Home | TOP500 Supercomputer Sites. Retrieved July 21, 2014, from http://www.top500.org/

7. F. Zhang, et al., Multi-objective scheduling of many tasks in cloud platforms, Future Generation Computer Systems (2013), http://dx.doi.org/10.1016/i.future.2013.09.006

8. Ge, R., Feng, X., Song, S., Chang, H. C., Li, D., & Cameron, K. W. (2010). Powerpack: Energy profiling and analysis of high-performance systems and applications. Parallel and Distributed Systems, IEEE Transactions on, 21(5), 658-671.

9. Intel® Developer Zone: Intel® Xeon Phi„¢ Coprocessor. (n.d.). IntelÂ® Xeon Phiâ„¢ Coprocessor. Retrieved July 19, 2014, from https://software.intel.com/en-us/mic-developer

10. Beacon. (n.d.). Home. Retrieved July 19, 2014, from http://www.nics.tennessee.edu/beacon

11. Berl, A., Gelenbe, E., Di Girolamo, M., Giuliani, G., De Meer, H., Dang, M. Q., & Pentikousis, K. (2010). Energy-efficient cloud computing. The computer journal, 53(7), 1045-1051.

12. Rodinia:Accelerating Compute-Intensive Applications with Accelerators. (2014, June 20). . Retrieved July 19, 2014, from https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerator

13. Ullman, J. D. (1975). < i> NP</i>-complete scheduling problems. *Journal of Computer and System sciences*, *10*(3), 384-393.

14. Gondzio, Jacek; Terlaky, Tamás (1996). "3 A computational view of interior point methods". In J. E. Beasley. Advances in linear and integer programming. Oxford Lecture Series in Mathematics and its Applications 4. New York: Oxford University Press. pp. 103–144. MR 1438311. Postscript file at website of Gondzio and at McMaster University website of Terlaky.

Appendix

Minimize_Execution_Time.cpp

// To be implemented in C++ using OpenMP API

For each task{

Primary Allocation Protocol: Find resource with MCT and assign step to determined thread stream.

MCT (task i, resource j) = Exec Time(task i, resource j) + CT(resource j);

CT(resource j) is defined as the calculated critical time of a particular thread resource.

The critical time of a particular thread is the execution time of the all the queued tasks.

CT(resource j) is the summation of all tasks queued on resource j prior to assignment of task i.

Execution Time of specific resource j given task i is largely situation specific. It can be assumed that all the MIC threads would retain the same execution time for a given task i at step T in an algorithm (thanks to the random assignment principle of the scheduling algorithm and a large thread number), as they are equally affected by overheads, memory contention, and scheduling lag. Tasks that are executed natively in the MICs have the fastest execution times for large tasks, but are plagued by memory contention and restrictions as well as parallelization overhead.

ArrayList<resource> resList = new ArrayList<resource>();

resList contains list of all threads available. The elements of resList are of the resource class, which contains useful data and methods, as well as the identity of the element, including its location either in the native core or in a specific MIC.

tasksToBeAllocated() is a method that returns the steps of code that have yet to be accounted for by the algorithm.

ArrayList<Double> MCTchart = new ArrayList<Double>();

for (all tasks that need to be allocated){

for (int i = 0; i++; i < resList.size()){

MCTchart.add(MCT(task i, resList.get(i)));

//Generates the MCT value of all available threads

}

int index = MCTchart.getIndexOfMin();

*/*returns the index of the resource with the fastest execution time. In the event of multiple resources with the same MCT value, an index of 0 through 15 will be preferred. (as this represents a Native thread) If no native thread with MCT is available, by the use of a random integer function, one of the available resources with the MCT will be arbitrarily selected, ensuring by the prinicples of randomization that no MIC unit is giving delegation priority and selectively forced to bear a greater burden than other MIC units. This ensures an approximate equality of memory contention and overheads for all MIC threads in a CAPER core. */*

resource desThread = resList.get(index); //returns the desire resource allocateTaskToThread(desThread); } }