Research Article

Measuring the Energy Efficiency Ratio of Parallelized Software Applications

Delbert Bonner and Akbar Siami Namin*

Computer Science Department, Texas Tech University, USA

***Corresponding author**
Akbar Siami Namin, Computer Science Department, Texas Tech University, Lubbock, TX, USA, Email: delbert.bonner | akbar.namin@ttu.edu

Submitted: 22 January 2013
Accepted: 20 February 2014
Published: 28 February 2014

Index Terms
• Energy model
• Multicore platforms
• Green computing

Abstract

Rising energy costs, the shrinking size of mobile devices and political influences have begun to force device and software developers to look at ways in which they can reduce their energy usage. While most energy savings models can be found in how hardware is designed, software plays a key role in how devices can be more energy efficient because software is what ultimately controls the hardware it runs on. At the same time a careful balance between performance and energy savings must be maintained. In order to examine this balance, researchers have begun to put forth energy models and metrics that rely on dynamic voltage and frequency scaling to optimize performance and energy usage. The problem with these models and measurements being that while most software is ran on devices capable of changing their processor frequency and voltage, most developers do not have the ability to change these settings due to the operating system safety and security restrictions. We present an alternative energy ratio that uses the work and idle times of the processor to examine energy efficiency gain by parallelization of software systems. Using this ratio we show how software developers can examine their parallelization efforts and decide not only which method will provide them with the performance they seek while not sacrificing energy usage, but also when it is expedient to reduce the amount of processors used by their application. The model is evaluated through a number of scheduling algorithms and case studies.

INTRODUCTION

In 1965, Moore made a series of predictions that have since become part of the standard for which improvements in computing performance and costs are judged. The basis of Moore's Law is that both the number and the density of transistors on inexpensive integrated circuits would double approximately every two years [1]. This increase has allowed integrated circuit designers to increase performance of their chips and reduce their cost as well. The reduction in size and cost of integrated circuits and the devices that utilize them has provided the means with which computer devices have been integrated into everyday life. Computer devices, such as modern smart cellphones, come in packages many orders of magnitudes smaller and are significantly cheaper. It is now possible to purchase fully functional computers for less than the cost of a tank of gas [2].Transistor sizes have continued to decrease to the point where significant performance gains can no longer be achieved by a simple reduction in size. This has forced chip designers to look for other ways to improve their devices. Data caching, pipelining, and instruction set reductions are a few of the ways that processor designers have used to increase performance without having to rely on reducing the size of the transistors used.

Recently, CPU designers have turned towards building multi-processor CPUs. While multi-processor systems are nothing new, packaging multiple processors on a single chip is a fairly recent advancement. Not only have CPU designers put multiple processors on a single chip, they have arranged the data caches in such a way that communication between the processors has also improved. By and large, these types of improvements have been the basis by which Moore's law continues to hold true.

Much like how decreases in cost and size made computing devices readily available, so has the inclusion of multiple processors on a CPU expanded the reach of multiprocessor software development. The average developer now has the ability to write truly parallel applications and take advantage of the additional throughput this provides.

Rising energy costs

Over the past few decades, several forces have collided to demand a reduction in the energy demands of our devices. This runs counter to what has happened as a consequence of the increased performance gained by increasing the transistor density of the integrated circuits these devices depend on. By increasing the transistor density of integrated circuits, we have also increased the amount of energy they require. The increase in energy demands due to density increases has begun to outpace the energy reduction provided by smaller transistors. While today's processors have far greater processing power than those of just a few years ago, they also require much more energy.

One of the forces that continually influence decisions, no matter what is being decided, is cost. Lately, the cost of the energy required to run computer devices has begun to be a concern. Both the cost of producing energy and the demand for energy have risen sharply over the past few years [3]. This has caused energy prices to become a major factor in the operating expenses for any decent sized computer network.

In 2011, Google disclosed that it uses an estimated 260 million watts of electricity continually [4]. This is roughly the same amount of energy that 200 thousand home use, which runs about $1,500 per household [5]. In other words, Google's computational energy costs were over $250 million in 2011. While Google made $38 billion in 2011, this still represents a significant amount of money spent on a single expense [6].

Political influence

While costs have a major impact on all industries, politics can have the same level influence as well. The past few decades have seen the rise in the political demand for resource conservation and sustainability. What is being called the green movement has begun to impact the IT industry as well in the form of "green computing."

Green computing works towards the goal of having computing resources with little to no impact on the environment [7]. The basic objectives of green computing are the same as the general green movement: those of reducing environmental impact of production and disposal of computers, an increase in recyclability of afterlife devices and waste, and reduction in computational energy usage. The last objective is the only one that most IT professionals have any real direct influence over and usually the most focused on.

In 1992, the US Environmental Protection Agency launched the Energy Star Program. This program's goals were to encourage companies to improve either their own energy consumption or that of their products [8]. Other governments have introduced similar initiatives or have adopted the Energy Star program as well. The primary way by which the Energy Star program achieves its goal in energy usage reduction is by certifying appliances as either using only the minimal amount of energy necessary or including methods that work towards reducing its overall energy usage. Energy Star estimates that their certification program helped Americans save over $20 billion in 2010 [9]. When Google disclosed how much energy their data centers were using, they also used that as an opportunity to discuss the ways in which they were trying to become greener [4]. Some of Google's green initiatives include an expansion in its usage of renewable energy sources and designing data centers that both run hotter than normal and use natural methods for temperature control.

As part of getting computers certified by the Energy Star program, hardware designers introduced the Advanced Control Power Interface (ACPI) industrial standard for reducing power consumption of idle computer components [10]. The basic concept for a device that is ACPI compliant is that as it remains idle it moves from a higher energy using power state to one with a lower energy usage. For processors, this usually means that each processor has at least two power states: an operational state and a stop or halted state.

Smart grid and energy optimization

Between 2000 and 2001 California experienced what has since been referred to as the California Energy Crisis [11]. While there were several influencing factors, the basic problem was that energy demands in California were allowed to outstrip the energy that was being supplied. The ultimate result was that over 1.5 million energy customers were affected by rolling blackouts, sometimes in the heat of summer. Investigation into the causes identified illegal market manipulations as the primary cause of the energy shortage, but the wide spread blackouts would have been lessened had the power grid been better able to handle the demands placed on it.

Events like the California energy crisis, coupled with rising energy costs and pressure for sustainability in our energy production has placed a greater emphasis on how our energy grids are managed. The key to this management is to making sure that energy production meets the demands. Unfortunately, energy usage demands and production methods are far from constant. For example, solar energy plants will produce more energy during bright sunny days than at night or on cloudy days. At the same time, the energy usage patterns of the different energy customers follow similar daily and seasonal patterns. Smart grids, or power grids that utilize computers and information gathering technology to manage how the energy is propagated across the grid, are being looked at as a means of optimizing the power grid along the lines of these energy usage and production patterns and reduce the likelihood that future energy crises occur.

Smart grid technology seeks to finds ways to improve the power grid by incorporating information technology into the various parts of the grid [12]. Currently, information technology is only deployed locally in the power grid as a measure of safety to protect power assets from overloading and failure. Smart grid proponents and policy makers seek to further increase the amount of information that is gathered and analyzed by computers deployed within the grid. These individuals hope that by turning the management of the power grid over to computers that not only will power management be increased but that the grid will also be more resilient to failure by giving it the ability to self-heal when a failure in one or more components occurs.

The ultimate goal of building a smart grid is finding ways in which power demands can be met using the most optimal methods, hopefully utilizing renewable or sustainable sources. By using smart grid technology, power grid managers have the ability to fully utilize excess energy that is produced cheaply, like that from a solar power plant at peak times, even if the peak production time does not match peak demand times by either shuffling the energy around to where it is needed or storing it off for later use. While this is possible without the use of a smart grid, smart grids make it much more affordable by being able to predict when it is more efficient to store excess energy or shuffle it across long distances.

Mobile energy demands

The decreasing size of circuits has given rise to a new type of energy management problem in form of mobile computing. What started as easy to setup portable computers now encompasses a variety of devices, including laptops, mobile phones, and tablet computers. Each of these devices utilizes some sort of battery to meet its on-the-go energy demands. To keep the total cost of these devices to a minimum, these devices almost always include a rechargeable battery.

In order to increase the mobility of their devices, developers strive to increase the amount of time a device will be able to operate between recharges or battery replacements. Regardless of whether the battery is rechargeable or not, the batteries in mobile devices only provide a finite amount of energy. This means that there are two different ways by which a device can increase its mobility: increase the capacity of the battery or decrease the energy demands of the device itself.

As current battery technology is limited in what it can do to increase the capabilities of the batteries powering mobile devices, the focus of mobile device development has shifted to minimizing the energy demands of mobile devices. Also, mobile devices have decreased in size, which places a severe limitation on the size of the battery included. For this reason, mobile devices usually do not use the same hardware as their desktop and server siblings. Instead, they use processors that are designed to both use less energy and generate less heat. In most cases, the tradeoff of performance for mobility is acceptable.

Software's impact on energy

While the largest amount of energy can be saved by improving computer hardware, software can also play a significant role [13]. The most basic way in which software affects energy consumption of a computer is in how the software utilizes and controls the hardware it has available. In other words, how an operating system manages the energy states of each of the devices can play have a major effect on how much energy a device use.

Early in 2011, the Linux foundation released the 2.6.38 version of the Linux kernel. Shortly after that, Tom's Hardware did a review of the latest Ubuntu release to that used the 2.6.38 kernel and found that the battery life for their testing rig dropped by almost 50%, confirming an early report of a possible problem with the kernel's power management [14].

The problem was found to be in the Active-State Power Management (ASPM) for PCI Express [15]. What happened was there is an issue certain BIOSes that have their ASPM support miss-configured, and this can cause various problems if the power mode is dropped on unsupported devices. To work around this issue, ASPM for the PCI express was disabled and its state cleared when it appeared that ASPM was not supported. The maintainers for the PCIe driver found that the proper solution was to only clear the ASPM state only when the BIOS handed control over to the operating system [16].

What this illustrates is that because hardware is ultimately controlled by the software ran on it, software still plays a major part in how much energy devices use. This means that ultimately software developers must pay close attention to how they write their software if they don't wish to negatively impact the power usages of the devices it runs on.

The structure of the paper

- In this paper we present an energy efficiency ratio for multiprocess applications that relies on CPU idle and work times. We derived this model from the power usage of a CPU and Amdahl's law [17] and show how it can be used to determine if a multiprocess solution to a problem will provide the desired energy savings. We then use this ratio to examine the energy efficiency of different applications and task schedulers, and then use this information to make arguments for which task scheduling method to use and the number of processors our application can be parallelized across. This paper's contributions are:Introduce a speedup ratio that relates CPU utilization with energy usage.
- Use this model to examine the energy efficiency of two different applications and several task schedulers.
- Examine how this ratio combined with Amdahl's law can be used to determine the optimal number of processors for a parallel application to use.

The rest of this paper is organized as follows: Section 2 reviews other models and energy saving techniques. Section 3 examines how software engineering can affect energy usage. Section 4 looks at task scheduling and how it is used to save energy in parallel applications. In section 5 we introduce our energy efficiency ratio. Section 6 is where we examine our case studies and apply the ratio. And section 7 presents our conclusion.

RELATED WORK

Software development researchers have recently begun turning toward looking at how software can be written with energy conservation in mind. The goal these researchers are looking for is to predict the energy usage of differing parallelization methods and choose the one that will meet the performance demands while minimizing energy usage. The basic idea is to slow the processors down during idle times or when there is more time than necessary to complete a given task [17]. Multithreaded task schedulers use this method of reducing the frequency of the processors when tasks have more time to execute than they need [18,19]. By using dynamic voltage these schedulers can manipulate how they schedule their tasks to minimize application energy usage.

Being able to accurately model energy usage is of extreme importance. Rountree et al. [20] pointed out that reducing 10% of the processors in a cluster by 50% or more would net an energy savings of over 5%, but only if this reduction did not delay critical tasks. If the system is forced to remain active due to these critical tasks by an additional 1%, then the system will actually use more energy than it would have if it had not reduced the processors operating frequency [20]. One of the key focuses of this line of research has been the development of software energy usage models and comparison ratios. These models' and ratios' goals are to give software developers the information they need in order to develop energy conscious software.

Ge and Cameron define an energy model that relies on the operating frequency of the processors [21]. This model allows developers to estimate the energy performance of a parallel application running at different frequencies. Ge et al. went on to use this model in developing a runtime power management system for high performance computing clusters, aimed at using dynamic voltage and frequency scaling to minimize a cluster's energy usage [22]. By utilizing this power management system, Ge et al. were able to achieve over 20% energy savings for the NAS parallel benchmarks.

Like Ge and Cameron, other researchers have applied energy modeling to Amdahl's law in order to get a good idea on how well an application's parallelization will save energy. Song et al. [23] developed a model based on a large range of different parameters that all have an effect on the amount of energy a computer system uses. They look at such things as what part of the workload takes place in the CPU versus memory reads/writes, parallelization overhead, and the various operating parameters of the device. Using these values, Song et al were able to estimate the iso-energy-efficiency of an application and provide application developers with a way to fine-tune the performance of their applications for energy conservation with only negligible performance loss [23].

Rountree et al decided to forgo simply modeling an application's energy by indirect methods, and instead proposed a framework whereby they were able to measure the workloads on and off the CPU [20]. This framework inserts memory load markers into the data caches, which indicate when a processor is waiting on a memory read or write. Using these markers this framework is able to determine when a processor's frequency can be reduced, thereby saving energy. Rountree et al used this approach to reduce the median absolute error by an order of magnitude.

The problem with most of these methods for both modeling and reducing energy usage is that they rely on things outside the reach of most software developers. While these methods are great for operating system, compiler and even hardware developers, software developers will rarely be able to make much use out of these methods due to either their complexity [23], their reliance on setting inaccessible to user level applications [21], or the need to add operating system or hardware monitoring methods [20]. Instead, software developers need models that contain information they have easy access to and can actively affect, such as CPU idle time.

ENERGY-CONSCIOUS SOFTWARE ENGINEERING

The combination of rising energy cost, increased mobile devices, and political influences has caused there to be a greater concentration on ways to reduce the energy demands of our computers and related devices. Most research and development into computer energy usage focuses on how to decrease the energy demands of computer hardware. There are many different parts of a computer that require their own energy conscious design, providing hardware developers plenty of things to look at.

As illustrated by the power issues that the Linux kernel recently experienced [14], software can play a major role in how much energy a device uses. Computer science research has begun to start looking into improving software engineering practices so that it not only focuses on application performance, but also energy usage as well.

Intel's response to green computing has been to release processors that are more energy conscious. A book published by Intel, *Energy Aware Computing*, helps software developers make their software more energy efficient [24]. In advance of the book, the authors have released a few papers covering some of its material. The key focus of what the authors present is minimizing what software is doing while the system is idle [25] and maximizing the time the computer can sit in an idle state [26]. For the later, they offer the basic suggestions of simply improving the overall performance of the software that gets written. Their suggestions reap the most fruit when applied during the design phase, since it focuses on picking the most optimum algorithms and data structures for the given tasks. Their reasoning is that if a computer is able to finish its work quickly, then it will be able to return to a lower power state sooner and begin saving energy. They call this the "*race to idle*". Due to the race to idle, programmers can greatly decrease the energy consumption of the devices their programs run on by simply optimizing their algorithms.

Effects of nested loops

The loops within an application offer a good straightforward area of optimization because by improving the tasks within a loop by a small amount, that small amount is multiplied by the number of iterations the loop goes through. The easiest way to improve the execution of a loop is by making sure that if there are data accesses within that they are in the same order as the data is stored in memory. Figure 1 shows the standard arrangement of the elements in a two dimensional array. If each row *i* is followed immediately by the next row, *i+1* , then *i* is the *major* index and *j* is the *minor* index. When iterating across this multi-dimensional array, the outer most loop should iterate across the most major index, *i*. Then the next nested loop should be for the next most major index until the inner most loop is iterating across the most minor index. For this two dimensional array the inner loop's index would be *j*. Arranging loop iterations so that memory accesses are sequential has a high chance of improving cache hits, which reduces both the energy expended in the memory accesses and the time it takes for each of those accesses [27].

Memory access *vs.* data caches

Memory and data caches are one of the improvements to hardware that has greatly increased performance without increasing clock speed. A successful cache hit can greatly reduce the amount of time a processor has to wait for a piece of data

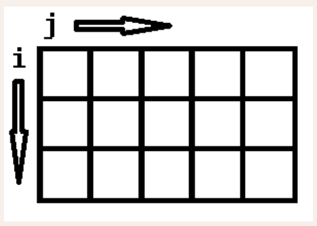

Figure 1 Two dimensional array arrangement.

to be available. A great way to improve performance is to limit how much memory an application needs at any given time to how much will fit in the CPU cache. By keeping the immediate memory needs of an application small enough to fit within the CPU cache, a software developer can insure that cache hits will rarely miss. If the memory needs of an application will not fit within the CPU cache, then another option is to focus the work being done on blocks that will and only moving on to the next block when all operations that can be completed on the current block have finished [28].

Effects of multithreading

With the advent of readily available and inexpensive multiprocessor system, a new avenue has opened up for the average software developer to improve the performance of their applications: multithreading. Multithreaded applications give developers the potential ability to decrease the amount of time it takes for an application to perform any given task. Additionally, multithreaded applications can provide a means of keeping the CPU busy when it has to wait on either devices or memory loads, which in turns helps improve an applications performance.

The first major availability of multiple processors came in the form of Intel's hyper threading technology. Essentially, this is a hardware trick where the processor reports double the number of actual computational units. A hyper threaded CPU is able to do this because it has doubled the number of state registers, allowing it to maintain the state of different execution threads simultaneously. By doing this, the CPU is able to be continuously executing tasks even when a non-hyper threaded CPU would stall do to a cache miss or some other wait operation [29]. Multithreading can have a massive effect on the runtime of an application. Even on a single processor, by adding additional thread software developers can still increase the performance of their applications since the processor will be kept busy. Multithreading has the potential to greatly improve the performance of an application, but only if care is given as a single mistake can be much more costly in a multithreaded application than in a sequential one. For example, mistakes in thread synchronization can cause dead locks or unintended delays which can add up to the point where a multithreaded application's performance is worse than its sequential equivalent.

As an example, in a recent exercise in application parallelization, we took a finite difference solver used as a way to model the propagation of an acoustic wave through a stochastic velocity model, in order to generate synthetic seismic data [30,31,32]. From the equation for the acoustic wave modeling, each cell is dependent on its past two values, as well as the most previous value of the two cells above, below, to the left, and to the right. Figure 2 shows an example of the dependencies that exist in calculating each cell. The core of the calculations took place in a simple sequential matrix data processing loop, itself in another loop so that the matrix could be recalculated for each time slice of the wave propagation.

The original application was written in MatLab, but for this exercise we decided to write it in C++ in order to gain direct access to the threading methods. Due to the nature of the dependencies across each time slice, only the matrix processing loops were parallelized. A simple block tiling method was used to cut the matrix up into equal blocks for each thread to work with.

Two different threading approaches were used in the final application. The first method started and stopped of each thread after it was done working for the current time slice. The other method only created the threads once, but used barriers to synchronize the threads with which time slice they should be working on. The application was run with between 1 and 32 threads on a system with a quad-core processor.

Figure 3 shows the execution times with the varying number of threads. A couple of noteworthy things can be seen in Figure 3. First, there is an immediate decrease in the execution time with only a few threads, and that adding more threads after a certain point does not continue the downward trend. This brings up the second interesting point: using the barriers to keep the threads synchronized ads enough overhead that after a certain number of threads the application actually performs worse than just running it with one thread. This is in contrast to starting and stopping the threads after each time slice where the performance usually gets slightly better with each thread added. Looking at Figure 3 and using the "race to idle" argument, we can make the assumption that the method of starting and stopping the threads will be more energy efficient. Additionally, we can assume that trying to use more than 6 or 7 threads will only decrease our energy efficiency, since the performance gains taper off after that point.

When there are more threads than there are available processors the system must swap the threads across those processors. This causes the barrier method performs worse than

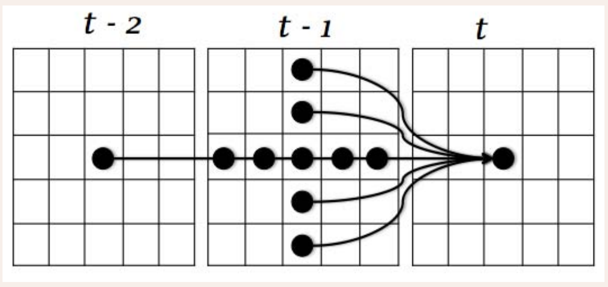

Figure 2 Finite difference solver cell dependencies.

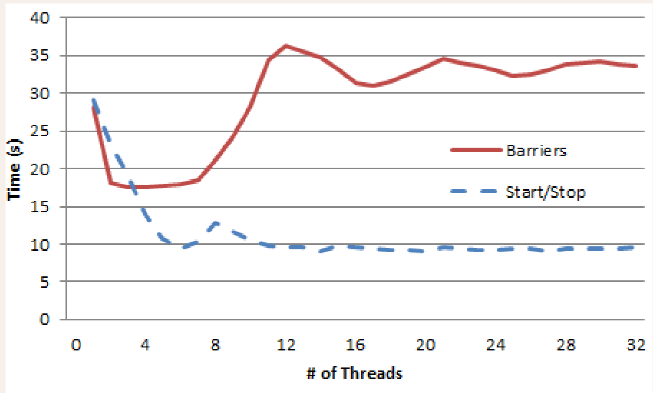

Figure 3 Finite difference approximator execution time by thread count.

the start and stop method due to context switching, cache misses, and synchronization costs. All of these things add up to severely decrease the performance of the barrier method.

On the other hand, the start and stop method does not have the context switching issue due to the fact that by the time each of the later threads are started, the first few have completed, thereby keeping the number of executing threads low enough to keep context switching to a minimum. Also, at a certain point the block that each thread works on gets small enough to fit within the processor cache. When all the data that a processor needs to complete a task is available in its cache, the processor's prefetch will be able to fill the cache with this data and the processor will have this data available there for the duration of the task's execution [28].

Adding additional threads to an application does not necessarily guarantee better performance. Instead, the only guarantee given by additional threads is an increase in complexity. In fact, if proper care is not given, application performance can actually degrade beyond usability due to threading complications such as resource locking and task scheduling.

POWER EFFICIENT TASK SCHEDULING

Researchers look for ways in which applications and tasks can be scheduled to both optimize their performance and energy usage. We look now at one of the ways in which tasks can be scheduled for both performance and energy conservation.

The job shop problem

One of the fundamental problems in developing efficient multiprocess applications is known as the job shop scheduling problem [33]. Graham described this problem as having a set of tasks $\mathrm{T}=\{\mathrm{T}_1,\ldots,\mathrm{T}_\mathrm{m}\}$ that are to executed uninterruptedly on n identical processing units P_i [33]. In addition, there exists a partial-order $\prec$ on T that states is if $\mathrm{T_i}\prec\mathrm{T_j}$ then T_j cannot start until T_i completes and a function of time $\mathrm{ì}:\mathrm{T}\rightarrow[0,\infty)$ The order in which the tasks are executed given by a linear ordering $L:\left(T_{k_1},\ldots,T_{k_m}\right)$ of T, called a task or priority list. Efficient task lists are the ones that minimize the makespan, or total time it takes to execute all the tasks.

Graham further demonstrated that the tasks, their partial ordering $\prec$ on T and the function μ could be represented by a directed graph $G(\prec,\mu)$. He set $G(\prec,\mu)$ such that the vertices corresponded to the tasks, T_i and the directed edges from T_i to T_j would indicate that $T_i\prec T_j$. Finally, Graham weighted the vertices of the graph, where the weight of each vertex is the length of time each task takes to execute. From this setup, application developers and researchers have been able to devise various priority lists, or schedules, by applying graph routing techniques [34].

Cyclic and acyclic task graphs

There are two different ways that tasks can be modeled using task graphs. The first is when each vertex represents a single task that executes only once [34]. This scenario produces an acyclic graph. The second way is to have each vertex represent a generic task that is executed infinitely often. This special case of the job scheduling problem is known as the basic cyclic scheduling problem and is "the most elementary formulation for studying repetitive applications," and is referred to as a "*reduced dependency graph*" [34]. This reduced dependence graph is denoted by *G*=(*V,E*), where *V* is the list of generic tasks, and *E* is the edges between the vertices which represent the partial-order dependencies. From this, researchers labeled each operation by a pair of indices $\{(v,k)\,|\,v\in V,0\leq k<N\}$, where *N* is the number of executions each task will undergo.

Researchers have further expanded the reduced dependence graph by weighting the edges, creating a weighted directed graph, denoted by *G*=(*V,E,d,w*), where the function $d:V\rightarrow\mathbb{N}^*$ is the duration of each task, and the function $w:E\rightarrow\mathbb{N}$ gives the dependence distance of each edge. The edge weight function, *w*, states that for any edge $e=(u,v)\in E$, and for any *k* such that $0\geq k<N-w(e)$, the operation $(v,k+w(e))$ cannot start before the operation (*u,k*). For example, in Figure 4 the duration of task A, *d*(*A*) is 4 and the edge between it and task B has an edge weight, $\mathrm{w}(\mathrm{e}_{\mathrm{A}\rightarrow\mathrm{B}})$, of 3.

As stated earlier, efficient schedules are ones that minimize their makespan. Scheduling problems solved by using an acyclical graph look at the total makespan of the schedule, considering each execution of each task. If there is a loop within the application, then the schedule is highly dependent on the number of iterations, *N*, within the loop. This means that a schedule is a function $\sigma:V\times\mathbb{N}\rightarrow\mathbb{N}$ that respects the dependence constraints:

$$\forall e=(u,v)\in E,\forall k\geq 0,\sigma(v,k+w(e))\geq\sigma(u,k)+d(u)$$

By using a reduced dependence graph when trying to schedule the tasks within a loop, researchers have been able to create schedule for any value of *N*. For these types of schedules, researchers measure the efficiency not by looking that the total makespan for the schedule, but by the average cycle time, *λ*, which is defined by [34].

$$\lambda=\liminf_{N\rightarrow\infty}\frac{\max\{\sigma(\mathrm{v},\mathrm{k})+\mathrm{d}(\mathrm{v})\,|\,\mathrm{v}\in\mathrm{V},0\leq\mathrm{k}<N\}}{\mathrm{N}}$$

Additional complexities arise when the reduced dependency graphs exhibit cyclic patterns. When there exist cyclic patterns within a graph, tasks end up depending on themselves. These graphs produce cyclic schedules, which is a schedule such that $\sigma(v,k)=c_v+\lambda k$ for some $c_v\in\mathbb{N}$ and $\lambda\in\mathbb{N}$. The schedule *σ* is a periodic schedule that schedules slices of the overall schedule with period *λ* units of time. Within each slice, only one instance of each generic task is executed.

Schedules can be produced from reduced dependence graphs by using two different methods. The first method schedules the

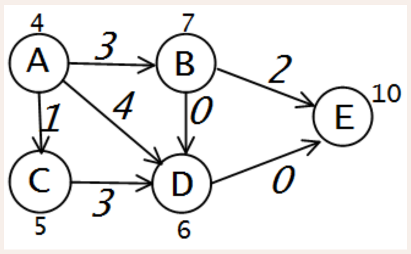

Figure 4 Weighted directed graph [34].

body of the loop without mixing up its iterations. The general idea is to remove the inter-iteration dependencies, or those edges with non-zero weight, $w(e) \neq 0$, and then simply use acyclical scheduling method. From here the scheduler can then calculate the makespan of $\sigma_a : \lambda = max_{v \in V}\left(\sigma_a(v) + d(v)\right)$ and the cyclic schedule σ by:

$$\forall \text{üüëüi} \;, \forall \;\; \in \mathbb{N}, \sigma(\;\;,\;\;) = \sigma_a(\;\;) + \lambda$$

The other method mixes up the iterations of the loop in such a way that the dependencies are still kept but minimized the average makespan. If given an unlimited number of resources, then the minimum makespan for a cyclic schedule is the maximum cyclic duration to distance ratio, $\tilde{n}\,(C) = \dfrac{d(C)}{w(C)}$ where C is any cycle in G [35].

Potential graphs

When there exists a cycle within a task graph, it introduces additional complexities that scheduling algorithms must deal with. By disregarding the edges of G where $w(e) \neq 0$, which removes the inter-iteration dependencies, developers are then able to use acyclical scheduling algorithms within the body of the loop. Schedulers can also take the reduced dependency graph and transform it into a "*potential graph*." A potential graph $G = (V, E, w)$ is a task graph where $w : E \rightarrow \mathbb{Z}$ defines the edge weights and the schedule σ for G is a function $\sigma : V \rightarrow \mathbb{N}$ such that all potential inequalities are satisfied:

$$\forall \mathrm{e} = (\mathrm{u}, \mathrm{v}) \in \mathrm{E}, \sigma(\mathrm{u}) + \mathrm{w}(e) \leq \sigma(\mathrm{v})$$

A potential graph primarily differs from a reduced dependency graph in that it contains a source task *s*. The source task has a start time σ *(s)=0* and duration *d (s)=0*. The source task is essentially a jumping off or starting point for the scheduling algorithm.

Using the source task, scheduler algorithms map the potential inequalities in the reduced dependency task graph to a potential graph. These scheduling algorithms first re-introduce all of the original dependencies as edges *e* where if *v* depends on μ the edg *e=(u,v)* but set the weight *w(e)=p(u).* These edges provide the inequality: *σ (u) +p (u) ≤ σ (v).* The dependency edges for the ready time *r(v)*of the tasks, or the time that the task has to wait before it is able to start, are added by the algorithm next. The algorithm assigns this edge, *e=(s,v)* the weight *w(e)= r(v)*and in return, it provides this inequality: *σ (s) +r(v) ≤ σ (v),* and since *σ (s) = 0* this satisfies the constraint that *r(v) ≤ σ (v)*. The final set of dependency edges the algorithm adds are for the due time, *d(v)*, which is the time that the tasks much be executed before. For this edge, *e(s,v)* the algorithm assigns the weight *w(e)= d(v),* and the resulting inequality is: *σ (u) +p (u) – d(v)≤ σ (s)=0.*

By introducing all of these different edge dependencies, cycles develop within the potential graph. Like a reduced dependency graph, having these cycles makes it difficult to easily determine if there is a valid schedule for the graph that obeys all of the inequalities. It has been shown that as long as all the simple circuits in the potential graph have nonpositive weight, there exists a schedule that obeys all of the inequalities [34].

Energy aware task scheduling

The goal of most scheduling algorithms is to minimize application execution time. Minimizing application execution time normally comes at the cost of powering additional processors or increasing the operating frequency of those processors. As previously stated, getting back to an idle state is a great way of reducing energy costs, but adding the cost of powering additional processors or having them run at higher speeds can offset the power gains of a quick return to idle. For this reason, several scheduling algorithms have been put forth to try and find the balance between quick executions and power conservation [21].

When scheduling each task in a task graph, the possibility exists that a task has more than enough time to execute. This extra time would be when the ready time plus the duration of a task is less than its due time, $r(v) + p(v) < d(v)$ scheduling techniques that seek to reduce power usage without negatively impacting performance use this time by reducing the operating frequency of the processor these tasks run on. By reducing the speed at which the task executes, the schedule increases the duration of the task eating up the excess time and in turn is able to reduce the amount of energy the task uses [18,19].

CPU load balancing presents another great way in which scheduling algorithms can decrease energy usage. CPU load balancing does not directly affect processor energy usage, but instead it is a means by which scheduling can reduce the cooling demands of the CPU. The more work a processor does the hotter it becomes, so CPU load balancing seeks to reduce this heat buildup by spreading out the work load across multiple processors. CPU load balancing does cause performance degradation due to the increased context switching and decrease in cache performance [36].

MODELING CPU POWER USAGE

Power and energy conscious computer and software designs look for ways in which power usage can be efficiently balanced with performance. For this reason, computer scientists have begun looking for ways in which they can model computer energy consumption. These models give them the ability to gage whether a particular energy saving technique provides the benefits necessary to overcome the loss in performance. Ge and Cameron proposed an energy model that showed the effects of changing the CPU operating frequency [21]. This model provides a great argument for not operating CPUs at the absolute highest possible speeds. They were able to prove that at a cost of only 1% in performance loss a savings of over 30% could be saved in energy usage by plainly reducing the CPU operating frequency.

Another method by which CPU energy usage can be modeled is by estimating the power usage of each individual instruction [37]. Measuring the power used by each instruction an application uses will give a fairly accurate model the energy a CPU will use while running the application. Utilizing the information provided by instruction level power analysis, application developers can optimize their applications to use less power intensive instructions thereby reducing the amount of energy their applications consume.

Energy ratio model

most CPU power models require that those who use them be able to know intimate details about how much power the CPU will use or be able to affect the physical properties and settings in some way. Unless a software developer is working directly with the hardware designers of a system, he/she will probably not have the required knowledge of the CPU. At the same time, most software developers will not be able to change the physical settings of the CPU due to security and system stability restrictions placed by the OS. For this purpose, we put forward a CPU energy model based off of the CPU utilization and the operating frequencies.

CPU power usage

Power is the rate at which a task uses energy, i.e. $P = \frac{\Delta E}{\Delta t}$, where the energy change the system under goes is, ΔE and Δt is the time it takes to complete the task. From this then the energy used by a system can be derived as the amount of power used over a given amount of time, $\Delta E = P * \Delta t$ [38]. If the amount of power a device uses varies with time, then the energy usage is $dE = P(t) * dt$. By taking the integral, we get the total energy used by the system as $E = \int P(t)dt$.

In a multiprocessor system, each processor will have its own power usage equation and therefore be using a different amount of energy. This means that the total energy consumed by the CPU is the sum of each of the processing cores' energy usage, giving us the following energy equation $E = \sum_{i=0}^{N} \int P_i(t)dt$, where N is the number of cores in the CPU, with $P_i(t)$ being the power equation for the *i*-th processing core.

This equation shows that there are two different ways in which energy usage can be decreased. The first way is to reduce power used while completing a task and the second way is to reduce the time it takes to complete the task. Computer researchers and designers have put more work on the physical properties of a CPU because by doing so energy can be saved no matter what the software running on it does. As previously mentioned, software developers can still have a significant impact on the energy usage of the devices their applications run on [16,25].

We can now use Ohm's law to get a good estimation of the power used by a CMOS chip. CMOS chips, upon which CPUs are generally based, use energy by charging and discharging set of capacitors, the power used to do so is given as $P = CV^2F$, where C is the capacitance, V is the voltage, and F is the frequency at which the chip changes state [39]. Of the three properties that make up this equation, only frequency can be altered without making physical changes to the CPU or its settings. For this reason, most power saving designs utilizes lower operating frequencies. Using the equation for the power of a CPU transforms the equation for the energy used by a CPU into *üüü* $\sum_{i=0}^{N} \int \quad ^2 \quad _i(\ )$.

The clocks that drive most processors are not capable of continuous frequency ranges, but are restricted to discrete values at which they can operate in order to keep their design simple and be cost effective. Limiting the operating frequencies means that each processor will only be operating in discreet power states, and the energy equation becomes:

$$E = \sum_{i=0}^{N} \sum_{j=0}^{S} CV^2 f_j t_{ij}$$

where N is the number of processors, and S is the number of states the processor operates in.

The ability to set the processor's frequency to different values has allowed operating system developers to have a great amount of control over power consumption. Unfortunately, as previously stated, applications running in user space do not have these capabilities due to security and system stability concerns. Since user space applications are restricted from having the ability to alter the operating frequency, software developers' only means of controlling the CPUs' energy usage is by altering the CPU utilization of each processor.

This means that developers can only affect the energy consumption of their application by changing how much time each processor is in use. This alters the energy equation to only having two CPU states that is must be concerned with, ON and OFF. Most processors, especially those that are ACPI compliant, have more than just two operating states. How many of those operating states that get used depend on what the operating system does when halting a process thread or when a thread waits on data accesses. Since how these CPU states get used are not within the capabilities of most software developers, we consider all the non-active states the same as if the CPU was not in use. With the model only considering two states (ON and OFF) and also assuming that the processors are homogenous causes the energy equation to become:

$$E = \sum_{i=0}^{N} CV^2 (f_{ON} t_{iON} + f_{OFF} t_{iOFF})$$

f_{on} and f_{off} being the frequency of the processors in the ON and OFF state respectively, and t_{i0n} and t_{ioff} as the time each processor spends in each state.

Sequential application power usage

A sequential application can only utilize a single processor. With only one processor to consider the energy used is then $E = CV^2(f_{ON} t_{ON} + f_{OFF} t_{OFF})$. Since the application is only using one processor, the other *N-1* processors are sitting idle and are operating at the OFF frequency for the duration of the execution. The energy these processors use is $E = CV^2 \sum_{i=0}^{N-1} f_{OFF}(t_{iON} + t_{iOFF})$. Without any changes between idle processors' energy usage, the summation reduces to simply $f_{OFF}(t_{ON} + t_{OFF}) * (N-1)$, meaning that the energy used by the idle processors is $E = CV^2 f_{OFF}(t_{ON} + t_{OFF}) * (N-1)$. For a sequential application the total energy used by the CPU is therefore:

$$E = CV^2 (f_{ON} t_{ON} + f_{OFF} t_{OFF}) + CV^2 f_{OFF}(t_{ON} + t_{OFF}) * (N-1)$$

By combining and rearranging terms, then using algebraic simplification, we can reduce the CPU energy used by a sequential application on a multiprocessor system to:

$$E = CV^2 (N f_{OFF}(t_{ON} + t_{OFF}) + t_{ON}(f_{ON} - f_{OFF}))$$

Parallel application power usage (applying amdahl's law)

Parallel processing researchers and developers use Amdahl's law to determine the theoretical limit to the performance an application can experience when parallelized. Amdahl's law states that the speedup of an application is the ratio of sequential to parallel execution time [40]. The goal of our model is to gage the improvement of a parallel application's energy usage compared to its sequential equivalent when ran on the same number of processors. To do this, we applied Amdahl's law to the CPU energy equations we derived. We look at the ratio between sequential and parallel energy usage on N processors, i.e. *ERN*. This ratio gives us

$$ERN = \frac{CV^2(Nf_{OFF}(t_{ON}+t_{OFF})+t_{ON}(f_{ON}-f_{OFF}))}{\sum_{i=0}^{N} CV^2(f_{ON}t_{iON}+f_{OFF}t_{iOFF})}$$

Since CV^2 is constant,

$$ERN = \frac{(Nf_{OFF}(t_{ON}+t_{OFF})+t_{ON}(f_{ON}-f_{OFF}))}{\sum_{i=0}^{N}(f_{ON}t_{iON}+f_{OFF}t_{iOFF})}$$

An Example

An example of how to apply this model would be to see the energy savings that would be gained by parallelizing an application on a quad-core processor. For this example the processors have an active frequency of 2.5 GHz and an idle frequency of 1 GHz. Assuming that the sequential version takes 2 minute to execute, where for 1.5 minutes the processor is active and the remaining 30 seconds it is inactive. For the parallelized version, it uses all 4 processors for 30 seconds and is idle for 15 seconds per processor. This gives us an *ERN* of 1.71 meaning that the sequential version of the application uses over 1 ½ times the energy as the parallel version. The Amdahl ratio for this application is 2.67, which means when we compare the ERN_S, we see that the energy savings are being out paced by the performance gains of parallelization.

CASE STUDIES

We now present two different task scheduling case studies and a set of task graph schedulers to compare the standard speed up ratio to the parallel energy efficiency ratio. The reason we look at both is to gage how well the improvement in performance translates to energy consumption reduction. We believe that just because a particular scheduling algorithm and parallelization method performs better that does not necessarily mean its performance gains outweigh the extra energy costs to do so.

Subject applications

We chose these particular applications to present as examples of how to apply the energy efficiency ratio due to the different parallelization issues they contain. The first application is an exercise that has been used to demonstrate how a cyclic scheduler works [34], but does not actually do anything useful. The other application is a chemistry application that calculates the total force between a set of molecules.

Cyclic loop: Below is the code that is used to demonstrate cyclic dependencies and how different task schedulers can schedule their tasks [34]. The cyclic dependencies give us the ability to look at schedulers designed to handle this special type of problem and how well they perform.

```
for(k = 0; k < N; k++)
{
A: a[k] = c[k-1];
B: b[k] = a[k-2] * d[k-1];
C: c[k] = b[k] + 1;
D: d[k] = f[k-1] / 3;
E: e[k] = sin(f[k-2]);
F: f[k] = log(b[k] + e[k]);
}
```

As we examine this code segment, we can see that there are both inter- and intra-iteration dependencies. For example, one of the inter-iteration dependencies is between tasks A and C. The operation A writes the value of *c[k-1]* to *a[k]*, and so therefore C must precede *A+1*. Similar dependencies exist throughout the code segment. Even though there are some schedulers that will try to schedule the task across multiple iterations, we have to start the loop at *k=2* otherwise we will end up with negative array indices.

From this code segment we construct the reduced dependency graph as show in Figure 5. The values in boxes near each task label are the duration or weight of the generic tasks. The other numbers near the edges are the weight of each edge. Looking at Figure 5 one can clearly see that the dependencies form three separate cycles: A *A→B→C→A*, *B→F→D→B*, and *E→F→E*. E We can also see that the only intra-iteration dependencies are *B→C* , *B→F* and *E→F*.

Force calculator: the force calculating example is a simple function that contains a set of nested FOR loops iterating across an array. The inner FOR loop performs some simple mathematical calculations on a 1-D array, iterating up to the current value of the outer loop's index. The calculation within the inner loop starts with the difference between the elements pointed to by the indexes of the two loops and performs some calculations which are then subtracted from the element pointed to by the inner loop's index and added to the outer loops indexed value.

With the Force Calculation function, the outer loop appears to be a perfect candidate for parallelization. Looking closer however reveals that there are dependencies on values that would be calculated in prior iterations of the loop. As part of the process of parallelizing the loop, we will have to find a way to eliminate this dependency. To keep our example simple, we focused our parallelization efforts to the inner loop instead.

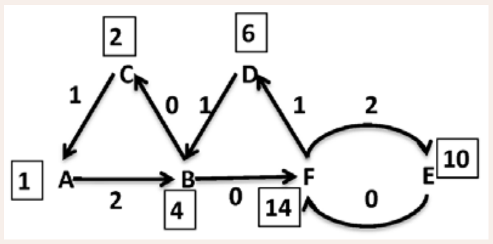

Figure 5 Reduced dependency graph of sample code segment [34].

To parallelize the inner loop we had to take into account the set of calculations that is added to the element indexed by the outer loop. Summation parallelizations are a trivial matter and required only a simple reduction to complete. The ease by which we could tile the inner loop across multiple threads provided us with a simple method for parallelizing this function. Figure 6 shows the resulting task graph of parallelizing the inner loop and reducing afterwards as part of the outer loop. Index *i* is the outer loop's index, while *j* is used for the inner loop with the calc nodes being the calculations done inside. The sum nodes represent the summation reduction that is done at the end of each iteration of the outer loop.

Task schedulers

The task schedulers we implemented were inherited a central scheduler class. Each of the scheduling algorithms we used in our schedulers use Graham's idea of mapping tasks to a reduced dependency task graph. Each task is included as an instance of the Task class, and it is the instances of this class that the scheduler uses to generate the schedule for the order the tasks will run. The instances of the Task class contain their task's duration or weight in addition to pointers to both their dependent tasks and their predecessor tasks. This allows them to be able to calculate their critical path and their dependent path values. This also provides them with the means to insure that their dependent tasks have been executed before they are.

The scheduler classes contain a list of all the tasks that it must schedule and a list of all the available threads. Each of the thread objects maintain a queue that contains each scheduled task and the time at which it is to start. As each task is scheduled, the scheduler will add it and its start time to the queue for the thread it is to be executed on. In most cases, the start time of each task is equal to the maximum schedule time $\sigma(v,k)$, and duration, d(v) of the intra-iteration dependencies, or those dependencies that have zero weight edges. If a task is to be queued up on a thread that is not yet ready, determined by the start time of the task, the scheduler then places it in the queue starting immediately after the previous task in that thread's queue. Schedule time, σ, of each task is:

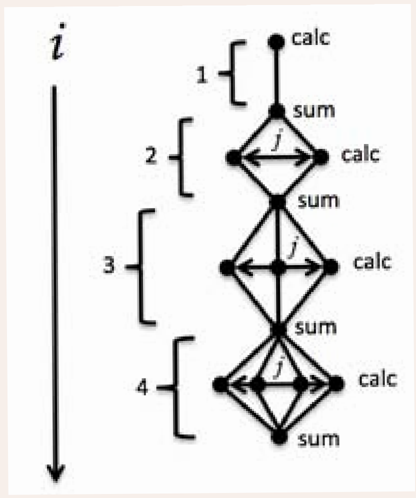

Figure 6 Force calculation task graph.

$$\sigma(v) = max\{P(t), \sigma(u,k) + d(u) \mid u \in D, 0 \le k \le N\}$$

Where *D.* is the list of tasks *v* is dependent on and P(t) is the current scheduled time of the selected thread. Each of the different scheduling techniques we used utilized different methods for deciding what order each of the tasks was scheduled and which of the dependencies were included in *D.*

Since the weight of each task is an estimate, the scheduler needs to keep track of the estimated time that the application is at. It does this by having each task signal when it completes. The scheduler adds the start time and duration of the task to get the task's completion time. If the task's completion time is greater than its current estimated time, it signals all the threads with the new estimated time.

As mentioned earlier, all of the threads maintain a queue of tasks it is to execute and the start times for each of those tasks. Each thread is comprised of a WHILE loop, with the exit condition set by the Thread object's Join method. Inside the loop, the thread waits until there are tasks in its execution queue and for the signal that the estimated application time is greater than its current task's start time. Before each task is executed by the thread, it is removed from the thread's queue. Once the queue is empty, the thread waits for additional tasks to be added or the signal that it is done and ready to rejoin the main thread.

Each scheduler we implemented inherits the basic methods for managing task and thread objects. The different schedulers are then only responsible for the actual scheduling of the tasks. The base scheduler object assumes that the task graph provided is a reduced dependency task graph and that it will be executed a set number of times, so it accepts as part of its constructor the number of iterations the tasks will be executed. It is then left up to the actual scheduling implementations to actually use this value or disregard it and assume that the task graph is an acyclical graph.

We developed four different task schedulers. Each of them is unique enough to provide a good look at the information provided by the energy model. We called the schedulers the Critical Path, Bottom-Up, First In/First Out, and the Bellman-Ford.

We previously mentioned that when there exists a cycle within a task graph, it introduces additional complexities that scheduling algorithms must deal with. One method that we used in all but the Bellman-Ford scheduler is to disregard the edges of G where $w(e) \neq 0$, this removes the inter-iteration dependencies and allows us to use acyclical scheduling algorithms within the body of the loop.

Critical path scheduler: The first method we used for ordering the tasks was by their length of each task's critical path. Since the critical path is defined as the longest path from each task to the end or stopping node, this method is classified as an acyclical scheduling method. This means that in order for this to work on a cyclical task graph, all non-zero edges are ignored. The critical path of each task is lazy loaded using a recursive depth-first search of the dependent tasks. Figure 7 shows a portion of the schedule that the critical path scheduler generated for the cyclic loop application.

The critical path scheduling method has two advantages:

simplicity and a guarantee of obeying dependencies. The simplicity comes from the fact that tree traversal algorithms are well known and easy to understand. The second advantage of dependency guarantee comes from the fact that by sorting by critical path means that since the tasks that need to be executed first will be, as they will have the largest critical path.

Bottom up scheduler: The next scheduler that we implemented organizes the tasks by looking at the path from each task back to the top of the task graph: what we call the dependent path. This is the opposite of the critical path. The fact that looking from the bottom up means that the possibility exists that the generated schedule will do so in a manner that does not maintain the dependencies between the tasks.

To make sure that the dependencies between tasks are kept, the Bottom Up scheduler schedules the tasks recursively. The order the scheduler looks at each task for scheduling is by the value of the dependent path. Before each task is scheduled, its dependent tasks are scheduled, also by their dependent path. Each task is scheduled this way until all the dependent tasks are scheduled, and then the depending tasks are. Figure 8 shows the schedule that the bottom-up scheduler generated for the cyclic loop, which for this particular problem is identical to the critical path. This is due to the fact that both schedule by sorting the longest paths.

First come scheduler: The least complex of all the schedulers is a first in first out scheduler. It assigns each task to a thread in the order it was added to the scheduler. It does not try to rearrange the tasks or do anything other than simply put each task on the next available thread. By scheduling the tasks in the order they were added to the scheduler, the first come scheduler gives all the control over the execution order to the developer. Figure 9 shows a portion of the resulting schedule that the first come scheduler generated.

Bellman-ford scheduler: The Bellman-Ford scheduler is by far the most complex, but it promises to produce a schedule with a makespan closest to the theoretical minimum. Unlike the other algorithms, the Bellman-Ford algorithm schedules the tasks across loop iteration.

The Bellman-Ford method works by first identifying the cycles within a potential graph and then using that information to calculate the values necessary for generating the schedule. To find the cycles within the task graph, we use Johnson's algorithm [41] for finding all of the elementary cycles within a graph [41]. Once the cycles have been identified, each cycle's duration to distance ratio, i.e. ρ(C), is calculated. The scheduler then uses ρ(C), to adjust the weights of the edges in the potential graph. Finally, the scheduler uses the Bellman-Ford method to calculate the longest path from that starting node to each task on the graph.

The Bellman-Ford algorithm normally is used to find the shortest path, so using it to find the longest path is a slight deviation that requires flipping the distance check from less than to greater than. The length of each of these paths is then used to determine when to start each task. The Bellman-Ford algorithm works best when there are an unlimited number of processors, but as this is unlikely to happen, the scheduler uses the next free processor if there are no free processors when a task is to be started. Figure 10 shows the portion of schedules that the Bellman-ford scheduler generates.

Applying the energy model: We now present how we applied our Energy Ratio model to the case studies. We look at both Amdahl's ratio and our own to see how much energy each application saves when parallelized. We also examine the energy

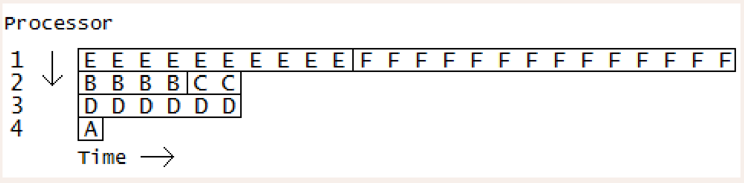

Figure 7 Critical path schedule.

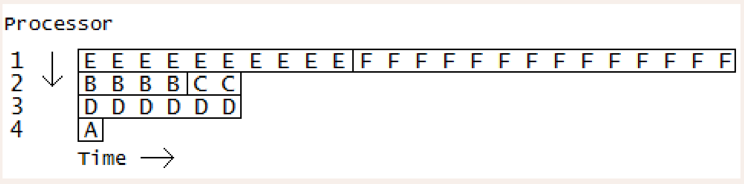

Figure 8 Bottom-up schedule.

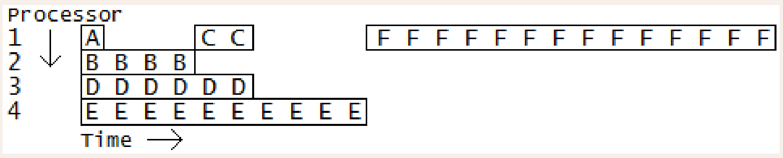

Figure 9 First come schedule.

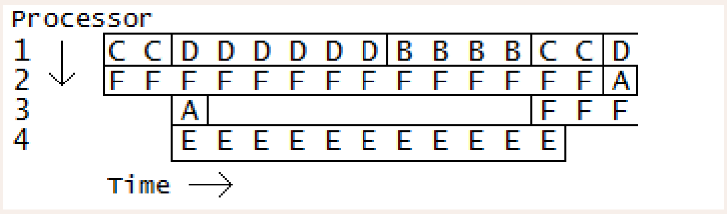

Figure 10 Bellman-Ford schedule.

savings each scheduler offers and if their performance gain is justified. Using this information we make a decision as to how best to parallelize each application to both balance performance with energy savings.

As *ERN* is based off of Amdahl's law, its values change much in the same way as those for Amdahl's. Increasing the energy used by the sequential version of an application, or reducing the energy used by the parallel version will increase the value of *ERN.* In addition, like Amdahl's law, the number of processors is the maximum value of *ERN*

Our testing setup was a laptop computer with an Intel Core I3 processor. Like all ACPI compliant processors, this processor has several different operating frequencies. We used Power Top [42], which reads the ACPI information from the kernel and outputs it into a human readable format, to identify the different operating frequencies. For this processor the active frequency was identified as 2.97 GHz and the lowest idle frequency as 913 MHz.

Cyclic loop: We ran the schedules and estimated the power usage for cases when two and four processors are utilized. We stopped at four processors due to the fact that both scheduling algorithms did not scale past four processors. Table 1 shows the CPU utilization each algorithm used when running the application on two and four processors. From looking at this table, we can assume that for this example the Bellman-Ford schedule will use less energy due to its higher CPU utilization.

Table 2 reports the energy efficiency ratios of two algorithms when compared to the sequential case. By looking at Figure 2 it would appear that we are getting poorer results by attempting to run each schedule on two processors as opposed to four. Actually considering that the theoretical maximum efficiency equals the number of processors we run the application on; running our application on two processors gets us closer to this value. Table 2 shows us that when we run the application on four processors we are not even able to achieve half of the theoretical maximum, which like Amdahl's law is equal to the number of processors available. While at the same time, when ran on two processors we are getting between 60% and 70% of this maximum. The closer we get to the maximum possible efficiency ratio, the better our energy usage will be.

It becomes apparent when looking at these three tables that overall CPU utilization plays a major part in how well an application performs and how much energy it uses. When this application was run on four processors each scheduler, except for the Bellman-Ford, used the processors a different amount but had the same average CPU utilization. This in turn resulted in having identical performance ratios. Clearly this demonstrates that when an application is running it will use less energy and perform better if the application fully utilizes the CPU cycles made available to it.

One of the points that Steigerwald et al make is that the energy efficiency of a computer reaches its maximum as the work load approaches 100% device utilization [25]. The reason for this is that even though a CPU or other part of the computer might be idle, it still uses energy. Keeping an idle device powered up is done so that it remains responsive and this in turn reduces latency. Also, if a device enters and exits an idle state repeatedly, powering it up and down not only increases the total energy usage but also can cause additional wear. Even still, like lights lighting empty rooms, powered up idle devices simply waste energy. It is this reason that the ACPI dictates that the longer a device remains idle, the lower the power state it enters.

Force calculator: The nature of the force calculation application presented some very interesting results with the schedulers used. Two aspects of the task graph we generated caused the ratio values to not vary between the different schedulers. First is the fact that there are no cycles in the graph, this means that there is no special cyclic handling necessary. Then the summation nodes limit how the application can be parallelized to only parallelizing the iterations of the inner loop.

With the ratio values not varying between the different schedulers our choice of scheduler will depend on how much overhead each adds to the application. Since we are looking at energy usage and execution time of the application, those are our biggest considerations. We might also rank our schedulers by memory usage, complexity or some other metric. The schedulers weren't parallelized; therefore their energy usage directly

Table 1: Cyclic loop CPU utilization.

Algorithm	Four Processors					Two Processors		
	1	2	3	4	Avg.	1	2	Avg.
Bellman-Ford	91.67%	63.64%	56.82%	68.18%	70.08%	97.38%	96.34%	96.86%
Bottom-Up	100.00%	25.00%	25.00%	4.17%	38.54%	100.00%	54.17%	77.09%
Critical Path	100.00%	25.00%	25.00%	4.17%	38.54%	100.00%	48.00%	74.00%
First Come	70.83%	16.67%	25.00%	41.67%	38.54%	100.00%	54.17%	77.09%

correlates to the amount of time they add to the application's run time.

Table 3 gives the time each scheduler took to generate the schedule for the force calculation application. Besides being the most complex and adding the most overhead via code and memory usage, the Bellman-Ford took two orders of magnitudes longer to generate its schedule. Its run time alone makes it a poor candidate for parallelizing an application like this. On the other hand, the simplest scheduler took the least amount of time to generate its schedule, making it the best choice of the four schedulers.

The way we setup the task graph for the force calculation application means that the application parallelizes across an almost unlimited number of processors. Being able to parallel across a large number of processors allowed us to look at this application's performance and energy usage as would be seen inside a data or high performance computing center. We are also able to gain a greater insight into the effects of parallelizing across a large number of processors. Figure 11 shows the results for the two ratios as we ran the force up to 32 processors. When we examine Figure 11 it become clear that after a certain number of threads there is no longer any performance gains to be achieved by adding more threads. Like not being able to parallelize the cyclic loop application past four processors, adding more processors past this point only wastes those resources as the application is not able to fully utilize them. Additionally, our energy efficiency ratio moves further and further away from the maximum the more processors that are made available to the application.

Looking at these results we can determine that after about 16 processors we are no longer able to achieve any higher performance gains and should not need more than that to run this application. Another observation we get from Figure 11 is that if we can sacrifice performance, then our energy conservation will be better when we run this application on fewer processors.

Further observations

After we completed our work with the two applications we continued to look at the behavior of the energy ratio. We were particularly interested in two extremes. The first we looked at

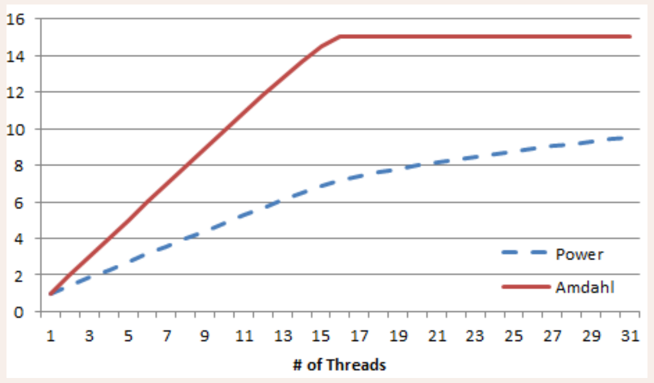

Figure 11 Parallelization ratios for the force calculation application.

is what happens when a device has the worst possible power management setup, or when the device does not have an idle power state to go to. We also looked at how our energy ratio plays out when a device simply is turned off when it becomes idle. Both of these situations provide insight into how we should be designing our computational devices in the future.

No idle state: Our energy efficiency ratio has an interesting characteristic when $f_{oN} = f_{off}$ the equation reduces to Amdahl's speed up ratio, mentioned earlier. By ERN being equal to the speedup ration, it tells us that if the frequency of the idle processors does not change, then the power improvements are completely dependent on being able to complete the tasks as quickly as possible.

With modern operating systems and processors, this should not usually be the case, unless the power management settings are set in such a way that the processors never go into an idle state. Even if the processors are set to never go into an idle state, then like what was stated earlier we want to complete the application run as quickly as possible. This still does not change that developers need to be optimizing their applications in order to reduce the energy usage.

Turning off idle processors: The other scenario we looked at is when the idle processors are turned off. When this happens f_{off} is zero, and the ratio becomes:

$$ERN = \frac{(t_{ON}\, f_{ON})}{\sum_{i=0}^{N}(f_{ON} t_{iON})}$$

Now if we assume that the total time it takes to execute all the tasks remains the same, then $(t_{ON}) = \sum_{i=0}^{N}(t_{iON})$ and E_N=1 Or in other words if the idle processors are completely turned off, then the power consumption of an application does not change between the parallel version and its sequential equivalent as long as the total execution time on all processors remains constant.

By having no improvement in the power consumption when parallelizing tells us that when the run time of the application remains fixed, the CPU utilization does not matter. At this point it becomes no longer necessary to add additional processors to the application solution. This means that by reducing the number of processors to the bare minimum necessary to complete the tasks on time, we can increase the CPU utilization for the duration of the application run, and this becomes the best energy usage

Table 2: Cyclic loop performance ratios.

Algorithm	Four Processors		Two Processors	
	ERN	Amdahl	ERN	Amdahl
Bellman-Ford	1.930	2.803	1.408	1.937
Bottom-Up	1.357	1.542	1.266	1.542
Critical Path	1.357	1.542	1.241	1.480
First Come	1.357	1.542	1.266	1.542

Table 3: Scheduler's run times.

Algorithm	Run Time(s)
Bellman-Ford	280.198
Bottom-Up	9.155
Critical Path	1.564
First Come	1.507

solution.

One example of where designers are placing too many CPUs in devices is the newer model of smart cell phones. Cell phone manufacturers have begun to increase the number of cores in a cell phone's CPU, much like those in traditional computers and servers. Unfortunately, applications on cell phones are fixed duration applications, so it does not matter how long an application takes to run. At the same time, most cell phones do not allow for multitasking, and since the applications are designed to run with limited resources the benefit to having additional CPUs is fairly limited.

CONCLUSION

As computer technology has progressed and become more inexpensive and mobile so has the need for energy conscious designs increased. Most of the done towards making devices use less energy is centered around improving the hardware, but software can and does play a major role in how much energy a device uses. If software developers are not trying to keep their applications optimized, then ultimately they are contributing to the excess usage of energy on the devices their programs run on.

One of the ways in which hardware has improved over the past few years is the addition of more than on processing core in CPUs. Multicore processors have the potential for both increasing the performance of an application and decrease its energy usage. Knowing the best method for parallelizing an application can be treacherous, as different parallelization methods are not guaranteed to give the performance desired and if care is not given can cause unanticipated errors. It is this reason we presented a ratio that software developers can use to gage if the performance gains achieved through the various parallelization methods justify the energy costs and increased application complexity.

Using the data from both Amdahl's ratio and our own energy efficiency ratio clearly shows that the greater the CPU utilization, the greater the energy efficiency an application will have. Even if the CPU were to use the same amount of energy when executing a parallelized version of the application as its sequential version, the less time it takes then the entire computer system will expend less energy overall.

By simply reducing the time that the application takes to execute, the sooner the CPU will return to an idle state. This "race to idle" allows a computer to return to the more energy efficient state sooner and thereby save energy [26]. At the same time, the more CPUs that are in use, then the more power will be used. Using more CPUs would lead one to believe that the system would be using more energy. On the contrary, by utilizing more of the processors an application will be more likely to finish quickly and leave fewer processors idle wasting energy.

Future work

This energy efficiency ratio presents a lot of good information for developers to use in making the decision to parallelize their applications. This does not mean that there are not ways in which it could be improved. A lot was left out in order to keep thing simple and easy to use or to keep it as independent of the hardware as possible.

One thing that was left out of the energy efficiency ratio is the relationship between the frequency of the processor's clock and the voltage the processor needs to maintain stability. Also the energy efficiency ratio does not take into account the energy usage of anything other than the CPU. The CPU actually account for only a small percentage of the overall energy used by a computer. Other components, such as the power supply and video adapter, use extremely more power than the CPU. In fact, for mobile devices the screens use the most energy, and users are advised to keep their screens off as much as possible.

REFERENCES

1. Moore GE. Cramming more components onto integrated circuits. Electronics. 1965; 114-117.
2. Rasberry Pi Foundation. August 2012. [Online].
3. Basheda G, Chupka MW, Fox-Penner P, Pfeifenberger JP, Schumacher A. Why Are Energy Prices Increasing? 2006.
4. Glanz J. Google Details, and Defends, Its Use of Electricity. New York Times. 2011.
5. Cauchon D. Household electricity bills skyrocket. USA Today. 2011.
6. Google Inc. Google Announces Fourth Quarter and Fiscal Year 2011 Results. 2011.
7. Murugesan S. Harnessing Green IT: Principles and Practices. IEEE IT Professional. 2008; 24- 33.
8. Energy Star. 2011. [Online].
9. Energy Star. Energy Star and Other Climate Protection Partnerships 2010 Annual Report. Washington DC. 2010.
10. Hewlett-Packard, Intel, Microsoft, Pheonix Technologies, Toshiba. Advanced Configuration and Power Interface Specification Rev 4.0a. 2010.
11. Lake L, Pease L. Energy Crisis in California. Pepperdine School of Public Policy. 2001.
12. Amin SM, Wollenberg BF. Toward a smart grid: power delivery for the 21st century. IEEE Power & Energy. 2005; 3: 34-41.
13. Brooks D, Tiwari V, Martonosi M. Wattch: a framework for architectural-level power analysis and optimizations. ACM SIGARCH Computer Architecture News. 2000; 28: 83-94
14. Overa A. Ubuntu 11.04 (Natty Narwhal), Reviewed In Depth. 2011. [Online].
15. Larabel M. The Leading Cause Of The Recent Linux Kernel Power Problems. Phoronix. 2011. [Online].
16. Larabel M. A Proper Solution To The Linux ASPM Problem. Phoronix. 2011. [Online].
17. Cho S, Melhem RG. Corollaries to Amdahl's Law for Energy. IEEE Computer Architecture Letters. 2008; 25-28.
18. Yao F, Demers A, Shenker S. A Scheduling Model for Reduced CPU Energy. 36th Annual Symposium on Foundations of Computer Science. Milwaukee, Wisconsin, USA. 1995.
19. King D, Ahmad I, Sheikh HF. Methods for optimizing the performance of directed acyclic graphs operating. Sustainable Computing: Informatics and System. 2011; 1: 99-112.
20. Rountree B, Lowenthal DK, Schulz M, de Supinski BR. Practical performance prediction under Dynamic Voltage Frequency Scaling. Green Computing Conference and Workshops (IGCC). Livermore, CA, USA. 2011.

21. Ge R, Cameron KW. Power-Aware Speedup. IEEE Parallel and Distributed Processing Symposium. 2007.

22. Ge R, Feng X, Feng W-C, Cameron KW. CPU MISER: A Performance-Directed, Run-Time System for Power-Aware Clusters. International Conference on Parallel Processing. 2007.

23. Song S, Su C-Y, Ge R, Vishnu A, Cameron KW. Iso-Energy-Efficiency: An Approach to Power-Constrained Parallel Computation. IEEE International Parallel & Distributed Processing Symposium (IPDPS). 2011; 128-139.

24. Steigerwald B, Lucero CD, Akella C. Agrawal AR. Energy Aware Computing. Intel. 2012.

25. Steigerwald B, Lucero CD, Akella C. Agrawal AR. Impact of Software on Energy Consumption. 2011.

26. Steigerwald B, Lucero CD, Akella C. Agrawal AR. Writing Energy-Efficient Software. 2011.

27. Naik K, Wei DSL. Software implementation strategies for power-conscious systems. Mobile Networks and Applications. *2001;* 6: 291-305.

28. M. Wolfe, "More Iteration Space Tiling," in *ACM/IEEE conference on Supercomputing* , New York, NY, USA, 1989.

29. Deborah T. Marr, Frank. B, David LH, Glenn Hinton, David A. Koufaty, et al. Hyper-Threading Technology Architecture and Microarchitecture. Intel Technology. 2002; 6: 4-15.

30. C. Poppeliers. Estimating vertical stochastic scale parameters from seismic reflection data: deconvolution with non-white reflectivity. Geophysical Journal International. 2007; 68: 769-778.

31. C. Poppeliers, "Estimation of Vertical Continuous Stochastic Parameters from Seismic Reflection Data," Mathematical Geosciences. 2009; 417: 761-777.

32. C. Poppeliers and A. Levander. Estimation of vertical stochastic scale parameters in the Earth's crystalline crust from seismic reflection data. Geophysical Research Letters. 2004; 31.

33. Graham RL. Bounds for Certain Multiprocessing Anomalies. The Bell System Technical Journal. 1966; 9: 1563-1581. Darte Y. Robert, Vivien F, Birkhauser . Scheduling and Automatic Parallelization, Boston. 2002; 5: 99-101.

34. M. Gondran and M. Minoux, Graphs and Algorithms, John Wiley & Sons, 1984.

35. Merke A, Bellosa F. Balancing power consumption in multiprocessor systems. Proceedings of the First ACM SIGOPS EuroSys. 2006; 4: 18-21.

36. Tiwari V, Malik S, Wolfe A, Lee MT C. Instruction Level Power Analysis and Optimization of Software. Ninth International Conference on VLSI Design. 1996; 1-18.

37. Serway RA, Beichner RJ. Physics for Scientists and Engineers with Modern Physics. Orlando, FL: Saunders College Publishing. 2000.

38. Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor. Intel. 2004.

39. Amdahl GM. Validity of the single processor approach to achieving large scale. AFIPS spring joint computer conference. 1967.

40. Johnson DB. Finding All the Elementary Circuits of a Directed Graph. Society for Industrial and Applied Mathematics Journal on Computing. 1975; 4: 77-84.

41. Rawshdeh T Do, S. Shi W. pTop: A Process-level Power Profiling Tool. Proceedings of the Workshop on Power Aware Computing and Systems. 2009.

Cite this article

Bonner D, Namin AS (2014) Measuring the Energy Efficiency Ratio of Parallelized Software Applications. Comput Sci Eng 1(1): 1002.