Avoiding Energy Wastage in Parallel Applications

Vijay Anand Korthikanti
Department of Computer Science
University of Illinois at Urbana Champaign
vkortho2@illinois.edu

Gul Agha
Department of Computer Science
University of Illinois at Urbana Champaign
agha@illinois.edu

***Abstract*—We propose a methodology to analyze algorithms in order to reduce energy waste in executing applications. Our methodology is based on three observations. First, the relation between power and frequency of a single core is approximately cubic. Thus it may be possible to run an application slower on a core in order to save energy. In the case of a parallel architecture, one has to also factor the effect (on performance and energy consumption) of the interaction between cores. Second, multicore architectures which aggressively manage power consumption by allowing cores to be operated at reduced frequencies are being developed. This means that parallel applications on a multicore architecture can be executed using a variable number of cores running at different frequencies–affecting both the performance of the application and the energy required to execute it. Lastly, there is a certain benefit (positive *utility*) in running an application faster and a cost (negative utility) in terms of the energy consumed. Expending energy that does not contribute to the overall utility wastes the energy. The precise trade-off between performance and energy consumption depends on the structure of a parallel algorithm and the associated utilities. We describe a methodology to do this trade-off and illustrate it with several parallel algorithms.**

***Keywords*-Utitlity; Energy; Performance; frequency;**

I. Introduction

A significant amount of energy is consumed by computers: by one estimate, in the United States alone computers consume 13% of all electricity [1]. It is should also be noted that other estimates have come up with a lower figure, all estimates are dated, and there is no consensus on the the numbers or even how to measure them. Nevertheless, it is clear that energy consumption by computers incurs both economic and environmental costs, contributing to an ever increasing degree to global warming (e.g. see [2]). In a series of papers, we have been studying how *performance* (inverse of time taken to execute an application) can be traded for *energy savings* in executing an application [3], [4].

The key idea is based on the following observations. Roughly speaking, power is approximately proportional to the cube of the speed (*frequency*) at which a core operates [5]. Recall that power is the rate at which energy is expended. This means that the energy E consumed by an algorithm (on a single core) is approximately proportional to the square of frequency at which it is executed. In other words, for a sequential algorithm, $E.T^2$ is constant with respect to varying frequency(where T represents the time taken by the algorithm to execute). Thus we can save energy by reducing the frequency at which a core operates. In case of an *embarassingly* parallel algorithm, the relation between frequency of cores and energy consumption is similar to that of the sequential algorithms. In this case running $2n$ cores at frequency f, for example, would take approximately a quarter the energy of n cores running at frequency $2f$, while providing the same performance.

In general, the relation between energy and performance of a parallel application is more complicated. This is because parallel algorithms typically involve interaction between different actors, and communication (or acess to shared memory) takes both time and energy. Moreover, the precise relation of performance and energy to the number of cores used, and the frequency at which those cores operate, can be complicated. For example, for certain parallel algorithms and architectures, communication delays can be masked by overlapping communication and computation. This can dramatically improve performance [6], [7], but the energy required for communication would be unaffected by such overlapped execution.

We associate a *utility* function with the performance of an application. Specifically, we assume utility is a monotonically increasing function of performance (time taken), and a monotonically decreasing function of the amount of energy used in executing the application. This suggests that we can maximize utility by trading performance for energy. The increased energy that is consumed by deploying more cores or increasing the frequency at which they operate, without a corresponding increase in the utility associated with improved performance of the application, is energy that is *wasted*.

For the analysis in this paper, we will use a cost function that is the inverse of the the utility associated with an execution. Thus our objective is to minimize this cost of executing parallel algorithms may be minimized by changing the number of cores and their frequency. In order to make our analysis concrete, we assume that cost is a linear function of the energy consumed by an application. Our simplifying assumption is quite reasonable. Energy is often linearly priced per unit (if environmental costs were assessed through a Carbon tax, that too would be factored in the price of the energy).

Moreover, we assume that cost is also a linear function of the time taken in executing an application. This assumption is quite reasonable in some contexts. If applications are run on a commercial supercomputer or in the cloud, what one pays a vendor is roughly linear with the time used. However, the assumption of linear cost of execution time is restrictive and may not hold in other contexts. For example, in case of

a laptop with hundreds of cores, user response time will not have a linear cost function. Concave utility functions would provide a more general model. Our motivation in restricting the analysis to linear utility (cost) functions is to keep the analysis simple.

An analysis of performance or energy consumption requires fixing a particular computational model. It is reasonable to assume that as the number of cores increase, multicore architectures are likely to be based on message-passing [8]. We therefore choose a message-passing model of parallel computating for our analysis. We make some further assumptions about the associated energy model. Specifically, we assume that all cores are homogeneous. This assumption keeps the analysis simpler and allows us to focus on the effect of the parallel algorithm. We also assume that cores that are idle consume minimal power, something that is already true in many architectures. We do not concern ourselves with a memory hierarchy but assume that local memory accesses are part of the time taken by an instruction. We assume that the communication time between cores is constant. This assumption is unlikely to be true with very large numbers of cores, but is reasonable as long as the time taken for sending and receiving a message is much greater than the time taken *en route* between the cores. We discuss ideas for possible extensions in Sec. VII.

Note that the architectural model we use in this paper is similar to the one we used earlier in [3], except that in the present paper we also consider energy leakage when a core is running at a lower frequency. In our earlier work [3], we fixed a performance requirement and analyzed parallel algorithms to minimize energy consumption given the performance requirement. The present paper shares the methodology for framing an energy equation with this earlier paper. However, the objective in this paper is far more general. Thus previous work can be seen as a special case of the work presented in this paper. Specifically, in the earlier work, one can think of the utility function used as step function of execution time: its value is some constant until some performance threshold and then 0 afterwards.

Note that this paper shares some of the text with the previous one: for completeness, we have included a description of the model and some of the details describing three of the algorithms that were also analyzed in earlier paper, as well as some of the relevant related work. The description of the *minimun spanning tree* algorithm is entirely new as this algorithm was not studied in [3]. Moreover, the analyses in this paper are original, as they consider linear utilities (including energy costs) rather than what amounts to step function utilities in earlier work.

Our analysis is sensitive to two ratios:

1) The ratio of energy required to execute an instruction and the energy required to send a message.
2) The ratio of the cost associated with single unit of energy to the cost associated with using the parallel system for a unit time.

In order to understand the energy behavior of parallel algorithms on a range of practical architectures, we study the optimal configurations as a function of a range of values of the two ratios.

Contributions of the paper: This paper is the first to propose a methodology to propose saving energy by considering the utility gain as a result of trading off performance and energy consumption as the number of cores used and the frequency at which they operate are varied in the execution of a parallel algorithm. We illustrate our methodology by analyzing different types of algorithms, ranging from algorithms that are almost embarrassingly parallel to those which have a strong sequential component. Specifically, we analyze *tree addition*, *LU factorization*, and two parallel *sorting* algorithms.

Outline of the paper: The next section discusses related work. Sec. III provides the background for our analysis, justifications for our assumptions and description of constants used in the analysis. Sec. IV explains our methodology to reason about the maximal utility configurations for parallel algorithms by means of an example. Sec. VI applies our methodology to other parallel algorithms–namely, Naive quicksort algorithm (inefficient parallel algorithm), parallel quicksort algorithm and LU factorization. Finally, Sec. VII discusses the results and future work.

II. Related Work

Several researcher have proposed the use of software-controlled *dynamic power management* in multicore processors. These dynamic power management techniques are based on using one or both of two *control knobs* for runtime power performance adaptation: namely, *dynamic concurrency throttling* to adapt the level of concurrency at runtime, and *dynamic voltage and frequency scaling* [9]–[13] to change power usage. The work on dynamic power management is useful for creating runtime tools which may be used in conjunction with profilers for the code. How accurate such tools can be remains to be seen. By contrast, we develop methods for theoretical analyzing parallel algorithms. Such an analysis can statically determine how to minimize the costs associated with the execution of a parallel application. One advantage is that one may be able to choose the right algorithm and resources *a priori*. Another advantage of our approach is that it can provide greater insight into the design of algorithms for energy conservation.

Bingham and Greenstreet have proposed a generic energy complexity metric, ET^α, for representing energy-time tradeoffs of CMOS technology [14]. Prior to this, various researchers had promoted the use of the ET [15] and ET^2 [16] metrics for modeling energy complexity. These models try to hide voltage/frequency scaling from the programmer, while enabling reasoning about the overall energy complexity of the computation. On the other hand, we use the metric $\alpha E + T$, to map the cost (utility) of running a parallel algorithm as a function of the number of cores and the frequency at which they operate, and view both concurrency throttling and voltage/frequency scaling as two orthogonal control knobs to change energy and execution time in order to maximize

the overall utility. Moreover, although energy for sending messages forms a significant proportion of the total energy consumed, the earlier generic complexity metrics does not account for the energy required for message passing. We consider both the energy required for message passing, as well as the energy lost to leakage, during the execution of a a parallel algorithm.

Some researchers have already considered the metric $\alpha E + T$ (termed as 'flow plus energy') [17], [18]. However, these researchers have studied a different problem: given a set of *independent processes* and processors, how can we develop a policy to assign processes to processors and scale the speed of the processors so that the assignment will be satisfy the objective of optimizing 'flow plus energy'. However, we analyze a single parallel algorithm and explicitly consider its potentially complex messaging structure to compute 'flow plus energy' of actors which may interact with each other.

In our earlier work, we have studied the relation between energy and performance for parallel algorithms. The goal of that work was to optimize the performance given an energy budget [19], or to optimize energy given a performance requirement [3], [4]. In this paper, we build on our prior work by considering the utility of performance and energy. This provides a general framework for cost-conscious energy conservation.

III. Problem Definition and Assumptions

In utility theory, the utility function of an agent is a function that ranks all pairs of consumption bundles by order of preference (completeness) such that any set of three or more bundles forms a transitive relation. This means that for each bundle (x, y) there is a unique relation, $U(x, y)$, representing the utility (satisfaction) relation associated with the bundle (x, y). The relation $(x, y) \rightarrow U(x, y)$ is called the utility function. The range of the function is a set of real numbers. The actual values of the function have no importance. Only the ranking of those values is significant in the theory.

We use a utility function based on the *cost* associated with running a parallel algorithm. In our case, the consumption bundle is the pair consisting of the number of cores used, P, and the frequency at which these cores operate, X. Formally, for a parallel algorithm, a utility function $U_{cost}(P, X)$ is defined as negative cost, and cost C is defined as follows:

$$C(P, X) = a \cdot E(P, X) + b \cdot T(P, X) \tag{1}$$

where $E(P, X)$ and $T(P, X)$ represent, respectively, the energy consumed by the parallel algorithm and the time taken by it, a denotes the cost associated with consuming a single unit of energy and b denotes the cost associated with using (e.g., with renting) a parallel computer for a single unit of time. We are interested in finding the appropriate bundle, (P, X), which maximizes the utility function (i.e., minimizes the cost). Without any loss for generality, we consider the following simplified cost function $C(P, X)$:

$$C(P, X) = \alpha \cdot E(P, X) + T(P, X) \tag{2}$$

where $\alpha = (a/b)$.

We are interested in following question: *given a parallel algorithm, an architecture model, and the ratio α, what is the optimal number of cores and their frequencies that minimizes the cost function $C(P, X)$ as a function of input size*. In other words, we are interested in finding the appropriate configuration of the parallel computer (number of cores and their frequencies) such that the cost of executing the parallel algorithm on the parallel system is minimized. Note that using more than the obtained optimal number of cores will lead to an increase in the amount of energy consumed without a corresponding advantage of decreasing the overall cost. In other words, using more cores than the optimal number will lead to an energy waste (as measured by the utility function).

As discussed in the introduction, we make a number of simplifying architectural assumptions. Specifically, we assume:

1) All active cores operate at the same frequency and the frequency of the cores can be varied using a frequency (voltage) probe. The cores switch to *idle* state if there is no computation left at them.
2) There is no memory hierarchy at the cores.
3) Each core has its own memory and cores synchronize through message communication.
4) The computation and memory access time of the cores can be scaled by varying the frequency of the cores.
5) Communication time between the cores is constant. We justify this assumption by noting that the time consumed for sending and receiving a message is usually high compared to the time taken to route the messages between the cores.

The computation time T_{busy} on a given core is proportional to the number of compute cycles (including cache accesses) that are executed on the core. Let X be the frequency of a core, then:

$$T_{busy} = \text{number of cycles} \times \frac{1}{X} \tag{3}$$

We denote the time for which a given core is active (not idle) as T_{active}.

The following equation approximates the power consumption in a CMOS circuit:

$$P = C_L V^2 f + I_L V \tag{4}$$

where C_L is the load capacitance, V is the supply voltage, I_L is the leakage current, and f is the operational frequency. The first term corresponds to the *dynamic power consumption* by an algorithm, while the second term corresponds to its *leakage power consumption*.

Recall that a linear increase in the voltage supply leads to a linear increase in the frequency of the core. However, a linear increase in the voltage supply also leads to a nonlinear (typically cubic) increase in power consumption. Thus, for simplicity, we model the dynamic and leakage energies consumed by a core, E, to be the result of the above mentioned critical factor:

$$E_{dynamic} = E_d \cdot T_{busy} \cdot X^3 \tag{5}$$

$$E_{leakage} = E_l \cdot T_{active} \cdot X \quad (6)$$

where E_d and E_l are some hardware constants [5].

We assume that a single message transfer consumes a constant amount of energy. Because recent processors have introduced efficient support for low power modes that can reduce the power consumption to near zero, it is reasonable to consider the energy consumed by idle cores as 0.

The following parameters and constants are used in the rest of the paper.

- E_m : Energy consumed for single message communication between cores.
- F : Maximum frequency of a single core.
- N : Input size of the parallel application.
- P : Number of cores allocated for the parallel application.
- K_c : Number of cycles executed at maximum frequency during the time it takes to send a single message.

IV. Generic Methodology

We now describe our methodology to determine for a given parallel algorithm as a function of the input size, the optimal number of cores and the frequencies at which they should operate. By optimal we mean that the cost function $C(P, X)$ associated with executing the algorithm is minimized at the particular values of P and X.

As an initial step, we evaluate energy consumed $E(P, X)$ by the execution of the parallel algorithm, and the (total) time taken $T(P, X)$ in the execution. We do this by the following series of steps:

Step 1. Consider the task dependence graph of the algorithm, where the nodes represent tasks and the edges represent task serialization. Find the *critical path* of the parallel algorithm, where the critical path is the longest path through the task dependency graph of the parallel algorithm. Note that the critical path length gives a lower bound on the execution time of the parallel algorithm.

Step 2. Partition the critical path into communication and computation steps.

Step 3. Evaluate the *message complexity* (total number of messages processed) of the parallel algorithm. The example algorithms we later discuss show that the message complexity of some parallel algorithms may depend only on the number of cores, while for others it depends on both the input size and the number of cores used.

Step 4. Evaluate the total number of computation cycles at all the cores.

Step 5. Evaluate the total active time (T_{active}) at all the cores as a function of the frequency of the cores. Scaling the parallel algorithm (critical path) may lead to an increase in active time in other paths (at other cores).

Step 6. Using the energy model discussed earlier, frame an expression for the energy consumed by the parallel algorithm as a function of the frequency of the cores $E(P, X)$. The energy expression E is the sum of the energy consumed by: 1) the computation carried out by the algorith, E_{comp}, 2) the communication required by the algorithm, E_{comm}, and 3) leakage when cores are idle, E_{leak}. Energy consumption for each of these components is given by the following equations:

$$\begin{aligned} E_{comp} &= E_d \cdot (\text{Total number of computation cycles}) \cdot X^2 \\ E_{comm} &= E_m \cdot (\text{Total number of communication steps}) \\ E_{leak} &= E_l \cdot T_{active} \cdot X \end{aligned}$$

Note that E_{comp} is lower if the cores run at a lower frequency, while E_{leak} may increase as the busy cores take longer to finish. E_{comm} may increase as more cores are used and the computation is in this case more distributed.

Step 7. Compute the execution time of the parallel algorithm, $T(P, X)$:

$$T(P, X) = \mu_{comm} \cdot \frac{K_c}{F} + \mu_{comp} \cdot \frac{1}{X}$$

where μ_{comm} and μ_{comp} denote, respectively, the total number of communication steps and the number of computation cycles in the critical path.

Note that the execution time of an algorithm corresponds to the inverse of its performance.

After obtaining expressions for energy consumption and execution time, we now frame an expression for the cost function $C(P, X)$ of the parallel algorithm using Eq. 2:

$$C(P, X) = \alpha \cdot (E_{comp} + E_{comm} + E_{leak}) + T(P, X)$$

Finally, we analyze the cost expression to obtain the number of cores and the frequencies at which they should operate in order to minimize the cost as a function of the input size.

Illustrative Example.

We illustrate our methodology using a simple *parallel addition algorithm*. The parallel addition algorithm adds N numbers using M cores. Initially all N numbers are assumed to be distributed equally among the P cores; at the end of the computation, one of the cores stores their sum. Without loss of generality, we assume that the number of cores available is some power of two. The algorithm runs in $\log_2 P$ steps. In the first step, half of the cores send the sum they compute to the other half so that no core receives a sum from more than one core. The receiving cores then add the number the local sum they have computed. We perform the same step recursively until there is only one core left. At the end of computation, a single core will store the sum of all N numbers.

In the above algorithm, the critical path is easy to find: it is the execution path of the core that has the sum of all numbers at the end (Step 1). We can see that there are $\log(P)$ communication steps and $((N/P) - 1 + \log(P))$ computation steps (Step 2).

We next evaluate the total number of message transfers required by the parallel algorithm (Step 3). It is trivial to see

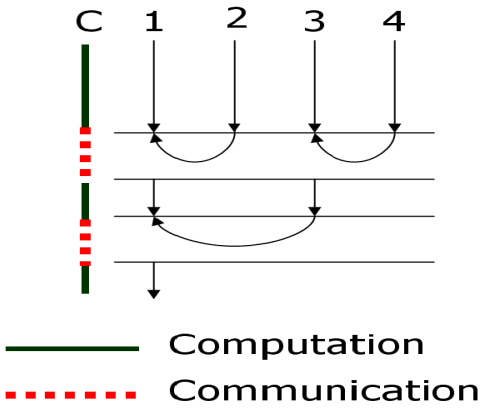

Fig. 1. Example scenario: Adding N numbers using 4 actors; Left most line represents the critical path

that the number of message transfers for this parallel algorithm when running on P cores is $(P-1)$. Note that, message complexity of this algorithm depends only on P and not on the input size N. Moreover, observe that the total number of computation steps at all cores is $N-1$ (Step 4).

We now evaluate the total 'active' time at all the cores, running at frequency X (Step 5). The total active time is:

$$T_{active} = \frac{\beta}{X} \cdot (N-1) + \frac{K_c}{F} \cdot (2 * (P-1))$$

where the first term represents the total active time spent by all cores performing computations, and the second term represents the total active time spent by all the cores during message transfers.

We now frame an expression for energy consumption $E(P,X)$ using the energy model (Step 6). The energy consumed for computation, communication and leakage while the algorithm is running on P cores at frequency X are given by the following equations:

$$\begin{aligned} E_{comp} &= E_d \cdot (N-1) \cdot \beta \cdot X^2 \\ E_{comm} &= E_m \cdot (P-1) \\ E_{leak} &= E_l \cdot T_{active} \cdot X \end{aligned}$$

where β is the number of computation cycles required per addition. Observe that the energy consumed by the parallel addition algorithm increases as either the number of cores used (P) increases, or the frequency at which these cores operate–when they are active (X)–increases.

The execution time of the addition algorithm as a function of frequency of the cores X (Step 7) is as follows:

$$T(P,X) = \log_2 P \cdot \frac{K_c}{F} + ((N/P) - 1 + \log(P)) \cdot \beta \cdot \frac{1}{X}$$

Note that time taken by the parallel addition algorithm decreases as either number of cores (P) or frequency of the cores (X) increase.

The cost of the parallel addition algorithm is given as follows:

$$C(P,X) = \alpha \cdot (E_{comp} + E_{comm} + E_{leak}) + T(P,X)$$

As noted earlier, E increases and T decreases as either P or X increase. Thus, there exist an optimal configuration at which the cost is minimum. The cost curve shown in Fig 2 depicts the energy-time trade-off assuming a (reasonable) set of values for the various parameters. In the next section, we analyze the cost expression obtained above in order to understand the properties of optimal configurations.

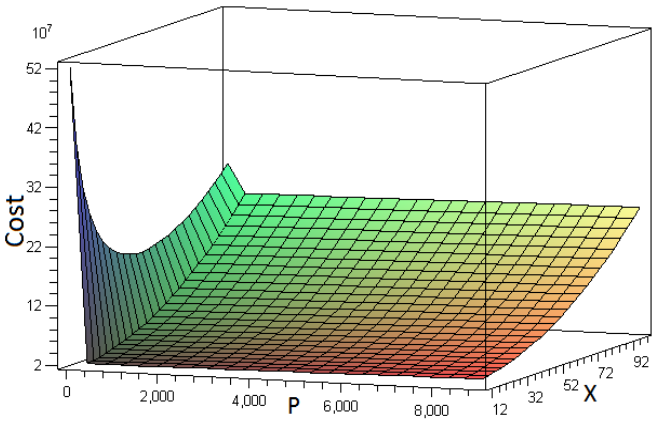

Fig. 2. Addition: Cost curve with $C(P,X)$ on Z axis, number of cores on X axis and frequency on Y axis with $k = 500$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$. Frequency is plotted in units $F/100$ where F is the maximum frequency. Number of cores is plotted in units of 10^4.

V. Analyzing Cost Expression

We now analyze the cost expression obtained above for the addition algorithm in order to determine the number of cores and their frequencies which minimize cost as a function of the input size. While we could differentiate the expression to compute the configuration which minimizes cost, symbolic differentiation results in a rather complex expression. Instead, we simply analyze the corresponding graphs.

Note that the cost expression is dependent on many parameters: N (input size), P (number of cores), β (number of instruction per addition), K_c (number of cycles executed at maximum frequency for single message communication time), E_m (energy consumed for single message communication between cores), E_l (leakage constant), the maximum frequency F of a core and the utilization ratio α. We can simplify a couple of these parameters without loss of generality. In most architectures, the number of cycles involved per addition is just one, so we assume $\beta = 1$. Without loss of generality, we set the leakage energy constant $E_l = 1$ and express all energy values with respect to this normalized energy value.

In order to graph the required differential, we must make some assumptions about the other parameters. While these assumptions compromise generality, we discuss the sensitivity of the analysis to a range of values for these parameters. One such parameter is the the energy consumed for single cycle at maximum frequency (as a multiple of the leakage energy constant). We assume this ratio to be 10, i.e., that $E_d \cdot F^2 = 10 \cdot E_l$. It turns out that this parameter is not very significant for our analysis; in fact, large variations in the parameter

do not affect the shapes of the graphs significantly. Another parameter, k, represents the ratio of the energy consumed for a communicating a single message, E_m, and the energy consumed for executing a single instruction at the maximum frequency. Thus, $E_m = k \cdot E_d \cdot F^2$. We analyze the sensitivity of our results to a wide range of values of k and α. For convenience, we denote $X = \gamma \cdot F$ such that $0 < \gamma < 1$ (since $0 < X < F$). Note that, with the above simplification, the cost expression will be free of F and the range of X is mapped correspondingly to the range of the parameter γ.

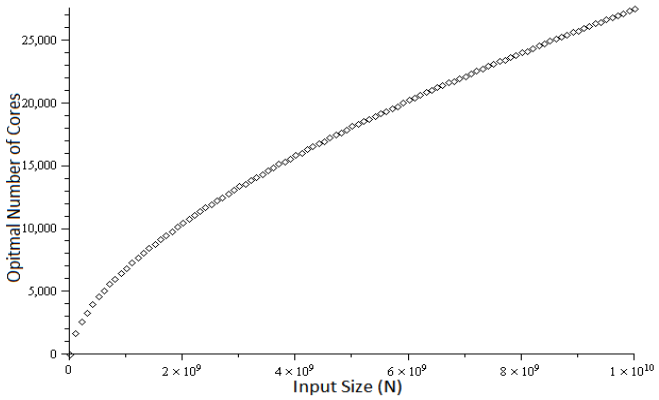

Fig. 3. Addition: optimal number of cores on Y axis and input size on X axis with $k = 500$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$.

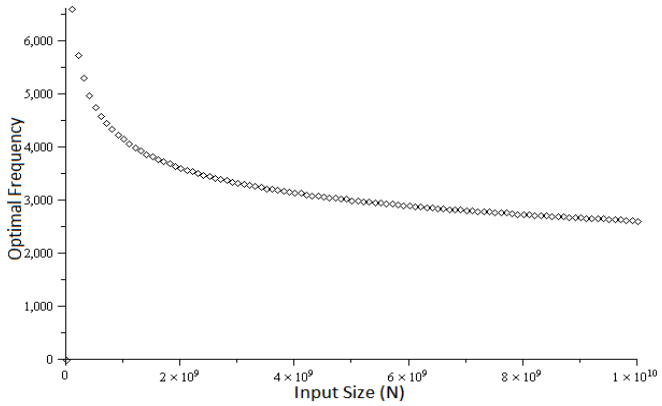

Fig. 4. Addition: Optimal frequency on Y axis and input size on X axis with $k = 500$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$. Frequency on Y axis is plotted in units $F/100000$ where F is the maximum frequency.

We use a brute force search technique to evaluate the optimal pair of number of cores and frequency of the cores required to minimize cost. Fig. 3 and Fig. 4 plot, respectively, the number of cores and the frequency of these cores (when active) which minimizes the cost as a function of input size. We see that optimal number of cores required increases with increasing input size (roughly following a negative exponential curve with a positive coefficient). However, the frequency of cores required to minimize the cost decreases with input size.

We now consider the sensitivity of this analysis with respect to k: recall that k is the ratio of energy used per message over

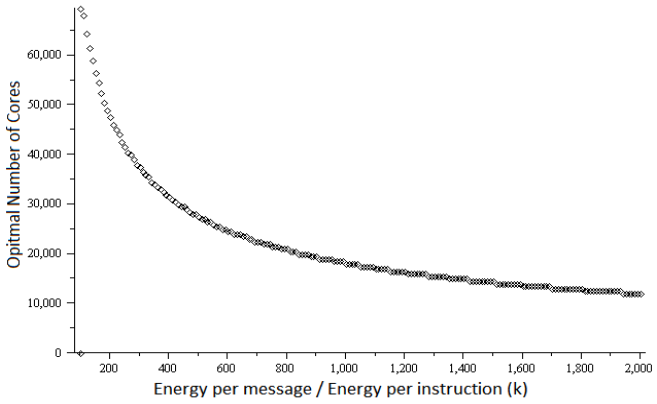

Fig. 5. Sensitivity analysis: optimal number of cores on Y axis and k (ratio of the energy consumed for single message transfer and the energy consumed for executing a single instruction at the maximum frequency) on X axis with $N = 10^8$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$.

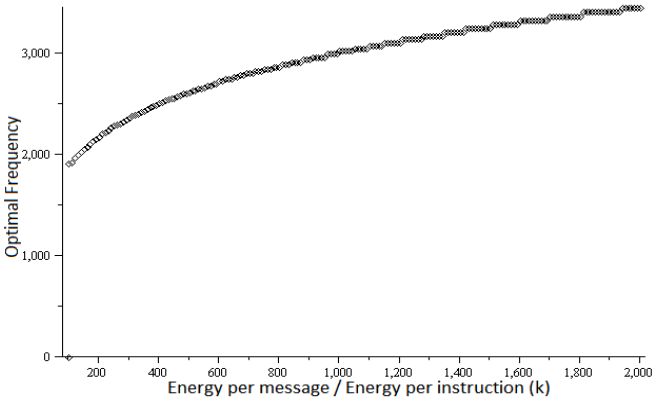

Fig. 6. Sensitivity analysis: Optimal frequency on Y axis and k (ratio of the energy consumed for single message transfer and the energy consumed for executing a single instruction at the maximum frequency) on X axis $N = 10^8$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$. Frequency on Y axis is plotted in units $F/100000$.

the energy used per instruction. Fig. 5 and Fig. 6 plot the number of cores and the frequency of the cores required to minimize cost by fixing the input size and varying k respectively. We see that optimal number of cores decreases with increasing k. Furthermore, the frequency of cores required to minimize cost increases with increasing k. We observe that this trend remains the same for the range of input values we studied (10^8 to 10^{10}).

We now consider the sensitivity of this analysis with respect to the ratio α, the cost of unit energy. Fig. 7 and Fig. 8 plot, respectively, the number of cores and their frequency (when active) that is required to minimize cost. We do this by fixing the input size and varying α. We note that both optimal number of cores and frequency of the cores required to minimize cost decreases with increasing α. We observe that this trend also remains the same for entire range of the input values considered (10^8 to 10^{10}).

The above graphs depict the exact behavior of the optimal number of cores and frequencies as a function of input size

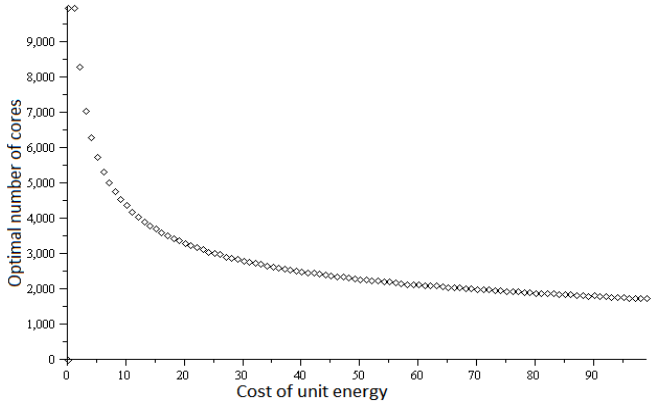

Fig. 7. Sensitivity analysis: optimal number of cores on Y axis and cost of unit energy (α) (measured relative to the cost of running the parallel system for unit time) on X axis with $N = 10^8$, $\beta = 1$, $K_c = 500$ and $k = 500$.

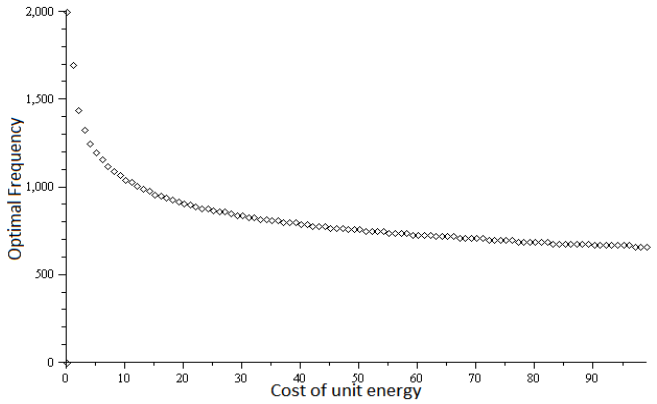

Fig. 8. Sensitivity analysis: Optimal frequency on Y axis and cost of unit energy (α) (measured relative to the cost of running the parallel system for unit time) on X axis with $N = 10^8$, $\beta = 1$, $K_c = 500$ and $k = 500$. Frequency on Y axis is plotted in units $F/100000$.

(over a specific input range) and, as far as can tell, appears to generalize to larger input sizes. However, we are unable to give an analytic expression for the asymptotic behavior of the optimal number of cores and frequencies as a function of input size as the cost expression is very complex.

VI. Case Studies

We now analyze four algorithms: two quicksort-based parallel algorithms, LU factorization, and Parallel Prim's Minimum Spanning Tree (MST) algorithm.

A. Naïve Parallel Quicksort Algorithm

Consider a naïve (and inefficient) parallel algorithm for quicksort. Recall that in the quicksort algorithm, an array is partitioned into two parts based on a pivot, and each part is solved recursively. In the naïve parallel version, an *input array* is partitioned into two parts by a single core (based on a pivot), and then one of the sub-arrays is assigned to another core. Now each of the cores partitions the arrays it is working on using the same approach as above, and assigns one of its sub-arrays to other cores. This process continues until all the available cores are used up. After the partitioning phase, in the average case, each core will have approximately an equal division of all elements of the input array. Finally, all the cores sort their arrays in parallel, using the serial quicksort algorithm on each core. The sorted input array can be recovered by traversing the cores. The naïve parallel quicksort algorithm is very inefficient (in terms of performance), as partitioning the array into two sub-arrays is done by single core which means that the execution time of the naïve parallel quicksort algorithm is bounded from below by the length of the input array.

Assume that the input array has N elements and the number of cores available for sorting are P. Without loss of generality, we assume both N and P are powers of two, so that $N = 2^a$, for some a and $P = 2^b$, for some b. For simplicity, we also assume that during the partitioning step, each core partitions the array into two equal sub-arrays by choosing the appropriate pivot (the usual average case analysis).

The critical path of this parallel algorithm is the execution done by the core which initiates the partitioning of the input array. The total number of communication and computation steps in the critical path are, respectively, $N(1-(1/P))$ and $2N(1-(1/P)) + K_q((N/P)\cdot\log(N/P))$, where K_q (1.4) is the quicksort constant.

Next, we evaluate the number of message transfers required in total by the parallel algorithm. It is trivial to see that number of message transfers for this parallel algorithm running on M cores is $\log(P)\cdot(N/2)$. Note that, unlike the previous example, the message complexity for naïve quicksort is dependent both on the number of cores and on the input size. Moreover, the total number of computation steps at all cores is $N \cdot log(P) + K_q \cdot N \cdot log(\frac{N}{P})$(Step 4).

We now evaluate the total active time at all the cores, running at frequency X. The total active time is given by the following equation

$$T_{active} = \frac{\beta}{X}\cdot(N\cdot log(P) + K_q \cdot N \cdot log(\frac{N}{P})) + \frac{K_c}{F}\cdot\log(P)\cdot N$$

where the first term represents the total active time spent by all cores performing computations, and second term represents the total active time spent by all the cores during message transfers.

We now frame an expression for energy consumption as a function of the frequency of the cores. The energy consumed for computation, communication and leakage while the algorithm is running on P cores at frequency X is given by:

$$\begin{aligned} E_{comp} &= E_d \cdot \left(N \cdot log(P) + K_q \cdot N \cdot log(\frac{N}{P})\right) \cdot \beta \cdot X^2 \\ E_{comm} &= E_m \cdot log(P) \cdot \frac{N}{2} \\ E_{leak} &= E_l \cdot T_{active} \cdot X \end{aligned}$$

where β is the number of cycles required per comparison.

The time taken $T(P, X)$ by the naïve quicksort algorithm as a function of X, the frequency at which the active cores

run, is as follows:

$$T(P,X) = N(1-(1/P)) \cdot \frac{K_c}{F} + 2N(1-(1/P)) \cdot \frac{\beta}{X} + (K_q((N/P) \cdot \log(N/P))) \cdot \frac{\beta}{X}$$

The cost of the naive quicksort algorithm is:

$$C(P,X) = \alpha \cdot (E_{comp} + E_{comm} + E_{leak}) + T(P,X)$$

Fig 9 depicts the cost of the naive quicksort algorithm for a wide range of configurations (P, X). It is clear from the curve that the cost is minimum when single core is used. In other words, there is no benefit from using more cores (using more cores will lead to energy wastage).

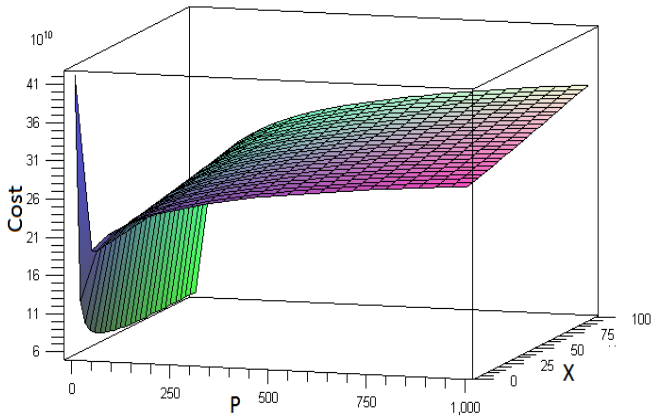

Fig. 9. Naive Quicksort: Utility curve with $C(P, X)$ on Z axis, number of cores on X axis and frequency on Y axis with $k = 500$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$. Frequency is plotted in units $F/100$ where F is the maximum frequency. Number of cores is plotted in units of 10^4.

We now analyze the cost expression obtained above for the naïve parallel quicksort algorithm in order to determine as a function of the input size, the optimal number of cores and their frequencies. For this analysis, we use the same assumptions that were used earlier in the analysis of the parallel addition algorithm ($k = 500$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$). We observe that both the optimal number of cores and frequencies of these cores required to minimize cost remain constant at 1 and $0.79F$ respectively (for the input sizes ranging from 10^8 to 10^{10}).

We now consider the sensitivity of this analysis with respect to the ratio k (representing the energy per message over the energy per instruction). For a fixed input size, Fig. 10 and Fig. 11 plot, respectively, the optimal number of cores and frequency of the cores as a function of k respectively. We see that number of cores required to minimize cost decreases significantly at very small k and attains the minimum value (number of cores = 1) for the remaining range of k. The frequency of the cores follows a similar trend except that it increases initially. We observe that this trend remains the same for entire range of input sizes we consider (10^8 to 10^{10}).

We next consider the sensitivity of this analysis with respect to the ratio α measuring the relative cost of a unit of

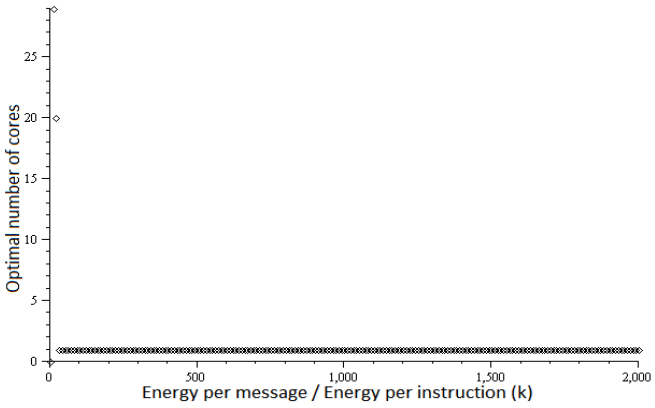

Fig. 10. Sensitivity analysis: optimal number of cores on Y axis and k (ratio of the energy consumed for single message transfer and the energy consumed for executing a single instruction at the maximum frequency) on X axis with $N = 10^8$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$.

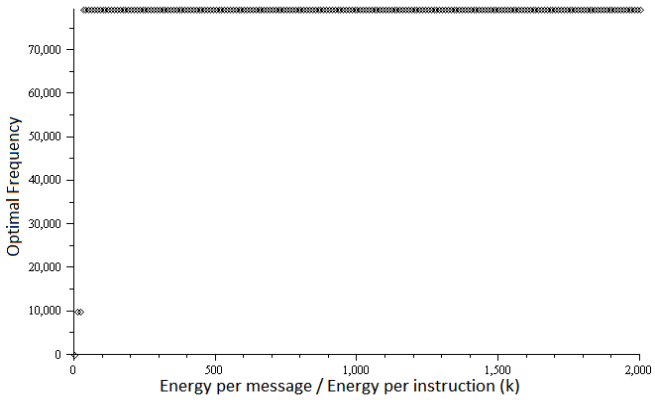

Fig. 11. Sensitivity analysis: Optimal frequency on Y axis and k (ratio of the energy consumed for single message transfer and the energy consumed for executing a single instruction at the maximum frequency) on X axis $N = 10^8$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$. Frequency on Y axis is plotted in units $F/100000$.

energy. We observed that the optimal number of cores remains constant at 1 for a wide range of values of α. Fig. 12 plots the frequency of the cores required to minimize the cost as a function of and varying α (assuming a fixed input size). We see that the frequency of the cores are required to operate at in order to minimize cost decreases with α. We also observe that this trend remains the same for entire range of input values considered (10^8 to 10^{10}). Based on these observations, we claim that, for the naïve parallel quicksort algorithm, using a single core at an appropriate frequencies is good enough for attaining minimizing cost (and avoiding energy wastage).

B. Parallel Quicksort Algorithm

The parallel quicksort formulation [20] works as follows. Let N be the number of elements to be sorted and $P = 2^b$ be the number of cores available. Each cores is assigned a block of N/P elements, and the labels of the cores $\{1, ..., P\}$ define the global order of the sorted sequence. For simplicity, we assume that the initial distribution of elements in each

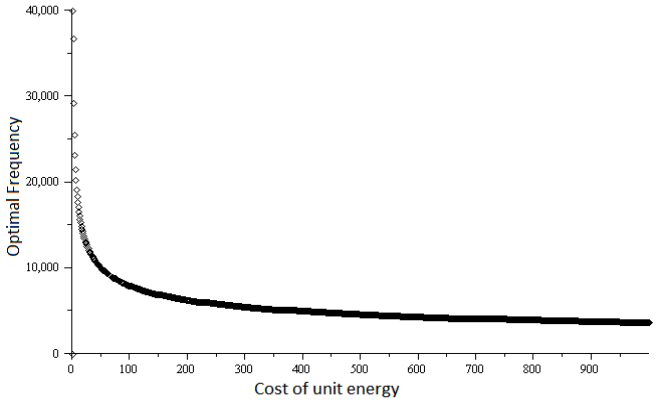

Fig. 12. Sensitivity analysis: Optimal frequency on Y axis and cost of unit energy (α) (measured relative to the cost of running the parallel system for unit time) on X axis $N = 10^8$, $\beta = 1$, $K_c = 500$ and $k = 500$. Frequency on Y axis is plotted in units $F/100000$.

core is uniform. The algorithm starts with all cores sorting their own set of elements (sequential quicksort). Then 'Core 1' broadcasts the median of its elements to each of the other cores. This median acts as the pivot for partitioning elements at all cores. Upon receiving the pivot, each cores partition its elements into elements smaller than the pivot and elements larger than the pivot. Next, each Core i where $i \in \{1...P/2\}$ exchanges elements with the Core $i + P/2$ such that core i retains all the elements smaller than the pivot, and Core $i + P/2$ retains all elements larger than the pivot. After this step, each Core i $i \in \{1....P/2\}$ stores elements smaller than the pivot, and remaining cores ($\{P/2+1, ...P\}$) store elements greater than the pivot. Upon receiving the elements, each core merges them with its own set of elements so that all elements at the core remain sorted. The above procedure is performed recursively for both sets of cores, splitting the elements further. After b recursions, all the elements are sorted with respect to the global ordering imposed on the cores.

Since all cores are busy all the time, the critical path of this parallel algorithm would be the execution path of any one of the cores. The total number of communication and computation steps in the critical path evaluates to $(1+N/P)\cdot \log P$ and $(\log(N/P)+N/P)\cdot \log P + K_q(N/P\cdot \log(N/P))$, where K_q (1.4) is the quicksort constant.

The number of message transfers for this parallel algorithm running on P cores is $(P\cdot \log(P) - P + 1) + \log(P)\cdot(N/2)$. Moreover, the total number of computation steps at all cores evaluates to $((\log N/P + N/P)\cdot \log P + K_q \cdot N/P \cdot \log N/P)\cdot P$.

We now evaluate the total active time at all the cores, running at frequency X. The total active time is given by the following equation:

$$
\begin{aligned}
T_{active} \\
= \quad & \frac{\beta \cdot P}{X} \cdot \left(\left(\log \frac{N}{P} + \frac{N}{P} \right) \cdot \log P + K_q \cdot \frac{N}{P} \cdot \log \frac{N}{P} \right) \\
& + \frac{K_c}{F} \cdot 2 \cdot \left((P \cdot \log(P) - P + 1) + \log(P) \cdot (\frac{N}{2}) \right)
\end{aligned}
$$

where the first term represents the total active time spent by all the cores performing computations, and second term represents the total active time spent by all the cores during message transfers.

We frame an expression for energy consumption E using the energy model. The energy consumed for computation, communication and leakage while the algorithm is running on P cores at frequency X is given by:

$$
\begin{aligned}
& E_{comp} = \\
& E_d \left(\left(\log \frac{N}{P} + \frac{N}{P} \right) \log P + K_q \cdot \frac{N}{P} \cdot \log \frac{N}{P} \right) \cdot P \cdot \beta \cdot X^2
\end{aligned}
$$

$$
\begin{aligned}
E_{comm} &= E_m \cdot \left(P \cdot \log P - P + 1 + \log P \cdot \frac{N}{2} \right) \\
E_{leak} &= E_l \cdot T_{active} \cdot X
\end{aligned}
$$

The time taken T by the parallel quicksort algorithm as a function of frequency of the cores X is as follows:

$$
\begin{aligned}
& T(P, X) = \left(1 + \frac{N}{P} \right) \cdot \log P \cdot \frac{K_c}{F} \\
& + \left(\log(\frac{N}{P}) + \frac{N}{P} \right) \cdot \log P + K_q \left(\frac{N}{P} \cdot \log(\frac{N}{P}) \right) \cdot \beta \cdot \frac{1}{X}
\end{aligned}
$$

The cost of the parallel quicksort algorithm is given by the following:

$$C(P, X) = \alpha \cdot (E_{comp} + E_{comm} + E_{leak}) + \beta \cdot T(P, X)$$

Fig 13 depicts the cost of the naive quicksort algorithm for a wide range of configurations (P, X). As in the case for the naïve parallel quicksort algorithm, the cost is minimum when a single core is used. In other words, there is no benefit to overall utility to be gained by using more cores.

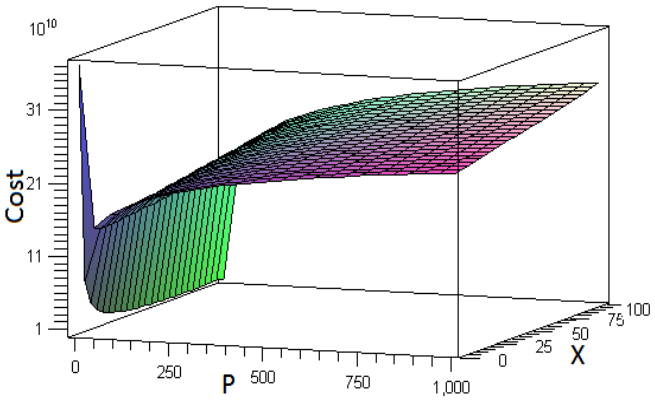

Fig. 13. Parallel Quicksort: Cost curve with $C(P, X)$ on Z axis, number of cores on X axis and frequency on Y axis with $k = 500$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$. Frequency is plotted in units $F/100$ where F is the maximum frequency. Number of cores is plotted in units of 10^4.

We now analyze the cost expression obtained above for the parallel quicksort algorithm in order to determine, as a function of the input size, the number of cores and their frequencies required for minimum cost. We use the same values that were used earlier in the analysis of the parallel

addition algorithm ($k = 500$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$). We observe that both the optimal number of cores and frequencies of the cores required to minimize cost remains constant at 1 and $0.79F$, respectively, for the range of input sizes varying from 10^8 to 10^{10}.

We now consider the sensitivity of this analysis with respect to the ratio k. By assuming $N = 10^8$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$, we observe that to minimize cost both the optimal number of cores and the frequencies of the cores required remains constant (in the range $10 < k < 2000$. We observe that this trend also remains the same in the entire range of input sizes considered (10^8 to 10^{10}).

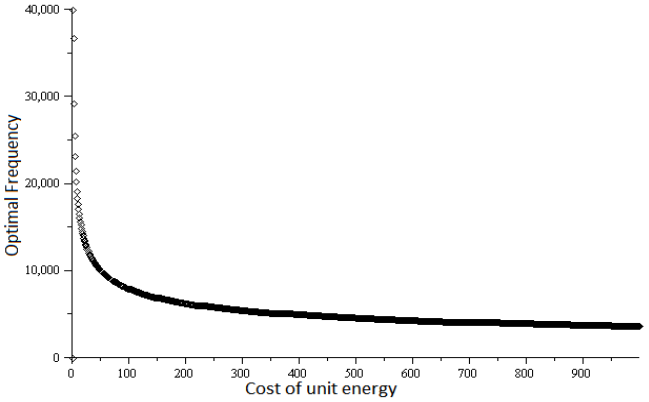

Fig. 14. Sensitivity analysis: Frequency on Y axis and cost of unit energy (α) (measured relative to the cost of running the parallel system for unit time) on X axis $N = 10^8$, $\beta = 1$, $K_c = 500$ and $k = 500$. Frequency on Y axis is plotted in units $F/100000$.

Finally, we consider the sensitivity of the analysis of this algorithm with respect to the relative cost of unit energy α. We observe that the optimal number of cores remains constant (at 1) for a wide range of values of α. Fig. 14 plots frequency of the cores required to minimize cost by fixing the input size and varying α. We can see that the frequency of cores required to minimize cost decreases with α. We observe that this trend remains the same for the entire range of input sizes considered (10^8 to 10^{10}). From above observations, we conclude that parallel quicksort algorithm–which it turns out has better scalability characteristics than naive quicksort algorithm–does not scale when it comes to minimizing cost (or avoiding energy wastage).

C. LU Factorization

1) Sequential Algorithm: Given a $N \times N$ matrix A, *LU factorization* involves coming up with a unit lower triangular matrix L and an upper triangular matrix U such that A = LU. A standard way to compute LU factorization is by Guassian elimination. In this approach, the matrix U is obtained by overwriting A. We presume that the reader is familiar with the Gaussian elimination algorithm. Recall that Guassian elimination requires about $N^3/3$ paired additions and multiplications and about $N^2/2$ divisions. (For performance analysis we ignore the later lower order term).

2) Parallel Algorithm: There are many parallel algorithms for LU factorization problem. Here we consider only the *coarse-grain 1-D column parallel algorithm* [21] for our performance analysis. Each core is assigned a few columns of the matrix and the cores communicate with each other and obtain the required matrix U. The algorithm at each of the P cores is described as follows:

LU Factorization

for k = 1 to $N - 1$ **do**
  if $k \in mycols$ **then**
    for i = $k + 1$ to N **do**
      $l_{ik} = a_{ik}/a_{kk}$ {multipliers}
    end for
  end if
  broadcast $\{l_{ik} : k < i \leq N\}$ {broadcast}
  for $J \in mycols, j > k$ **do**
    for i = $k + 1$ to N **do**
      $a_{ij} = a_{ij} - l_{ik}a_{kj}$ {update}
    end for
  end for
end for

In the algorithm, matrix rows need not be broadcast vertically, since any given column is contained entirely in only one process. But there is no parallelism in computing multipliers or updating a column. Horizontal broadcasts are required to communicate multipliers for updating. On average, each core performs about $N^3/(3 \cdot P)$ operations (one addition and one multiplication). Moreover, each core broadcasts about $N^2/2$ messages, under the assumption that overlap of broadcasts for successive steps is allowed.

The number of message transfers required in total by the parallel algorithm evaluates to $P \cdot (N^2/2)$. Furthermore, the total number of computation steps at all cores evaluates to $(N^3/3)$.

We now evaluate the total active time at all the cores, running at frequency X. The total active time is given by the following equation:

$$T_{active} = \frac{\beta}{X} \cdot (N^3/3) + \frac{K_c}{F} \cdot 2 \cdot (P \cdot (N^2/2))$$

where the first term represents the total active time spent by all cores performing computations, and second term represents the total active time spent by all the cores during message transfers.

We frame an expression for energy consumption E using the energy model. The energy consumed for computation, communication and leakage while the algorithm is running on P cores at frequency X is given by:

$$\begin{aligned} E_{comm} &= E_m \cdot P \cdot \frac{N^2}{2} \\ E_{comp} &= E_d \cdot \frac{N^3}{3} \cdot \beta \cdot X^2 \\ E_{leak} &= E_l \cdot T_{active} \cdot X \end{aligned}$$

.

The time taken T(P,X) by the parallel LU-factorization algorithm as a function of frequency of the active cores X is as follows:

$$T(P,X) = \frac{N^2}{2} \cdot \frac{K_c}{F} + \frac{N^3}{3 \cdot P} \cdot \beta \cdot \frac{1}{X}$$

The cost of the parallel addition algorithm is given by the following:

$$C(P,X) = \alpha \cdot (E_{comp} + E_{comm} + E_{leak}) + T(P,X)$$

Fig 15 depicts the cost of the LU-factorization algorithm for a wide range of configurations (P,X). Note that the cost curve is very similar to the one obtained for the parallel addition algorithm.

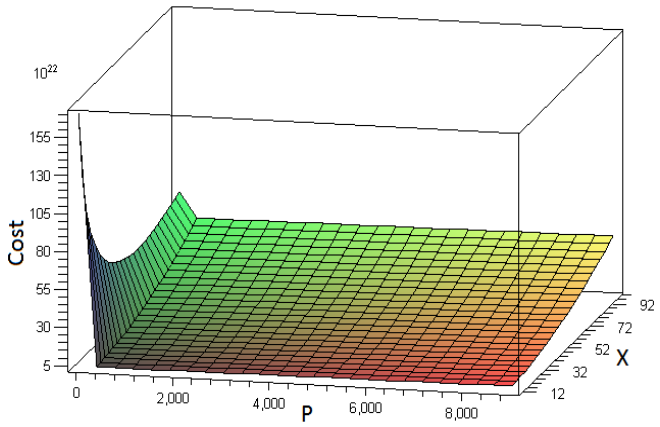

Fig. 15. LU Factorization: Cost curve with $C(P,X)$ on Z axis, number of cores on X axis and frequency on Y axis with $k = 500$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$. Frequency is plotted in units $F/100000$ where F is the maximum frequency.

We now analyze the cost expression obtained above for the LU-factorization algorithm in order to determine the optimal number of cores and their frequencies as a function of the input size. We again use the same assumptions about parameter values that were used earlier in the analysis of the previous algorithms.

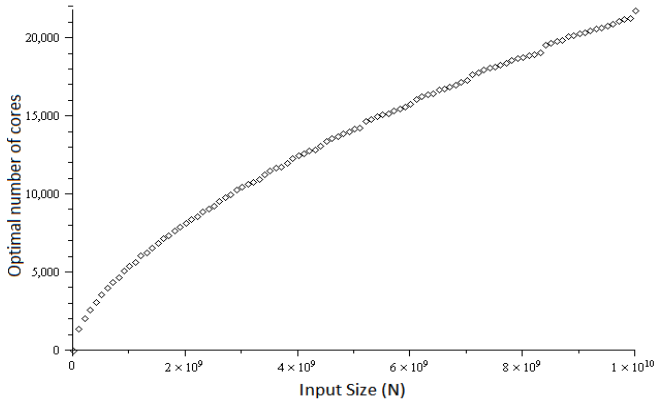

Fig. 16. LU Factorization: optimal number of cores on Y axis and input size on X axis with $k = 500$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$.

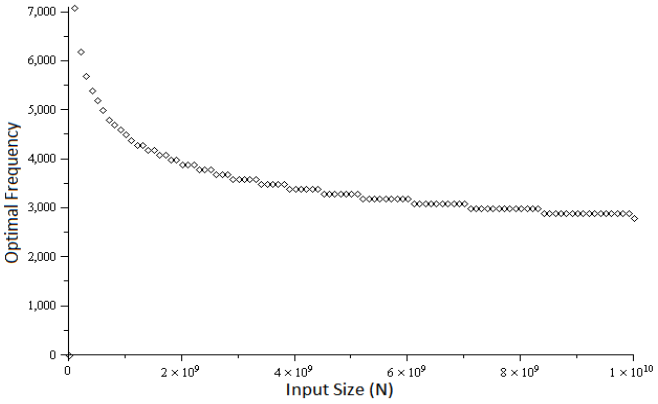

Fig. 17. LU Factorization: Optimal frequency on Y axis and input size on X axis with $k = 500$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$. Frequency on Y axis is plotted in units $F/100000$ where F is the maximum frequency.

Fig. 16 and Fig. 17 plots optimal number of cores and frequency of the cores required to minimize cost as a function of input size respectively. We see that optimal number of cores required to minimize cost increases with increasing input size (roughly follows a negative exponential curve with coefficient). Note that for all input sizes, the optimal number of cores obtained for parallel addition algorithm is greater than the number obtained for this algorithm. Moreover, the frequency of cores required to minimize cost decreases with input size.

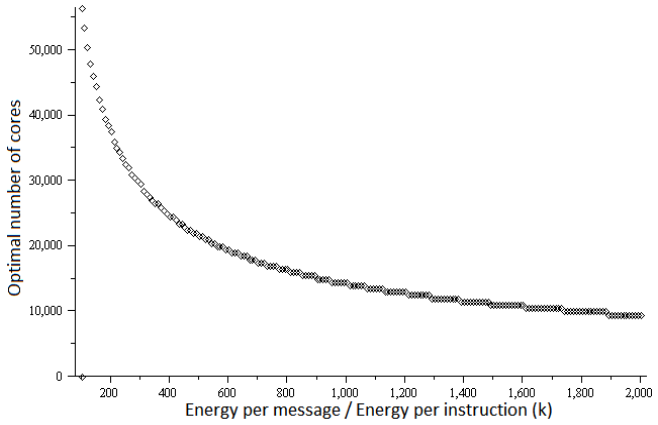

Fig. 18. Sensitivity analysis: optimal number of cores on Y axis and k (ratio of the energy consumed for single message transfer and the energy consumed for executing a single instruction at the maximum frequency) on X axis with $N = 10^8$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$.

We consider the sensitivity of this analysis with respect to the ratio k. Fig. 18 and Fig. 19 plot, respectively, the optimal number of cores and the frequency at which the cores are required to operate in order to minimize cost, if the input size is fixed and k is varied. We can see that, as in parallel addition algorithm, the optimal number of cores decreases with increasing k. Moreover, as in the parallel addition algorithm, the frequency of the active cores required to minimize cost increases with increasing k. We observe that this trend remains

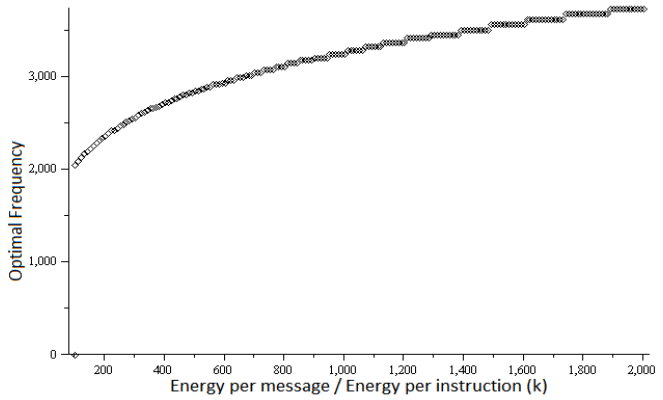

Fig. 19. Sensitivity analysis: Optimal frequency on Y axis and k (ratio of the energy consumed for single message transfer and the energy consumed for executing a single instruction at the maximum frequency) on X axis $N = 10^8$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$. Frequency on Y axis is plotted in units $F/100000$.

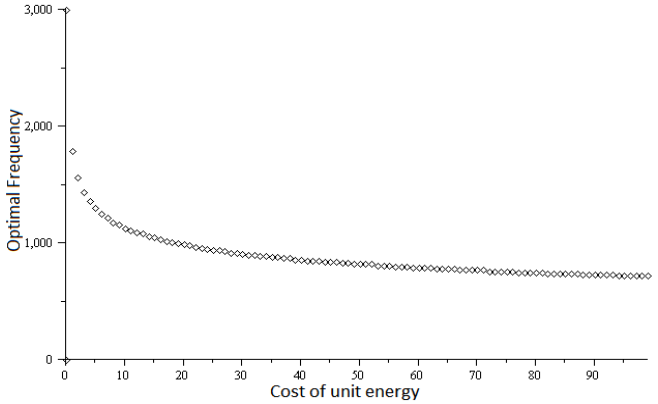

Fig. 21. Sensitivity analysis: Frequency on Y axis and cost of unit energy (α) (measured relative to the cost of running the parallel system for unit time) on X axis $N = 10^8$, $\beta = 1$, $K_c = 500$ and $k = 500$. Frequency on Y axis is plotted in units $F/100000$.

the same for entire range of input values considered (10^8 to 10^{10}).

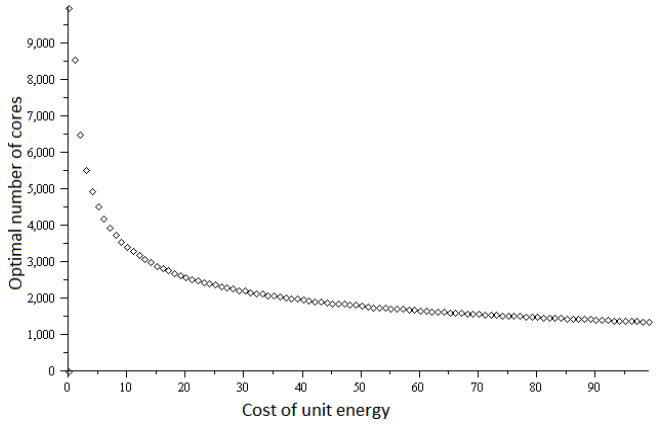

Fig. 20. Sensitivity analysis: optimal number of cores on Y axis and cost of unit energy (α) (measured relative to the cost of running the parallel system for unit time) on X axis with $N = 10^8$, $\beta = 1$, $K_c = 500$ and $k = 500$.

Finally, we now consider the sensitivity of this analysis with respect to the ratio α. Fig. 20 and Fig. 21 plots as a function of k, respectively, the optimal number of cores and the optimal frequency of these cores operate when active (for a fixed input size). We note that, as in parallel addition algorithm, both the optimal number of cores and the frequency of these cores required to minimize cost decreases with increasing α. We observe that this trend also remains the same for a range of input sizes (10^8 to 10^{10}). Based on the above observations, we claim that both the parallel LU-factorization algorithm and the parallel addition algorithm have similar optimal configuration characteristics.

D. Minimum Spanning Tree: Prim's Algorithm

a) Sequential Algorithm: A spanning tree of an undirected graph G is a subgraph of G that is a tree containing all vertices of G. In a weighted graph, the weight of a subgraph is the sum of the weights of the edges in the subgraph. A minimum spanning tree for a weighted undirected graph is a spanning tree with minimum weight. Prim's algorithm for finding an MST is a greedy algorithm. The algorithm begins by selecting an arbitrary starting vertex. It then grows the minimum spanning tree by choosing a new vertex and edge that are guaranteed to be in the minimum spanning tree. The algorithm continues until all the vertices have been selected. We provide the code for the algorithm below.

PRIM_MST(V, E, w, r)

1: $V_T = \{r\}$;
2: $d[r] = 0$;
3: **for** all $v \in (V - V_T)$ **do**
4: **if** edge(r, c) exists **then**
5: set $d[v] = w(r, v)$
6: **else**
7: set $d[v] = \infty$
8: **end if**
9: **while** $V_T \neq V$ **do**
10: find a vertex u such that $d[u] = \min\{d[v]|v \in (V - V_T)\}$;
11: $V_T = V_T \cup \{u\}$
12: **for** all $v \in (V - V_T)$ **do**
13: $d[v] = \min\{d[v], w(u, v)\}$;
14: **end for**
15: **end while**
16: **end for**

In the above program, the body of the while loop (lines 10–13) is executed $n - 1$ times. Both the number of comparisons performed for evaluating $\min\{d[v]|v \in (V - V_T)\}$ (Line 10) and the number of comparisons performed in the for loop (Lines 12 and 13) decreases by one for each iteration of the main loop. Thus, by simple arithmetic, the overall number of comparisons done by the algorithm is around n^2 (ignoring lower order terms).

1) Parallel Algorithm: We consider the parallel version of Prim's algorithm in [22]. Let P be the number of cores, and let N be the number of vertices in the graph. The set V is partitioned into P subsets such that each subset has N/P consecutive vertices. The work associated with each subset is assigned to a different core. Let V_i be the subset of vertices assigned to core C_i for $i = 0, 1, \cdots, M-1$. Each core C_i stores the part of the array d that corresponds to V_i. Each core C_i computes $d_i[u] = \min\{d_i[v] | v \in (V \setminus V_T \wedge V_i)\}$ during each iteration of the while loop. The global minimum is then obtained over all $d_i[u]$ by sending them to core C_0. The core C_0 now holds the new vertex u, which will be inserted into V_T. Core C_0 broadcasts u to all cores. The core C_i responsible for vertex u marks u a belonging to set V_T. Finally, each processor updates the values of $d[v]$ for its local vertices. When a new vertex u is inserted into V_T, the values of $d[v]$ for $v \in (V \setminus V_T)$ must be updated. The core responsible for v must know the weight of the edge (u, v). Hence each core C_i needs to store the columns of the weighted adjacency matrix corresponding to the set V_i of vertices assigned to it.

On average, each core performs about N^2/P comparisons. Moreover, each core is involved in $2 \cdot N$ (ignoring lower order constants) message communications. The number of message transfers required in total by the parallel algorithm evaluates to $2 \cdot P \cdot N$. Furthermore, the total number of computation steps at all cores on average evaluates to N^2.

We now evaluate the total active time at all the cores, running at frequency X. Total active time is given by the following equation

$$T_{active} = \frac{\beta}{X} \cdot (N^2) + \frac{K_c}{F} \cdot 2 \cdot (2 \cdot P \cdot N) \tag{7}$$

where the first term represents the total active time spent by all cores performing computations, and second term represents the total active time spent by all the cores during message transfers.

Now, we frame an expression for energy consumption as a function of the frequency of the cores. The energy consumed for computation and communication while the algorithm is running on P cores at frequency X is given by the following equations:

$$\begin{aligned} E_{comm} &= E_m \cdot 2 \cdot P \cdot N \\ E_{comp} &= E_d \cdot N^2 \cdot \beta \cdot X^2 \\ E_{leak} &= E_l \cdot T_{active} \cdot X \end{aligned}$$

The time taken T(P,X) by the parallel Prim's minimum spanning tree algorithm as a function of the frequency of the cores X is as follows:

$$T(P, X) = 2 \cdot N \cdot \frac{K_c}{F} + \frac{N^2}{2 \cdot P} \cdot \beta \cdot \frac{1}{X}$$

The cost of the parallel MST algorithm is given as follows:

$$C(P, X) = \alpha \cdot (E_{comp} + E_{comm} + E_{leak}) + T(P, X)$$

Fig 22 depicts the cost of the parallel MST algorithm for a wide range of configurations (P, X). Note that the cost curve is very similar to the ones obtained for parallel addition algorithm and LU factorization.

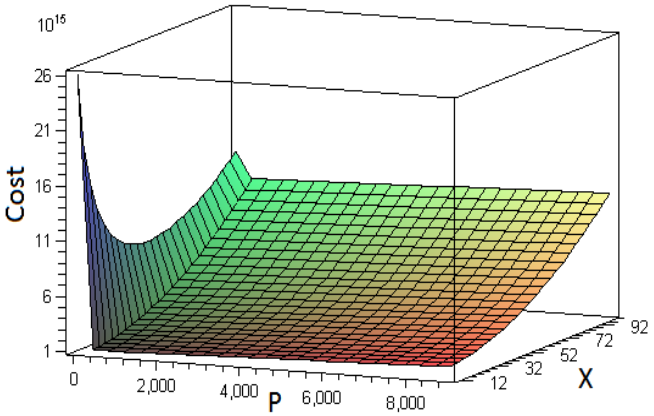

Fig. 22. Parallel MST: Cost curve with $C(P, X)$ on Z axis, number of cores on X axis and frequency on Y axis with $k = 500$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$. Frequency is plotted in units $F/100$ where F is the maximum frequency. Number of cores is plotted in units of 10^4.

We now analyze the cost expression obtained above for the parallel MST algorithm in order to determine the optimal number of cores and their frequencies as a function of the input size. We use the same assumptions that were used earlier in the analysis of the parallel addition algorithm.

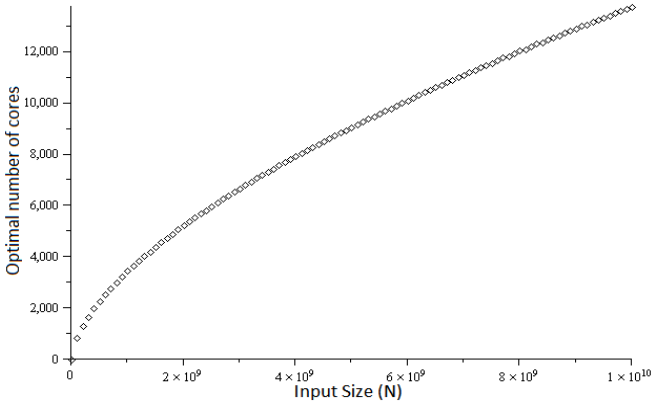

Fig. 23. Parallel MST: optimal number of cores on Y axis and input size on X axis with $k = 500$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$.

Fig. 23 and Fig. 24 plot, respectively, the optimal number of cores and their frequency which minimize cost as a function of input size. We see that the optimal number of cores increases with increasing input size (roughly follows a negative exponential curve with a positive coefficient). Note that for all input sizes, the optimal number of cores obtained for this algorithm is comparatively much smaller than the numbers obtained for both the parallel addition and the LU-factorization algorithms. The frequency of cores required to minimize cost decreases with the input size.

We now consider the sensitivity of this analysis with respect to the ratio k. Fig. 25 and Fig. 26 plot, respectively, the optimal number of cores and the frequency of these cores

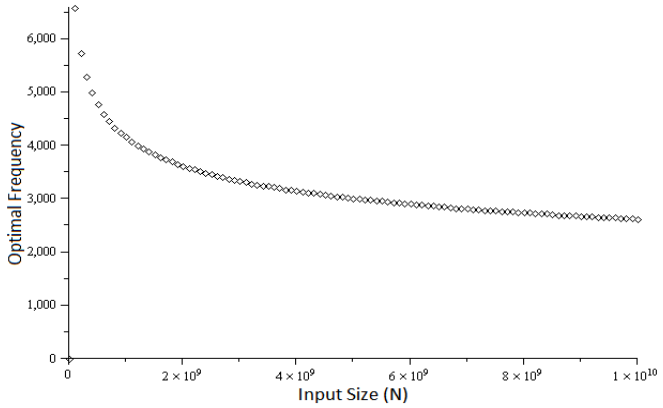

Fig. 24. Parallel MST: Optimal frequency on Y axis and input size on X axis with $k = 500$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$. Frequency on Y axis is plotted in units $F/100000$ where F is the maximum frequency.

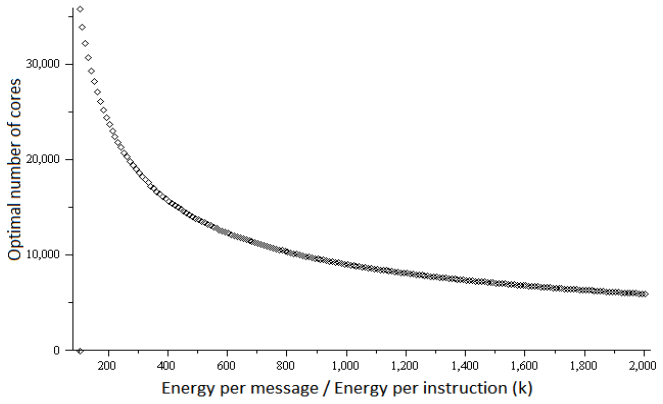

Fig. 25. Sensitivity analysis: optimal number of cores on Y axis and k (ratio of the energy consumed for single message transfer and the energy consumed for executing a single instruction at the maximum frequency) on X axis with $N = 10^8$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$.

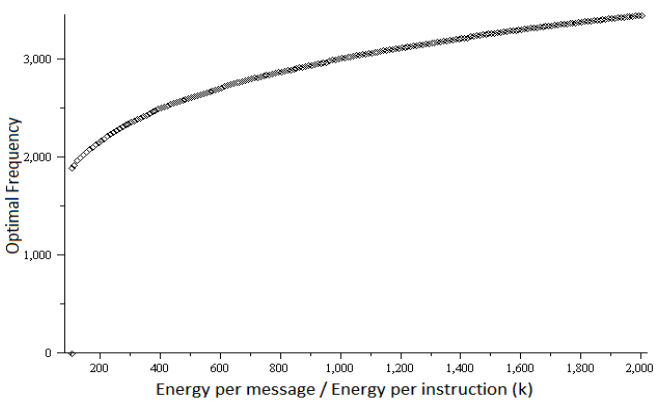

Fig. 26. Sensitivity analysis: Optimal frequency on Y axis and k (ratio of the energy consumed for single message transfer and the energy consumed for executing a single instruction at the maximum frequency) on X axis $N = 10^8$, $\beta = 1$, $K_c = 500$ and $\alpha = 0.1$. Frequency on Y axis is plotted in units $F/100000$.

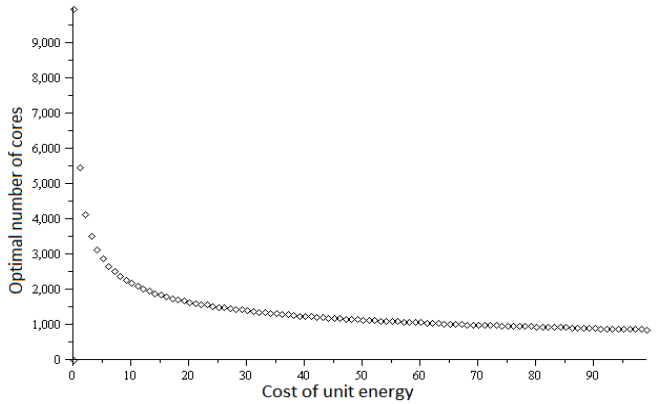

Fig. 27. Sensitivity analysis: optimal number of cores on Y axis and cost of unit energy (α) (measured relative to the cost of running the parallel system for unit time) on X axis with $N = 10^8$, $\beta = 1$, $K_c = 500$ and $k = 500$.

as a function of k (asssuming a fixed input size). We see that, as was the case in both the parallel addition and the LU-factorization algorithms, the optimal number of cores decreases with increasing k. Furthermore, the frequency of cores required to minimize cost increases with increasing k. We observe that this trend remains the same entire range of input sizes considered (10^8 to 10^{10}).

We now consider the sensitivity of this analysis with respect to the ratio α. Fig. 27 and Fig. 28 plot, respectively, the optimal number of cores and the optimal frequency for these cores (fixing the input size and varying k). We note that, as was the case in both the parallel addition and the LU-factorization algorithms, the optimal number of cores and the frequency of these cores required to minimize cost decreases with increasing α. We observe that this trend also remains the same the range of input sizes considered (10^8 to 10^{10}). Based on the above observations, we claim that LU-factorization, parallel addition and Prim's algorithm asymptotically possesses similar optimal configuration characteristics for maximizing utility.

VII. Discussion

As expected, our analysis shows that the optimal configuration (number of cores and their frequency when active) that is required to maximize utility depends on the structure of a parallel algorithm. For example, the optimal number of cores and frequencies of the cores of the parallel addition algorithm, the LU factorization, and the Parallel MST algorithm behave similarly as a function of the input size (i.e.,they have approximately same structure), at least in the range of input sizes we considered. However, for a particular input size, the optimal number of cores and frequencies of the cores may vary significantly with the algorithm. It appears that for the above three parallel algorithms, the optimal number of cores and frequencies of the cores required to maximize utility exhibit the same asymptotic growth.

On the other hand, the two parallel versions of the quicksort algorithm possesses similar characteristics but differ from the other three algorithms. We observe that the optimal number of cores and the frequencies of these cores for both the quicksort algorithms appear to have the same asymptotic growth.

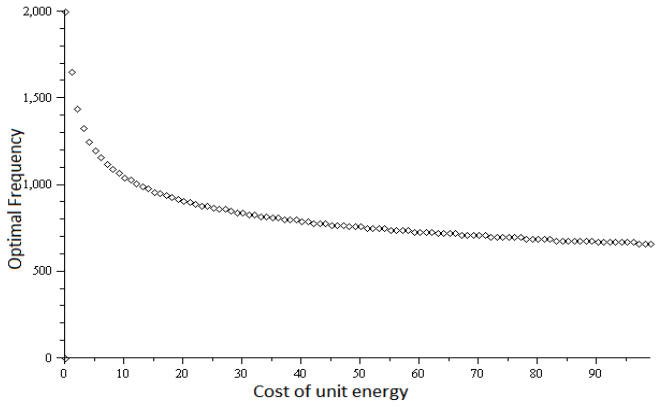

Fig. 28. Sensitivity analysis: Optimal frequency on Y axis and cost of unit energy (α) (measured relative to the cost of running the parallel system for unit time) on X axis $N = 10^8$, $\beta = 1$, $K_c = 500$ and $k = 500$. Frequency on Y axis is plotted in units $F/100000$.

This is an interesting considering that both algorithms have dramatically different performance scalability characteristics.

For purposes of interpreting our results concretely, we fixed the ratio of time required for a message transfer versus that required for a computation cycle (K_c). This ratio will vary depending on the architecture. In general, K_c is in the range 200 to 500 for current architectures. Varying the value of K_c over this range does not affect our results much (i.e., the optimal number of cores and frequencies of the cores do not change much as a result of varying K_c). We also fixed the ratio of the energy required for a message transfer versus that required for a computation cycle (k). However, there is greater uncertainty as to what the value of k should be. Therefore, we studied the sensitivity of our analysis to a wide range of possible values for k. We observe that varying the values of k can dramatically affect the optimal number of cores for some parallel algorithms.

The sort of analysis done in this paper is more similar in spirit to parallel complexity analysis, than it is to performance evaluation on an architecture. However, the analysis could be refined to be closer to some proposed multicore architectures– for example, by modeling a memory hierarchy and using specific values of the constants. One abstract way to do this would be to develop a variant of the BSP model of parallel computation [23], specifically, one that takes into account the fact that for multicore architectures, the memory hierarchy may include a level consisting of shared memory between a small number of cores.

In this paper, we only considered linear utility functions to model the costs associated with the execution of a parallel algorithm. One could generalize our approach to a wide range of convex utility functions (wuch functions are a norm for utilities in economics). Furthermore, in this work, we have analyzed the utilities of existing parallel algorithms. It would be an interesting to design new parallel algorithms which provide maximal utility (minimum cost) for a particular problem.

ACKNOWLEDGMENT

The authors would like thank Soumya Krishnamurthy (Intel) for motivating us to look at this problem. We would also like to thank George Goodman and Bob Kuhn (Intel), and MyungJoo Ham (Illinois) for helpful feedback and comments on this line of research.

REFERENCES

[1] M. P. Mills, "The internet begins with coal," *Green Earth Society, USA*, 1999.

[2] "http://www.greenpeace.org/raw/content/international/press/reports/make-it-green-cloud-computing.pdf," *Greenpeace International*, 2010.

[3] V. A. Korthikanti and G. Agha, "Analysis of parallel algorithms for energy conservation in scalable multicore architectures," in *ICPP*, 2009, pp. 212–219.

[4] ——, "Towards optimizing energy costs of algorithms for shared memory architectures," in *SPAA*, 2010.

[5] A. Chandrakasan, S. Sheng, and R. Brodersen, "Low-power cmos digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 4, pp. 473–484, 1992.

[6] G. Agha and W. Kim, "Parallel programming and complexity analysis using actors," in *Third Working Conference on Massively Parallel Programming Models*, 1997, pp. 68–79.

[7] ——, "Actors: A unifying model for parallel and distributed computing," *Journal of systems architecture*, vol. 45, no. 15, pp. 1263–1277, 1999.

[8] R. Murphy, "On the effects of memory latency and bandwidth on supercomputer application performance," *IEEE International Symposium on Workload Characterization*, pp. 35–43, Sept. 2007.

[9] M. Curtis-Maury, A. Shah, F. Blagojevic, D. Nikolopoulos, B. de Supinski, and M. Schulz, "Prediction Models for Multi-dimensional Power-Performance Optimization on Many Cores," in *PACT*, 2008, pp. 250–259.

[10] J. Li and J. Martinez, "Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors," in *HPCA*, 2006, pp. 77–87.

[11] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi, "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget," in *MICRO*, vol. 9, 2006, pp. 347–358.

[12] S. Park, W. Jiang, Y. Zhou, and S. Adve, "Managing Energy-Performance Tradeoffs for Multithreaded Applications on Multiprocessor Architectures," in *ACM SIGMETRICS*. ACM New York, NY, USA, 2007, pp. 169–180.

[13] R. Ge, X. Feng, and K. Cameron, "Performance-Constrained Distributed DVS Scheduling for Scientific Applications on Power-Aware Clusters," in *ICS*. IEEE Computer Society Washington, DC, USA, 2005.

[14] B. D. Bingham and M. R. Greenstreet, "Computation with energy-time trade-offs: Models, algorithms and lower-bounds," in *ISPA*, 2008, pp. 143–152.

[15] R. Gonzalez and M. A. Horowitz, "Energy dissipation in general purpose microprocessors," *IEEE Journal of Solid-State Circuits*, vol. 31, p. 12771284, 1995.

[16] A. J. Martin, "Towards an energy complexity of computation," *Information Processing Letters*, vol. 39, p. 181187, 2001.

[17] H.-L. Chan, J. Edmonds, and K. Pruhs, "Speed scaling of processes with arbitrary speedup curves on a multiprocessor," in *SPAA*, 2009, pp. 1–10.

[18] A. Wierman, L. L. H. Andrew, and A. Tang, "Power-aware speed scaling in processor sharing systems," in *INFOCOM*, 2009, pp. 2007–2015.

[19] V. A. Korthikanti and G. Agha, "Energy bounded scalability analysis of parallel algorithms," *Technical Report, UIUC*, 2009.

[20] V. Singh, V. Kumar, G. Agha, and C. Tomlinson, "Scalability of Parallel Sorting on Mesh Multicomputer," in *ISPP*, vol. 51. IEEE, 1991, p. 92.

[21] G. Geist and C. Romine, " LU Factorization Algorithms on Distributed-Memory Multiprocessor Architectures," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, p. 639, 1988.

[22] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Co., Inc. Redwood City, CA, USA, 1994.

[23] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.