Energy-Bounded Scalability Analysis of Parallel Algorithms

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

The amount of energy available in some contexts is strictly limited. For example, in mobile computing, available energy is constrained by battery capacity. As multicore processors with a large number of processors, it will be possible to significantly vary the number and frequency of cores used in order to manage the performance and energy consumption of an algorithm. We develop a method to analyze the *scalability* of an algorithm given an energy budget. The resulting *energy-bounded scalability* analysis can be used to optimize performance of a parallel algorithm executed on a scalable multicore architecture given an energy budget. We illustrate our methodology by analyzing the behavior of four parallel algorithms on scalable multicore architectures: namely, parallel addition, two versions of parallel quicksort, and a parallel version of Prim's Minimum Spanning Tree algorithm. We study the sensitivity of energy-bounded scalability to changes in parameters such as the ratio of the energy required for a computational operation versus the energy required for communicating a unit message. Our results shows that changing the number and frequency of cores used in a multicore architecture could significantly improve performance under fixed energy budgets.

I. Introduction

The amount of energy available in some contexts is strictly limited. For example, in mobile computing, available energy is constrained by the battery capacity. As energy costs for computing have continued to increase, it may also be useful to budget energy consumption for a particular application. On a sequential processor, energy consumption can be reduced by lowering the frequency at which the processor runs. It is obvious that lowering the frequency in a uniprocessor is proportional to the performance of an algorithm (i.e., the frequency is inversely proportional to time taken by the algorithm). However, the picture is more complex in the case of parallel processors.

Parallel computing involves some sequential subcomputations, some parallel computation, and communication between nodes. Parallel performance and energy costs are not only dependent on the number of cores (and the frequency at which they operate), they are also dependent on the structure of the parallel algorithm. We provide a methodology to analyze the performance characteristics of an algorithm executed on a parallel computer under a given energy budget.

Parallel computing is typically used to improve performance by dividing the problem into subtasks and executing the subtasks in parallel. By increasing the number of cores, computation at each core is reduced, which in turn improves performance. For parallel algorithms in which there is no communication between the cores, doubling cores halves the computation per core. If the frequency of each core is 0.8 of the original frequency, two cores consume about the same amount of energy as the original core while the overall performance increases by about 60%.

More generally, parallel algorithms involve communication between the cores. In this case, as the number of cores increases, the number of messages between cores also increases. This in turn means that more energy is required for communication. Thus, under a given energy budget, increasing cores leads to a decrease in the amount of energy left for computation at each core, requiring the cores to run at lower speeds. In this paper, we study how to optimize the performance of parallel algorithms by changing how many cores are used and at what frequency, so that the gain in performance from parallelism is maximal.

Note that measuring the energy consumed and the performance of a parallel algorithm does not amount to the same thing. This difference between is due to two important factors:

- There is a nonlinear relationship between power and frequency at which the cores operate in multicore processors. In fact, the power consumed by a core is (typically) proportional to the cube of its frequency.
- Executing parallel algorithms typically involves communication (or shared memory accesses) as well as computation. The energy and performance characteristics of communication and computation may be different. For example, in many algorithms, communication time may be masked by overlapping communication and computation (e.g., see [1]). However, the energy required for communication would be unaffected by whether or not the communication overlaps with the computation.

Scalability of a parallel algorithm measures the relation between performance and the number of cores used. We define *energy-bounded scalability* as a measure of the relation between performance and the number of cores used given a *fixed energy budget*. Specifically, energy-bounded scalability analysis answers the following question: *For a given parallel algorithm and a fixed energy budget, how does the number of cores required to maximize performance vary as a function of input size?*.

Although the current generation of multicore computers has a limited number of cores, industry expects to double the number of cores every 18 months. Thus we are particularly interested in scalable architectures. Current multicore computers are based on shared memory. Scaling such an architecture results in memory access bottlenecks and, as a recent paper by Murphy [11] suggests, increasing cores with a global shared memory in hardware will not improve performance. Therefore, we assume that memory there is no global shared memory; instead, there is a message-passing architecture between cores with local memory.

In order to focus on some essential aspects of the problem, we make a few other simplifying assumptions. We assume that all cores are homogeneous and that cores that are idle consume minimal power. We do not concern ourselves with a memory hierarchy but assume that local memory accesses are part of the time taken by an instruction. Since the time consumed for sending and receiving a message may be high compared to the time consumed *en route* between the cores, we assume that the communication time between cores is constant. We discuss ideas for possible extensions in Sec. VII.

Contributions of the paper: This paper is the first to propose a methodology to analyze energy-bounded scalability. We illustrate our methodology by analyzing different types of algorithms, ranging from algorithms that are embarrassingly parallel to those which have a strong sequential component. Specifically, we analyze tree addition, Prim's minimum spanning tree, and two sorting algorithms. Not surprisingly, the ratio of energy consumed in executing an instruction and the energy consumed in sending a message is critical in determining energy-bounded scalability. For each of our examples, we analyze how sensitive energy-bounded scalability is to the above energy ratio.

II. Related Work

Previous research has studied software-controlled dynamic power management in multicore processors. Researchers have taken two approaches for dynamic power management. Specifically, they have used one or both of two *control knobs* for runtime power performance adaptation: namely, *dynamic concurrency throttling*, which adapts the level of concurrency at runtime, and *dynamic voltage and frequency scaling* [4]–[6], [10], [12]. This work provides a runtime tool which may be used with profilers for the code. By contrast, we develop methods for theoretical analyzing parallel algorithms which can statically determine how to maximize the performance under fixed energy budget.

Li and Martinez develop an analytical model relating the power consumption and performance of a parallel code running on a multicore processors [9]. This model considers parallel efficiency, granularity of parallelism, and voltage/frequency scaling in relating power consumption and performance. However,

the model does not consider total energy consumed by an entire parallel application, or even the structure of the parallel algorithm. Instead, the algorithmic structure (communication and computation) of a parallel algorithm is assumed to represented by a parallel efficiency metric and a generic analysis is used irrespective of the algorithmic structure.

The notion of energy-bounded scalability is in some ways analogous to performance scalability under iso-efficiency as defined by Kumar et al. [8] which is a measure of a algorithm's ability to effectively utilize an increasing number of processors in a multicomputer architecture. Recall that efficiency measures the ratio of the speed-up obtained by an algorithm and the number of processes used. Kumar measures scalability by observing how large a problem size has to grow as a function of the number of processors used in order to maintain *constant efficiency*. By analogy, we can consider our analysis as *performance scalability under iso-energy*.

Wang and Ziavras have analyzed performance energy trade offs for matrix multiplication on a FPGA based mixed-mode chip multiprocessors [14]. Their analysis is based on a specific parallel application executed on a specific multiprocessor architecture. In contrast, our general methodology of evaluating energy scalability can be used for a broad range of parallel applications and multicore architectures.

In [7] we considered a dual problem to the one we are analyzing here: namely, the problem of characterizing energy scalability under iso-performance.[1] Specifically, the analysis in that paper studies how, given an algorithm and a performance requirement, the number of cores required to minimize the energy consumption varies as a function of input size. Obviously, for embarrassingly parallel algorithms, whether one wants to optimize the number of cores for a given energy budget, or the energy for a given performance requirement, it is best to use a maximal number of cores. However, in general, the two analyses will give different results. This is can be understood by appreciating the following difference. Energy scalability under iso-performance minimizes total energy consumed by an algorithm. The total energy consumed is a sum of energy consumed in *all* paths executed by the parallel algorithm. On the other hand, energy-bounded scalability analysis optimizes performance: performance is measured by considering the length of the *longest* path in the execution of a parallel algorithm.

III. Assumptions

Our analysis should be thought of as providing a first order of magnitude value. As a first cut, we make a number of simplifying architectural assumptions, some of which could be relaxed in future work, where a more detailed and specific architectural performance model is used. Specifically, our simplifying assumptions are as follows:

1) All cores operate at same frequency and frequency of the cores can be varied using a frequency (voltage) probe.
2) The computation time of the cores can be scaled (by scaling the frequency of the cores).
3) 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.
4) There is no memory hierarchy at the cores (memory access time is constant).
5) Each core has its own memory and cores synchronize through message communication.

The running time T on a given core is proportional to the number of cycles μ executed on the core. Let X be the frequency of a core, then:

$$T = \textit{(number of cycles)} \times \frac{1}{X} \tag{1}$$

[1]Note that some of the discussion of the related work above is similar to that in the earlier paper. Some of the example algorithms studied in the two papers are also the same, allowing us to compare the results of the analyses.

Recall that a linear increase in voltage supply lead to a linear increase of frequency of the core. Moreover, a linear increase in voltage supply also leads to a nonlinear (in principle cubic) increment in power consumption. While the energy consumed will also be the result of other factors, for simplicity, we model the energy consumed by a core, E, to be the result of the above mentioned critical factor:

$$E = E_c \times T \times X^3 \tag{2}$$

where E_c is some hardware constant (see [2]).

The following additional 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
- M : Number of cores allocated for the parallel application.
- K_c : Number of cycles executed at maximum frequency for single message communication time
- P_s : Static power consumed (i.e., by an idle core).

IV. Methodology

We now present our methodology to evaluate energy-bounded scalability of a parallel algorithm as 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 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 idle time (T_{idle}) at all the cores as a function of frequency of the cores. Scaling the parallel algorithm (critical path) may lead to an increase in idle time in other paths (at other cores).

step 5 Evaluate the total number of computation cycles at all the cores.

Step 6 Frame an expression for the energy consumed by the parallel algorithm as a function of the frequency of the cores, using the energy model discussed above. The energy expression is the sum of the energy consumed by 1) computation, E_{comp}, 2) communication, E_{comm} and 3) idling (static power), E_{idle} where

$$E_{comp} = E_c \cdot (\text{Total number of computation cycles}) \cdot X^2 \tag{3}$$

$$E_{comm} = E_m \cdot (\text{Total number of communication steps}) \tag{4}$$

$$E_{idle} = P_s \cdot T_{idle} \tag{5}$$

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

Step 7 Given an energy budget E, evaluate the frequency X with which the cores should run, as a function of E. Note that both E_{comp} and E_{idle} depend on the frequency of the cores.

Step 8 Express the time taken (inverse of performance) by the parallel algorithm as a function of frequency of the cores:

$$\text{Time Taken} = \text{Number of communication steps} \cdot \frac{K_c}{F} + \text{Number of computation cycles} \cdot \frac{1}{X} \quad (6)$$

where the first term represents the time taken for communication in the critical path and the second term represents the time taken for executing the computation steps in the critical path, at frequency X.

Step 9 Analyze the equation to obtain the number of cores required for maximum performance as a function of input size. In particular, compute the appropriate number of cores that are required to maximize the performance under budget constraints.

A. Example: Adding Numbers

Consider a simple parallel algorithm to add N numbers using M cores. Initially all N numbers are equally distributed among the M cores; at the end of the computation, one of the cores stores their sum. Without loss of generality, assume that the number of cores available is some power of two. The algorithm runs in $\log(M)$ 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, one core will store the sum of all N numbers.

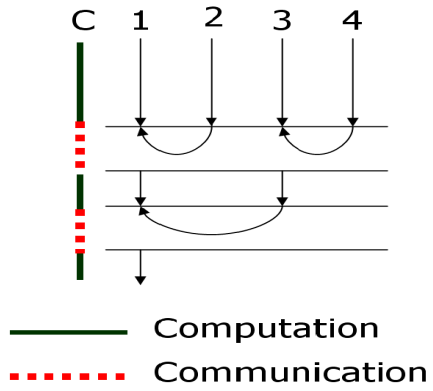

Fig. 1. Example scenario: Adding N numbers using 4 actors; Left most line represents the critical path; embarrassingly parallel application but represents a broad class of tree algorithms

Now we describe the steps needed to evaluate the energy-bounded scalability. 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 $\log(M)$ communication steps and $((N/M) - 1 + \log(M))$ computation steps (step 2).

We next evaluate number of message transfers in total required by the parallel algorithm (Step 3). It is trivial to see that number of message transfers for this parallel algorithm when running on M cores is $(M - 1)$. Note that in this algorithm, the message complexity is only dependent on M and not on the input size N. We now evaluate the total idle time at all the cores, running at frequency X (Step 4). The

total idle time is:

$$T_{idle} = \frac{\beta}{X} \cdot (M(log(M) - 1) + 1) + \frac{1}{F} \cdot K_c \cdot (M(log(M) - 2) + 2) \quad (7)$$

where the first term represents the total idle time spent by idle cores while other cores are busy computing, and second term represents the total idle time spent by idle cores while other cores are involved in message communication. Moreover, observe that the total number of computation steps at all cores is $N-1$ (Step 5).

Now, we frame an expression for energy consumption as a function of the frequency of the cores, using the energy model. (Step 6). The energy consumed for computation, communication and idling while the algorithm is running on M cores at frequency X is given by:

$$E_{comp} = E_c \cdot (N-1) \cdot \beta \cdot X^2 \quad (8)$$

$$E_{comm} = E_m \cdot (M-1) \quad (9)$$

$$E_{idle} = P_s \cdot T_{idle} \quad (10)$$

where β is the number of cycles required per addition.

Given an energy budget E, the frequency X with which the cores should run (step 7) is obtained by solving the resultant cubic equation:

$$E = E_{comp} + E_{comm} + E_{idle} \quad (11)$$

Due to the complex structure of the solution to the cubic equation, we approximate the solution (frequency) as follows:

$$X = \left(\frac{E - E_m \cdot (M-1) - P_s \cdot \frac{K_c}{F} \cdot ((M(log(M) - 2) + 2))}{E_c \cdot (N-1) \cdot \beta} \right)^{1/2} \quad (12)$$

The restriction that $X^2 > 0$ provides an upper bound on the number of cores that can be used to increase the performance, as a function of N, E, F, P_s and K_c.

The time taken (inverse of performance) by the addition algorithm as a function of frequency of the cores X (step 8) is as follows:

$$\text{Time Taken} = \log(M) \cdot \frac{K_c}{F} + ((N/M) - 1 + \log(M)) \cdot \beta \cdot \frac{1}{X} \quad (13)$$

Finally, Step 9 involves analysis of the equation that we have obtained above. We consider this step below.

V. Analyzing Performance Equation

We now analyze the performance expression obtained above for the addition algorithm to evaluate energy-bounded scalability. While we could differentiate the function with respect to the number of cores to compute the minimum, this results in a rather complex expression. Instead, we simply analyze the graphs expressing energy-bounded scalability.

Note that the performance expression is dependent on many variables such as N (input size), M (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), P_s (static power) and the maximum frequency F of a core. 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 $\beta = 1$. We also set idle energy consumed per cycle as $(P_s/F) = 1$, where the cycle is at the maximum frequency F. We express all energy values with respect to this normalized energy value.

In order to graph the required differential, we must make some specific 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 compared to idle energy consumed per cycle. We assume this ratio to be 10, i.e., that $E_c \cdot F^2 = 10 \cdot (P_s/F)$. 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 sending a single message, E_m, and the energy consumed for executing a single instruction at the maximum frequency. Thus, $E_m = k \cdot E_c \cdot F^2$. We fix the energy budget E to be that of the energy required for the sequential algorithm, running on a single core at maximum frequency F and analyze the sensitivity of our results to a range of values of k.

The sequential algorithm for this problem is trivial: it takes $N-1$ additions to compute the sum of N numbers. By Eq. 2, the energy required by the sequential algorithm is given by $E_{seq} = E_c \cdot \beta \cdot (N-1) \cdot F^2$. Fig. 2 plots performance (time taken) as a function of N and M. Substituting E_{seq} for E in Eq. 31), and by considering the restriction on X, an upper bound on the number of cores would be $((N-1)/k \cdot F^2)+1$.

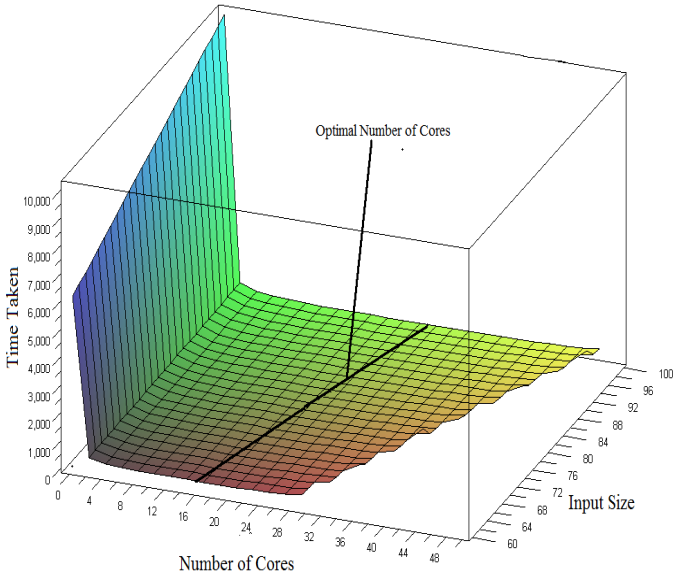

Fig. 2. Addition: performance curve with time taken on Z axis, number of cores on X axis and input size on Y axis with $k = 10$, $\beta = 1$, $k_c = 5$. Time taken is plotted in units $1/F$ where F is the maximum frequency. Number of cores is plotted in units 10^5. Input size is plotted from 6×10^7 to 10^8 in units of 10^6. Black curve on the XY plane is the plot of optimal number of cores required for maximum performance with varying input size. For any input size, the strict upper bound on the number of cores is depicted by the distorted portion at the end of the curve.

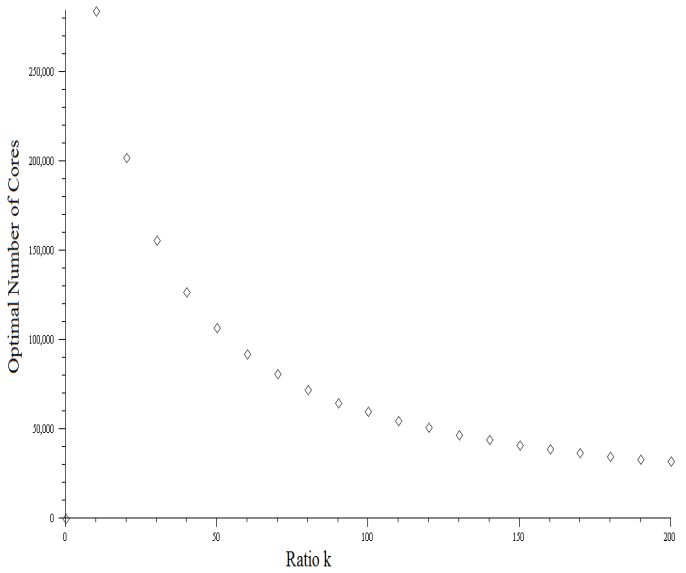

Fig. 3. Sensitivity analysis: optimal number of cores on Y axis, and k (ratio of energy consumed for single message communication to the energy consumed for executing single instruction at maximum frequency) on X axis with input size $N = 10^7$.

Fig. 2 plots time taken (inverse of performance) as a function of input size and number of cores. We can see that for any input size N, initially the time taken by the algorithm decreases with increasing M and later on increases with increasing M. As explained earlier, this behavior can be understood by the fact that performance increases with an increase in number of cores, and energy left out for computation decreases with increasing cores (the difference between the energy budget and the energy used for communication). However, the behavior shows that the optimal number of cores required for maximum performance is in the order of input size. It turns out that compared to both parallel quicksort algorithms (considered later in the paper), the addition algorithm has better energy-bounded scalability characteristics. Furthermore, we can see that increasing the input size leads to an increase in the optimal number of cores. We now consider the sensitivity of this analysis with respect to the ratio k. Fig. 3 plots the optimal number of cores required for maximum performance by fixing the input size and varying k. The plot shows that

for a fixed input size, the optimal number of cores required for maximum performance decreases with increasing k, approximating a c/k curve where c is some constant.

VI. Case Studies

We now analyze two parallel quicksort based algorithms and Parallel Prim's Minimum Spanning Tree (MST) algorithm. Although the two parallel quicksort algorithms have similar *energy* scalability under iso-performance [7], it turns that their energy-bounded scalability graphs are quite different. Note that due to space constraints, we do not provide detailed derivations for the formulas.

A. Naïve Parallel Quicksort Algorithm

Consider a naïve (and inefficient) parallel algorithm for quicksort. Recall that in the quicksort algorithm, an array is partitioned in to two parts based on a pivot and each part is solved recursively. In the naïve parallel version, array is partitioned into two parts by a single core (based on a pivot) and then one of the sub array is assigned to another core. Now each of the cores partitions its arrays using the same approach as above, and assigns one of its subproblems to other cores. This process continues until all the available cores are used up. After this partitioning phase, in the average case, all cores will have approximately equal division of all elements of the array. Finally, all the cores sort their arrays using the serial quicksort algorithm in parallel. Sorted array can be recovered by traversing the cores. Algorithm is very inefficient, as partitioning the array in to two sub arrays is done by single core. Since one core must partition the original array, the runtime of the parallel algorithm is bounded below by array length.

Assume that the input array has N elements and the number of cores available for sorting are M. Without loss of generality, we assume both N (2^a) and M (2^b) to be power of two's. For simplicity of the analysis, 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 of core that initiates the partitioning of the array. The total number of communication and computation steps in the critical path evaluates to $N(1-(1/M))$ and $2N(1-(1/M))+K_q((N/M)\cdot\log(N/M))$, where K_q (1.4) is the quicksort constant.

Next, we evaluate the number of messages transfers required in total by the parallel algorithm (Step 3). It is trivial to see that number of message transfer for this parallel algorithm running on M cores is $\log(M)\cdot(N/2)$. Note that, unlike the previous example, the message complexity for naïve quicksort is dependent both on number of cores and on the input size. We now evaluate the total idle time at all the cores, running at frequency X (Step 5). Total idle time is given by the following equation

$$T_{idle} = \frac{\beta}{X}\cdot N(2M-\log(M)-2)+\frac{1}{F}\cdot K_c\cdot N(M-\log(M)-1) \tag{14}$$

where β is the number of cycles required per comparison.

The first term represents the total idle time spent by idle cores while other cores are busy computing and second term represents the total idle time spent by idle cores while other cores are involved in message communication. Moreover, the total number of computation steps at all cores is $N\cdot log(M)+K_q\cdot N\cdot log(\frac{N}{M})$ (Step 5).

Now, we frame an expression for energy consumption as a function of the frequency of the cores, using the energy model. (Step 6). The energy consumed for computation, communication and idling while the algorithm is running on M cores at frequency X is given by:

$$E_{comp} = E_c\cdot\left(N\cdot log(M)+K_q\cdot N\cdot log(\frac{N}{M})\right)\cdot\beta\cdot X^2 \tag{15}$$

$$E_{comm} = E_m\cdot log(M)\cdot\frac{N}{2} \tag{16}$$

$$E_{idle} = P_s\cdot T_{idle} \tag{17}$$

Given an energy budget E, the frequency X with which the cores should run (step 7) is obtained by solving the resultant cubic equation:

$$E = E_{comp} + E_{comm} + E_{idle} \tag{18}$$

Due to the complex structure of the solution to the cubic equation, we approximate the solution (frequency) as follows:

$$X = \left(\frac{E - E_m \cdot log(M) \cdot \frac{N}{2} - P_s \cdot \frac{K_c}{F} \cdot N(M - \log(M) - 1)}{E_c \cdot \left(N \cdot log(M) + K_q \cdot N \cdot log(\frac{N}{M}) \right) \cdot \beta} \right)^{1/2} \tag{19}$$

In order to achieve performance improvement given the energy budget, we require $0 < X < F$. This restriction provides a lower bound on the input size as a function of M, E and K_c.

Time taken (inverse of performance) by the Naive quicksort algorithm as a function of frequency of the cores X (step 8) is as follows

$$\text{Time Taken} = N(1 - (1/M)) \cdot \frac{K_c}{F} + (2N(1 - (1/M)) + K_q((N/M) \cdot \log(N/M))) \cdot \beta \cdot \frac{1}{X} \tag{20}$$

Finally, Step 9 involves analysis of the equation obtained above. We consider it below.

Energy-bounded Scalability Analysis We use the same assumptions that were used before in the energy-bounded scalability analysis of the parallel addition algorithm. In that analysis, we fix the energy budget E to be that of the energy required for the sequential algorithm, running on a single core at maximum frequency F. Sequential quicksort algorithm performs on average $O(N \log(N))$ comparisons for sorting an array of size N. By Eq. 2, energy required by the sequential algorithm is given by $E_{seq} = E_c \cdot \beta \cdot (K_q \cdot N \cdot log(N)) \cdot F^2$.

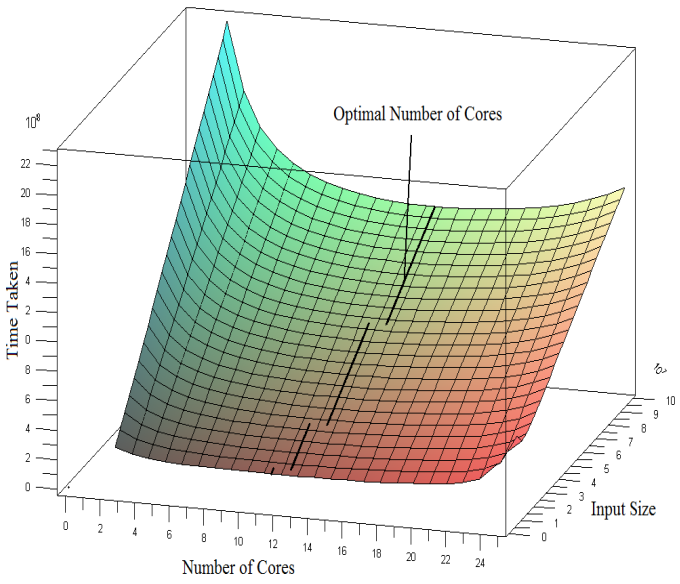

Fig. 4. Naïve Parallel Quicksort: performance curve with time taken on Z axis, number of cores on X axis and input size on Y axis with $k = 10$, $\beta = 1$, $k_c = 5$. Time taken is plotted in units $10^8 \cdot 1/F$ where F is the maximum frequency. Input size is plotted from 10^7 to 10^8 in units of 10^7. Black curve on the XY plane is the plot of optimal number of cores required for maximum performance with varying input size.

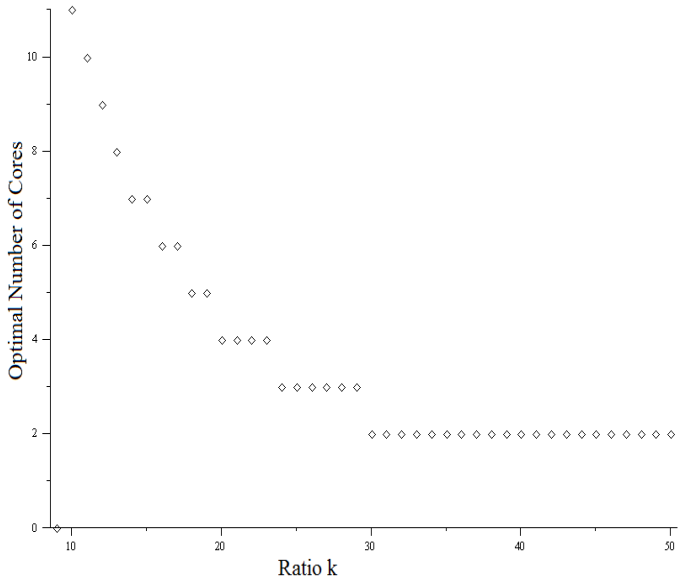

Fig. 5. Sensitivity analysis: optimal number of cores on Y axis, and k (ratio of energy consumed for single message communication to the energy consumed for executing single instruction at maximum frequency) on X axis with input size $N = 10^7$.

Fig. 4 plots performance (time taken) as a function of input size and number of cores. We can see that for any input size N, initially time taken by the algorithm decreases with increasing M and later on increases with increasing M. This behavior is very similar to the curve we obtained for the addition algorithm. However, the optimal number of cores required for optimal performance in this case is far less compared to that of the addition algorithm. The reason for this change is due to the fact that the energy for communication is a function of both the input size and the number of cores. Because the sorting algorithm is communication intensive, given that the energy budget is fixed and we do not control the energy required for communication, the energy remaining for computation is far less compared to the case of the addition algorithm. In other words, naïve quicksort algorithm possesses worse energy-bounded scalability characteristics to that of addition algorithm. Furthermore, we can see that increasing the input size leads to an increase in the optimal number of cores.

We now consider the sensitivity of this analysis with respect to the ratio k. Fig. 5 plots the optimal number of cores required for maximum performance by fixing the input size and varying k. The plot shows that for a fixed input size, the optimal number of cores required for maximum performance decreases with increasing k. Note that the structure of sensitivity curve of the naïve parallel quick sort algorithm is very different from that of the addition algorithm. Note that in the quicksort algorithms, we assumed that messages sent to other cores contain only one element. However, sending multiple elements in a single message may reduce the energy consumption per element. The sensitivity analysis suggests that this will not change the overall shape of the curve.

B. Parallel Quicksort Algorithm

The parallel quicksort formulation [13] works as follows. Let N be the number of elements to be sorted and $M = 2^b$ be the number of cores available. Each cores is assigned a block of N/M elements, and the labels of the cores $\{1, ..., M\}$ defines the global order of the sorted sequence. For simplicity of the analysis, we assume that initial distribution of elements in each 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 all the remaining 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 $\{1...M/2\}$ exchange elements with core $i + M/2$ such that core i retains all the elements smaller than the pivot and core $i + M/2$ retains all elements larger than the pivot. After this step, all the cores $\{1....M/2\}$ stores elements smaller than the pivot and remaining cores $\{M/2 + 1, ...M\}$ stores elements greater than the pivot. Upon receiving the elements, each core merges them with its own set of elements such 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.

Now we perform the energy-bounded scalability analysis for this algorithm. 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 $(1 + N/M) \cdot \log M$ and $(\log(N/M) + N/M) \cdot \log M + K_q(N/M \cdot \log(N/M))$, where K_q (1.4) is the quicksort constant.

The number of message transfer for this parallel algorithm running on M cores is $(M \cdot \log(M) - M + 1) + \log(M) \cdot (N/2)$. Since all the cores are busy all the time T_{idle} (idle time) evaluates to zero. Moreover, the total number of computation steps at all cores evaluates to $((\log N/M + N/M) \cdot \log M + K_q \cdot N/M \cdot \log N/M) \cdot M$ (Step 5).

Now, we frame an expression for energy consumption as a function of the frequency of the cores, using the energy model. (Step 6). The energy consumed for computation, communication and idling while the

algorithm is running on M cores at frequency X is given by:

$$E_{comp} = E_c \cdot \left(\left(\left(\log \frac{N}{M} + \frac{N}{M} \right) \cdot \log M + K_q \cdot \frac{N}{M} \cdot \log \frac{N}{M} \right) \cdot M \right) \cdot \beta \cdot X^2 \quad (21)$$

$$E_{comm} = E_m \cdot \left(M \cdot \log M - M + 1 + \log M \cdot \frac{N}{2} \right) \quad (22)$$

$$(23)$$

Since $T_{idle} = 0$, energy consumed due to idle computation is 0.

Given an energy budget E, the frequency X with which the cores should run (step 7) is obtained by solving the resultant quadratic equation:

$$E = E_{comp} + E_{comm} + E_{idle} \quad (24)$$

and the frequency is as follow:

$$X = \left(\frac{E - E_m \cdot \left(M \cdot \log M - M + 1 + \log M \cdot \frac{N}{2} \right)}{E_c \cdot \left(\left(\left(\log \frac{N}{M} + \frac{N}{M} \right) \cdot \log M + K_q \cdot \frac{N}{M} \cdot \log \frac{N}{M} \right) \cdot M \right) \cdot \beta} \right)^{1/2} \quad (25)$$

In order to achieve performance improvement given the energy budget, we require $0 < X < F$. This restriction provides a lower bound on the input size as a function of M, E and K_c.

Time taken (inverse of performance) by the parallel quicksort algorithm as a function of frequency of the cores X (step 8) is as follows

$$\text{Time taken} = (1 + N/M) \cdot \log M \cdot \frac{K_c}{F} + (\log(N/M) + N/M) \cdot \log M + K_q(N/M \cdot \log(N/M)) \cdot \beta \cdot \frac{1}{X} \quad (26)$$

Finally, Step 9 involves analysis of the equation obtained above. We consider it below.

Energy-bounded Scalability Analysis We use the same assumptions mentioned earlier for the energy-bounded scalability analysis of the parallel addition algorithm. In the analysis, We fix the energy budget E to be that of the energy required for the sequential algorithm, running on a single core at maximum frequency F. In particular, $E_{seq} = E_c \cdot \beta \cdot (K_q \cdot N \cdot log(N)) \cdot F^2$ is substituted for E. Fig. 6 plots performance as a function of N and M.

Fig.6 shows similar trend as seen for the case of addition algorithm. However, for the input range considered, the optimal number of cores required for maximum performance is far less compared to that of the addition algorithm. Also, note that the behavior observed here is different from the case of naïve quick sort algorithm. For the input range considered, the optimal number of cores required for maximum performance for the parallel quicksort algorithm is greater than that of the naïve quicksort version. In other words, the parallel quicksort algorithm possesses better energy-bounded scalability characteristics compared to that of naïve quicksort version. The above observation is in contrast to the fact that both the algorithms have similar energy scalability under iso-performance characteristics, as shown in [7]. We now consider the sensitivity of this analysis with respect to the ratio k. Fig. 7 plots the optimal number of cores required for maximum performance by fixing the input size and varying k. The plot shows that for a fixed input size (10^7), the optimal number of cores required for maximum performance decreases with increasing k. Note that the structure of the sensitivity curve of the parallel quicksort algorithm is very similar to that of parallel addition algorithm (approximates c/k for some constant c). However, the graph is very different compared to the naïve quicksort version.

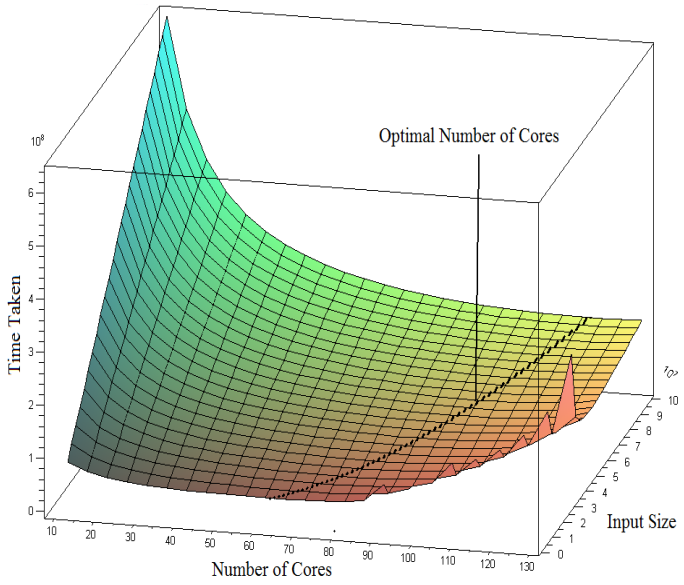

Fig. 6. Parallel Quicksort: performance curve with time taken on Z axis, number of cores on X axis and input size on Y axis with $k = 10$, $\beta = 1$, $k_c = 5$. Time taken is plotted in units $10^8 \cdot 1/F$ where F is the maximum frequency. Input size is plotted from 10^7 to 10^8 in units of 10^7. Black curve on the XY plane is the plot of optimal number of cores required for maximum performance with varying input size.

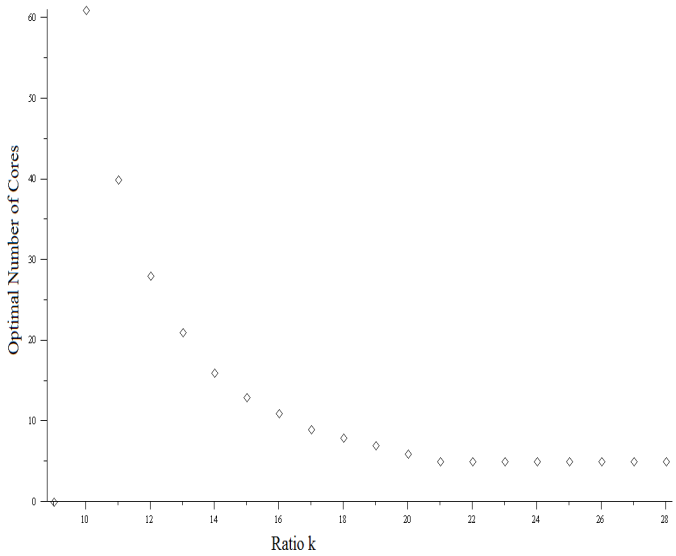

Fig. 7. Sensitivity analysis: optimal number of cores on Y axis, and k (ratio of energy consumed for single message communication to the energy consumed for executing single instruction at maximum frequency) on X axis with input size N = 10^7.

C. Minimum Spanning Tree: Prim's Algorithm

1) 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. Detailed algorithm PRIM_MST(V, E, w, r) is as follows:

In the above program, the body of the while loop(lines 10 - 13) us executed $n - 1$ times. Both the number of comparison performed for evaluating $\min\{d[v]|v \in (V - V_T)\}$ (line 10) and the number of comparison 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 required by the algorithm is around n^2 (ignoring lower order terms). Energy consumed by the algorithm on a single core, running at maximum frequency is given by following equation.

$$E_{seq} = E_c \cdot \beta \cdot N^2 \cdot F^2 \tag{27}$$

where β is number of cycles required for single comparison operation

2) Parallel Algorithm: Here, we consider the parallel version of Prim's algorithm taken from the text book [8]. Let M be the number of cores, and let N be the number of vertices in the graph. The set V is partitioned into M subsets such that each subset has N/M 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

Algorithm 1 PRIM_MST(V, E, w, r)

```
1: V_T = {r};
2: d[r] = 0;
3: for all v ∈ (V − V_T) do
4:    if edge(r, c) exists then
5:       set d[v] = w(r, v)
6:    else
7:       set d[v] = ∞
8:    end if
9:    while V_T ≠ V do
10:      find a vertex u such that d[u] = min{d[v]|v ∈ (V − V_T)};
11:      V_T = V_T ∪ {u}
12:      for all v ∈ (V − V_T) do
13:         d[v] = min{d[v], w(u, v)};
14:      end for
15:   end while
16: end for
```

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 in to 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 in to 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 set V_i of vertices assigned to it.

On average, each core performs about N^2/M 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 $2 \cdot M \cdot N$. Since all the cores are busy all the time T_{idle} (idle time) evaluates to zero. Moreover, the total number of computation steps at all cores on average evaluates to N^2.

Now, we frame an expression for energy consumption as a function of the frequency of the cores, using the energy model. (Step 6). The energy consumed for computation, communication and idling while the algorithm is running on M cores at frequency X is given by:

$$E_{comm} = E_m \cdot 2 \cdot M \cdot N \tag{28}$$

$$E_{comp} = E_c \cdot N^2 \cdot \beta \cdot X^2 \tag{29}$$

Since $T_{idle} = 0$, energy consumed due to idle computation is 0.

Given an energy budget E, the frequency X with which the cores should run (step 7) is obtained by solving the equation:

$$E = E_{comp} + E_{comm} + E_{idle} \tag{30}$$

and the frequency is as follow:

$$X = \left(\frac{E - E_m \cdot 2 \cdot M \cdot N}{E_c \cdot N^2 \cdot \beta} \right)^{1/2} \tag{31}$$

In order to achieve performance improvement given the energy budget, we require $0 < X < F$. This restriction provides a lower bound on the input size as a function of M, E and K_c.

Time taken (inverse of performance) by the parallel Prim's minimum spanning tree algorithm as a function of frequency of the cores X (step 8) is as follows

$$\text{Time taken} = 2 \cdot N \cdot \frac{K_c}{F} + \frac{N^2}{2 \cdot M} \cdot \beta \cdot \frac{1}{X} \tag{32}$$

We use the same assumptions mentioned earlier for the energy-bounded scalability analysis of the parallel addition algorithm. In the analysis, We fix the energy budget E to be that of the energy required for the sequential algorithm, running on a single core at maximum frequency F. In particular, E_{seq} is substituted for E. Since the expression for time taken is relatively simple, we algebraically differentiate the expression with respect to M to evaluate the optimal number of cores required for maximum performance. The optimal number of cores evaluates to $(N \cdot \beta)/(3 \cdot k)$. In particular, the optimal number of cores required for maximum performance is in the range of input size $(\theta(n))$. Thus, the parallel Prim's algorithm possesses good energy-bounded scalability characteristics (similar to that of addition algorithm). The expression for optimal number of cores also shows that for fixed input size the optimal number of cores decreases inversely with increasing k.

VII. Conclusions

The work in this paper is a preliminary step toward understanding how parallel algorithms work under fixed energy budgets, given that the frequency at which cores are run can be scaled. We analyzed four examples which showed different energy-bounded scalability characteristics. The Parallel Prims MST algorithm and the Parallel Addition algorithm show order N energy-bounded scalability (i.e., the number of cores required for optimal performance grows linearly with input size). The Quicksort algorithms have a much lower order of energy-bounded scalability, although the parallel Quicksort has better energy-bounded scalability than the naïve Quicksort. Analyzing a larger number of parallel algorithms may yield insight into how algorithms with different structures can be classified by membership in *energy-bound scalability classes*.

The goal of the present work is to develop an understanding of the gross effect on performance as the number of cores used grow in an energy constrained environment. The sort of analysis done in this paper is more similar in spirit to a parallel complexity analysis of an algorithm, than to its performance evaluation on a real architecture. However, the analysis could be refined to be closer to some architectures by modeling the memory hierarchy. One abstract way to do this would be to develop a variant of the LogP model of parallel computation [3], specifically, taking into account the fact that for multicore architectures, the memory hierarchy is may include a level consisting of shared memory between a small number of cores.

Acknowledgments

This research has been supported in part by the National Science Foundation grant CNS 05-09321. 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.

References

[1] G. Agha and W. Kim, "Parallel programming and complexity analysis using actors," *Massively Parallel Programming Models*, vol. 0, p. 68, 1997.

[2] 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.

[3] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. Von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," *ACM SIGPLAN Notices*, vol. 28, no. 7, pp. 1–12, 1993.

[4] 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 *Proceedings of the 17th International Conference on Parallel architectures and compilation techniques*. ACM New York, NY, USA, 2008, pp. 250–259.

[5] R. Ge, X. Feng, and K. Cameron, "Performance-Constrained Distributed DVS Scheduling for Scientific Applications on Power-Aware Clusters," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Washington, DC, USA, 2005.

[6] 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 *International Symposium on Microarchitecture: Proceedings of the 39 th Annual IEEE/ACM International Symposium on Microarchitecture*, vol. 9, no. 13, 2006, pp. 347–358.

[7] V. A. Korthikanti and G. Agha., "Analysis of Parallel Algorithms for Energy Conservation in Scalable Multicore Architectures," in *International Conference on Parallel Processing (ICPP)*, 2009.

[8] 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.

[9] J. Li and J. Martinez, "Power-Performance Considerations of Parallel Computing on Chip Multiprocessors," *ACM Transactions on Architecture and Code Optimization*, vol. 2, no. 4, pp. 1–25, 2005.

[10] ——, "Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors," in *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, 2006, pp. 77–87.

[11] R. Murphy, "On the effects of memory latency and bandwidth on supercomputer application performance," *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pp. 35–43, Sept. 2007.

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

[13] V. Singh, V. Kumar, G. Agha, and C. Tomlinson, "Scalability of Parallel Sorting on Mesh Multicomputer," in *Parallel Processing: 5th International Symposium: Papers.*, vol. 51. IEEE, 1991, p. 92.

[14] X. Wang and S. Ziavras, "Performance-Energy Tradeoffs for Matrix Multiplication on FPGA-Based Mixed-Mode Chip Multiprocessors," in *Proceedings of the 8th International Symposium on Quality Electronic Design*, 2007, pp. 386–391.