On The Energy Complexity 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

Mark Greenstreet
Department of Computer Science
University of British Colombia
mrg@cs.ubc.ca

***Abstract*—For a given algorithm, the energy consumed in executing the algorithm has a nonlinear relationship with performance. In case of parallel algorithms, energy use and performance are functions of the structure of the algorithm. We define the *asymptotic energy complexity* of algorithms which models the minimum energy required to execute a parallel algorithm for a given execution time as a function of input size. Our methodology provides us with a way of comparing the orders of (minimal) energy required for different algorithms and can be used to define energy complexity classes of parallel algorithms.**

I. Introduction

Energy is a critical concern in computing [1], [17], and specifically in cloud computing [2]. It has previously been observed that energy and performance are related in different ways to input size and the number of cores used. While parallel time complexity provides an appropriate way of comparing the performance of algorithms, there has been no comparable characterization of the energy consumed by a parallel algorithms on an architecture.

In this paper, we define the *asymptotic energy complexity* of parallel algorithms. Given a performance requirement (based on the complexity of the sequential algorithm), the metric characterizes the minimum energy required to execute a parallel algorithm on a parallel architecture, as a function of the input size. In other words, as the input size is increased, the energy spent by a parallel algorithm would grow at least by its asymptotic energy complexity, regardless of how many cores are used or at what speeds they operate. Our methodology provides us with a way of defining energy complexity classes of parallel algorithms.

In the case of sequential algorithms, it is possible to lower the frequency of a core to reduce energy consumption at a cost of increased execution time. However, for a fixed performance target, the amount of energy required is also fixed. Furthermore, both the execution time and the energy consumption of a sequential algorithm are proportional to the total number of operations. Thus, both time complexity and energy complexity are indistinguishable for sequential algorithms.

In the case of parallel algorithms, the precise relation of execution time and energy consumption, with the number of cores used and the frequency at which these cores operate, can be more involved. For example, for certain parallel algorithms and architectures, communication delays are masked by overlapping communication and computation. This can significantly improve performance, although the energy required for communication would be unaffected by such overlapped execution. As with the sequential case, it is always possible to lower the frequency to reduce the energy consumption while increasing execution time. Thus, the energy consumed by a parallel algorithm is meaningful only under a performance bound.

An analysis of performance (inverse of execution time) or energy consumption requires fixing a particular parallel computation model. In cloud computing message passing between nodes is the norm. It is reasonable to assume that as the number of cores increase, multicore architectures are likely to be based on message-passing [18]. We therefore choose a message-passing model of parallel computing for our analysis. We further assume that all cores are homogeneous, and in particular, we assume that they all operate at the same frequency. This assumption is realistic for many architectures as frequency scaling may be achieved by globally controlling the voltage. More importantly for us, the homogeneity assumption keeps the analysis simpler and allows us to focus on the energy complexity of algorithms. We also assume that the 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 consider alternative models of communication time. First, we assume that the communication time between cores is constant; although this is a reasonable assumption in cloud computing, it is unlikely to be true in multicore computers with a very large number of cores. However, constant communication time provides a reasonable approximation 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 then consider a more scalable model of communication time.

Contributions: This paper is the first to propose a method for analyzing the *energy complexity* of parallel software. We illustrate our methodology by analyzing different types of parallel algorithms, specifically, algorithms which have different communication patterns.

II. Related Work

A number of researchers have studied software-controlled *dynamic power management* in multicore processors. The idea is to use some *control knobs* for runtime power performance adaptation. Two such knobs have been proposed: *dynamic concurrency throttling* [5], [15], [19] which enables the level

of concurrency to be changed at runtime, and *dynamic voltage and frequency scaling* [5], [6], [10], [15] which changes how much power is used. Runtime tools supporting dynamic power management work in conjunction with profilers for the code. How accurate such tools can become remains to be seen; in part this depends on the effectiveness of the profilers in determining the structure of the algorithms being executing. On the other hand, we are pursuing a complementary approach which involves theoretically analyzing the energy complexity of parallel algorithms. Our analysis can statically determine how to minimize the energy costs associated with the execution of a parallel application. One advantage of such an analysis 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 reduction.

Bingham and Greenstreet used an ET^{α} metric to analyze energy-time trade-offs for hardware implementations of arithmetic and sorting algorithms using a model inspired by properties of CMOS hardware [3]. Prior to this, various researchers promoted the use of the ET [8] and ET^2 [16] metrics for modeling the trade-offs. These models present an abstraction of voltage/frequency scaling to the system designer to enable reasoning about the overall energy/time trade-offs of the computation. However, there is a significant conceptual gap between these hardware inspired models and the programming paradigms used to code parallel algorithms. Rather than directly modeling the physical limits of CMOS circuits, this paper presents models that reflect current parallel computers and how they are programmed. We define the energy complexity metric for an algorithm to be the minimum energy required for parallel computation as function of the input size under a performance bound. We view both concurrency throttling and voltage/frequency scaling as two orthogonal parameters which affect energy consumption and execution time. We consider the energy lost to leakage currents in the processors, during the execution of a parallel algorithm. This approach enables us to map parallel algorithms to energy classes based on the optimal configuration of parameters for the algorithm.

In our earlier work, we studied the relation between energy and performance for parallel algorithms. The goal of that work was to optimize the performance given an energy budget [12], or to optimize energy given a performance requirement [11], [13]. In this paper, we build on our prior work by developing the notion of energy complexity which reflects the benefit of parallelism for reducing energy asymptotically in parallel applications without loosing on the performance front.

III. Problem Definition and Assumptions

Given a parallel algorithm, an architectural model and a performance requirement, in [11], we defined energy scalability under iso-performance to be the optimal number of cores, as a function of input size, required for minimum energy consumption. In this work, we are interested in knowing how the minimal energy required by a parallel algorithm varies *asymptotically* with input size, under the constraint that the performance of parallel algorithm is not smaller than the optimal (performance) of the corresponding sequential algorithm. We answer this question by analyzing the energy complexity of a parallel algorithm.

Given a parallel algorithm, its energy complexity can be obtained by analyzing energy scalability under iso-performance with the performance being the optimal sequential performance. As discussed in the introduction, we make a few simplifying architectural assumptions. We assume all active cores operate at the same frequency and the frequency of the cores can be varied using a frequency (voltage) probe. A core switches to the *idle* state if there is no computation left at it. For simplicity, we do not model the memory hierarchy at each core. Each core possesses a fast local cache, and cores communicate through a point-to-point interconnection network. The computation and cache access time of the cores can be scaled by varying the frequency of the cores. Communication time between the cores is constant. We justify this assumption by noting that the time consumed for sending and receiving a message is 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:

$$T_{busy} \quad = \quad (\text{number of compute cycles})/X \tag{1}$$

We denote the time for which a given core is active (not idle) as T_{active}. The time taken to transfer a message of size M between two cores is $t_s + t_w \cdot M$, where t_s is the message startup time and t_w is the per-word transfer time.

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

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

where C_L is the effective load capacitance (accounting for activity factors), V is the supply voltage, I_L is the leakage current, and f is the operating frequency. The first term corresponds to the *dynamic power consumption* by an algorithm, while the second term corresponds to its *leakage power consumption*.

In a simple model of CMOS circuits, a linear increase in the power supply voltage leads to a linear increase in the frequency of the core. As seen from Equation 2, such a linear increase in the power supply voltage also leads to a nonlinear (typically cubic) increase in power consumption. Thus, we model the dynamic and leakage energies consumed by a core, to scale cubically and linearly respectively with respect to the operating voltage:

$$\begin{aligned} E_{dynamic} &= E_d \cdot T_{busy} \cdot X^3 \\ E_{leakage} &= E_l \cdot T_{active} \cdot X \end{aligned} \tag{3}$$

where E_d and E_l are some hardware constants [4]. From Equation 1, T_{busy} is inversely proportional to X, thus $E_{dynamic}$ is proportional to X^2. On the other hand, T_{active} can include

the time that a processor is waiting for a message; so, T_{active} does not necessarily scale as X^{-1}, and $E_{leakage}$ and *increase* when X decreases.

Because recent processors have introduced efficient support for low power sleep 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_0}: Number of cycles executed at maximum frequency in message startup time($K_{c_0} = t_s F$);
K_{c_1}: Number of cycles executed at maximum frequency in per-word transfer time($K_{c_1} = t_w F$).

IV. Generic Analysis

This section presents an energy complexity analysis of generic parallel algorithms according to their communication patterns. In subsequent sections, we classify several representative parallel algorithms using the energy complexity metric.

For simplicity, we assume that the communication energy required for transferring a message (word) between any two cores is constant. A more detailed analysis is done in Sec. VII with a more realistic communication model. Also, we do not consider the leakage energy in our initial analyses. However, in Sec. VI, consider leakage and show that leakage energy component has significant impact on the asymptotic behavior of the energy complexity of a parallel algorithm. In all models, we assume that all communication actions are point-to-point.

As mentioned before, we fix the performance requirement to be the time taken by the sequential algorithm at maximum frequency. Let the total number of operations for the sequential algorithm be W. The time taken and the energy consumed by the sequential algorithm are given by $T_{seq} = W/F$ and $E_{seq} = E_d \cdot W \cdot F^2$.

We evaluate the energy complexity of a parallel algorithm in a series of steps. Step 1 involves evaluating the frequency with which the cores should operate such that the parallel performance matches the sequential performance. In other words, we scale the computation time of the critical path to the difference between (a) the required performance and (b) the time taken for message transfers along the critical path. Thus, the new reduced frequency X at which all cores should run is given by:

$$X = F \cdot \frac{\text{Number of computation cycles in the critical path}}{T_{seq} - \text{Communication time of the critical path}} \quad (4)$$

where the critical path is the longest path through the task dependency graph (where edges represents task serialization) of the parallel algorithm.

The second step of our method involves framing an expression for the energy consumed by the parallel algorithm running on P cores at the reduced frequency X. The energy expression has two parts: the energy for computation (E_{comp}) and the energy for communication (E_{comm}). For the algorithms we consider when $P \ll N$, the total number of computation steps at all cores is asymptotically same as the number of computation steps of the sequential algorithm. Thus, the energy consumed by a parallel algorithm is

$$E \quad = \quad E_d \cdot W \cdot X^2 + E_{comm}$$

Now we analyze the above energy expression to obtain the energy complexity of parallel algorithms with three different communication patterns namely, no communication, single message for a constant number of computations and, single message per each core.

Case 1: No communication. Consider the parallel algorithm running on P cores. For simplicity, we assume that computations are equally distributed among the cores i.e, each core has to perform W/P operations and there is no communication between the cores. Energy consumption can be reduced by lowering the frequencies of the cores, so that the parallel performance matches the performance bound (sequential performance). The reduced frequency of the cores and the reduced total energy consumed by the parallel algorithm are

$$X = F/P \; ; \qquad E = E_d \cdot W \cdot F^2/P^2$$

It is trivial to see that the energy consumed by the parallel algorithm under the performance bound decreases with number of cores. However, the number of cores is bounded by the input size. Thus, the optimal number of cores required for minimum energy for this case is N. Thus, the energy complexity for computations with no communication is $O(W/N^2)$.

Case 2: Single message per constant number of computations. In this case, we assume that a core sends a message to all cores for every fixed number of computations. Under the assumption that computations are equally distributed among the cores, each core performs W/P computations and W/k communications for some constant k. Further, we assume that communication time is overlapped by computation time. In other words, we assume that the time taken by the parallel algorithm can be determined by only considering the computations. As in the earlier case, we scale the frequencies of the cores so that the parallel performance matches the performance bound. The reduced frequency of the cores and the total energy consumed by the parallel algorithm is given by

$$X = F/P \; ; \qquad E = E_d \cdot W \cdot F^2/P^2 + E_m \cdot W \cdot P/k$$

For energy reduction, we require E to be less than E_{seq}. This condition gives an upper bound on the number of cores that can be used. The optimal number of cores required for minimum energy for this case is given by

$$P_{opt} \quad = \quad \left(2 \cdot E_d \cdot F^2 \cdot k/E_m\right)^{1/3}$$

Note that the optimal number of cores is not dependent on the input size. Thus, the energy complexity for this case is $O(W/k^{2/3})$.

Case 3: Single message per each core. In this case we assume that on average each core sends a single message. Under the assumption that communications are masked by the computations, the reduced frequency of the cores and total energy consumed by the parallel algorithm is given by

$$X = F/P \; ; \qquad E = E_d \cdot W \cdot F^2/P^2 + E_m \cdot P$$

The optimal number of cores required for minimal energy consumption for this case is given by

$$P_{opt} = \left(2 \cdot E_d \cdot F^2 \cdot W/E_m\right)^{1/3}$$

Thus, the energy complexity of the parallel algorithm is $O(W^{1/3})$ if $P_{opt} < N$ and $O(W/N^2)$ otherwise, where N is the input size.

V. Analysis of Parallel Algorithms

In this section, we analyze various parallel algorithms for their energy complexity and try to classify them with respect to the aforementioned three classes.

a) Parallel Addition: The parallel addition algorithm adds N numbers using P 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.

The sequential algorithm for this problem is trivial: it takes $N-1$ additions to compute the sum of N numbers. By Eq. 1 the running time of the sequential algorithm is given by $T_{seq} = \beta \cdot (N-1) \cdot (1/F)$ where, β denotes the number of CPU cycles required for single addition.

For the parallel algorithm described above, 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. We can see that there are $\log(P)$ communication steps and $((N/P) - 1 + \log(P))$ computation steps.

By Eq. 4, the reduced frequency with which the cores should run so that the parallel running time is same as the running time of the sequential algorithm is given by

$$X \quad = \quad F \cdot \frac{(N/P - 1 + \log(P)) \cdot \beta}{\beta \cdot (N-1) - \log(P) \cdot (K_{c_0} + K_{c_1})}$$

If $N/P \gg \log(P)$ i.e.,$N \gg P \cdot \log(P)$, then $X \approx F/P$.

The number of message transfers for this parallel algorithm when running on P cores is $(P-1)$ (on average a single message per core). Moreover, observe that the total number of computation steps at all cores is $N-1$. The energy consumed by the parallel addition algorithm running on P cores at frequency X is given by:

$$E = E_d \cdot \beta \cdot N \cdot F^2/P^2 + E_m \cdot P \tag{5}$$

The optimal number of cores required for minimum energy consumption is given by

$$P_{opt} = \left(2 \cdot E_d \cdot F^2 \cdot N/E_m\right)^{1/3} = O(N^{1/3})$$

Thus, the asymptotic energy complexity of the parallel addition algorithm is $O(N^{1/3})$. Note that, N should be greater than $P \cdot \log(P)$ for this asymptotic result to apply.

b) Naïve Quicksort: 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 core partitions its array using the same approach as above, and assigns one of its sub-arrays to an idle core. This process continues until all the available cores have received tasks. After the partitioning phase, in the average case, each core will have roughly the same number of elements from the input array. Finally, all of 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), because the initial partitioning of the array into two sub-arrays is done by single core; thus, the execution time of this 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 size sub-arrays by choosing the appropriate pivot (the usual average case analysis).

The sequential algorithm for this problem takes $K_q \cdot N \cdot \log(N)$ comparisons to sort N numbers, where $K_q \approx 1.4$ is the quicksort constant. By Eq. 1 the running time of the sequential algorithm is given by $T_{seq} = \beta \cdot K_q \cdot N \cdot \log(N) \cdot (1/F)$ where, β denotes the number of CPU cycles required for single comparison

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 computation steps in the critical path is $2N(1-(1/P)) + K_q((N/P) \cdot \log(N/P))$. The total number of communication steps and communication messages (words) in the critical path are, respectively $\log(P)$ and $N(1-(1/P))$.

By Eq. 4, the reduced frequency with which the cores should run so that the parallel running time is same as the running time of the sequential algorithm is given by Eq. 6 below.

$$X = F \cdot \frac{(2N(1-(1/P)) + K_q((N/P)\cdot\log(N/P)))\cdot\beta}{\beta\cdot(K_q\cdot N\cdot\log(N)) - (\log(P)\cdot K_{c_0} + N\cdot(1-(1/P))\cdot K_{c_1})} \quad (6)$$

If $2N(1-(1/P)) \ll K_q((N/P)\cdot\log(N/P))$ and $K_q\cdot N\cdot\log(N) \gg \log(P)\cdot K_{c_0} + N\cdot(1-(1/P))\cdot K_{c_1}$, then the frequency X can be approximated by F/P The constraint for N simplifies to requiring N to be exponentially larger than P i.e., $P \ll \log(N)$.

The number of message transfers for this parallel algorithm running on P cores is $\log(P)\cdot(N/2)$. Moreover, the total number of computation steps at all cores is approximately $K_q\cdot N\cdot\log(N)$. The energy consumed by the naïve quicksort algorithm running on P cores at frequency X is given by:

$$E = E_d\cdot\beta\cdot K_q\cdot N\cdot\log(N)\cdot F^2/P^2 + E_m\cdot\log(P)\cdot N$$

The optimal number of cores required for minimum energy consumption is given by

$$\begin{aligned} P_{opt} &= \left(2\cdot E_d\cdot\beta\cdot K_q\cdot F^2\cdot\log(N)/E_m\right)^{1/2} \\ &= O(\log^{1/2}(N)) \end{aligned}$$

and thus the energy complexity of the naïve quicksort algorithm is $O(N\log(\log(N)))$. Again, P should be less than $\log(N)$ for this result to apply.

c) Parallel Quicksort: 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 core is uniform. The algorithm starts with all cores sorting their own set of elements (sequential quicksort). Then 'Core 1' broadcasts via a tree of processes 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 (for $i \in \{1....P/2\}$) stores elements smaller than the pivot, and the 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.

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

By Eq. 4, the reduced frequency with which the cores should run so that the parallel running time is same as the running time of the sequential algorithm is given by Eq.7 below.

If $\beta\cdot K_q\cdot N\cdot\log(N) \gg ((1+\log(P))\cdot K_{c_0} + (\log(P)+(N/P))\cdot K_{c_1})\cdot\log(P)$, then the frequency X can be approximated by F/P. The constraint for N simplifies to requiring $N \gg \max(2^{(K_{c_1}/(\beta\cdot K_q))}, P\cdot\log(P)\cdot K_{c_0}/K_{c_1})$.

The number of message transfers for this parallel algorithm running on P cores is approximately $\log(P)\cdot(N/2)$. Moreover, the total number of computation steps summed over all cores is approximately $K_q\cdot N\cdot\log(N)$. The energy consumed by the parallel quicksort algorithm running on P cores at frequency X is given by:

$$E = E_d\cdot\beta\cdot K_q\cdot N\cdot\log(N)\cdot F^2/P^2 + E_m\cdot\log(P)\cdot N$$

Note that energy expression above is same as the energy expression we derived for the naïve quicksort algorithm. Thus the energy complexity of the parallel quicksort algorithm is also $O(N\log(\log(N)))$. It is interesting to see that both parallel quicksort and naïve quicksort have same energy complexity given that they have different performance characteristics [14]. However, for the naïve quicksort algorithm running on a multicore with a moderate (or larger) number of cores, there are no practical values of N for which the asymptotic results apply. On the other hand, the parallel quicksort algorithm can do well for reasonable values of N as long as the interconnect bandwidth is reasonably large (i.e. K_{c_1} is small).

d) LU factorization: Given a $N \times N$ matrix A, *LU factorization* computes 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 Gaussian 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 Gaussian elimination requires about N^3/3 paired additions and multiplications and about N^2/2 divisions. (For our asymptotic performance analysis we ignore the latter, lower-order term). The time taken by the algorithm on a single core, running at maximum frequency is given by $T_{seq} = (\beta\cdot N^3)/(3\cdot F)$ where β is number of cycles required for a paired addition and multiplication

Parallel Algorithm: There are many parallel algorithms for LU factorization problem. Here we consider only the *coarse-grain 1-D column parallel algorithm* [7] for our performance analysis. Each core is assigned roughly N/P columns of the

$$X = F \cdot \frac{\beta \cdot (N/P) \cdot (K_q \log(N) - (K_q - 1) \log P)}{\beta \cdot (K_q \cdot N \cdot \log(N)) - ((1 + \log(P)) \cdot K_{c_0} + (\log(P) + (N/P)) \cdot K_{c_1}) \cdot \log(P)} \tag{7}$$

matrix and the cores communicate with each other to obtain the required matrix U. The algorithm at each of the P cores is described as follows:

LU Factorization

1: **for** k = 1 to $N-1$ **do**
2: **if** $k \in mycols$ **then**
3: **for** i = $k+1$ to N **do**
4: l_{ik} = a_{ik}/a_{kk} {multipliers}
5: **end for**
6: **end if**
7: broadcast $\{l_{ik} : k < i \leq N\}$ {broadcast}
8: **for** $J \in mycols, j > k$ **do**
9: **for** i = $k+1$ to N **do**
10: $a_{ij} = a_{ij} - l_{ik}a_{kj}$ {update}
11: **end for**
12: **end for**
13: **end for**

In the algorithm, matrix rows need not be broadcast vertically, since any given column is stored entirely by a single processor. However, there is no parallelism in computing multipliers or updating a column. Horizontal broadcasts are required to communicate multipliers for the updates.

On average, each core performs roughly $N^3/(3 \cdot P)$ operations (one addition and one multiplication). Moreover, each core broadcasts about $N^2/2$ matrix elements in N communication actions. We assume that broadcasts for successive steps can be overlapped. By Eq. 4, the reduced frequency with which the cores should run so that the parallel running time is the same as the running time of the sequential algorithm is given by

$$X \quad = \quad F \cdot \frac{(N^3/(3 \cdot P)) \cdot \beta}{(N^3/3) \cdot \beta - (N \cdot K_{c_0} + \frac{N^2}{2} \cdot K_{c_1})}$$

If $(N^3/3) \cdot \beta \gg (N \cdot K_{c_0} + \frac{N^2}{2} \cdot K_{c_1})$, then the frequency X can be approximated by F/P The constraint for N simplifies to $N \gg \max(K_{c_1}/\beta, K_{c_0}/K_{c_1})$.

The parallel algorithm performs a total of $P \cdot (N^2/2)$ message transfers and a total of $N^3/3$ computation steps. This yields a total energy for the parallel algorithm when run on P cores at frequency X of:

$$E \quad = \quad (E_d \cdot N^3 \cdot F^2)/(3 \cdot P^2) + E_m \cdot P \cdot N^2/2$$

The optimal number of cores for minimum energy consumption is given by

$$P_{opt} \quad = \quad \left((4 \cdot E_d \cdot F^2 \cdot N)/(3 \cdot E_m)\right)^{1/3} = O(N^{1/3})$$

and thus the energy complexity of the parallel LU factorization algorithm is $O(N^{7/3})$.

e) Prim's Parallel MST: 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 is a greedy method for finding a MST. 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, where N is the number of vertices in the graph. 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 roughly N^2 (ignoring lower order terms). The time taken by the algorithm on a single core, running at the maximum frequency is given by $T_{seq} = \beta \cdot N^2/F$, where β is number of cycles required to compare the weights of two edges.

Parallel Algorithm: We consider the parallel version of Prim's algorithm in [14]. 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 $d_i[u] = \min\{d_i[v]|v \in ((V \setminus V_T) \cap V_i)\}$ during each iteration of the while loop. The global minimum is then obtained over all $d_i[u]$ by sending the $d_i[u]$ values from each core to core C_0. 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 as 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.

For the parallel algorithm, each core performs about N^2/P comparisons. Each edge to add to the tree is selected by each core finding the best local candidate followed by a global reduction. Under the assumption that reductions and broadcasts happen in a tree fashion, the total number of communication steps at each core is $2 \cdot \log(P) \cdot N$ (ignoring lower order terms).

By Eq. 4, the reduced frequency with which the cores should run so that the parallel running time is same as the running time of the sequential algorithm is given by

$$X \quad = \quad F \cdot \frac{(N^2/P) \cdot \beta}{N^2 \cdot \beta - 2 \cdot \log(P) \cdot N \cdot (K_{c_0} + K_{c_1})}$$

If $N \gg \max(P, 2 \cdot \log(P) \cdot ((K_{c_0} + K_{c_1})/\beta))$, then the frequency X can be approximated by F/P.

The number of message transfers required in total by the parallel algorithm is $2 \cdot P \cdot N$. Furthermore, the total number of computation steps at all cores on average is N^2. The energy consumed by the parallel MST algorithm running on P cores at frequency X is given by:

$$E \quad = \quad E_d \cdot N^2 \cdot F^2/P^2 + E_m \cdot P \cdot 2 \cdot N$$

The optimal number of cores required for minimum energy consumption is given by

$$P_{opt} \quad = \quad \left(E_d \cdot F^2 \cdot N/E_m\right)^{1/3} = O(N^{1/3})$$

and thus the energy complexity of the parallel MST algorithm is $O(N^{4/3})$.

VI. The Effect of Leakage Energy

We now consider the effect of leakage power on the energy complexity of parallel algorithms. While the recent introduction of high-k dielectrics has reduced the magnitude of gate leakage [9], drain-to-source leakage currents remain a significant design concern. Thus, we consider the consequences of including leakage energy in our model for energy-optimal computation.

By Eq. 3, the leakage energy consumed by a parallel algorithm running on P cores at frequency X is given by $E_{leakage} = E_l \cdot T_{active} \cdot X$ where

$$\begin{aligned} T_{active} \quad = \quad & \tfrac{1}{X} \cdot (\text{Total number of computation cycles}) \\ & + \tfrac{K_c}{F} \cdot (\text{Total number of communications steps}) \end{aligned}$$

The first term of this equation represents the total active time spent by all cores that are performing some computation, and the second term represents the total active time spent by all cores during message transfers.

After substitution, the leakage energy $E_{leakage}$ can be expressed:

$$\begin{aligned} & E_{leakage} \; = \\ & \quad E_l \cdot (\text{Total number of computation cycles}) + \\ & \quad E_l \cdot \tfrac{K_c \cdot X}{F} \cdot (\text{Total number of communications steps}) \end{aligned}$$

By our assumption that $P \ll N$, the total number of computation cycles at all cores has the same order as that of the number of computation cycles of the sequential version of the algorithm. As a result, for all parallel algorithms, the first term in the leakage energy expression has a higher complexity than any other component in the energy expression, irrespective of how many cores are used. In other words, the energy complexity of a parallel algorithm would be the same, to within constant factors, as the energy complexity of the corresponding sequential algorithm. Thus, if we assume a model with energy leakage that is proportional to execution time, asymptotically there is no benefit in using an increasing number of cores.

VII. A More Realistic Communication Model

We now consider the effect of using a more realistic model of time and energy required for communication in scalable multicore architectures. We do so by performing a generic analysis, as we have done earlier, except that we use a more realistic terms for communication energy. Specifically, we assume that the energy consumed by a single message transfer between any two cores is proportional to the distance between the cores. For example, in a 2D grid, the average energy required for sending a single message between any two cores in the grid is given by $E_m \cdot \sqrt{P}$. However, as was the case earlier, we assume that the communication time between any two cores is constant.

For the 2D grid communication model, we analyze the energy expression given by Eq. 5 to obtain the energy complexity of parallel algorithms with three different communication patterns, namely, no communication, a single message for a constant number of computations and, a single message for each core.

Case 1: No communication. This is a trivial case: because no communication is involved, the energy complexity is the same as earlier: $O(W/N^2)$.

Case 2: Single message per constant number of computations. If one message is sent for every k operations for some constant k, then the total energy consumed by the parallel algorithm running at reduced frequency $X = F/P$ is given by:

$$E = E_d \cdot W \cdot F^2/P^2 + E_m \cdot W \cdot P^{3/2}/k$$

Parallel Algorithm	Optimal Cores	Energy Complexity	Improvement (sequential/ parallel energy)
Addition	$O(N^{\frac{2}{7}})$	$O(N^{\frac{3}{7}})$	$O(N^{\frac{4}{7}})$
Naïve Quicksort	$\left(\frac{\log(N)}{\log(\log(N))}\right)^{\frac{1}{3}}$	$O(N \cdot \log^{\frac{1}{3}}(N) \cdot (\log(\log(N)))^{\frac{2}{3}})$	$\frac{\log^{\frac{2}{3}}(N)}{(\log(\log(N)))^{\frac{2}{3}}}$
Parallel Quicksort	$\left(\frac{\log(N)}{\log(\log(N))}\right)^{\frac{1}{3}}$	$O(N \cdot \log^{\frac{1}{3}}(N) \cdot (\log(\log(N)))^{\frac{2}{3}})$	$\frac{\log^{\frac{2}{3}}(N)}{(\log(\log(N)))^{\frac{2}{3}}}$
LU Factorization	$O(N^{\frac{2}{7}})$	$O(N^{\frac{17}{7}})$	$O(N^{\frac{4}{7}})$
Prim's MST	$O(N^{\frac{2}{7}})$	$O(N^{\frac{10}{7}})$	$O(N^{\frac{4}{7}})$

TABLE I
ENERGY COMPLEXITY OF PARALLEL ALGORITHMS UNDER 2D GRID COMMUNICATION MODEL

The optimal number of cores required for minimum energy in this case is given by:

$$P_{opt} = \left((4 \cdot E_d \cdot F^2 \cdot k)/(3 \cdot E_m)\right)^{2/7}$$

Note that the optimal number of cores is again not dependent on the input size. Thus, the energy complexity for this case is $O(W/k^{4/7})$ – the dependence on W is the same as for the earlier case, but the sensitivity to the number of computation actions per message, k is slightly *less* than for the earlier model.

Case 3: Single message per each core. For this case, the total energy consumed by the parallel algorithm running at reduced frequency $X = F/P$ is given by

$$E = E_d \cdot W \cdot F^2/P^2 + E_m \cdot P^{3/2} \tag{8}$$

The optimal number of cores required for minimal energy consumption for this case is given by

$$P_{opt} = \left((4 \cdot E_d \cdot F^2 \cdot W)(3 \cdot E_m)\right)^{2/7}$$

Thus, the energy complexity of the parallel algorithm is $O(W^{3/7})$ if $P_{opt} < N$ and $O(W/N^2)$ otherwise, where N is the input size.

Table 1 shows the energy complexity for parallel algorithms considered earlier under this communication model. Observe that all algorithms considered have a better energy complexity than the corresponding sequential algorithms. However, the improvement (sequential energy complexity/parallel energy complexity) obtained in this model is asymptotically smaller than the improvements obtained for the earlier model that disregarded the impact of communication distance. Furthermore, we note that, for each of parallel algorithms considered, as the message complexity increases, parallel energy complexity decreases.

VIII. DISCUSSION

Our analysis shows that the energy consumed by a parallel application can be reduced asymptotically by determining the optimal number of cores and their frequency for a given algorithm, without losing performance. The asymptotic energy reduction depends on the communication and computation structure of the parallel application. For example, asymptotic energy reduction (improvement) of the parallel addition algorithm, the LU factorization, and the Parallel MST algorithm are same – $O(N^{\frac{2}{3}})$.

On the other hand, the two parallel versions of the quicksort algorithm have similar characteristics but differ from the other three algorithms. The two versions of quicksort have the same asymptotic growth, but the algorithm that was designed for parallel processors approaches this asymptote for practical input sizes, whereas the naïve algorithm only requires an input size that is exponentially larger than the number of processors. This illustrates a key contribution of our approach: by identifying constraints on the input size an number of processors required to achieve the asymptotic performance bounds, we our model can distinguish practical algorithms from ones that are of only a purely theoretical interest.

Our analysis shows that leakage energy can have a dramatic effect on the energy complexity of parallel algorithms. Specifically, we found that there is no asymptotic energy reduction possible if leakage energy is significant.

We observed that message complexity plays a major role in determining the energy complexity. Specifically, our model quantifies how higher message complexities result in lower improvements in the parallel energy complexity with respect of the sequential energy complexity. In particular, when the effect of communication distance was considered, the energy savings of parallel computation were reduced for all of the algorithms that we considered. Thus, we expect that an appropriate model for communication time and energy will be essential for any practical analysis of algorithms for highly parallel computers.

Our analysis could be extended to be closer to some proposed multicore architectures. One abstract way to do this would be to develop a variant of the BSP model of parallel computation [21], 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.

IX. ACKNOWLEDGMENT

This material is based in part on research sponsored by the Air Force Research Laboratory and the Air Force Office of Scientific Research, under agreement number FA8750-11-2-0084. The U.S. Government is authorized to reproduce and

distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

REFERENCES

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

[2] A. Berl, E. Gelenbe, M. Di Girolamo, G. Giuliani, H. De Meer, M. Q. Dang, and K. Pentikousis, "Energy-efficient cloud computing," *The Computer Journal*, vol. 53, no. 7, pp. 1045–1051, 2010.

[3] B. D. Bingham and M. R. Greenstreet, "Modeling energy time trade-offs in vlsi computation," *IEEE Transactions on Computers*, 2011 (to appear).

[4] 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.

[5] 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.

[6] R. Ge, X. Feng, and K. W. Cameron, "Performance-constrained distributed dvs scheduling for scientific applications on power-aware clusters," in *SC*, 2005, p. 34.

[7] 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.

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

[9] Intel, "Intel and core i7 (nehalem) dynamic power management," 2008.

[10] 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.

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

[12] ——, "Energy bounded scalability analysis of parallel algorithms," *Technical Report, UIUC*, 2009.

[13] ——, "Towards optimizing energy costs of algorithms for shared memory architectures," in *SPAA*, 2010, pp. 157–165.

[14] 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.

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

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

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

[18] 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.

[19] 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.

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