Understanding Energy Behaviors of Thread Management Constructs

Gustavo Pinto
Federal University of Pernambuco
ghlp@cin.ufpe.br

Fernando Castor
Federal University of Pernambuco
castor@cin.ufpe.br

Yu David Liu
SUNY Binghamton
davidL@binghamton.edu

Abstract

Java programmers are faced with numerous choices in managing concurrent execution on multicore platforms. These choices often have different trade-offs (*e.g.*, performance, scalability, and correctness guarantees). This paper analyzes an additional dimension, *energy consumption*. It presents an empirical study aiming to illuminate the relationship between the choices and settings of *thread management constructs* and energy consumption. We consider three important thread management constructs in concurrent programming: explicit thread creation, fixed-size thread pooling, and work stealing. We further shed light on the energy/performance trade-off of three "tuning knobs" of these constructs: the number of threads, the task division strategy, and the characteristics of processed data. Through an extensive experimental space exploration over real-world Java programs, we produce a list of findings about the energy behaviors of concurrent programs, which are not always obvious. The study serves as a first step toward improving energy efficiency of concurrent programs on parallel architectures.

1. Introduction

IT energy consumption keeps rising steeply in spite of advances in many areas [2]. Energy-efficient solutions are highly sought after across the compute stack, with more established results through innovations of hardware/architecture [14–16, 19, 33, 35], operating systems [11, 24, 30, 39], and runtime systems [7, 31, 38]. In recent years, there is a growing interest in studying energy consumption from higher levels of the compute stack, such as program analysis [4, 13], programming models [3, 6, 18, 32, 34], and applications [29, 40]. A higher-level study is often endowed with a broader application space. For example, programming model solutions can bring energy-aware programmers into energy optimization. Despite their promise, few language-level or application-level energy-efficient solutions address concurrent software running on parallel architectures [4, 10, 31, 37]. This is unfortunate for at least two reasons: (1) thanks to the proliferation of multicore CPUs, concurrent programming is a standard practice in modern software engineering [36]; (2) a CPU with more cores (say 32) often consumes more power than one with fewer cores (say 1 or 2). Energy optimization over programs on such platforms has the potential to yield larger savings, but may also face more challenges [15, 16].

We believe a first step to optimize energy consumption of concurrent programs is to gain a comprehensive understanding of their energy behaviors. This paper presents an empirical study to illuminate and understand energy behaviors of concurrent programs on multi-core architectures. In particular, our study is unique in its focus on how programmer decisions — the choices and settings of thread management constructs — may impact energy consumption and its close relative, performance. Our research is motivated by the following questions:

- **RQ1.** Do alternative *thread management constructs* have different impacts on energy consumption?
- **RQ2.** What is the relationship between *the number of threads* and energy consumption?
- **RQ3.** What is the relationship between *task division strategies* and energy consumption?
- **RQ4.** What is the relationship between *data* volume/access and energy consumption?

To answer **RQ1**, we select three thread management constructs influential in concurrent language design:

- *Explicit threading* ("the Thread style"): programmers manually map logically independent units of work to threads, *i.e.,* the scheduling unit of the virtual machine and/or the underlying operating system. Explicit threading is the most widely used approach in Java multi-threaded programming [36].
- *Thread pooling* ("the Executor style"): programmers create a pool of threads — often fixed in size — and further submit logically independent units of work to the thread pool. The relationship between threads and the units of work is often 1:n. Threads select and execute submitted units of work from a centralized buffer managed by the language runtime. In Java, this mechanism is known as

executors and is part of the `java.util.concurrent` library.

- *Work stealing* ("the ForkJoin style"): similar to thread pooling, programmers also create a pool of operating system threads and submit logically independent units of work to the pool. What is unique to work stealing is that each thread maintains its own buffer of units of work. When one such buffer becomes empty, its maintaining thread may "steal" work from other threads. Its incarnation in Java is the ForkJoin framework [21].

Given these constructs, our investigation is further aimed at how their settings —"tuning knobs" of concurrent programming for programmers — may impact energy consumption. Among them, the number of threads and the size of data are two classic knobs, addressing the dual control vs. data aspects of concurrency. Their respective impacts on energy consumption are focuses of our study. *Tasks*, *i.e.*, logically independent units of work, have an intimate relationship with both. Just as the Executor and ForkJoin styles indicate, the ratio between the number of tasks and the number of threads is a design consideration of concurrent programmers. When the number of tasks increases while the size of data remains the same, each task will process a smaller "slice" of data, *de facto* tuning *task granularity*. We call the programmer job of dividing work to achieve desirable task granularity *task division*. The impact of task division strategies on energy consumption is another focus of our study.

Our study produces a list of findings, many of which are not obvious. We summarize them in Section 5, at the end of each **RQ**'s discussion. We now highlight two of them.

First, our study reveals the context-dependent nature of the energy behaviors of thread management constructs. Each thread management construct has its own "15 minutes of fame." Despite the highly complex landscape, some patterns do seem to recur. For example, as the number of threads for running a concurrent program continues to increase, we observe its energy consumption often increases first, and then decreases later, a phenomenon we term the Λ *curve*. The shape of the curve differs significantly from the one that describes performance (execution time).

Second, our experiments further demonstrate that "faster" is not a synonym with "greener" for concurrent programs: for multi-threaded programs, a faster program generally does not consume the least energy. For example, one benchmark we analyzed achieved a speedup of 9.5x when running with 32 threads, while its energy consumption grew 1.97x. A corollary of this observation is that performance as an indicator to estimate energy consumption is unreliable at best for multi-threaded Java programs. Moreover, we have observed that, for some of the benchmarks, a sequential variant is the one with the lowest overall energy consumption. On the other hand, our experimental results suggest that, except for embarassingly serial applications, effective use of the thread management constructs achieves the best compromise in terms of performance and energy consumption.

Throughout our exploration, a recurring theme is to illuminate the intricate relationship between energy and performance. There exists a rich literature on this topic [4, 6, 15, 16, 31, 32]. We enrich existing work by offering a programming-level perspective.

This paper makes the following contributions:

1. It describes an empirical study — the first of its kind to the best of our knowledge — to correlate energy behaviors of concurrent programs with thread management constructs and their knobs.
2. It conducts an extensive experimental exploration that involves a combination of factors, ranging from thread management constructs, the number of threads, task division strategies, task granularity choices, data sizes, and data access characteristics. The exploration carves out a landscape that involves thousands of distinct points in the experiment space. In addition, the paper describes a preliminary study on the stability and portability of our results under different settings of heap size, garbage collection, just-in-time compilation, and platforms.
3. It offers insights into energy behaviors of real-world concurrent Java programs, with a detailed list of often non-obvious findings.

2. Related Work

Studying energy efficiency of concurrent programs at the application/language level is an emerging direction. Most of the existing work concentrates on energy behaviors in the presence of synchronization. Park *et al.* [27] developed several synchronization-aware runtime techniques to balance the trade-off between energy and performance. Gautham *et al.* [10] studied the relative energy efficiency of synchronization implementation techniques (such as spin locks and transactions). A recent short paper [22] called for energy management based on different synchronization patterns, a concrete instance of which based on futures has been formally defined [23]. Trefethen and Thiyagalingam [37] surveyed energy-aware software, including multi-threaded programs with different workload settings. Bartenstein and Liu [4] designed a data-centric approach to improve energy efficiency for multi-threaded stream programs. Ribic and Liu [31] designed an algorithm to improve the energy efficiency of the work-stealing runtime of Intel Cilk Plus through managing the relative speed of threads. In an recent study [28], some of us have scratched the surface of this problem. Findings as faster is not greener and different thread management constructs have different impact on energy consumption were also discussed in this paper. However, the current study greatly extends the previous one. Under this backdrop, our work is unique in its focus on the im-

pact of programming models for managing thread execution and program design choices on energy consumption.

There are many approaches for energy management of multi-threaded programs at the architecture- and OS-levels. Examples in the former category include investigating the impact of Dynamic Voltage and Frequency Scaling on multi-core architectures [16], meeting power budget based on hardware performance counters [15], and leveraging hardware heterogeneity [19] and processor topology [33]. Examples in the latter include studying the impact of energy consumption based on workloads [11], thread schedules [24, 39], and thread migration [30]. Our work and related work cited here are complementary. Together, they attempt to understand energy behaviors of multi-threaded programs through the perspectives of different levels of the compute stack.

More broadly, there is a growing interest in understanding and managing energy consumption from software-centric approaches. Tiwari *et al.* [35] correlated energy consumption with CPU instructions. Vijaykrishnan *et al.* [38] and Farkas *et al.* [7] performed two early studies on the energy consumption of the JVM. More recently, Hao *et al.* [13] designed a dynamic analysis to estimate energy consumption of Android bytecode. They also observed that there is no strong correlation between performance and energy consumption. Within the programming language community, it is an active area of research to design energy-aware programming languages, with examples such as Eon [34], Green [3], EnerJ [32], Energy Types [6], and LAB [18]. None of these software-centric energy management approaches focuses on multi-threaded programs.

Performance analysis of multi-threaded Java programs has a long history, leading to a rich literature we cannot cite in full. In recent years, there are numerous results based on the DaCapo benchmark suite [5] for this purpose. Kalibera *et al.* [17] conducted a comprehensive study on the benchmarks in Dacapo itself.

Lea [21] described the work stealing algorithm implemented by Java's ForkJoin framework. Work stealing was popularized by the Cilk language [8], and there is a growing interest in designing multi-threaded language runtimes with work-stealing thread management [12, 20, 31].

Earlier versions of Java use *green threads* [26]. The term is unrelated to energy consumption; it refers to VM-managed threads. After Java 1.3, green threads have been replaced by native threads, where programmer-created threads are directly mapped to OS threads.

3. Programming Patterns for Thread Management

We use an (overly) simplified version of the `sunflow` benchmark [5] to illustrate the distinct programming patterns of the three thread management constructs. The code snippets are in Figure 1, 2, and 3 respectively. The three parame-

```
class Main {
  int counter = 0; int coords[DATAN];
  void main() {
    for (int i = 0; i < THREADN; i++)
      (new Bucket()).start();
  }
  class Bucket extends Thread {
    ...
    public void run() { while(counter<DATAN){ dowork
        (); }}
    public void dowork() {
      int start;
      synchronized (Main.this) {
        if (counter >= DATAN) return;
        start=counter; counter+=DATAN/TASKN;
      }
      for (int j = 0; j < DATAN/TASKN; j++) {
        render(coords[start + j]);
      } //end for
}}}
```

Figure 1. (Simplified) Concurrent Programming in Thread style

```
class Main {
  int counter = 0; int coords[DATAN];
  void main() {
    ExecutorService es = Executors.
        newFixedThreadPool(THREADN);
    for (int i = 0; i < TASKN; i++)
      es.execute(new Bucket());
  }
  class Bucket extends Thread {
    ...
    public void run() { dowork(); }
}}
```

Figure 2. (Simplified) Concurrent Programming in Executor style

```
class Main {
  int counter = 0; int coords[DATAN];
  void main() {
      (new ForkJoinPool(THREADN)).submit(new Bucket
          (1));
  }
  class Bucket extends RecursiveAction {
    ...
    int taskcounter;
    Bucket(int taskcounter) {
      this.taskcounter = taskcounter;
    }
    public void compute() {
      if (taskcounter <= TASKN) {
        (new Bucket(taskcounter+1)).fork();
        dowork();
      }
}}}
```

Figure 3. (Simplified) Concurrent Programming in ForkJoin style

ters related to **RQ2**-**RQ4** are `THREADN` for the number of threads (**RQ2**), and `TASKN` for the number of tasks (**RQ3**), and `DATAN` for the data size (**RQ4**), respectively. `sunflow` is a rendering algorithm (ray tracing) where coordinates are stored in array `coords` and method `render` takes one coordinate to render. The rendering logic is encompassed in a method called `dowork`. The coordinate to be processed next is indicated by an index named `counter`, which is accessed from within a `synchronized` block. For brevity, the code snippets here omit the body of the `render` method, and further omit program logic unrelated to our discussion here, such as post-rendering processing (typically performed through placing a barrier at the end of the `main` function).

In the Thread style, the program explicitly bootstraps `THREADN` threads, through messaging the `start` method of a `Bucket` object, whose class is a subclass of the JDK `Thread` class. The `run` method of the `Bucket` class (an inner class of `Main` in the example) is executed by each bootstrapped thread. Here, each thread continuously processes tasks through a busy `while` loop, and each task is defined as executing an instance of `dowork`. Since there are `TASKN` tasks, each task will work on a "slice" of coordinates of size `DATAN/TASKN`.

In the Executor style, `THREADN` threads are created in a fixed-size thread pool, managed by an instance of the `ExecutorService` class of the JDK. The inner class `Bucket` now only encompasses a task and its `run` method only executes the `dowork` method (definition identical to that in Figure 1) once. In the `main` method, `TASKN` tasks will be managed by the pool of `THREADN` threads. The submission for management is achieved through the use of the `execute` method of the `ExecutorService` object.

The ForkJoin style is similar to the Executor style in that a fix-sized pool – the `ForkJoinPool` object – will manage `THREADN` threads. Unlike Executor however, ForkJoin adopts a work stealing algorithm to manage threads. Instead of submitting all tasks to a centralized service such as in Executor, each thread under a work-stealing scheduler maintains its own deque for tasks. A thread running out of tasks (a *thief*) will "steal" a task from the deque of another randomly selected thread (a *victim*). Programs written in work-stealing languages or language frameworks often demonstrate distinct patterns, usually involving recursively dividing work into smaller tasks. For our program, the thread pool of the `main` method is only `submit`'ed with one (initial) task, a `Bucket` task subclassed from the JDK class `RecursiveAction`. A thread in the pool will pick up the task, *i.e.,* run its `compute` method. As we can see, the `compute` method may further `fork` new tasks "on the go," where forking can be viewed as placing the task on the thread's own deque. Such a task in turn may either be picked up by the current thread, or be stolen and picked up by other threads in the pool. Work stealing is a classic yet sophisticated algorithm, with subtleties detailed in prior work [8, 21].

In the rest of the paper, we manually refactor each benchmark into the three programming patterns. Figures 1, 2, and 3 serve as examples of what we view as "comparable" programs in our benchmarking process. We routinely fix two of the three parameters — `THREADN`, `TASKN`, `DATAN` — and observe the impact on energy/performance when the 3rd parameter varies.

4. Experiment Setup

In this section we describe the benchmarks that we analyzed, the infrastructure and the methodology that we used to perform the experiments.

4.1 Benchmarks

We use a variety of benchmarks for evaluation. Benchmarks 1-3 are from a Debian-based language benchmark suite [1]. Benchmark 4 and 5 were developed by us. The rest of the benchmarks are from the well-known DaCapo benchmark suite [5].

1. `knucleotide`: This benchmark receives a DNA as the input stream. It then extracts the DNA sequence, and writes the code and percentage frequency for all the 1-nucleotide and 2-nucleotide sequences, and counts all the 3-, 4-, 6-, 12- and 18-nucleotide sequences. Finally, it writes the counts and codes for specific sequences. This benchmark is memory-intensive because it employs string manipulation intensively. It does not scale up well when running with a high `DATAN`. This benchmark does not have synchronization points, but uses one atomic variable.
2. `mandelbrot`: A mandelbrot is a mathematical set of points whose boundary is a distinctive and easily recognizable two-dimensional fractal shape. Mandelbrot set images are created by sampling complex numbers and determining for each one whether the result tends towards infinity when a particular mathematical operation is iterated on it. According to its website, this benchmark spends 99% of the time using CPU, and uses IO only to print the results. This benchmark does not have synchronization points, but uses one atomic variable.
3. `spectralnorm`: The spectral norm is the maximum singular value of a matrix. Intuitively, one can think of it as the maximum 'scale', by which the matrix can 'stretch' a vector. It is also CPU-intensive, and scales up well in multicore machines. This benchmark synchronizes threads using a barrier, and uses one atomic variable.
4. `largestimage`: This IO-intensive benchmark performs a recursive search into the file system, looking for image files. During traversal, it keeps track of the number of

[1] `http://benchmarksgame.alioth.debian.org`

image files it encountered and the largest among them. This benchmark has two synchronization points and is strongly IO-bound.

5. `n-queens`: This puzzle can be summarized as problem to place N chess queens on an NxN chessboard so that no two queens attack each other. It is a computationally intensive, CPU-bound problem. This benchmark does not have synchronization points, but uses one atomic variable.
6. `sunflow`: renders a set of images using ray tracing[2].
7. `xalan`: transforms XML documents into HTML.
8. `h2`: executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application.

We selected the benchmarks based on their diverse characteristics. For instance, according to a recent study [17], `sunflow` scales well when the number of CPU cores increases, `h2` scales rather poorly, and `xalan` is the middle-of-the-road benchmark in terms of scalability. Benchmark `largestimage` is I/O-intensive, `knucleotide` is memory-intensive, and benchmarks `mandelbrot`, `n-queens`, and `spectralnorm` are CPU-intensive.

For the benchmarks, `DATAN` represents the size of the text file for `knucleotide`, the size of the vector for both `mandelbrot` and `spectralnorm`, the size of a matrix for `n-queens`, the number of directories for `largestimage`, the size of an image for `sunflow`, the number of converted files for `xalan`, and the number of database transactions for `h2`.

4.2 Experimental Environment

Unless noted otherwise, all experiments were conducted on a machine with 2×16-core AMD Opteron 6378 processors (Piledriver microarchitecture) running Debian 3.2.46-1 x86-64 Linux (kernel 3.2.0-4-amd64) and 64GB of DDR3 1600 memory and Oracle HotSpot 64-Bit server VM, JDK version 1.7.0_11, build 21. All experiments were performed with no other load on the OS. We conform to the default settings of both the OS and the JVM. Several default settings are relevant to this context: (1) the power management of Linux is the default *ondemand* governor, which dynamically adjusts CPU core frequencies based on system workloads. (2) For the JVM, the parallel garbage collector is used, and just-in-time (JIT) compilation is enabled. The initial heap size and maximum heap size are set to be 1GB and 16GB respectively. We run each benchmark 10 times within the same JVM; this is implemented by a top-level 10-iteration loop over each benchmark. The reported data is the average of the last 3 runs. We justify this decision in Section 6.

[2] The description for the DaCapo benchmark were taken directly from the DaCapo website: http://www.dacapobench.org/

Energy consumption is measured through current meters over power supply lines to the CPU module. Data is converted through an NI DAQ and collected by NI LabVIEW SignalExpress with 100 samples per second and the unit of the current sample is *deca-ampere* (10 ampere). Since the supply voltage is stable at 12V, energy consumption is computed as the sum of current samples multiplied by $12 \times 0.01 \times 10$. We measured the "base" power consumption of the OS when there is no JVM (or other application) running. The reported results are the measured results *modulo* the "base" energy consumption.

5. Study Results

In this section, we report the results of our experiments. Results for **RQ1** and **RQ2** are presented in Section 5.1, which describes the impact of different thread management constructs in the presence of varying numbers of threads. In Section 5.2 we attempt to answer **RQ3** by investigating the impact of different task division strategies. Finally, in Section 5.3 we present answers to **RQ4** by exploring different data characteristics.

5.1 Energy Behaviors with Alternative Programming Abstractions and Varying Numbers of Threads

In this group of experiments, we fix the number of tasks and the size of the data, and study how variations on the number of threads and the choice of different thread management constructs impact energy consumption. The results of our experiments are presented in Figure 4. Here, the odd rows are energy consumption results, whereas the even rows are the corresponding performance results.

The Λ Curve. One interesting observation throughout our study is that energy consumption typically increases as the number of threads increases, and then gradually decreases as the number of threads approaches the number of CPU cores. In the energy consumption figures, the curves typically display a Λ shape, which we term the Λ *curve*. Nearly all benchmarks display the Λ curve.

We believe the Λ curve results from a combination of multicore processor characteristics and program performance traits. Under the default setting of the ondemand governor, power management modules of multicore CPUs work in an "adaptive" fashion: when a particular core stays idle, the operating frequency of the core will be dynamically adjusted to a lower level. When a 32-core CPU is only loaded with 4 threads for instance, a large number of cores will operate on the lowest frequency (the specific number of cores is likely to be slightly more than 4, because of the running of VM/OS threads). It is standard knowledge that power consumption is reduced when the operating frequency is lower. For that reason, a program running 4 threads is likely to consume less power than one running 8 threads. This helps us explain the / part of the Λ curve.

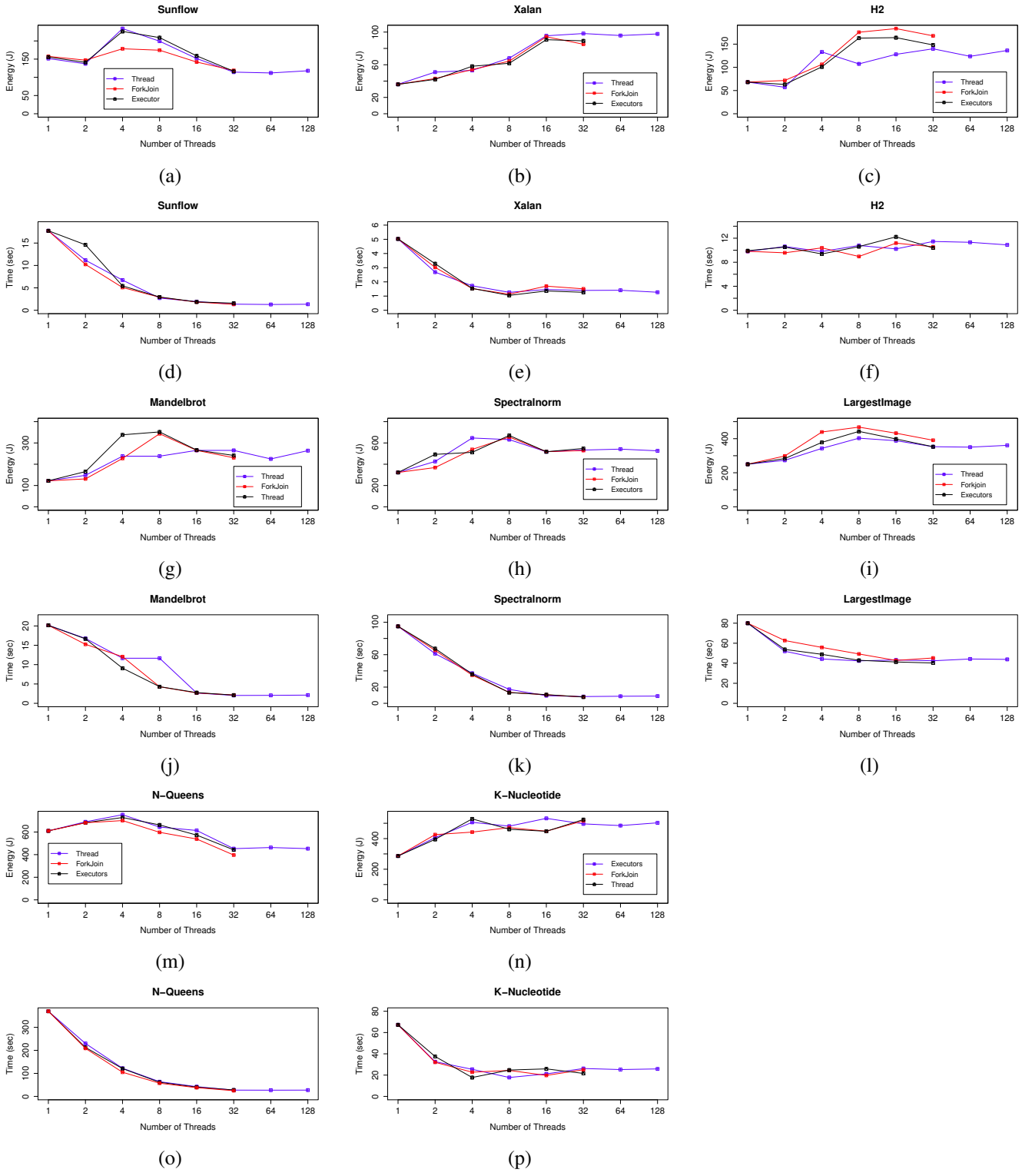


Figure 4. Energy/Performance with Alternative Programming Abstractions and Varying Numbers of Threads

To see why energy consumption often decreases after the initial increase, note that energy consumption, by definition, is the multiplication of power and time. As more threads are used, program execution time tends to shorten. The extent of the drop — the $\backslash$ part of the Λ curve — is determined by the increase in performance (and thus decrease in time) and the increase in power consumption. The greater the ratio between speedup and increase in power, the steeper the $\backslash$ part of the curve will be.

The specific shape details of the Λ curve, including the "peaking" point and the slope of the increase/decrease, are application-specific. Take `sunflow` and `h2` for example. Power consumption for the two benchmarks is 55.9 W and 49.8 W on average, respectively, when using 1 thread, and 152.0 W and 55.1 W when using 32 threads. Execution time is 18.74 and 5.23 seconds, respectively, when using 1 thread, and 1.55 and 5.71 seconds when using 32 threads. Since the power consumption for `sunflow` increases about 3x and performance improves 12x, energy consumption significantly decreases. For `h2` however — a benchmark known to scale rather poorly as the number of cores increases — power consumption increases 1.03x but performance decreases 1.09x. Thus, in this extreme case, the $\backslash$ part of the curve does not exist.

Embarrassingly Parallel vs. Embarrassingly Serial. Our selection of benchmarks range from "embarrassingly parallel" ones (`spectralnorm` and `sunflow`), to "leaning parallel" (`xalan` and `knucleotide`), to "leaning serial" (`mandelbrot` and `largestimage`), to "embarrassingly serial" ones (`h2`). The performance results of three Dacapo benchmarks — Figure 4(d)(e)(f) — are consistent with recent studies (*e.g.*, [17]).

We find the (more) embarrassingly parallel benchmarks are likely to "peak" earlier on the Λ curve, *i.e.*, reaching the highest energy consumption with the smallest number of threads. For example, `sunflow`'s Λ curve peaks at 4 threads, whereas `xalan` peaks at 16. We think this is reasonable: the speed-up of `sunflow` is almost 8x when the number of threads increases from 1 to 8 (linear speedup), so the reduction in performance can quickly offset the increase in power consumption early on. In comparison, `xalan` produces a 5x speedup with the same variation in threads and its performance does not improve with more threads. Hence, its Λ curve peaks later.

Faster $\neq$ Greener. In most of our benchmarks, additional threads would initially lead to improved performance; see Figure 4(d)(e) for example. Following the / part of the Λ curve however, the energy consumption increases as the number of threads increases initially. Furthermore, for 6 of our 8 benchmarks, the lowest energy consumption was achieved by the sequential (1 thread) version. Being "faster" clearly has little correlation with being "greener" for concurrent programs on multicore architectures.

Moreover, since benchmarks "peak" at different parts of the Λ curve, it is not possible to generalize that an improvement in time could be seen as an improvement in energy, and *vice versa*.

Which Programming Style Should I Use? As Figure 4 shows, it is possible to detect differences in the amount of energy used when different concurrent programming abstractions are employed. For some benchmarks, this difference is small, *e.g.*, `xalan` in Figure 4(b)(e). However, the difference is more noticeable in others. Every programming abstraction may have its "15 minutes of fame." In one configuration of `sunflow`, ForkJoin outperforms Thread and Executor by reducing energy consumption by 30%, as shown in Figure 4(a). In one configuration of `h2` however, ForkJoin underperforms Thread and Executor by increasing energy consumption by 50%, as shown in in Figure 4(c). Our experiments do show that there are scenarios where one style is more likely to outperform the others, which we summarize now.

First, the Thread style consistently saves more energy than Executors and ForkJoin in I/O- and memory-bound benchmarks, such as `largestimage` and `h2`. In particular, tasks in the `largestimage` benchmark frequently employs expensive blocking I/O primitives to read from the disk. This scenario is not favorable for the ForkJoin style because it prevents tasks from being stolen and thus suppress load balancing. Moreover, the more a ForkJoin worker attempts unsuccessfully to acquire a task, the more computational resources are wasted. The Executor style performs slightly better, but still consumes 12.33% more energy than the Thread style with 8 threads. One possible reason is that an Executor needs to manage a queue of worker threads. Updates to the queue are protected from clients by a lock, thus increasing synchronization costs when a new task is submitted. Such overhead does not exist in the Thread style.

Second, the energy consumption of the ForkJoin-style programming is sensitive to the degree of parallelism latent in the benchmarks. It outperforms the other two strategies when the benchmarks are embarrassingly parallel (Figure 4(a)), but underperforms the other two strategies in the presence of more serial benchmarks (Figure 4(c)). We believe this can be explained through the nature of the work stealing algorithm: it excels through balancing the deques of individual threads. For benchmarks involving significant serial portions, synchronization (such as a barrier) is often used during the execution of a task. The work stealing algorithm is oblivious to such intra-task synchronizations, but the impacts of stealing a task with long synchronization delays and one without are clearly different. In other words, the natural strength of work stealing in balancing tasks among threads is broken in such benchmarks.

Another ForkJoin characteristic is that it performs better than Thread and Executor when the benchmark needs to join during execution or after completion, such as `xalan`

and `spectralnorm`. The reason is that `Thread.join()` is a blocking operation. However, when one ForkJoin worker is blocked waiting to join a stolen task, the worker could (i) execute some task that it would be running if the steal had not occurred, or (ii) unless there are already enough live threads, the framework may create or re-activate a spare thread to compensate for blocked joiners until they unblock.

Energy-Performance Trade-offs. An energy-related question arises when we move from single-threaded programming to multi-threaded programming, or from 16 threads to 32 threads. One well-known metric to evaluate the energy/performance trade-off is the Energy-Delay Product (EDP): the product of energy consumption and performance. We compute the EDP for the benchmarks, with selected results presented in Fig. 5 [3], where a smaller EDP value indicates the more favorable trade-off.

We observed that a parallel execution is generally more favorable for energy-performance trade-offs than its single-threaded counterpart. This is particularly true for embarrassingly parallel programs: the EDP for `sunflow` with 32 threads is only 5.8% of its single-threaded execution. The degree of improvement on EDP appears to be in sync with the potential of parallelism in applications, and for specific benchmarks, increasing the number of threads is most likely *not* aligned with the improvement of EDP. For instance, when the number of threads increases for `xalan` from 8 to 16, EDP for all three programming constructs deteriorates significantly. The most unfortunate case among our benchmarks is perhaps `h2`. As the number of threads increases, the benchmark produces no gain in performance, but its energy consumption triples. As a result, EDP degrades as we move from sequential to parallel execution.

Overpopulating Cores with Threads. For the Thread style of thread management, we have also constructed experiments where the number of threads goes beyond the number of cores. In all experiments, we did not notice significant change in energy consumption, but there is often a small but noticeable increase in execution time. We speculate this results from the overhead of thread management and context switching. For instance, in the `sunflow` benchmark, the number of context switches increases 3.57 times when varying from 32 to 128 threads, as Figure 6 shows.

The almost negligible difference suggests that the JVM and the OS are very well-versed in handling cases where threads outnumber cores. We choose not to perform experiments over the case where there are more threads than CPU cores for Executor and ForkJoin styles. The "comparable" (Section 3) implementation would create a thread pool that outnumbers the number of cores. We do not believe that is the intended use for these thread management constructs.

[3] Throughout the rest of the paper, we select a subset of benchmarking results to illustrate ideas. Additional graphs can be found in the Appendix.

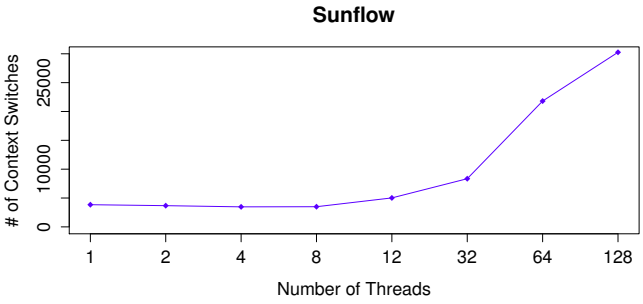


Figure 6. Context Switches and Thread Overpopulation

RQ1 Summary*: Different thread management constructs have different impacts on energy consumption. For I/O-bound programs, the Thread style exhibits the best energy consumption, whereas the ForkJoin style has the worst. For embarrassingly parallel benchmarks, the opposite holds.*

RQ2 Summary*: The relationship between energy consumption and the number of threads often forms the Λ curve. Being faster is not synonymous with being greener. Sequential execution often leads to the least energy consumption, whereas parallel execution leads to improved energy/performance trade-off for non-embarassingly-serial programs.*

5.2 Energy Behaviors and Task Division Strategies

In this section, we fix the number of threads and the size of data, and study how the variations on the number of tasks have effects on energy consumption. To thoroughly explore the experimental space, we further refine our benchmarks into two versions: a *task-centric* division strategy and a *data-centric* division strategy.

In the former, we directly fix the number of tasks in the fashion of Figures 2 and 3. In the later, we set a *sequential threshold* to data, *i.e.*, the size of data a task will work on, and not explicitly set the number of tasks. The two strategies lead to different programming patterns, but they are indeed two sides of the same coin for task granularity: given the overall data, fixing the number of tasks will implicitly set the data size per task, whereas fixing (sequential cutoff) data size will implicitly determine the number of tasks.

Task Granularity with Task-Centric Division. In this style, we divide the work based on `TASKN`. Figure 7 demonstrates the effect of task granularity on `xalan` benchmark. We observed a similar energy consumption behavior in the other benchmarks. Here the data suggests remarkable uniformity: the number of tasks submitted/executed as logically independent units of work has little impact on energy consumption, independently of the thread management construct.

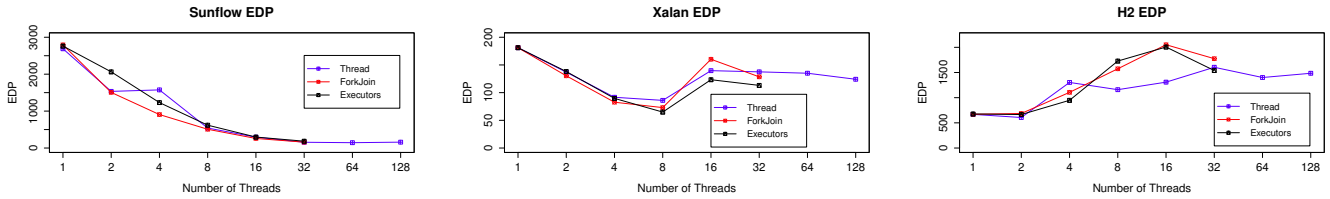


Figure 5. EDP for sunflow, xalan and h2

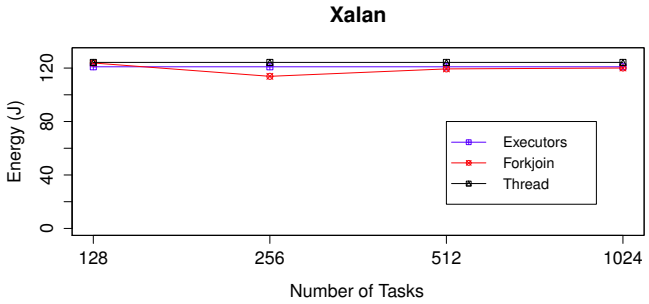


Figure 7. Energy and Task-Centric Division

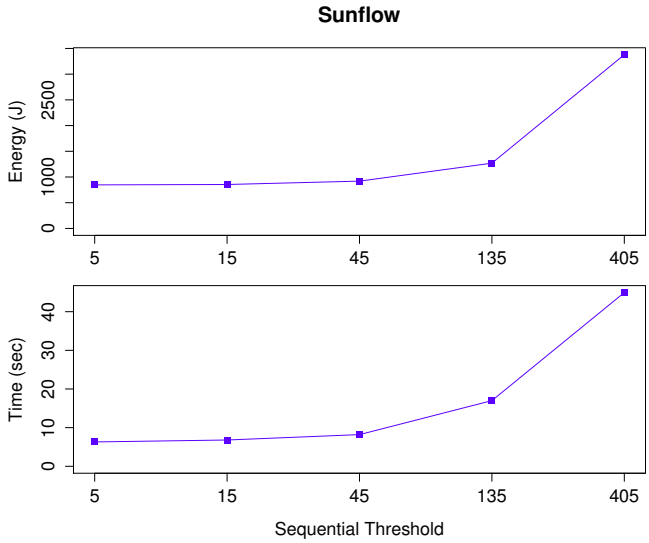


Figure 8. Energy/Performance and Data-Centric Division

At first glance, the results may be disappointingly "boring." We believe, however, that no news is news. The data reveals that task granularity matters little to energy consumption of concurrent programs. For instance, we have at a point increased the number of tasks to 1024, for a benchmark whose overall `DATAN` is 2048. In other words, every task only takes 2 pieces of data. In this case, no noticeable energy consumption increase was observed. Its version in the Executor style submits 1024 tasks to the `ExecutorService` and its version in the ForkJoin style recursively creates 1024 `RecursiveAction` objects. Though such programming patterns may appear to be "extreme," our experiments show they place little burden on energy consumption.

Task Granularity with Data-Centric Division. Under a data-centric approach, ForkJoin can also be seen as a divide-and-conquer algorithm, where in each recursive call two new tasks are spawned until a certain threshold is reached. Using this approach, Figure 8 shows the energy/performance behavior of different sequential threshold configurations for the `sunflow` benchmark. We now discuss three observations. We choose not to perform experiments of the case using Thread and Executors styles because their programming patterns do not naturally fall into the divide-and-conquer style as ForkJoin does.

First, energy consumption and performance both increase when the sequential threshold changes from 135 to 405, a 2.66x increase in energy consumption. Performance has a similar increase of 2.64x. In this example, we used 1024 as our `DATAN`, which creates an array with 2048 positions. Thus, when we use 405 as our sequential threshold, the benchmark creates only 6 tasks and operates on 6 cores. With about 82% of the cores sleeping, the benchmark is not able to take advantage of the multiprocessors, and both performance and energy consumption suffer. With the sequential threshold set at 135, the program operates on 16 cores. As we have discussed earlier, the energy consumption of running `sunflow` on 16 cores is only 1.21x higher than when using 32 cores.

Second, the overhead of scheduling a high number of tasks does not seem to impact energy consumption. This phenomenon appears to recur in all benchmarks.

Third, energy consumption and performance do not always increase in sync. For example, there is a small energy consumption variation (7.85%) when the sequential threshold changes from 5 (1023 tasks) to 45 (127 tasks). Performance, on the other hand, degrades 23.8%.

One possible reason is that, when tasks are coarse-grained, it is less likely that a ForkJoin thief will steal a task, because the total number of available tasks decreases. Thus, after few unsuccessful attempts, the processor goes idle and the average power consumption decreases. Yet, when we move from 135 to 405, we also move from using 16 to using 6 processors. Then, as explained in terms of the Λ curve,

energy consumption is higher when not using all processors available (the / part).

Asymmetric Workload. So far, we have created tasks where the data is divided uniformly. Another important characteristic to take into consideration is the use of asymmetric workloads. With different amounts of work, some ForkJoin workers will finish their work faster than others. Hence, the likelihood of steals may increase. Figure 9 shows the average number of steals per task granularity in the presence of symmetric load, a random asymmetric work division, and an 80-20 asymmetric work division.

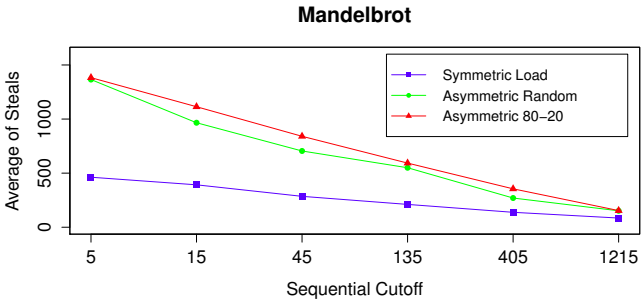


Figure 9. Number of steals per task granularity.

The figure shows that the number of steals is strongly correlated to the symmetric vs. asymmetric nature of task workloads. Further, the number of steals is also correlated to task granularity: the smaller the tasks, the greater the number of steals. Still, we observed an average energy savings of 3.26% using asymmetric workloads. We have experienced similar results in CPU-bound benchmarks, as Figure 10 shows.

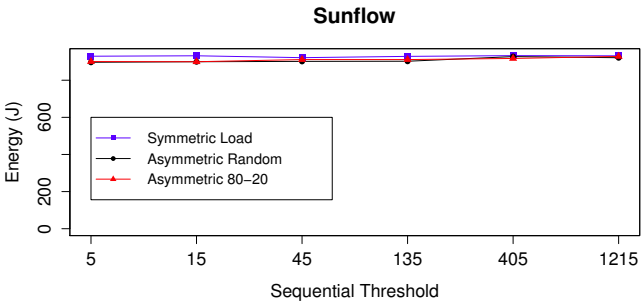


Figure 10. Energy Savings With Asymmetric Workload.

The Width of Forking. The ForkJoin-style can be configured to divide the work into n desirable tasks, instead of two per recursive call, which we term the *width* of forking. We have analyzed 4 different forking widths. For the `sunflow` benchmark in Figure 11, we observed a negligible difference of energy consumption from 2 to 4 forks, and from 4 to 8 forks per recursive call (about 0.96% and 1.21% respectively). From 8 forks to 16 forks, however, we observed an increase of 5.78% over the total energy consumption, and a similar increase in the execution time of 5.67%. This result is consistent with the other benchmarks.

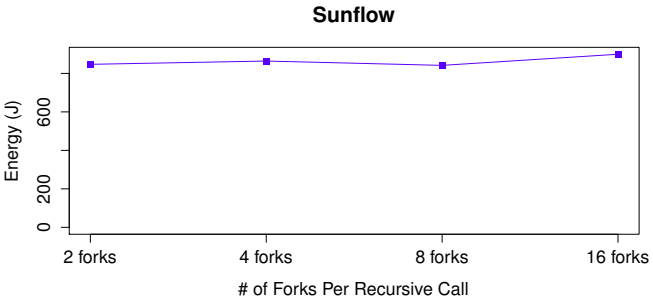


Figure 11. Energy and Forking Width

***RQ3 Summary**: In task-centric division, the granularity of tasks appears to matter little to energy consumption. In data-centric division, asymmetric workloads in ForkJoin are more energy-friendly, and excessive forking width can lead to increased energy consumption.*

5.3 Energy Behaviors and Data

We now focus on **RQ4**, studying the impact of data — its size and access patterns — on program energy behaviors.

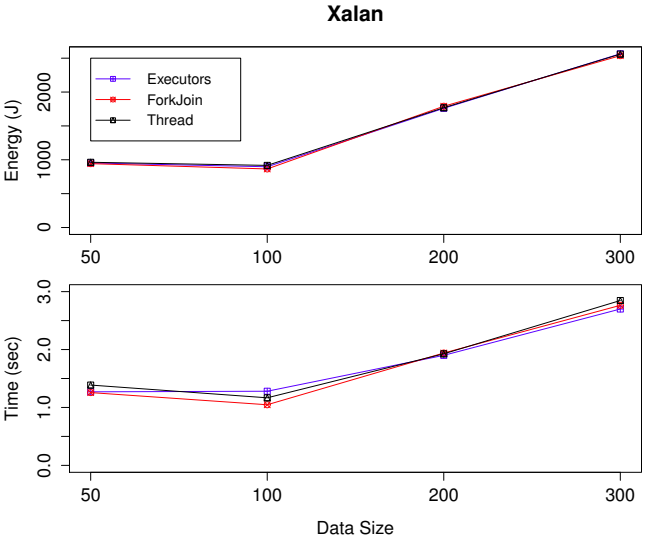


Figure 12. Energy and Performance Varying Data Size

Data Size. Fixing the number of threads and the number of tasks, we now study how the variations on data size have effect on energy consumption. Figure 12 shows the energy behavior for the `xalan` benchmark, where the analogous `DATAN` value (as used in examples Fig. 1/2/3) represents the number of XML files to be converted. `THREADN` and `TASKN` were fixed at 32 and 256, respectively.

As predicted, energy consumption increases when a larger number of files are processed. Observe however, the

increase in energy consumption is not necessarily linear to data size. Generally speaking, the precise relationship is application-specific: it depends on the algorithm complexity relative to the data size. In cases of data-parallel benchmarks, one phenomenon we observe is that the curve is often *convex*, especially for the part of the curve where the data sizes are relatively small. Take `xalan` for instance. When data size increases from 50 to 100, the energy consumption and performance remain almost unchanged. We think this has to do with the programming pattern itself. In data-parallel programs, there is usually a barrier at the end of data processing, and performance is determined by the slowest processing thread. When overall data size is small, the execution time of processing each data "slice" is also small. Variations on processors and scheduling may contribute to a larger proportion on the progress of individual threads, and differences in data size may be masked. When data size increases, the masking effect is reduced.

In `xalan`, the energy behaviors with the 3 thread management constructs are nearly identical, but there is a detectable difference in performance for the three constructs, with ForkJoin taking the least time and Executor taking the most. Since execution time is the accumulated effect of power over time, this indicates ForkJoin is likely to have completed the task faster with a higher power consumption. Work stealing systems are most known for their ability for load balancing, where CPU core idling is reduced, improving performance while presenting fewer opportunities for cores to fall into lower power modes. This phenomenon is reduced when data size becomes larger, because data processing time would be proportionally larger, reducing the effect for frequent steals.

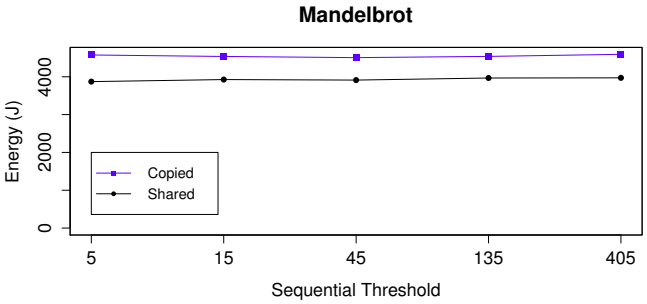


Figure 13. Energy and Data Sharing Strategies

Data Sharing vs. Copying. We now study how memory-intensive tasks may impact energy consumption. In this set of experiments, we change the ForkJoin versions of our benchmarks to so that the tasks operate on copies of the data, instead of working in-place. Many ForkJoin tasks operate on an indexable data structure, with subtasks operating on contiguous partitions of this data structure. As part of the recursive step, it is often necessary to split the input data structure into two smaller ones on which the subtasks can operate. Such data structures are often not mutated during the process, so the standard programming practice is to pass the start/end indices, leaving the data structure shared. For our experiments, we change these benchmarks, so that the part of data structure operated by subtasks are *copied*. Given an array-based representation, each recursive call in this scenario will create n new arrays, where n is the width of forking. In Figure 13 we compare both approaches using the `mandelbrot` benchmark.

As the figure shows, we experienced an energy consumption increase of 15.38%. The performance counterpart of this figure in the appendix shows a performance loss of 20.85% when copying is performed. Observe, however, that copying has a more severe impact on performance than energy. This is indeed natural: when long-latency main memory request is issued, the issuing cores can often be reduced to a lower frequency, and a lower level of power consumption.

```
public void compute() {
   ...
   NQueensSolver[] tasks = new NQueensSolver[size
       ];
   for (int i = 0; i < tasks.length; i++) {
      int[] newElements = new int[depth + 1];
      System.arraycopy(currentElements, 0,
                       newElements, 0, depth);
      tasks[i] = new NQueensSolver(newElements);
      tasks[i].fork();
   }
   ...
   for (int i = 0; i < tasks.length; i++) {
      if(tasks[i] != null) tasks[i].join();
   }
}
```

Figure 14. `ForkJoin`: Spreading Out Data Copying

```
public void compute() {
   ...
   List<NQueensSolver> tasks = new ArrayList<>(
       size);
   for (int i = 0; i < tasks.size(); i++) {
      int[] newElements = new int[depth + 1];
      System.arraycopy(currentElements, 0,
                       newElements, 0, depth);
      tasks.add(new NQueensSolver(newElements));
   }
   ...
   invokeAll(tasks);
}
```

Figure 15. `ForkJoin`: Aggregating Data Copying

Data Locality. Next, we investigate the impact of data locality on energy consumption. We modify the (data copying flavor of the) `n-queens` benchmark into two versions: Figure 14 and Figure 15. The two versions are functionally

identical. In the first version, the execution of a task follows the sequence of `abababab`c where `a` is copying memory for a subtask, `b` is forking the subtask, and `c` is computing the current task. In the second version, the execution of task follows the sequence of `aaaacbbbb`[4].

Which version should fare better? On the surface, the second version indeed admits less parallelism on the execution of the current task: its forks the subtasks only after the current task has finished. Therefore, it cannot be executed in parallel with any of the `a` steps or the `c` step of the subtasks. Our benchmarking results on the other hand show the opposite: the second version yields energy savings of 10.11% and a performance improvement of 10.66%.

We hypothesize that data locality plays an important role. Note that in the first version, we interspersed data copying with thread forking (together with other operations in a loop iteration). Any of the latter operations may potentially pollute the cache, increasing the chance of memory round-trips. In the second version however, the same memory area is repeated requested, leading to significant data locality.

To make sure data locality is the main cause here, we also investigated the same two-version approach, but using a data sharing strategy. There is no noticeable difference in energy consumption and performance for the two versions.

RQ4 Summary: *Data size has non-linear impact on energy consumption. In data-parallel programs, the curve to demonstrate the relationship between data size (X-axis) and energy consumption (Y-axis) is often convex. Performance is closely related to energy consumption in the presence of data size variations, but the two do not follow identical trajectories. Significant data copying leads to increased energy consumption, but its relative effect is often smaller than the performance loss it imposes. Data locality plays an important role in energy consumption in multi-threaded programs.*

6. Threats to Validity

In experimental systems research, a fundamental challenge is the vast number of factors across the compute stack. For instance, it is a valid question to ask whether different OS scheduling policies [24, 39], different processor and interconnect layouts [19, 33], and different VLSI circuit designs [1], have impact on results. They clearly all do. Our study takes a route common in experimental programming language research, by constructing experiments over representative system software and hardware, and the results are empirical by nature.

To take a step further, we seek to gain a preliminary understanding of how platform variations impact our results. In particular, we focus on configuration variations of the language runtime. The primary goal is to understand the stability and portability of our results.

Heap size. Heap size settings are known to impact JVM performance (*e.g.*, [9]). Figure 16 shows the energy consumption and performance under different settings of maximum heap sizes (to trigger GC) for `sunflow`; the rest of the JVM settings are identical to those described in Section 4. When maximum heap size is restricted to a very low level – such as 20MB for `sunflow` – both energy consumption and performance go higher significantly. We speculate the additional overheads result from VM allocation and garbage collection. Variations in energy consumption that stem from heap size appear to be small if the maximum heap size is higher. While examining this benchmark without setting a fixed maximum heap size, we observed that its heap usage reaches a peak of more than 50MB before GC is triggered. Fixing the heap size at 20MB may have triggered significantly more GC.

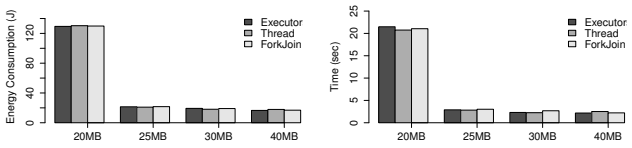


Figure 16. Heap Size Effect (`sunflow`, 32 threads, 256 tasks, 256 image size)

Garbage collection strategy. To gain a preliminary understanding of how GC strategies may pose a threat to the validity of our results, we construct experiments over 5 GC options over Hotspot: (a) SerialGC: the stop-the-world serial collector, (b) ParallelGC: the parallel collector, (c) PrallelOldGC: the parallel collector with data compression, (d) ConcMarkSweepGC: concurrent mark sweep collector, and (e) G1GC): the garbage-first collector. All have been specified by Oracle[25]. Figure 17 shows the results for `xalan`.

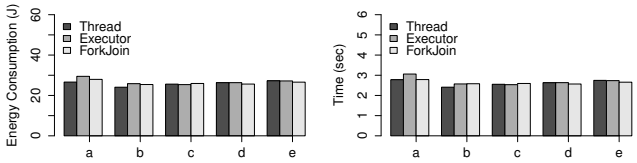


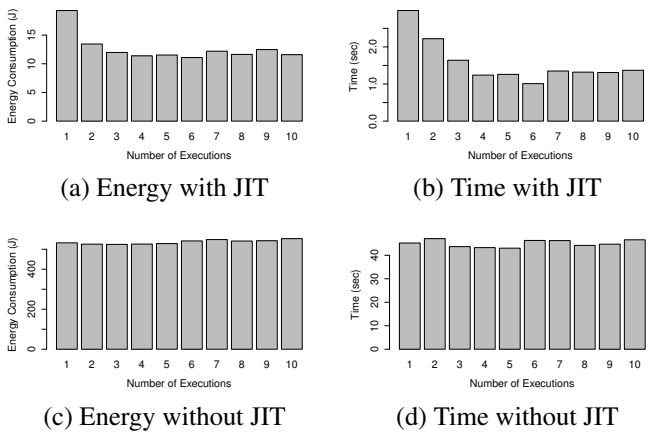
Figure 17. GC Effect (`xalan`, 32 threads, 64 tasks, 300 transformation files. GC strategies are: `a`: SerialGC, `b`: ParallelGC, `c`: `PrallelOldGC`, `d`: `ConcMarkSweepGC`, `e`: `G1GC`)

As shown, GC strategies do have observable impact on program energy consumption. In the context of this study, the effect is relatively mild, within $\pm 10\%$. A precise relationship between GC and energy consumption is a complex topic beyond the scope of this paper.

Just-In-Time compilation. Just-In-Time (JIT) compilation dynamically optimizes the program and is known to have

[4] The `invokeall` method in the second version is part of the Java ForkJoin API. It forks all tasks and then joins them all. Through inspecting its source code, we find no "magic" that would otherwise skew the results.

significant impact on performance. Predictably, JIT also has direct impact on energy consumption. Figure 18 shows the effect of JIT on `sunflow`.



(a) Energy with JIT (b) Time with JIT

(c) Energy without JIT (d) Time without JIT

Figure 18. JIT Effect (`sunflow`, 32 threads, 256 tasks, 256 image size, 10 runs on X-axis)

Here, the X-axis represents 10 "hot" runs of `sunflow`, *i.e.* a top-level loop that encompasses 10 executions within one JVM. With JIT, early runs incur higher energy/time overhead than later runs, as illustrated in Figure 18(a) and Figure 18(b). Also note that energy/performance behaviors do stabilize after a number of runs. With JIT disabled, both energy consumption and performance are uniform, as shown in Figure 18(c) and Figure 18(d). Both of them however are also significantly worse than their JIT counterparts.

Moreover, the growth of energy and time is not proportional. Performance increases 33 times from not using JIT to using JIT, whereas energy consumption increases more than 45 times. For instance, for the 10th `sunflow` execution, the average power consumption using JIT was 98.7 W, and when not using it was 142.8 W. Analyzing the curves we observed that, although the JIT execution achieves the highest power consumption (175.3 W using JIT and 166.3 W not using JIT), the non-JIT execution spends much more time in the highest part of the graph (3rd quartile: 163.2 W), that is, consuming more power, than the approach using JIT (3rd quartile: 154.6 W).

In Section 4, we explained our data collection strategy as averaging the last runs of JIT-enabled executions. This decision stems from our observations here: (1) JIT-disabled executions incur energy/performance overhead unrealistic to common use of Java applications, and (2) later runs of JIT-enabled executions do stabiize in terms of energy consumption and performance.

Platform Variations. As a final experiment, we ran some of the benchmarks on a different machine: an 8-core AMD FX-8150 processor (Bulldozer architecture) with 16GB of DDR 1600 memory, running Debian 3.2.46-1 Linux (kernel 3.2.0-4-amd64) and Oracle HotSpot 64-Bit server VM, JDK version 1.7.0_45, build 18. Figure 19 shows the results for `n-queens` benchmark.

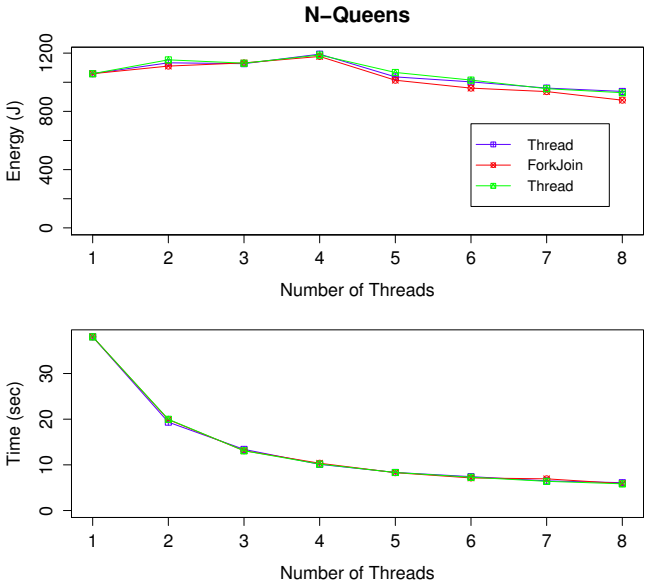


Figure 19. Results on Alternative Platform

The benchmarking results show similar trends. For instance, the Λ curve recurs, peaking at half of available processors – the same behavior for this benchmark when using 32 processors. The thread management styles behave similarly when compared to the 32 processors machine. Still, ForkJoin style outperforms Thread and Executor.

7. Conclusion

In this paper, we present a study on how concurrent programming practices may have impact on energy consumption. Our results suggest that different constructs for managing concurrent execution can impact energy consumption in different ways, and energy consumption is determined by the choice of thread management constructs, the number of threads, the granularity of tasks, the size of the data, and the nature of data access. This study is a step toward a better understanding of the interplay between energy efficiency and performance.

References

[1] A.Chandrakasan, S. Sheng, and R. Brodersen. Low power cmos digital design. *IEEE Journal of Solid State Circuits*, 27:473–484, 1992.

[2] J. Asafu-Adjaye. The relationship between energy consumption, energy prices and economic growth: time series evidence from asian developing countries. *Energy Economics*, 22(6):615 – 625, 2000.

[3] W. Baek and T. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.

[4] T. Bartenstein and Y. Liu. Green streams for data-intensive software. In *ICSE*, 2013.

[5] S. Blackburn, R. Garner, C. Hoffmann, A. Khang, K. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA*, 2006.

[6] M. Cohen, H. Steve Zhu, S. Emgin, and Y. Liu. Energy types. In *OOPSLA*, 2012.

[7] K. Farkas, J. Flinn, G. Back, D. Grunwald, and J. Anderson. Quantifying the energy consumption of a pocket computer and a java virtual machine. In *SIGMETRICS*, 2000.

[8] M. Frigo, C. Leiserson, and K. Randall. The implementation of the cilk-5 multithreaded language. In *PLDI*, 1998.

[9] Andy G., Dries B., and Lieven E. Statistically rigorous java performance evaluation. In *OOPSLA*, 2007.

[10] A. Gautham, K. Korgaonkar, P. Slpsk, S. Balachandran, and K. Veezhinathan. The implications of shared data synchronization techniques on multi-core energy efficiency. In *HotPower*, 2012.

[11] R. Ge, X. Feng, W. Feng, and K.W. Cameron. Cpu miser: A performance-directed, run-time system for power-aware clusters. In *ICPP*, 2007.

[12] Y. Guo, J. Zhao, V. Cavé, and V. Sarkar. Slaw: A scalable locality-aware adaptive work-stealing scheduler. In *IPDPS*, 2010.

[13] S. Hao, D. Li, W. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *ICSE*, 2013.

[14] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. IEEE Symposium*, 1994.

[15] 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*, 2006.

[16] A. Iyer and D. Marculescu. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. In *ICCAD*, 2002.

[17] T. Kalibera, M. Mole, R. Jones, and J. Vitek. A black-box approach to understanding concurrency in dacapo. In *OOPSLA*, 2012.

[18] A. Kansal, T. Saponas, A. Brush, K. McKinley, T. Mytkowicz, and R. Ziola. The latency, accuracy, and battery (lab) abstraction: programmer productivity and energy efficiency for continuous mobile context sensing. In *OOPSLA*, 2013.

[19] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO 36*, 2003.

[20] V. Kumar, D. Frampton, S. Blackburn, D. Grove, and O. Tardieu. Work-stealing without the baggage. In *OOPSLA*, 2012.

[21] Doug Lea. A java fork/join framework. In *Java Grande*, 2000.

[22] Y. Liu. Energy-efficient synchronization through program patterns. In *GREENS*, 2012.

[23] Y. Liu. Variant-frequency semantics for green futures. In *PLACES'12*, 2012.

[24] A. Merkel and F. Bellosa. Balancing power consumption in multiprocessor systems. In *EuroSys*, 2006.

[25] Oracle. Java hotspot garbage collection. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-140228.html, 2014. [Online; accessed 21-Mar-2014].

[26] Oracle. Java language and virtual machine specifications. http://docs.oracle.com/javase/specs/, 2014. [Online; accessed 21-Mar-2014].

[27] S. Park, W. Jiang, Y. Zhou, and S. Adve. Managing energy-performance tradeoffs for multithreaded applications on multiprocessor architectures. In *SIGMETRICS*, 2007.

[28] G. Pinto and F. Castor. On the implications of language constructs for concurrent execution in the energy efficiency of multicore applications. In *SPLASH*, 2013.

[29] G. Pinto, F. Castor, and Y. Liu. Mining questions about software energy consumption. In *MSR*, 2014.

[30] K. Rangan, G. Wei, and D. Brooks. Thread motion: fine-grained power management for multi-core systems. In *ISCA*, 2009.

[31] H. Ribic and Y. Liu. Energy-efficient work-stealing language runtimes. In *ASPLOS*, 2014.

[32] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[33] A. Solernou, J. Thiyagalingam, M. Duta, and AnneE. T. The effect of topology-aware process and thread placement on performance and energy. In *Supercomputing*, Lecture Notes in Computer Science, 2013.

[34] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: a language and runtime system for perpetual systems. In *SenSys'07*, pages 161–174, 2007.

[35] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on VLSI Systems*, 2:437–445, 1994.

[36] W. Torres, G. Pinto, B. Fernandes, J. Oliveira, F. Ximenes, and F. Castor. Are java programmers transitioning to multicore?: a large scale study of java floss. In *TMC*, 2011.

[37] A. Trefethen and J. Thiyagalingam. Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science*, 2013.

[38] N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam, and M. J. Irwin. Energy behavior of java applications from the memory perspective. In *JavaTM Virtual Machine Research and Technology Symposium*, 2001.

[39] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *SOSP*, 2003.

[40] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang. Refactoring android java code for on-demand computation offloading. In *OOPSLA*, 2012.

A. Appendix

A.1 Figures: EDP

A.2 Figures: Varying Size of Data

A.3 Figures: Task-Centric Approach

A.4 Figures: Data-Centric Approach

A.5 Figures: Coping vs Sharing

A.6 Figures: Forking Width

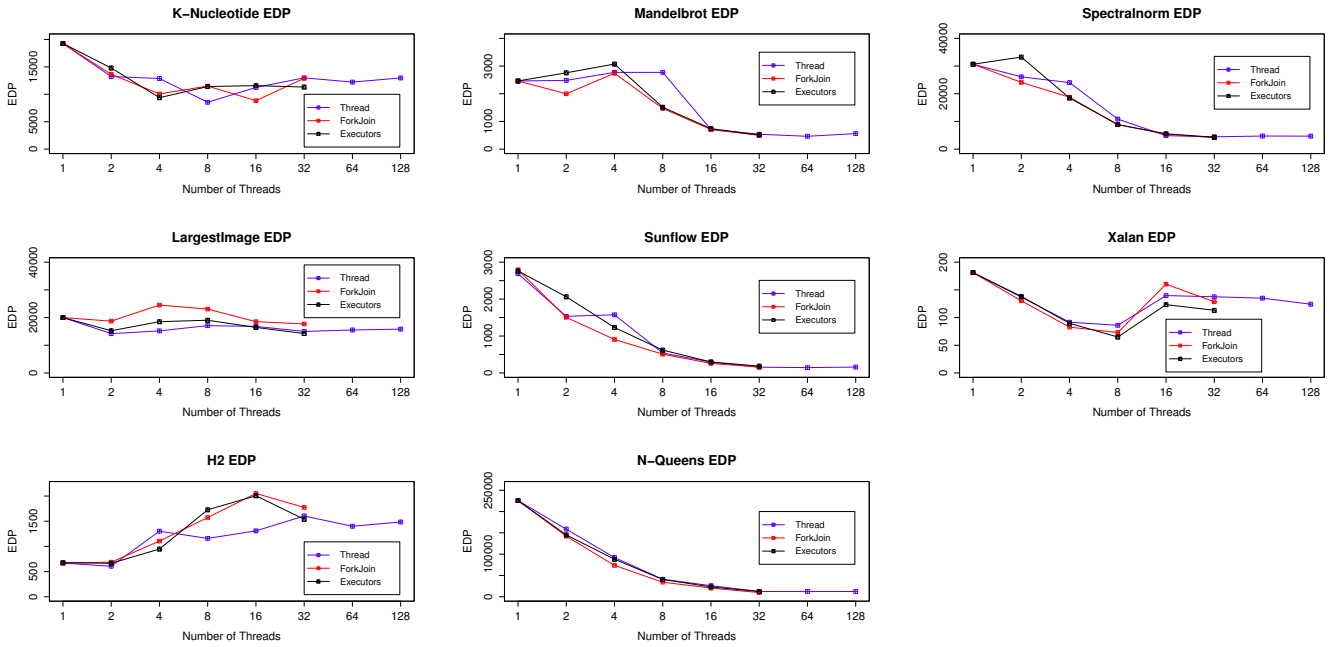


Figure 20. EDP (a smaller value is better)

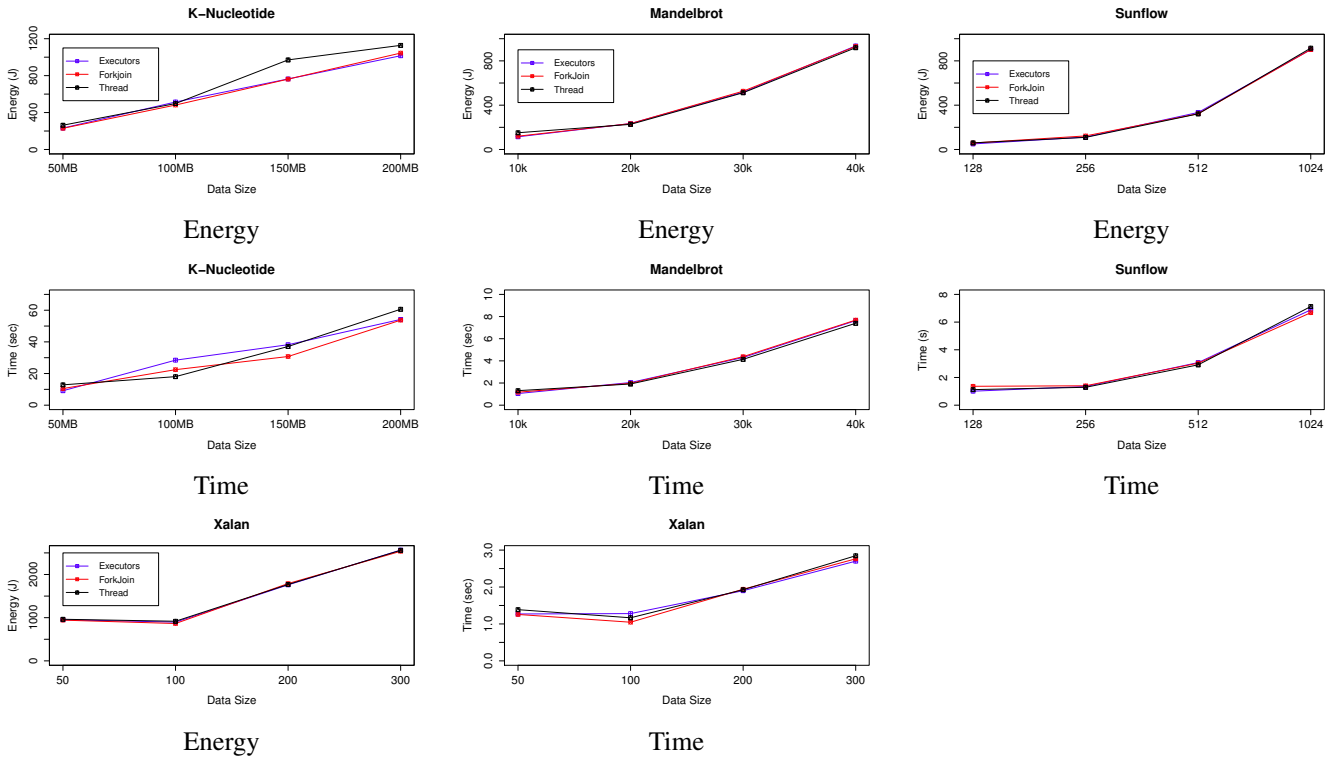


Figure 21. Energy/Performance: Data Size

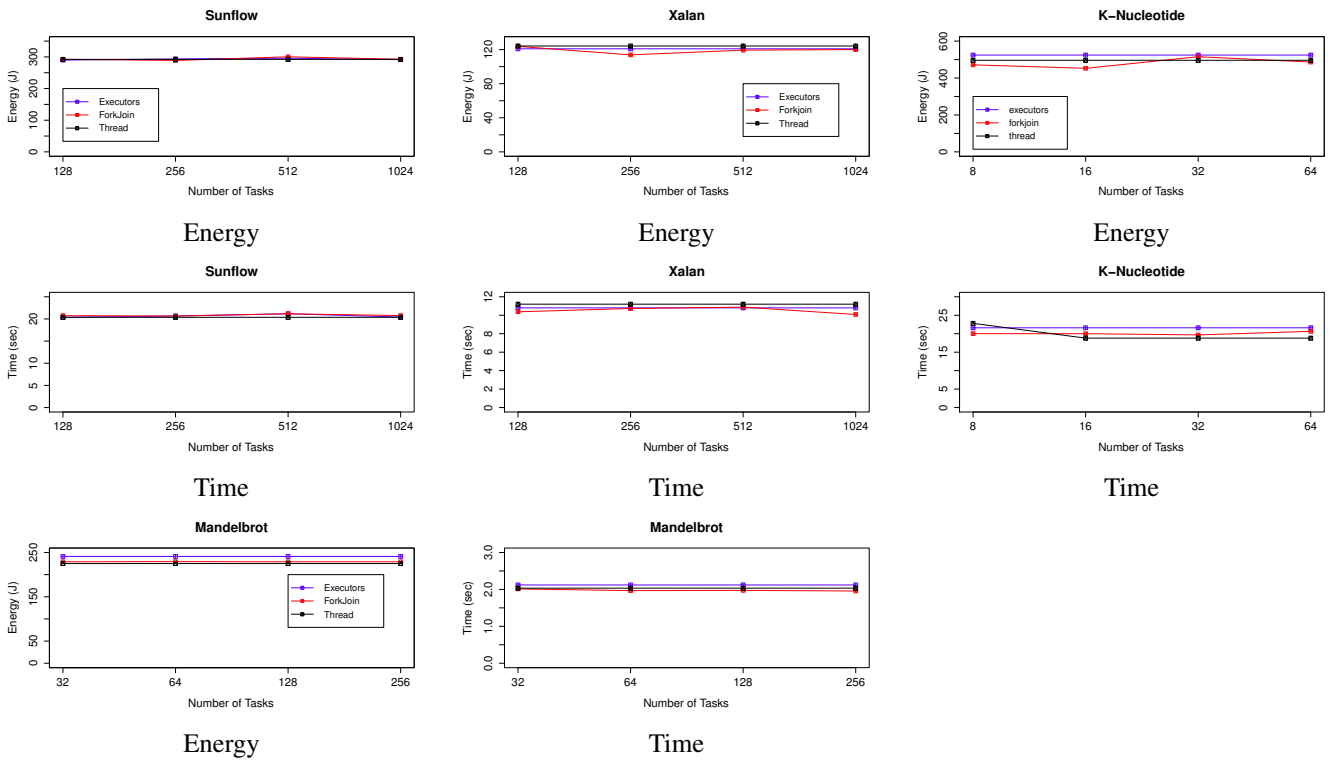


Figure 22. Energy/Performance: Task Granularity in a Task-Centric Approach

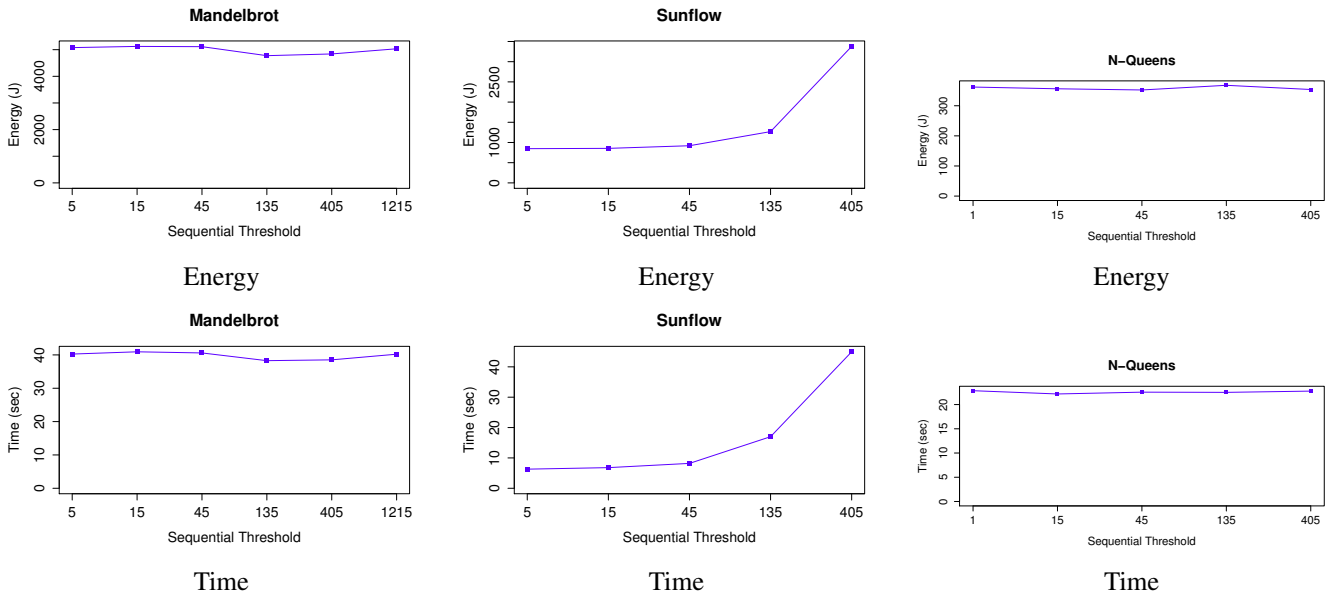


Figure 23. Energy/Performance: Task Granularity in a Data-Centric Approach

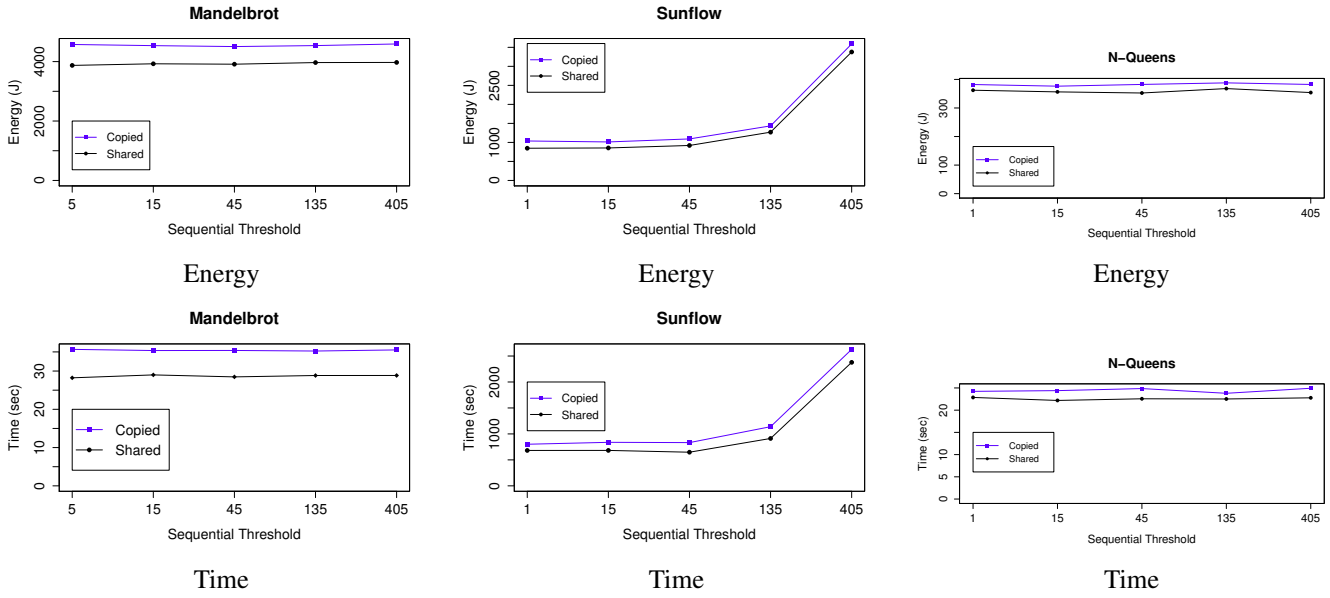


Figure 24. Energy/Performance: Data Copying vs. Sharing

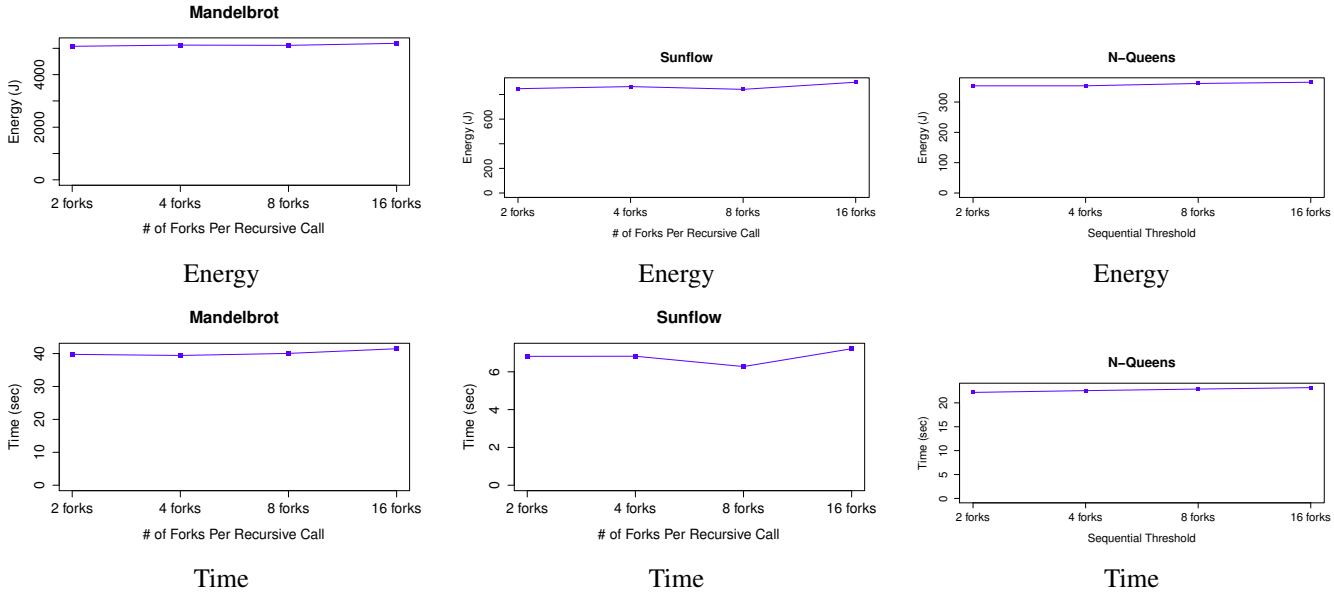


Figure 25. Energy/Performance: Forking Width