Isoefficiency in Practice: Configuring and Understanding the Performance of Task-based Applications

Sergei Shudler
Technische Universität Darmstadt
Darmstadt, Germany
shudler@cs.tu-darmstadt.de

Alexandru Calotoiu
Technische Universität Darmstadt
Darmstadt, Germany
calotoiu@cs.tu-darmstadt.de

Torsten Hoefler
ETH Zurich
Zurich, Switzerland
htor@inf.ethz.ch

Felix Wolf
Technische Universität Darmstadt
Darmstadt, Germany
wolf@cs.tu-darmstadt.de

Abstract

Task-based programming offers an elegant way to express units of computation and the dependencies among them, making it easier to distribute the computational load evenly across multiple cores. However, this separation of problem decomposition and parallelism requires a sufficiently large input problem to achieve satisfactory efficiency on a given number of cores. Unfortunately, finding a good match between input size and core count usually requires significant experimentation, which is expensive and sometimes even impractical. In this paper, we propose an automated empirical method for finding the isoefficiency function of a task-based program, binding efficiency, core count, and the input size in one analytical expression. This allows the latter two to be adjusted according to given (realistic) efficiency objectives. Moreover, we not only find (i) the actual isoefficiency function but also (ii) the function one would yield if the program execution was free of resource contention and (iii) an upper bound that could only be reached if the program was able to maintain its average parallelism throughout its execution. The difference between the three helps to explain low efficiency, and in particular, it helps to differentiate between resource contention and structural conflicts related to task dependencies or scheduling. The insights gained can be used to co-design programs and shared system resources.

PPoPP '17, February 4–8, 2017, Austin, Texas, USA.

DOI: http://dx.doi.org/10.1145/3018743.3018770

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Modeling techniques; D.1.3 [*Concurrent Programming*]: Parallel programming; D.2.8 [*Metrics*]: Performance measures; D.4.8 [*Performance*]: Modeling and prediction

Keywords parallel programming; tasking; isoefficiency; performance modeling; performance analysis; co-design

1. Introduction

Task-based programming models, such as Cilk [7] or OpenMP [28], are well known and as the number of cores per node continues to increase, they gain more and more attention. One major advantage of task-based programming is that it allows parallelism to be expressed in terms of tasks, which are units of computation that can be either independent, dependent on a previous task, or a prerequisite to a subsequent task. Explicitly expressing parts of the code as tasks allows the compiler to take care of all the thread management intricacies, thereby sparing the user from tedious low-level details. Good task delineation also helps the scheduler better exploit the inherent parallelism and can lead to improved load balance. For these reasons, task-based programming will play an even more prominent role in exascale systems.

Normally, when the user receives an allocation of computing resources, the nodes are not shared. This means the user has to use all of the cores on each node efficiently, otherwise, computing resources are wasted. In an exascale system this problem will be even more pronounced because the available node concurrency is predicted to be larger by at least one order of magnitude compared to the systems available today [34].

Although the optimization of task-based algorithms has been studied extensively in the past [12, 14, 21, 27, 33],

the size of the input in these studies usually remained fixed. Since the critical path length in a task dependency graph limits the speedup of the algorithm [8], fixed input size means that no matter how well the algorithm is optimized, the speedup, and thus the efficiency, is limited. Starting from a certain core count the speedup will stop increasing unless the input size increases as well. Moreover, scaling is often not perfect, meaning that the speedup rate is too slow to maintain constant efficiency. Figure 1 is an example of this phenomenon. It shows the speedup and efficiency for the applications Sort and Strassen from the Barcelona OpenMP Tasks Suite (BOTS) [14]. Although the inputs for these applications are 128M integers and $8{,}192 \times 8{,}192$ doubles, respectively, their speedup does not increase fast enough, leading to a sharp drop in efficiency. Even if we try to optimize the code and achieve better speedups, the effect will not last at higher scales, as an optimized version will still be limited by the length of the critical path. The only way to ensure that efficiency remains constant, as the number of cores increases, is to increase the input size as well. This concept is embodied in the isoefficiency relation [20], which binds the number of processing elements (PEs) the application uses to the input size. It specifies by which factor the input size has to increase, with respect to the increase in the number of PEs, to maintain constant efficiency. Isoefficiency can be generalized to a two-parameter efficiency function that provides efficiency values as a function of both the PE count and the input size. The contour lines of this function are exactly isoefficiency lines (cf. Section 3).

Although isoefficiency analysis is useful in understanding the scalability behavior of algorithms, it is not straightforward to apply and requires deep knowledge of the algorithm. Moreover, it only provides theoretical insight, much like traditional complexity analysis. In practice, however, task-based algorithms experience hardware limitations in the form of resource contention in general and memory contention in particular. Resources such as cache and memory controllers are limited and can negatively impact application scalability [35]. These might render theoretical isoefficiency functions not accurate enough to be used in practice. To be able to make informed decisions as to how big the input size should be in order to use all of the allocated cores efficiently, the user not only has to have a realistic isoefficiency model but also needs to understand the severity of resource contention at higher scales.

In this study, we propose a novel practical method to automatically model the empirical efficiency functions of task-based applications and their contention-free replay runs. Modeling the efficiency function allows us to easily derive an isoefficiency relation for any realistic target efficiency. To this end, we developed a task replay engine that executes empty task skeletons, thereby emulating execution without resource contention. Resource contention includes cache accesses, memory bandwidth, coherence traffic, network communication, and disk I/O. After performing the replay runs we derive the efficiency function from the results and compare the models we obtain with the models of normal runs. A big discrepancy suggests that resource contention overhead is a major scalability bottleneck. We also analyze the task dependency graph (TDG) and model an upper bound efficiency based on TDG metrics. A discrepancy between the upper bound efficiency and the contention-free efficiency suggests that there is still room to improve either the algorithm itself or the scheduler. Our approach is applicable to both pure shared memory applications, as well as to task-based parts of hybrid applications (e.g., OpenMP parallel regions in hybrid MPI / OpenMP applications). Our approach helps users, application developers, and hardware designers answer the following important questions, related to both application analysis and deployment:

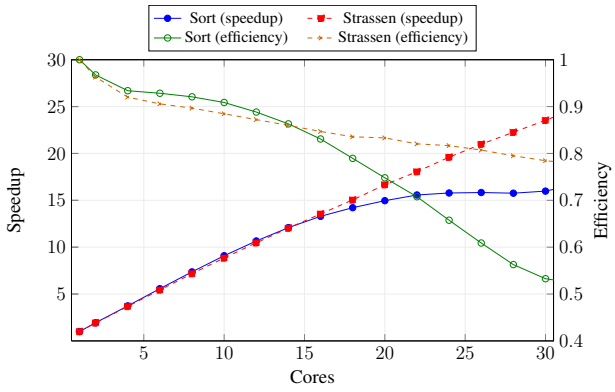


Figure 1: Speedup and efficiency for Sort and Strassen applications from BOTS benchmark.

1. Are there any fundamental scalability limitations in the algorithm or the implementation of a task-based application? This is helpful to compare implementation alternatives independently of the target system.
2. Is poor scaling of a task-based application a result of resource contention overhead? The answer helps application developers analyze the causes of bottlenecks in their applications and system designers to respond to pressure on shared resources.
3. Is there any optimization potential, in terms of task dependencies, scheduling, and granularity in a task-based application? The answer helps application developers optimize their applications on the level of the task graph and its execution.
4. What is the required input size for a given core count such that we maintain a constant, given efficiency? The answer helps users efficiently utilize all the computing resources they have. They can aim for the right problem sizes based on the number of available cores.

5. What is the required core count for a given input size such that we maintain a constant, given efficiency? Which efficiency can we expect for a given number of cores and input size? Both questions are related to the co-design process when hardware designers have to understand how to make future systems suitable for both existing and future applications.

The remainder of this paper is organized as follows. In Section 2, we describe the task dependency graph analysis in greater detail. In Section 3 we describe the efficiency modeling approach and present the task replay engine. Section 4 evaluates our technique, analyzes the results, and, at the end, shows examples of deriving application input sizes for a target machine. Finally, we review related work in Section 5, before drawing our conclusion in Section 6.

2. Task Dependency Graphs and Efficiency Analysis

There is an important difference between tasks and threads: a task is a work package containing a collection of instructions to be executed, whereas a thread is a light-weight process that executes given instructions in an independent context. A task-based code can be executed by one or more threads running on one or more cores. Each instance of a task is executed at any given time by only one thread. For the purpose of our discussion, we will assume that the number of threads is equal to the number of cores, and that each thread runs on a separate core.

A task dependency graph (TDG) is a directed acyclic graph (DAG) that represents the execution of a task-based application. The nodes of this graph are tasks, and the edges represent dependencies between tasks, meaning that a task cannot begin execution before the tasks connected to it via incoming edges have been completed. Except for some trivial cases, most of the interesting and useful problems have dependencies in their algorithm flow, thereby producing complex TDGs.

In the process of execution, the scheduler assigns tasks, which are ready to be executed, to threads. Depending on the scheduler, tasks might be stopped (i.e., preempted) and resumed later. In this regard, tasking environments, including OpenMP, distinguish between *tied* and *untied* tasks. A tied task can only be executed by the thread that started executing it. This implies that if this task is preempted it can only be resumed by the thread that executed it before. An untied task, on the other hand, can be resumed by any available thread after preemption. Both types of tasks have advantages—tied tasks provide guarantees for private data (i.e., data on the stack), whereas untied tasks provide more flexibility to the scheduler. In this study we use applications with both types of tasks.

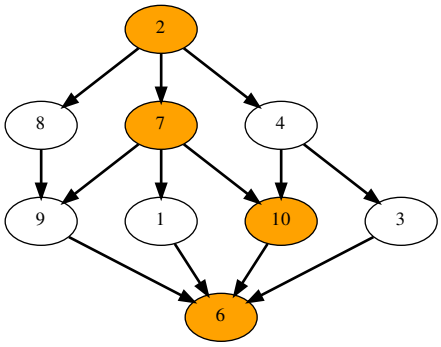


Figure 2: Task dependency graph; each node contains the task time and the highlighted tasks form the critical path.

2.1 TDG Metrics and Laws

We characterize TDGs using a set of key metrics [6, 11]. The *work* of the computation is the total execution time on one core, or the sum of all the task times. The *depth* of the computation (also known as *span*) is the total sum of all task times on the *critical path*, which is the longest path, in terms of task times, from any source node to any target node. A source node is a node with no incoming edges, and a target node is a node without any outgoing edges. Figure 2 shows an example of a TDG in which work equals 50 and depth equals 25. In the rest of the paper, we use the following notations:

- p, n: the number of threads and the input size, respectively.
- $T_p(n)$: the execution time of a computation with p threads with input size n.
- $T_1(n)$: the work of the computation, or the computational effort for input size n.
- $T_\infty(n)$: the depth of the computation for input size n.
- $S_p(n) = \frac{T_1(n)}{T_p(n)}$: the speedup of the computation for specific p and n.
- $\pi(n) = \frac{T_1(n)}{T_\infty(n)}$: the *average* parallelism of the computation for n.

From these TDG metrics we can derive important laws [11] and boundaries on speedup and efficiency.

Work law The execution cannot be faster than when we divide the whole work $T_1(n)$ equally between cores:

$$T_p(n) \geq \frac{T_1(n)}{p} \tag{1}$$

A direct consequence of the work law is an upper bound on the speedup: $S_p(n) \leq p$. We ignore super-linear speedups for the sake of simplicity.

Depth law Since the critical path is a chain of dependencies, the tasks on this path must be executed one after the

other, giving us another lower bound on the execution:

$$T_p \geq T_\infty \tag{2}$$

In this case we can derive another upper bound on the speedup: $S_p(n) \leq \pi(n)$.

2.2 Efficiency and Isoefficiency

The two-parameter efficiency function is defined as efficiency $E(p,n) = \frac{S_p(n)}{p}$ with two parameters p and n. The isoefficiency, which binds together the core count and the input size [19, 20] for a specific, constant efficiency, is simply a contour line on the surface of $E(p,n)$. To clarify this point, we first define the total overhead time:

$$T_o(p,n) = pT_p(n) - T_1(n) \tag{3}$$

This is the total amount of time that all of the threads spend without contributing to the solution of the problem, including resource contention, idle time, and scheduling overhead. Rearranging Eq. 3 and using the efficiency definition (i.e., $E = \frac{S_p(n)}{p}$) yields the isoefficiency relation:

$$T_1(n) = \frac{E}{1-E} T_o(p,n) \tag{4}$$

This relation binds $T_1(n)$, p, and E. Normally, in isoefficiency analysis, the efficiency E is constant and we are able to form an expression that relates the core count p to the work of the computation $T_1(n)$. However, if we rearrange Eq. 4 such that $E = f(p,n)$, we obtain an expression that relates the core count p and the input size n to the efficiency E. In other words, we obtain the efficiency function. It is easy to see now that isoefficiency is a special case of the more general efficiency function limited to a specific, constant efficiency.

Isoefficiency is a useful tool in the theoretical analysis of parallel algorithms. It allows users and developers to compare different alternatives and choose the one in which the problem size grows more slowly in relation to the core count. In practice, however, resource contention overhead might overshadow other types of overheads and render a thought-to-be-scalable algorithm unscalable. Our methodology tackles this problem by modeling the empirical efficiency functions of both the application itself and the contention-free replay of the application. We can identify three different efficiency functions for a task-based application:

1. $E_{ac}(p,n)$: The actual efficiency function of the application, modeled after the empirical results of runtime benchmarks. In this case the application runs as it is and experiences contention. Therefore, this function reflects realistic application performance including resource contention and scheduling overhead.
2. $E_{cf}(p,n)$: The contention-free efficiency function, modeled after the results of replaying empty task skeletons according to the application's TDG. The replay uses the same TDG and scheduling policy as in the original runs that were benchmarked to produce $E_{ac}(p,n)$. Since the replay is free of resource contention, this efficiency function reflects an ideal situation in which the application does not experience resource contention caused by threads accessing the same resource simultaneously.
3. $E_{ub}(p,n)$: An upper bound on the efficiency of the application. Since efficiency is defined as $\frac{S_p(n)}{p}$, an upper bound on the speedup also limits the efficiency. From Section 2.1 we know that $S_p(n) \leq \min\{p, \pi(n)\}$, thus we define $E_{ub}(p,n) = \min\{1, \frac{\pi(n)}{p}\}$. This function describes an ideal situation of maximum speedup that is hardly achievable in practice.

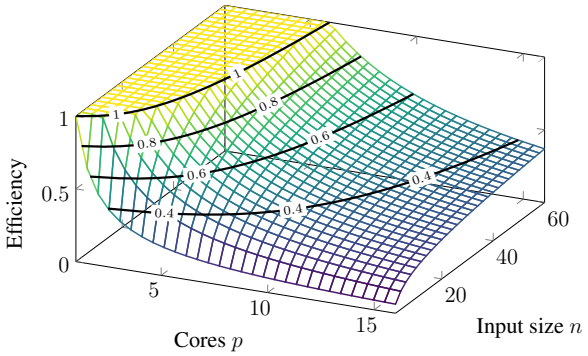


Figure 3: Upper-bound efficiency function $E_{ub}(p,n) = \min\{1, \frac{\log n}{p}\}$. The contour lines are isoefficiency functions for the efficiency values 1.0, 0.8, 0.6, and 0.4.

As a concrete example for an efficiency function, consider the task-based version of the Mergesort algorithm. A theoretical analysis of its TDG, for increasing input size n, gives us: $T_1(n) = \Theta(n \log n)$ and $T_\infty(n) = \Theta(n)$ [11]. Without loss of generality, we assume that the constant factor is 1 and get: $\pi(n) = \log n$. Figure 3 depicts the upper-limit efficiency function $E_{ub}(p,n) = \min\{1, \frac{\log n}{p}\}$ that we obtain in this case. It is a 3D surface graph in which the X and Y axes are the core count and the input size, respectively; whereas, the Z-axis, limited to the range $[0,1]$, gives us the efficiency values. The contour lines at Z-axis values of $E = 1$, $E = 0.8$, $E = 0.6$, and $E = 0.4$ are isoefficiency functions for these efficiencies.

By analyzing the differences between the efficiency function we can gain a number of important insights:

- $\Delta_{con} = E_{cf}(p,n) - E_{ac}(p,n)$: The contention discrepancy between actual and contention-free efficiencies characterizes how severe the resource contention overhead is. Essentially, it tells us whether this overhead is a significant obstacle to application scalability. A big discrepancy suggests that optimization efforts should focus

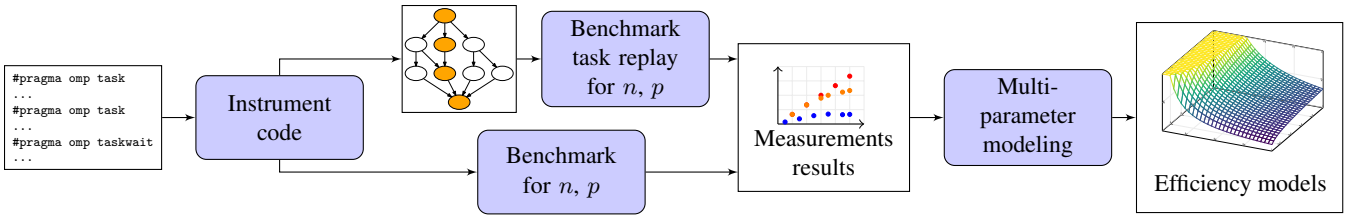


Figure 4: The modeling workflow for actual and replay efficiency models.

on reducing the resource contention either on the level of the application or the underlying system.

- $\Delta_{str} = E_{ub}(p,n) - E_{cf}(p,n)$: The structural discrepancy between upper bound and contention-free efficiencies characterizes the optimization potential of the application on the level of the task graph. A big discrepancy suggests that developers should explore optimization steps beyond reducing resource contention, such as reducing task dependencies, adjusting the task granularity, or using a more efficient scheduler. A small discrepancy, on the other hand, means that – disregarding contention – an algorithm's implementation is close to the maximum efficiency that it can achieve. Δ_{str} can only provide insights into an observed behavior of an application's algorithm. However, there might be other, possibly better algorithms that would produce different TDGs with different $E_{ub}(p,n)$ functions, and potentially, even a better maximum efficiency.

3. Modeling Approach

In this section we present our approach to modeling the efficiency functions, and consequently, the isoefficiency functions. Performance modeling, and automated performance modeling in particular, was shown to be useful and practical for analyzing the performance of parallel applications [9, 23, 26, 31, 32]. In this work, we combine multi-parameter performance modeling with benchmarking of real task-based applications to automatically generate the empirical efficiency functions of both the application and the contention-free replay of the application's TDG.

Figure 4 shows the modeling workflow. It starts with instrumenting the code, continues with the construction of the code's TDG, and then proceeds with benchmarking the code for increasing n and p. The TDG is used as an input to the replay engine, which we will present later in this section, and the replay is benchmarked in the same way as the code itself. After benchmarking both the application and the replay, we continue with producing empirical models using the performance-modeling tool Extra-P [2].

For our study we use the OmpSs [15] threading environment. Similar to OpenMP, OmpSs offers the ability to annotate functions or blocks of code as tasks. Although task dependencies were already introduced in OpenMP 4.0, not all compilers support them in full yet. OmpSs, on the other hand, provides a more mature task dependency support that allows experimentation with more complex TDGs. The OmpSs runtime, Nanos++, provides an instrumentation plugin that instruments the code, measures the task execution times, generates the TDG, and saves it as a Graphviz [3] `.dot` file. We modified the instrumentation plugin to compute T_1 and T_∞, and produce a simplified `.dot` file, better suited as an input to the replay engine.

Moreover, OmpSs and OpenMP offer the same syntax for task creation and synchronization, namely, `#pragma omp task` and `#pragma omp taskwait` work in both environments. This allows the OmpSs compiler to compile OpenMP task-based applications, and it also allows the Nanos++ runtime to successfully instrument them.

OmpSs provides a number of choices for task scheduling policies during application execution. Using the *breadth first* scheduler (`--schedule=bf` flag) for tied tasks and the *work first* scheduler (`--schedule=wf` flag) for untied tasks was shown to produce good runtimes [13]. The breadth first scheduler uses a single, FIFO-ordered global ready queue for the tasks. Whenever a task is ready (i.e., its dependencies are fulfilled) it is enqueued in the queue and later dequeued to be executed by an available thread. The work first scheduler, on the other hand, uses one ready queue per thread. Whenever a task is created by a thread, the thread begins to execute it immediately, preempting the current task and placing it in the queue. If a thread becomes idle and its queue is empty, it attempts to steal tasks from the queues of the other threads to improve load balance. This scheduling policy is similar to the default scheduling policy used in Cilk [7].

3.1 Task Replay Engine

The goal of the task replay engine is to emulate the execution of a task graph without resource contention. The TDG, constructed by the instrumentation plugin, contains task times when the code is executed by one thread. Since a single thread does not have to share memory bandwidth or wait for other threads to access shared data structures, these times are free of resource contention overhead.

Instead of specifying the tasks implicitly using pragmas, the replay engine uses the OmpSs runtime API to specify the tasks and their dependencies explicitly. This API is defined in the `nanox/nanos.h` file that is part of the OmpSs installation. Each task is specified as a function that receives the

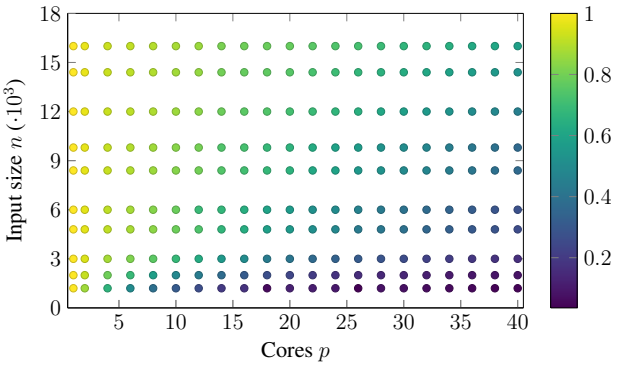


Figure 5: Typical benchmark results; the color of each point represents the measured efficiency.

task time as an argument and then busy-waits in a loop for the duration of this time. To query the time efficiently and with minimal overhead we use the timer of the *LibSciBench* library [22]. The library provides high-resolution timers for a number of common architectures. For the x86 architecture, on which we tested our methodology, the timer of the library uses the RDTSC register, and in order to prevent problems with out-of-order execution it issues the `CPUID` instruction before querying the register. The overhead in the function that emulates task execution is minimal. It only includes querying the time, repeating a loop counter, and accessing one local variable to store the accumulated time.

Since we use the same OmpSs runtime for the replays as we use for the normal runs, we ensure that the combination of the scheduler and tied/untied tasks during the replay is the same as it is during the normal runs.

3.2 Multi-Parameter Modeling with Extra-P

We start by selecting a range of threads p and a range of input sizes n. The benchmark then runs the application for each combination of p and n from these ranges. The results can be viewed as a 2D grid of points: the X-axis is the number of threads and the Y-axis is the input size. Figure 5 shows an example of such a 2D grid. Each point represents a single result and its color the measured efficiency.

After the benchmarking is done we run Extra-P [2] to produce two-parameter models of efficiency. These models are a special case of the more general multi-parameter models that aim to capture how a number of independent parameters, such as core count, problem size, and algorithmic parameters, influence a target metric, such as runtime, floating-point operations, and so on. The *performance model normal form* (PMNF) for multiple parameters [10] is defined as:

$$f(x_1, x_2, ..., x_m) = \sum_{k=1}^{n} c_k \cdot \prod_{l=1}^{m} x_l^{i_{k_l}} \cdot log^{j_{k_l}}(x_l) \qquad (5)$$

In this form, parameters x_l are represented by m combinations of monomials and logarithms, which are summed up in n different terms to form the whole model. The exponents i_{k_l} and j_{k_l} are chosen from sets $I, J \subset \mathbb{Q}$, respectively. Essentially, these sets define the scope of all the possible terms. Consider, for example, $n = 3$, $m = 2$, and $I = \{0, 0.25, 0.5\}$, $J = \{0, 1\}$. In this case, the search space for possible terms would be $\{1, log(x), x^{0.25}, x^{0.25}log(x), x^{0.5}, x^{0.5}log(x)\}$, and an example model could be: $f(x_1, x_2) = c_1 + c_2 \cdot x_1^{0.5} + c_3 \cdot x_1^{0.25} x_2^{0.25} log(x_2)$.

The modeling technique is based on an iterative modeling refinement process that stops when $\bar{R}^2$, the adjusted coefficient of determination – a standard statistical fit factor $\in [0, 1]$ such that a value of 1 indicates an optimal fit – cannot be improved any further. In cases when n and m increase and the search space of possible terms becomes too big, the technique uses a heuristic that greatly reduces the number of candidate models, while retaining a high degree of accuracy. This makes the multi-parameter modeling mechanism a usable technique in practice [10].

Table 1: Evaluated task-based applications.

App.	Origin	Description
Cholesky	BAR	Cholesky factorization of dense matrices
FFT	BAR	Fast Fourier transform of a matrix
Fibonacci	BOTS	Calculates Fibonacci numbers
NQueens	BOTS	Solution of the N-Queens problem
Sort	BOTS	Integer sorting with parallel Mergesort
SparseLU	BAR	LU decomposition over a sparse square matrix
Strassen	BOTS	Strassen's matrix multiplication

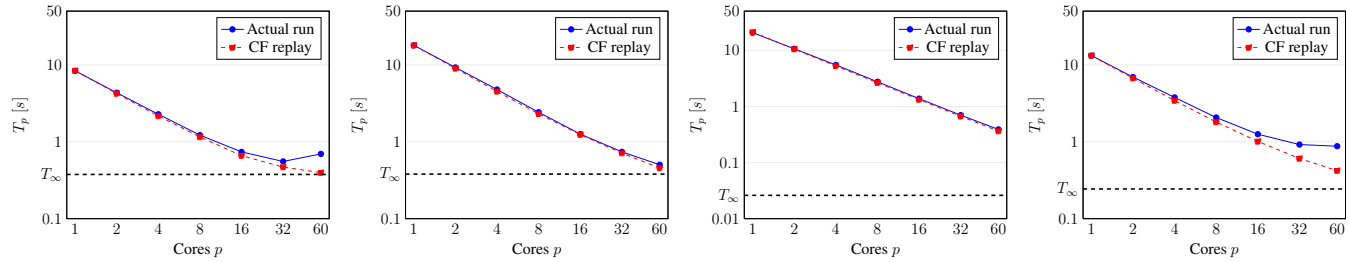
4. Evaluation

In this section, we model the efficiency, and hence the isoefficiency, of a number of task-based applications using our methodology and evaluate the results. We start with a discussion of the benchmarking setup, and then continue with the analysis of the results, including depth and parallelism models, isoefficiency models, and co-design use cases.

4.1 Experimental Setup

Table 1 presents the applications we evaluated. Since the focus is on task-based OmpSs/OpenMP applications, we selected our candidates from well known benchmark suites that target these programming models, namely, the Barcelona OpenMP Tasks Suite (BOTS) [14] and the Barcelona Application Repository (BAR) [1]. We were able to use the OmpSs compiler, which supports both the OmpSs and the OpenMP syntax, to successfully compile BOTS. While applications from BAR only have tied tasks, BOTS offers both tied and untied versions of its applications. To have a better coverage of potential use cases, we chose to run untied versions of BOTS applications and selected the scheduling policy accordingly.

We ran our experiments on a single NUMA node that consists of four Xeon E7-4890 v2 processors with 15 cores



(a) Cholesky (8,400 × 8,400 doubles) (b) Fibonacci ($n = 49$) (c) NQueens (14 × 14 board) (d) Strassen (4K×4K doubles)

Figure 6: Runtimes of actual runs and contention-free (CF) replays (on log scale) with constant input. The horizontal dashed lines, labeled T_∞, show the depth of the computation for the given inputs.

in each processor. Together they comprise 60 cores in one shared-memory machine.

For measuring both the runtime of each application as well as the task times we used the timer of the *LibSciBench* library [22] (as in the task replay engine). Each execution and replay of a particular combination of (p, n) was repeated multiple times. To reduce the effects of noise and increase the accuracy of the models we measured the confidence intervals of our measurements and increased the number of repetitions accordingly. As a rule of thumb, we deemed the number of repetitions to be enough when the 95% confidence interval was no larger than 5% of the mean. For most of the benchmarked applications, five repetitions was enough, but for some of them ten repetitions were necessary. In the special case of $p = 1$, we ran both the instrumented version of the code to produce the TDG and the uninstrumented version to measure a perturbation-free runtime as input for efficiency calculations.

Some applications have more than one input parameter, and depending on a particular combination of input parameters, the TDG can increase in different ways. In some cases, the number of tasks stays constant, while the task sizes increase; in other cases, the tasks sizes stay constant, but the number of tasks increases. Sometimes both the number of tasks and the task sizes increase. In all these cases, both the work (i.e., $T_1(n)$) and n increase.

In the case of Cholesky, the smallest input was a 1,200 × 1,200 matrix with 200 × 200 blocks, and the largest was a 16,000×16,000 matrix with 800×800 blocks. The inputs for FFT ranged from 5,280×5,280 to 30,000×30,000 matrices.

The input for the Fibonacci application is the index of the Fibonacci number. In our benchmarks, the smallest input was 47 and the largest 53. Smaller inputs resulted in very short runtimes, which did not serve the purposes of this study. The input of NQueens is the board dimension, which ranged from 10 to 15. As in the case of Fibonacci, smaller inputs result in runtimes that are too short.

The application Sort in BOTS is a parallel variant of the Mergesort algorithm that expects the number of elements in the input to be a power-of-two value. Our inputs, therefore, were arrays with a power-of-two number of integers, which ranged from 1M to 512M. The application SparseLU works on matrices and the inputs, in this case, ranged from 2,500 × 2,500 to 12,500 × 12,500 matrices.

The Strassen application implements a parallel version of the sequential Strassen algorithm. Since this algorithm recursively subdivides each side of the matrix into two, the dimension sizes have to be powers of two. Therefore, the smallest input in this case was a 256 × 256 matrix and the largest a 8,192 × 8,192 matrix.

Table 2: Depth and parallelism models of the evaluated applications.

Application	Model		$\bar{R}^2$
Cholesky	$T_\infty(n)$	$4.31 \cdot 10^{-9} \cdot n^{1.75} \log n$	0.99
	$\pi(n)$	$2.29 + 2.35 \cdot n$	0.98
FFT	$T_\infty(n)$	$0.08 + 1.33 \cdot 10^{-14} \cdot n^{2.75} \log n$	0.92
	$\pi(n)$	$1.19 \cdot 10^{-2} \cdot n^{0.67} \log n$	0.91
Fibonacci	$T_\infty(n)$	0.35	——
	$\pi(n)$	$25.48 + 0.49 \cdot n^{2.75} \log n$	0.99
NQueens	$T_\infty(n)$	$6.57 \cdot 10^{-4} \cdot n^2 \log n$	0.99
	$\pi(n)$	$2.18 \cdot n^{2.875} \log n$	0.98
Sort	$T_\infty(n)$	$3.03 \cdot 10^{-6} \cdot \sqrt{n}$	0.93
	$\pi(n)$	$3.53 + 3.32 \cdot 10^{-2} \cdot \sqrt{n}$	0.94
SparseLU	$T_\infty(n)$	$5.12 \cdot 10^{-5} \cdot n^{0.75} \log n$	0.96
	$\pi(n)$	$5.8 \cdot 10^{-5} \cdot n^{1.75} \log n$	0.99
Strassen	$T_\infty(n)$	$1.47 \cdot 10^{-9} \cdot n^2 \log n$	0.99
	$\pi(n)$	$0.25 \cdot n^{0.75}$	0.99

4.2 Analysis of the Results

Figure 6 shows the runtimes of some of the evaluated applications and their contention-free replays for constant inputs of medium size. In every subfigure, the horizontal dashed line represents $T_\infty(n)$ (the depth), which is a lower bound on the execution time. Note that the Y-axis is on a logarithmic scale. The figure illustrates that in some cases application runtimes reach $T_\infty(n)$ quite quickly.

In the cases of Cholesky and Fibonacci, the convergence is very quick, and by the time p equals 60, the runtime

Table 3: Efficiency models of the evaluated applications. The last column shows the required input sizes (n) for $p = 60$ and an efficiency of 0.8.

Application	Model		rRMSE	Input size for $p = 60$
Cholesky	E_{ac}	$1.09 - 0.51 \cdot \sqrt{p} + 3.11 \cdot 10^{-2} \cdot \sqrt{p} \log n$	9.7%	37,718 × 37,718
	E_{cf}	$1.14 - 0.54 \cdot \sqrt{p} + 3.4 \cdot 10^{-2} \cdot \sqrt{p} \log n$	7.8%	24,685 × 24,685
	E_{ub}	$\min\{1, (2.29 + 2.35 \cdot 10^{-3} \cdot n) \cdot p^{-1}\}$	2.4%	19,500 × 19,500
FFT	E_{ac}	$0.96 - 0.1 \cdot \log p + 5.08 \cdot 10^{-22} n^{4.5} \log p$	19.5%	30,310 × 30,310
	E_{cf}	$1.03 - 0.16 \cdot p^{0.67} + 1.04 \cdot 10^{-2} \cdot p^{0.67} \log n$	4.8%	15,800 × 15,800
	E_{ub}	$\min\{1, (1.19 \cdot 10^{-2} \cdot n^{0.67} \log n) \cdot p^{-1}\}$	4.1%	5,800 × 5,800
Fibonacci	E_{ac}	$0.98 - 5.11 \cdot 10^{-3} \cdot p^{1.25} + 1.76 \cdot 10^{-3} \cdot p^{1.25} \log n$	3.5%	51
	E_{cf}	$0.97 - 1.46 \cdot 10^{-2} \cdot p^{1.25} + 9.26 \cdot 10^{-3} \cdot p^{1.25} \log n$	3.0%	51
	E_{ub}	$\min\{1, (25.48 + 0.49 \cdot n^{2.75} \log n) \cdot p^{-1}\}$	1.5%	49
NQueens	E_{ac}	$1.04 - 0.66 \cdot \sqrt{p} + 0.17 \cdot \sqrt{p} \log n$	13%	14
	E_{cf}	$1.0 - 6.21 \cdot 10^{-2} \cdot p + 1.61 \cdot 10^{-2} \cdot p \log n$	3%	13
	E_{ub}	$\min\{1, (2.18 \cdot n^{2.875} \log n) \cdot p^{-1}\}$	6.6%	12
Sort	E_{ac}	$0.99 - 9.2 \cdot 10^{-3} \cdot p^{1.5} + 2.29 \cdot 10^{-4} \cdot p^{1.5} \log n$	1.9%	350G
	E_{cf}	$1.0 - 4.61 \cdot 10^{-2} \cdot p^{0.75} + 1.62 \cdot 10^{-3} \cdot p^{0.75} \log n$	5.7%	6.6M
	E_{ub}	$\min\{1, (3.53 + 3.32 \cdot 10^{-2} \cdot \sqrt{n}) \cdot p^{-1}\}$	6.7%	1.7M
SparseLU	E_{ac}	$1.02 - 0.46 \cdot p^{0.67} + 3.28 \cdot 10^{-2} \cdot p^{0.67} \log n$	6.3%	12,000 × 12,000
	E_{cf}	$1.05 - 0.48 \cdot p^{0.67} + 3.49 \cdot 10^{-2} \cdot p^{0.67} \log n$	6.1%	11,000 × 11,000
	E_{ub}	$\min\{1, (5.8 \cdot 10^{-5} \cdot n^{1.75} \log n) \cdot p^{-1}\}$	1.7%	7,800 × 7,800
Strassen	E_{ac}	$1.55 - 1.02 \cdot p^{0.25} + 4.59 \cdot 10^{-2} \cdot p^{0.25} \log n$	9.5%	83,600 × 83,600
	E_{cf}	$1.26 - 0.65 \cdot p^{0.33} + 3.89 \cdot 10^{-2} \cdot p^{0.33} \log n$	5.9%	12,680 × 12,680
	E_{ub}	$\min\{1, (0.25 \cdot n^{0.75}) \cdot p^{-1}\}$	2.4%	1,200 × 1,200

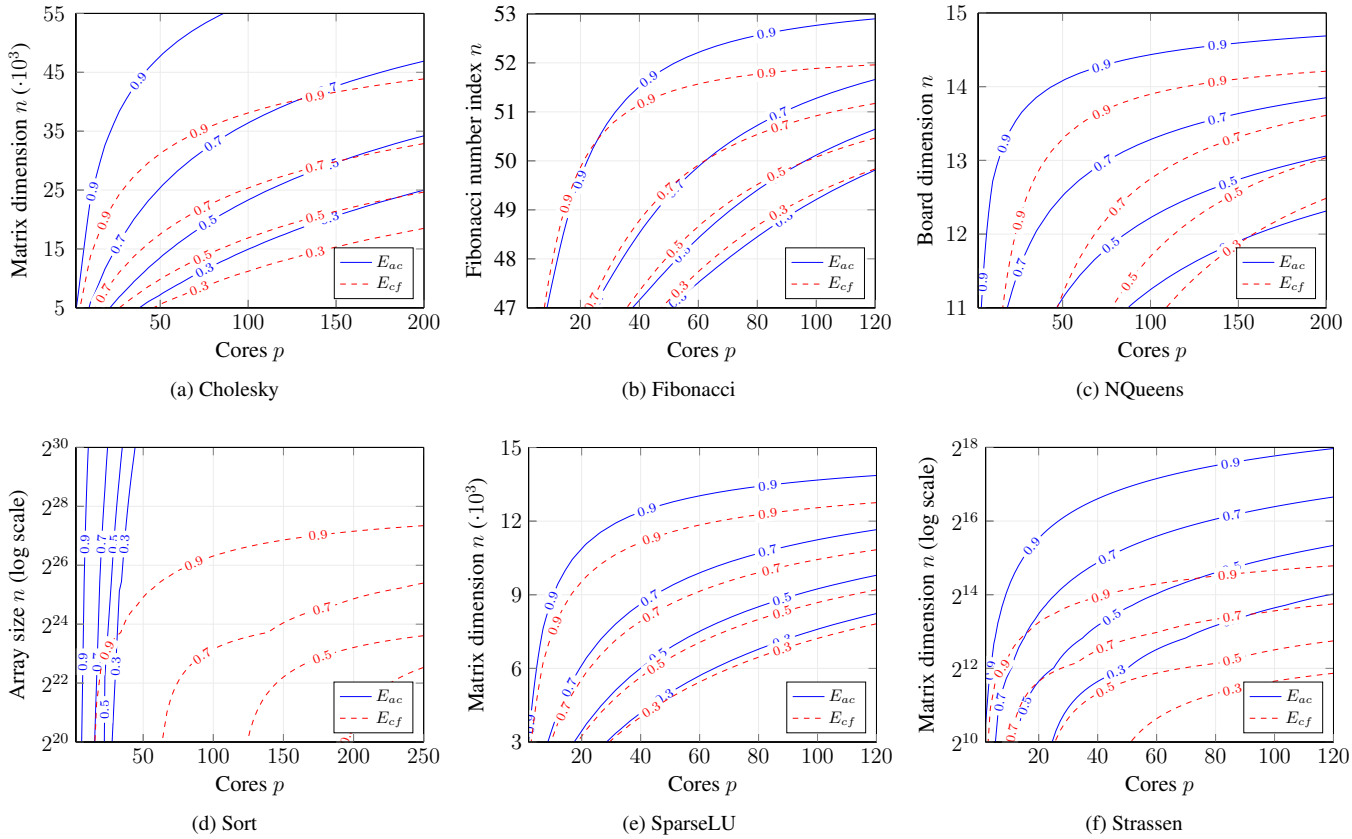
would have almost reached $T_\infty(n)$. In other cases, however, the runtime converged more slowly or stagnated due to prohibitive resource contention. For all of these examples, it makes no sense to continue increasing the core count further, unless the problem size is increased as well. This phenomenon is hardly surprising, but the difficult part is to understand what happens to the efficiency when the problem size changes, or how severe the effects of resource contention are. Even if we consider more optimized versions of the applications, the same questions still remain. The figure also shows that in some cases the difference between the actual run and the replay increases as the core count increases, meaning that the resource contention becomes more severe. In some of the benchmarked applications, we observed that the replay time for $p = 1$ is slightly bigger than the execution time of the original code. This happens due to small perturbation effects of task instrumentation [25]; the impact of this effect, however, is minimal.

4.2.1 Scaling of Depth and Average Parallelism

Table 2 presents the models for $T_\infty(n)$ and $\pi(n)$ (average parallelism) that were created using the results from the TDG analysis. In all of the models the logarithms are binary (i.e., base 2 logarithms). The $\bar{R}^2$ column is the adjusted coefficient of determination (cf. Section 3.2). Although theoretical analysis of the average parallelism in an algorithm is an established practice, these results are the first successful attempt to produce empirical $\pi(n)$ models that are able to uncover potential scalability bugs in real implementations. A $\pi(n)$ that grows more slowly than $T_\infty(n)$ indicates that the implementation is asymptotically not scalable, and hence, contains a scalability bug. Surprisingly, the growth of $\pi(n)$ in Cholesky, FFT, and Strassen is slow compared to $T_\infty(n)$. This suggests that there are potential scalability bugs in these applications. Moreover, a fast growing $T_\infty(n)$ is an indication that the algorithm structure could be improved so that the depth would not become the limiting factor as n increases. The $\pi(n)$ models are used as the basis for the $E_{ub}(p, n)$ models in Table 3, since $E_{ub}(p, n) = \min\{1, \frac{\pi(n)}{p}\}$.

The Fibonacci application implements a trivial algorithm in which each task performs very little work. The TDG in this case is a tree, in which the work grows exponentially with n, and the depth linearly with n. The increase in the depth is proportional to the size of a single task, and therefore very small. This is the reason why the $T_\infty(n)$ model for Fibonacci is constant. Since a constant model is essentially an average of the measured values, the $\bar{R}^2$ is undefined in this case. As an alternative, we could consider the model for the height of the tree, which would be exactly $\mathcal{O}(n)$. The parallelism model, however, accurately reflects the fact that Fibonacci has plenty of available parallelism. Since our PMNF does not contain exponential terms, the model in the table is the best approximation of the exponential behavior in the measured data.

The analysis of the models in Table 2 is an example of how we can discover fundamental scalability limitations in



(a) Cholesky (b) Fibonacci (c) NQueens

(d) Sort (e) SparseLU (f) Strassen

Figure 7: Isoefficiency models of evaluated task-based applications and their replays. The label on each line denotes the efficiency of the line. Each model identifies lower-bounds on the inputs necessary to maintain the constant efficiency underlying the model.

task-based applications and help users answer Question 1 in Section 1.

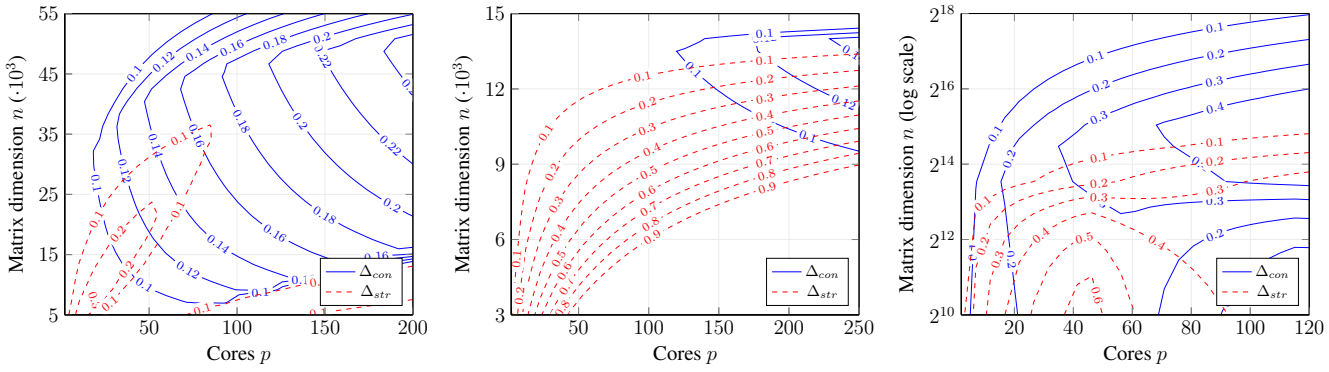
4.2.2 Efficiency Models

Table 3 presents the efficiency models of the evaluated applications. There are three rows for each application listing the three efficiency models that we created (i.e., $E_{ub}(p,n)$, $E_{ac}(p,n)$, and $E_{cf}(p,n)$). In all the models the logarithms are binary. The *rRMSE* column is the relative root-mean-square error. It is a standard statistical factor that measures the relative differences between the observed data and the model, and is defined as: rRMSE $= \sigma/\bar{y}$, where: $\sigma = \sqrt{\sum_{i=1}^{n}(f(x_i)-y_i)^2/n}$, y_i are observed data, and $\bar{y}$ is the mean of the y_i values. For two-parameter models, the rRMSE factor reflects the accuracy of the fit better than $\bar{R}^2$, which is used for the single-parameter models in Table 2. The last column shows the input size n, derived from our models by letting the efficiency E be 0.8 and the core count p be 60, which is the total number of cores on our test machine. In Section 4.2.3, we discuss in greater detail how the input sizes were calculated.

All of the $E_{ac}(p,n)$ and $E_{cf}(p,n)$ models follow the same pattern $C - A \cdot f(p) + B \cdot f(p)g(n)$ that empirically emerged from our measurements. The interpretation of this pattern is that the first term, the constant C, is approximately 1 and it denotes the maximum attainable efficiency. The second term, $-A \cdot f(p)$, reflects the reduction in efficiency that occurs when we increase the core count. The last term, $B \cdot f(p)g(n)$, denotes the efficiency that we gain when we increase the input size. Together these terms reflect the interplay between the core count and the input size, and the effect it has on the efficiency. In the case of FFT, the constant B in the last term of $E_{ac}(p,n)$ is very small, which means that resource contention is a very significant factor and even large increases of the input size are not enough to offset the drop in the efficiency.

Figure 7 depicts the isoefficiency lines $E = 0.9$, $E = 0.7$, $E = 0.5$, and $E = 0.3$ for most of the evaluated applications. The isoefficiency lines start from 0.9, because $E = 1$ is an ideal situation which can hardly be achieved in practice, and therefore has less practical value.

Figure 8 presents the Δ_{con} and Δ_{str} discrepancies for Cholesky, SparseLU, and Strassen. For the sake of brevity,



(a) Cholesky (b) SparseLU (c) Strassen

Figure 8: The Δ_{con} and Δ_{str} discrepancies of selected applications plotted as contour lines. The label of each line is the value of the discrepancy along the line.

we chose only these three applications; the other applications exhibit a simpler behavior regarding Δ_{con} and Δ_{str}. Similar to the efficiency functions, Δ_{con} and Δ_{str} are two-parameter functions that range from 0 to 1. The figure depicts the contour lines of these functions at constant intervals, and the label on each line specifies the value of the discrepancy along this line.

In the case of Cholesky, E_{cf} scales better than E_{ac} and, as Figure 8a shows, Δ_{con} exceeds 0.2. For example, consider $p = 100$ and the 0.7 isoefficiency, in this case, E_{ac} yields approximately $n = 36{,}000$, whereas E_{cf} yields approximately $n = 25{,}000$. This is a significant difference between the input sizes required to achieve the same efficiency, and it suggests that contention is a potential scalability bottleneck.

In the cases of Fibonacci, NQueens, and SparseLU, E_{cf} and E_{ac} scale almost at the same rate, and the isoefficiency lines with the same labels (i.e., efficiencies) are close to each other. In Figure 8b, for example, Δ_{con} stays well below 0.2. Considering that Fibonacci and NQueens are not memory-bound, this result is not surprising. It is, however, surprising to see that SparseLU is not affected by resource contention as one might have initially expected. We can conclude that resource contention is not an obstacle to scalability in these cases.

Sort is clearly affected by resource contention as the differences between the isoefficiency lines of E_{cf} and E_{ac} are very big. In the model $E_{ac} = 0.99 - 9.2 \cdot 10^{-3} \cdot p^{1.5} + 2.29 \cdot 10^{-4} \cdot p^{1.5} \log n$, the presence of $p^{1.5}$ in the second term means that the efficiency drops very quickly as the core count increases. Even though $p^{1.5}$ is also present in the third term, the combination of a small coefficient $2.29 \cdot 10^{-4}$ and $\log n$ makes it hard to offset the efficiency drop. It is not surprising that Sort is impaired by resource contention, but the severity of this impact, as evident from the behavior of E_{ac} and E_{cf}, is unexpected.

Not surprisingly, Strassen, which is heavily memory-bound, is also affected by resource contention. In some cases, as Figure 8c shows, Δ_{con} reaches 0.4, and if we consider, for example, $p = 100$ and the 0.7 isoefficiency lines, the input size n in E_{ac} would be about four times as large as in E_{cf}. The discrepancy is big when both the core count and the input sizes are either low or high. In the former case, the threads most likely contend for shared caches; whereas, in the latter case, they contend for memory bandwidth.

From the Δ_{con} analysis we can conclude that, for Cholesky, Sort, and Strassen, poor scaling is a result of a prohibitive contention overhead. This conclusion is an example of how, using our approach, we can answer Question 2 in Section 1.

As suggested by Figure 8a and the example input sizes for E_{ub} and E_{cf} in Table 3, the Δ_{str} of Cholesky, Fibonacci, and NQueens is rather small. However, Figures 8b and 8c show, for SparseLU and Strassen, that Δ_{str} is clearly bigger for certain ranges of p and n. Although this discrepancy becomes smaller as n increases, there is still room for improvement of either task dependencies, scheduling, or granularity. This insight is an example of how our approach helps to answer Question 3 in Section 1.

4.2.3 Co-Design Use Cases

We can use the efficiency models to derive a realistic approximation of the input size n that should be used to run an application with constant efficiency on a given core count p. For example, the actual efficiency model for Strassen is $E_{ac} = 1.55 - 1.02 \cdot p^{0.25} + 4.59 \cdot 10^{-2} \cdot p^{0.25} \log n$. For an efficiency of 0.8 and $p = 60$ we can derive the equation $0.8 = 1.55 - 1.02 \cdot 60^{0.25} + 4.59 \cdot 10^{-2} \cdot 60^{0.25} \log n$, and after solving it we would obtain $n = 83{,}600$, which means the application's input in this case should be a $83{,}600 \times 83{,}600$ matrix. This directly answers Question 4 in Section 1, and helps users efficiently utilize all the computing resources

they have. We used the Symbolic Math Toolbox in MATLAB [4] both to solve the equation in this example and to derive the example input sizes in Table 3.

For some of the applications, such as Fibonacci, NQueens, and SparseLU, the example input sizes in the table are within the range of the inputs that we used for benchmarking. This means that the efficiency scaling in these cases is generally good. For other applications, such as Cholesky, we validated the example input size by running the application on all of the 60 cores of our test machine. In the cases of Sort and Strassen, however, validating the input size was impossible due to prohibitive memory requirements.

The inputs in Table 3 provide examples for co-design use cases. By calculating input sizes for future core counts, hardware designers can see whether the inputs are realistic and feasible. The input size for Sort, for example, shows that utilizing all of the 60 cores efficiently also requires adding a substantial amount of memory to a future machine.

Similar to the input size case, we can calculate the required core count, given a specific input size n. In this case, hardware designers can estimate the number of cores they will need for a predefined input size. Unlike the previous case, this approach can provide an answer to how many processing elements and memory controllers in a future machine would be suitable for an existing application with realistic inputs. We can see, for example, that Fibonacci, NQueens, and SparseLU would be suitable for a future machine with higher core counts. This is an example of how our approach can help hardware designers answer Question 5 in Section 1.

Hardware designers could use the generated contention models to design shared resources for future systems-on-chip. For example, the sizes of shared resources such as last-level cache, coherence networks, memory controllers, or input/output channels could be tuned to a specific set of applications using scaling and contention models. We leave details of such micro-architectural discussions for future work, as it lies outside the scope of this paper.

5. Related Work

Directed acyclic graphs, or more specifically task dependency graphs, are an established model of multithreading [6, 8, 11, 16, 18]. They were used in earlier work, even before the emergence of task-based programming models, for analyzing and understanding parallel computations. Perhaps the biggest strengths of the TDG model are its simplicity and that it allows two fundamental metrics to be defined—*work* and *depth*–which provide important bounds on performance and speedup.

An earlier work by Eager et al. [16] used TDGs, as well as work and average parallelism metrics, to investigate the tradeoffs between speedup and efficiency in parallel computations. Blelloch [6] used work and depth metrics to analyze parallel algorithms on a PRAM machine model in the context of the NESL parallel language [5].

The designers of Cilk [7, 17], an early task-based programming model, used TDGs, as well as work and depth metrics, to analyze and understand Cilk's performance. The Cilkview scalability analyzer [21] is a more recent work for profiling and benchmarking Cilk applications. It first instruments the code, and then constructs the TDG once the code has finished running. After benchmarking the code, Cilkview visualizes the measured speedup along with both lower and upper speedup limits (cf. Section 2.2). Although Cilkview is a useful tool for understanding the performance and some limitations of a task-based application, it focuses on just one problem size, ignoring the isoefficiency concept.

Previous studies explored the problem of overheads in task-based applications [27, 33]. The authors identified metrics to characterize the time spent by the threads doing either unrelated work or no useful work at all. They then suggested a number of techniques to improve the runtime of an application on a given number of cores. In another study [35], the authors created low-level models of memory bandwidth that allow the bandwidth usage to be predicted. In our study, in contrast, we take a more general approach to understanding resource-contention effects. The multi-parameter models that we create take both the core count and the problem size into account, allowing the user to understand the interplay between these two parameters.

Our task replay engine shares some similarities with Prometheus [24] and TaskSim [30]. Both tools enable the accurate simulation of task-based codes by either, in the case of Prometheus, constructing a TDG and simulating hardware contention, or, in the case of TaskSim, gathering execution traces. Since our aim is not an accurate reproduction of execution, but a contention-free execution, we used a simpler approach for the task replay.

Although isoefficiency is a well-known concept [19, 20, 29], the empirical analysis of it has not received much attention so far. To the best of our knowledge, there are no studies that explicitly model empirical isoefficiency of task-based applications.

6. Conclusion

In this paper, we propose a novel method that helps users, application developers, and hardware designers identify the causes of limited scalability in task-based applications. The method provides insights into the effects of resource contention on efficiency, and allows users to analyze how severe this contention is. By modeling how the depth and the average parallelism change as the input increases, our method also allows users to identify scalability bugs in task-based applications. Average parallelism that scales poorly compared to the depth indicates that the application would not run optimally for bigger inputs.

We identify three efficiency functions that describe the application behavior in different scenarios, namely, an ideal upper-bound efficiency, actual efficiency reflecting the application behavior, and contention-free efficiency based on the replay of the TDG. By analyzing the discrepancies between these efficiency functions, we are able to provide answers to questions regarding co-design aspects, the connection between poor scaling and resource contention, optimization potential, and the presence of scalability bugs.

We conclude that our methodology is a viable approach for analyzing both the effects of resource contention on efficiency and further optimization potential. It provides users with an insight into whether the obstacle to scaling is resource contention or insufficient parallelism in the structure of the TDG. In addition, users can also calculate the required input sizes to keep efficiency constant on a given core count. This approach can be used in co-design analysis to understand how many processing elements to put in a future machine, such that we can have high efficiency with realistic application inputs. We envision this methodology will be adopted for analyzing present and future task-based applications as many-core hardware becomes ever more ubiquitous.

Acknowledgments

This work was supported in part by the German Research Foundation (DFG) and the Swiss National Science Foundation (SNSF) through the DFG Priority Programme 1648 *Software for Exascale Computing* (SPPEXA). Additional support was provided by the German Federal Ministry of Education and Research (BMBF) through the Score-E project under grant no. 01IH13001G, and by the US Department of Energy under grant no. DE-SC0015524. We would also like to acknowledge the University Computing Center (Hochschulrechenzentrum) of TU Darmstadt for providing us with access to their supercomputers. Finally, we wish to thank the OmpSs team at Barcelona Supercomputing Center (BSC) for their support.

References

[1] BSC Application Repository. `https://pm.bsc.es/projects/bar/wiki/Applications`.

[2] Extra-P – Automated Performance-modeling Tool. `http://www.scalasca.org/software/extra-p`.

[3] Graphviz – Graph Visualization Software. `http://www.graphviz.org`.

[4] MATLAB. `http://www.mathworks.com`.

[5] G. E. Blelloch. NESL: A Nested Data-Parallel Language. Technical report, 1992.

[6] G. E. Blelloch. Programming Parallel Algorithms. *Commun. ACM*, 39(3):85–97, 1996.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(8):55–69, 1997.

[8] R. P. Brent. The Parallel Evaluation of General Arithmetic Expressions. *Journal of the ACM (JACM)*, 21(2):201–206, April 1974.

[9] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes. In *Proc. of the 2013 ACM/IEEE Conference on Supercomputing (SC '13)*, pages 45:1–45:12. ACM, 2013.

[10] A. Calotoiu, D. Beckingsale, C. W. Earl, T. Hoefler, I. Karlin, M. Schulz, and F. Wolf. Fast Multi-Parameter Performance Modeling. In *Proc. of the 2016 IEEE International Conference on Cluster Computing (CLUSTER '16)*, pages 1–10, September 2016. (to appear).

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and S. Clifford. Multithreaded Algorithms. In *Introduction to Algorithms, Third Edition*, pages 772–812. The MIT Press, 3rd edition, 2009.

[12] A. Duran, J. Corbalán, and E. Ayguadé. An Adaptive Cut-off for Task Parallelism. In *Proc. of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*, pages 36:1–36:11. IEEE Press, 2008.

[13] A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of OpenMP Task Scheduling Strategies. In *Proc. of the 4th International Conference on OpenMP in a New Era of Parallelism (IWOMP '08)*, pages 100–110. Springer-Verlag, 2008.

[14] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proc. of the 2009 International Conference on Parallel Processing (ICPP '09)*, pages 124–131. IEEE, 2009.

[15] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[16] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup Versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[17] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*, pages 212–223. ACM, 1998.

[18] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[19] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2002.

[20] A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *Parallel Distributed Technology: Systems Applications*, 1(3): 12–21, 1993.

[21] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview Scalability Analyzer. In *Proc. of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*, pages 145–156, 2010.

[22] T. Hoefler and R. Belli. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results. In *Proc. of the 2015 ACM/IEEE Conference on Supercomputing (SC '15)*, pages 73:1–73:12. ACM, 2015.

[23] T. Hoefler, W. Gropp, R. Thakur, and J. L. Träff. Toward Performance Models of MPI Implementations for Understanding Application Scaling Issues. In *Proc. of the European MPI Users' Group Meeting (EuroMPI '10)*, pages 21–30. Springer-Verlag, 2010.

[24] G. Kestor, R. Gioiosa, and D. Chavarra-Miranda. Prometheus: scalable and accurate emulation of task-based applications on many-core systems. In *Proc. of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '15)*, pages 308–317. IEEE, 2015.

[25] D. Lorenz, P. Philippen, D. Schmidl, and F. Wolf. Profiling of OpenMP Tasks with Score-P. In *Proc. of the 2012 41st International Conference on Parallel Processing Workshops (ICPPW '12)*, pages 444–453, September 2012.

[26] M. R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, and S. Poole. Modeling and Predicting Performance of High Performance Computing Applications on Hardware Accelerators. *International Journal of High Performance Computing Applications*, 27(2):89–108, May 2013.

[27] S. L. Olivier, B. R. de Supinski, M. Schulz, and J. F. Prins. Characterizing and Mitigating Work Time Inflation in Task Parallel Programs. In *Proc. of the 2012 ACM/IEEE Conference on Supercomputing (SC '12)*, pages 65:1–65:12. IEEE Computer Society Press, 2012.

[28] OpenMP Architecture Review Board. OpenMP application programming interface, version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[29] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[30] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, and M. Valero. On the simulation of large-scale architectures using multiple application abstraction levels. *ACM Transactions on Architecture and Code Optimization*, 8(4):1–20, 2012.

[31] S. Shudler, A. Calotoiu, T. Hoefler, A. Strube, and F. Wolf. Exascaling Your Library: Will Your Implementation Meet Your Expectations? In *Proc. of the 29th ACM International Conference on Supercomputing (ICS '15)*, pages 165–175. ACM, June 2015.

[32] N. R. Tallent and A. Hoisie. Palm: Easing the Burden of Analytical Performance Modeling. In *Proc. of the 28th ACM International Conference on Supercomputing (ICS '14)*, pages 221–230. ACM, 2014.

[33] N. R. Tallent and J. M. Mellor-Crummey. Effective Performance Measurement and Analysis of Multithreaded Applications. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09)*, pages 229–240. ACM, 2009.

[34] U.S. Department of Energy. The Opportunities and Challenges of Exascale Computing. Office of Science, Washington, D.C., 2010. `http://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf`.

[35] W. Wang, T. Dey, J. W. Davidson, and M. L. Soffa. DraMon: Predicting Memory Bandwidth Usage of Multi-threaded Programs With High Accuracy and Low Overhead. In *Proc. of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA '14)*, pages 380–391, February 2014.