# Provably and Practically Efficient Granularity Control

Umut A. Acar
Carnegie Mellon University and Inria
USA and France
umut@cs.cmu.edu

Vitaly Aksenov
Inria and ITMO University
France and Russia
vitalii.aksenov@inria.fr

Arthur Charguéraud
Inria & Université de Strasbourg, CNRS, ICube
France
arthur.chargueraud@inria.fr

Mike Rainey
Indiana University and Inria
USA and France
me@mike-rainey.site

## Abstract

Over the past decade, many programming languages and systems for parallel-computing have been developed, e.g., Fork/Join and Habanero Java, Parallel Haskell, Parallel ML, and X10. Although these systems raise the level of abstraction for writing parallel codes, performance continues to require labor-intensive optimizations for coarsening the granularity of parallel executions. In this paper, we present provably and practically efficient techniques for controlling granularity within the run-time system of the language. Our starting point is "oracle-guided scheduling", a result from the functional-programming community that shows that granularity can be controlled by an "oracle" that can predict the execution time of parallel codes. We give an algorithm for implementing such an oracle and prove that it has the desired theoretical properties under the nested-parallel programming model. We implement the oracle in C++ by extending Cilk and evaluate its practical performance. The results show that our techniques can essentially eliminate hand tuning while closely matching the performance of hand tuned codes.

***CCS Concepts*** • **Software and its engineering → Parallel programming languages**;

***Keywords*** parallel programming languages, granularity control

**ACM Reference Format:**
Umut A. Acar, Vitaly Aksenov, Arthur Charguéraud, and Mike Rainey. 2019. Provably and Practically Efficient Granularity Control. In *PPoPP '19: Symposium on Principles and Practice of Parallel Programming, February 16–20, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 36 pages. https://doi.org/10.1145/3293883.3295725



*PPoPP '19, February 16–20, 2019, Washington, DC, USA*

ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3295725

## 1 Introduction

The proliferation of multicore hardware in the past decade has brought shared-memory parallelism into the mainstream. This change has led to much research on *implicit threading*, or, alternatively, implicit parallelism, which goes back to early parallel programming languages such as Id [Arvind and Gostelow 1978] and Multilisp [Halstead 1985]. Implicit threading seeks to make parallel programming easier by delegating certain tedious but important details, such as the scheduling of parallel tasks, to the compiler and the run-time system. There has been much work on specialized programming languages and language extensions for parallel systems, for various forms of implicit threading, including OpenMP [OpenMP Architecture Review Board 2008], Cilk [Frigo et al. 1998], Fork/Join Java [Lea 2000], Habanero Java [Imam and Sarkar 2014], NESL [Blelloch et al. 1994],TPL [Leijen et al. 2009], TBB [Intel 2011], X10 [Charles et al. 2005], parallel ML [Fluet et al. 2011; Guatto et al. 2018; Jagannathan et al. 2010; Raghunathan et al. 2016], and parallel Haskell [Chakravarty et al. 2007; Keller et al. 2010].

Nearly all of these languages support a lightweight syntax for expressing parallelism. For example, in Cilk Plus, the programmer needs two keywords to express parallelism, **`spawn`** and **`sync`**, which respectively indicate a computation that can be executed in parallel and a computation that must be synchronized with. These two keywords suffice to express parallel loops and *nested parallelism*, wherein parallel computations may themselves start and synchronize with other parallel computations. When combined with the powerful features of modern programming languages, these simple constructs enable elegant expression of parallelism. For example, we can implement a "parallel map" that applies some function `f` to each element of an array in a C++-like parallel language by using a parallel-for loop (`pfor`), as shown in the `map` function in Figure 1. By using templates, this implementation enables many different forms of mapping, e.g., `map` can be used to increment the elements in an input array of integers or it can be used to implement a matrix-vector multiplication by passing as an argument an array of arrays (vectors) representing the matrix.

```
template <F,T,S>
void map(F f, T* a, S* b, int n)
  pfor (i = 0; i < n; i++)
    b[i] = f(a[i])


int grain = ... // to be determined
template <F,T,S>
void map_coarsened(F f, T* a, S* b, int n)
  pfor (i = 0; i < (n + grain - 1) / grain; i++)
    for (j = i · grain; j < min(n, (i + 1) · grain); j++)
      b[j] = f(a[j]);
```

**Figure 1.** Parallel map, uncoarsened and coarsened implementations.

Even though implicit parallelism can be elegant, and offer important asymptotic efficiency guarantees [Acar et al. 2002, 2013; Blelloch and Greiner 1996; Blumofe and Leiserson 1999; Frigo et al. 1998], there remains a key concern: the constants hidden in these asymptotic analyses, a.k.a., the *overheads of parallelism* can easily overwhelm the benefits of parallelism in practice. This challenge is especially difficult in high-level, declarative codes, such as `map`, and is sometimes considered a "holy grail" in parallelism research (e.g., [Tzannes et al. 2014]). For example, the simple `map` example above could run as much as 10-100x slower than an optimized implementation due to these overheads.

To control the overheads of parallelism, the current state of the art requires the programmer to *tune* the code to perform *granularity control* or *coarsening* so as to amortize the overheads [Intel 2011]. To do so, the programmer identifies for each potential parallel computation a *sequential alternative*, a semantically equivalent sequential piece of code, and makes sure that this sequential alternative is executed for small (and only for small) computations. As an example, imagine an instance of `map` where the programmer increments each element of an integer array by one. To perform this operation efficiently, the programmer should "bunch" iterations into blocks that are executed sequentially and select the bunch size to be large enough to amortize the cost of parallelism. The function `map_coarsened` in Figure 1 shows such a tuned implementation of `map`, where the value `grain` determines the bunch size.

The optimized `map` implementation is far from its elegant predecessor, but elegance aside, it is not easy to make `map` perform well, because the programmer still has to choose the setting for `grain`. There are at least two problems with this approach.

- The optimal setting for the grain depends on the architecture as well as the specific inputs and requires a manual, labor-intensive search [Bergstrom et al. 2012; Feeley 1992, 1993a; Hiraishi et al. 2009; Intel 2011; Lopez et al. 1996; Rainey 2010; Sanchez et al. 2010a; Tzannes et al. 2014].
- As noted by many researchers, such optimizations could also harm performance portability by overfitting the code to a particular machine [Acar et al. 2016; Huelsbergen et al. 1994; Lopez et al. 1996; Pehoushek and Weening 1990; Tzannes et al. 2014; Weening 1989].
- It can be impractical to tune generic codes such as `map`, because the optimal `grain` depends on the template parameters as well as the actual arguments to `map`, some of which, such as the function argument, might not be known a priori.

Prior work therefore proposed techniques for (semi-) automating this process [Huelsbergen et al. 1994; Lopez et al. 1996; Pehoushek and Weening 1990; Weening 1989]. Yet, these techniques remain heuristics that fall short of delivering strong theoretical and practical efficiency guarantees. A more recent paper on oracle-guided scheduling [Acar et al. 2016] shows that strong bounds can be achieved, but assumes an oracle that can predict the future. The authors do present an implementation for an oracle, but their implementation technique can support only certain flat-parallel programs, where parallel computations cannot be nested in other parallel computations. As such, this limitation excludes nearly all interesting parallel programs expressible in high-level parallel programs today.

In this paper, we present provably efficient and practical techniques for controlling granularity for nested-parallel programs. As in oracle-guided scheduling [Acar et al. 2016], we ask the programmer to provide an *abstract cost function*, e.g., asymptotic cost, for each piece of parallel code. We then provide an online prediction algorithm that uses such information to implement the oracle by predicting the actual work (1-processor run-time) of parallel codes. Our major contribution is this prediction algorithm and the result that the algorithm delivers efficient and practical granularity control, for any nested-parallel program, under a certain number of assumptions that we present in detail. The key insight behind our algorithm is to break a circular dependency encountered in implementing the oracle in prior work [Acar et al. 2016] by using an iterative refinement algorithm that, after a small number of iterations, provably converges to the desired values. We present a theoretical analysis of the key efficiency properties of the algorithm and evaluate the approach by comparison to a highly (hand) optimized suite of benchmarks from the Problem Based Benchmark Suite [Blelloch et al. 2012]. Our results show that our techniques can eliminate the need for hand tuning in many cases and closely match the performance of the hand optimized code.

The specific contributions of this paper include:

- an online algorithm for automatic granularity control in lightly annotated, nested-parallel computations,
- end-to-end tight bounds on run time or parallel codes,

```
T f(x) {
  T r // local variable to store the result
  spguard([&] { return c(x) },      // cost function
          [&] { if |x| == 1 ...      // parallel body
                else (x1, x2) = divide(x)
                     r1 = spawn f(x1)
                     r2 =       f(x2)
                     sync
                     r = conquer(r1, r2) },
          [&] { r = g(x) })      // sequential body
  return r }
```

**Figure 2.** A sample use of spguard. Higher-order functions are expressed using the C++ lambda-expression syntax.

- an implementation for Cilk Plus, which implements work stealing for C++, and
- an evaluation on a broad collection of hand-tuned benchmarks in Cilk Plus.

## 2 Algorithmic granularity control

Our goal is to transfer the burden of granularity control to a capable, library implementation. To this end, we ask the programmer to provide for each parallel function a *series-parallel guard*, by using the keyword `spguard`. A spguard consists of: a *parallel body*, which is a lambda function that performs a programmer-specified parallel computation; a *sequential body*, which is a lambda function that performs a purely sequential computation equivalent to the parallel body, i.e., performing the same side-effects and delivering the same result, a *cost function*, which gives an abstract measure, as a positive number, of the work (run-time cost) that would be performed by the sequential body.

At a high level, a spguard exploits the result of the cost function to determine whether the computation involved is small enough to be executed sequentially, i.e., without attempting to spawn any subcomputation. If so, the spguard executes the sequential body. Otherwise, it executes the parallel body, which would typically spawn smaller subcomputations, each of them being similarly guarded by a spguard.

To see the basic idea behind our algorithm, consider the example shown in Figure 2, involving a single spguard. Suppose that we have a parallel function `f(x)` which, given an argument `x`, performs some computation in divide-and-conquer fashion, by recursively calling itself in parallel, as shown above. Assume the body of this function to be controlled by a spguard, with `c(x)` denoting the cost function, and `g(x)` denoting the sequential body, that is, a purely sequential function that computes the same result as `f(x)`.

The cost function may be any programmer-specified piece of code that, given the context, computes a value in proportion to the one-processor execution time of the sequential body. Typically, the cost function depends on the arguments provided to the current function call. A good choice for the cost function is the average asymptotic complexity of the sequential body, e.g., $n \log n$, or $n$, or $\sqrt{n}$, where $n$ denotes the size of the input. The programmer need not worry about constant factors because spguards are able to infer them on-line, with sufficient accuracy.

In a real implementation, the sequential body can be left implicit in many cases, because it can be inferred automatically. For example, the sequential body for a parallel-for loop can be obtained by replacing the parallel-for primitive with a sequential-for. Likewise, in many instances, the complexity function is linear, allowing us to set it to the default when not specified. In our library and experiments, we use this approach to dramatically reduce the annotations needed.

### 2.1 Sequentialization decisions

Our algorithm aims at sequentializing computations that involve no more than a small amount of work. To quantify this amount, let us introduce the *parallelism unit*, written $\kappa$, to denote the smallest amount of work (in units of time) that would be profitable to parallelize on the host architecture. The value of $\kappa$ should be just large enough to amortize the cost of creating and managing a single parallel task. On modern computers, this cost can be from hundreds to thousands of cycles. Practical values for $\kappa$ therefore range between 25 and 500 microseconds.

Intuitively, we aim at enforcing the following policy: if the result of `f(x)` can be obtained by evaluating the sequential body `g(x)` in time less than $\kappa$, then `g(x)` should be used. Under a small number of assumptions detailed in further, this policy leads to provably efficient granularity control.

One central question is how to predict whether a call to `g(x)` would take less than $\kappa$ units of time. Assume, to begin with, a favorable environment where (1) the hardware is predictable in the sense that the execution time is in proportion to the number of instructions, and (2) the cost function `c(x)` gives an estimate of the asymptotic number of instructions involved in the execution of `g(x)`. For a given input `x`, let $N$ denote the value of `c(x)`, and let $T$ denote the execution time of `g(x)`. By definition of "asymptotic", there exists a constant $C$ such that: $T \approx C \cdot N$. Our algorithm aims at computing $C$ by sampling executions, and then it exploits this constant to predict whether particular calls to `g(·)` take fewer than $\kappa$ units of time.

More precisely, for an input `x_i` being evaluated sequentially, that is, through a call to `g(x_i)`, we may measure the execution time of `g(x_i)`, written $T_i$, and we may compute the value of `c(x_i)`, written $N_i$. From a collection of samples of the form $(T_i, N_i)$, we may evaluate the value of the constant $C$ by computing the ratios $T_i/N_i$. In reality, the actual value of the constant can vary dramatically depending on the size of the computation, in particular due to cache effects—we will return to that point. There is a much bigger catch to be addressed first.

In order to decide which computations are safe to execute using the sequential body `g(·)`, our algorithm needs to first know the constant $C$. Indeed, without a sufficiently accurate estimate of the constant, the algorithm might end up invoking `g(·)` on a large input, thereby potentially destroying all available parallelism. Yet, at the same time, in order to estimate the constant $C$, the algorithm needs to measure the execution time of invocations of `g(·)`. Thus, determining the value of $C$ and executing the algorithm `g(·)` are interdependent. Resolving this critical circular dependency is a key technical challenge.

At a high level, our algorithm progressively sequentializes larger and larger computations. It begins by sequentializing only the base case, and ultimately converges to computations of duration $\kappa$. Each time that we sequentialize a computation by calling `g(·)` instead of `f(·)`, we obtain a measure. This measure may be subsequently used to predict that another, slightly larger input may also be processed sequentially. We are careful to increase the input size progressively, in order to always remain on the safe side, making sure that our algorithm never executes sequentially a computation significantly longer than $\kappa$.

Because our algorithm increases the cost (as measured by the cost function) of sequentialized computations each time at least by a multiplicative factor, called $\alpha$, the estimation converges after just a logarithmic number of steps. The *growth rate*, $\alpha$ controls how fast sequentialized computations are allowed to grow. Any $\alpha > 1$ could be used; values between 1.2 and 5 work well in practice.

### 2.2 Nested parallelism

When dealing with a single spguard, the process described above generally suffices to infer the constant associated with that spguard. However, the process falls short for programs involving nested parallelism, e.g., nested loops or mutually-recursive functions. To see why, consider a function `h(·)` that consists of a spguard whose parallel body performs some local processing and then spawns a number of calls to completely independent functions. Because `h(·)` is not a recursive function with a base case, the algorithm described so far does not have a chance to follow the convergence process; the spguard of `h(·)` would have no information whatsoever about its constant, and it would always invoke the parallel body, consequently failing to control granularity.

To address this issue and support the general case of nested parallelism, we introduce an additional mechanism. When executing the parallel body of a spguard, our algorithm computes the sum of the durations of all the pieces of sequential computation involved in the execution of that parallel body. This value gives an upper bound on the time that the sequential body would have taken to execute. This upper bound enables deriving an over-approximation of the constant. Our algorithm uses this mechanism to make *safe* sequentialization decisions, i.e., to sequentialize computations that certainly require less than $\kappa$ time. By measuring the duration of such sequential runs, our algorithm is then able to refine its estimate of the constant. It may subsequently sequentialize computations of size closer to $\kappa$.

Overall, our algorithm still progressively sequentializes larger and larger subcomputations, only it is able to do so by traversing distinct spguards with different constant factors.

### 2.3 Dealing with real hardware

Our discussion so far assumes that execution times may be predicted by the relationship $T \approx C \cdot N$. But in reality, this assumption is not the case. The reason is that the ratios $T/N$ may significantly depend on the input size. For example, one can observe on a microbenchmark that processing an input that does not fit into the L3 cache may take up to 10 times longer to execute than a just slightly smaller input that fits in the cache. In fact, even two calls to the same function on the same input may have measured time several folds apart, for example, if the first call needs to load the data into the cache but not the second one.

We design our algorithm to be robust in the face of large variations of the execution times typical of modern hardware. In addition to validating empirically that our algorithm behaves well on hundreds of runs of our full benchmark suite, we formally prove its robustness property. To that end, we consider a relatively realistic model that takes into account the variability of execution times typical of current hardware. More precisely, we develop our theory with respect to an abstract notion of "work", which we connect both to runtime measures and to results of costs functions, as described next.

First, we assume that runtime measures may vary by no more than a multiplicative factor $E$ from the work, in either direction. Second, we require that, for the program and the growth rate $\alpha$ considered, there exists a value $\beta$ such that, for any cost function involved in the program, the following property holds: if the cost function applied to input $J$ returns a value no more than $\alpha$ bigger than for input $I$, then the work associated with input $J$ is at most $\beta$ times bigger than the work for input $I$. Intuitively, this property ensures that the work increases with the cost, but not exceeding a maximal rate of increase.

One important feature of our algorithm is that its execution does not require knowledge of $E$ or $\beta$. These parameters are only involved in the analysis. It may nevertheless be useful to estimate the values of $E$ and $\beta$ that apply on the target hardware. In the technical appendix, we show the results of estimating these parameters on our test harness.

### 2.4 Analysis

Our algorithm relies on several important assumptions. These assumptions, whose formal statements may be found in the

technical appendix, are matched by a large class of realistic parallel algorithms.

First, our algorithm assumes that, for each spguard, the sequential body evaluates no slower than the corresponding parallel body on a single processor. (The sequential body might always be obtained by replacing spawns with sequences in the parallel body.) Furthermore, the sequential body should not run arbitrarily faster than the parallel body when executed with all of its inner spguards sequentialized. This assumption is required to know that it is safe to exploit the execution time of the parallel body to over-approximate that of the sequential body.

To better understand this latter part of the assumption, consider the following example. Assume a purely sequential algorithm, whose cost is $a \cdot N$ for an input of size $N$, for some constant $a$. Assume also a divide-and-conquer parallel algorithm that recursively divides its input in halves, such that for an input of size $N$, the division process (and, symmetrically, the conquer process) takes time $b \cdot N$, for some constant $b$; the cost of this parallel algorithm is thus $b \cdot N \log N$. On the one hand, the sequential algorithm may be assumed to be faster than the parallel algorithm, because it is always possible to instantiate the sequential algorithm as the sequential execution of the parallel algorithm. On the other hand, let us check the requirement that "the sequential body does not run arbitrarily faster than the parallel body executed with all its inner spguards being sequentialized." Executing one step of the divide-and-conquer process (as described by the parallel body) and then immediately calling the sequential body in both branches has cost: $b \cdot N + 2 \cdot a \cdot \frac{N}{2}$. Executing the sequential body directly has cost: $a \cdot N$. The ratio between these two costs is equal to: $1 + \frac{b}{a}$, which is bounded by a constant, as required. Many practical algorithms exhibit a similar pattern.

Second, our algorithm assumes that spguards evaluate in constant time and are called regularly enough through the call tree. Without this latter assumption, the program could have a spguard called on an input that takes much longer than $\kappa$ to process, with the immediately nested spguard called on an input that takes much less than $\kappa$, leaving no opportunity for our algorithm to sequentialize computations of duration close to $\kappa$.

Third and last, our algorithm assumes some balance between the branches of a fork-join. Without this assumption, the program could be a right-leaning tree, with all left branches containing tiny computations. Such an ill-balanced program is a poorly-designed parallel program that would be slow in any case.

As we prove through a careful analysis detailed in the technical appendix, under the aforementioned assumptions, our algorithm is efficient. Our bound generalizes Brent's bound ($T_P \leq \frac{w}{P} + s$) [Zadeh 2017] by taking into account the overheads of thread creation (cost of spawning and managing threads) and granularity control (including the cost of evaluating cost functions, which are assumed to execute in constant time). A simplified statement of the bound, using big-O notation, appears next.

**Theorem 2.1.** *Under the above assumptions, with parallelism unit $\kappa$, the runtime on $P$ processors using any greedy scheduler of a program involving work $w$ and span $s$ is bounded by:*

$$T_P \;\leq\; \left(1 + \frac{O(1)}{\kappa}\right) \cdot \frac{w}{P} \;+\; O(\kappa) \cdot s \;+\; O(\log^2 \kappa).$$

The most important term in this bound is the first term: the overheads impact the work term $w$ by only a small factor $O(1)/\kappa$, which can be reduced by controlling $\kappa$. The second term states that doing so increases the span by a small factor $O(\kappa)$, and that the granularity control comes at a small logarithmic overhead $O(\log^2 \kappa)$, which is due to our granularity control algorithm. Although our theorem is stated, for simplicity, in terms of greedy schedulers, the bound could be easily adapted, e.g., to work-stealing schedulers.

### 2.5 Pseudo-code for the estimator and spguard

In what follows, we present pseudo-code for the core of our algorithm. For each syntactic instance of a spguard, we require a unique *estimator* data structure to store the information required to estimate the corresponding constant factor. (In the case of templated code, we instrument the template system to ensure that each template instantiation allocates one independent estimator data structure.) The top of Figure 3 shows the code of the estimator data structure. The estimator data structure maintains only one variable: `Nmax`, which tracks the maximum (abstract) work of a sample computation that took less than $\kappa$ time.

The function `report` provides the estimator with samples. It takes as argument `T`, the execution time, and `N`, the abstract work. If `T` is less than the parallelism threshold $\kappa$ and `N` is greater than `Nmax`, then `Nmax` is updated to `N`. The function `report` is protected against data races by using an `atomic` block. Our library implements this atomic block by using a single compare-and-swap (CAS) operation on `Nmax`. In the case of a race, compare-and-swap would fail and our code would try again until either it succeeds, or `Nmax` becomes greater than `N`.

The function `is_small`, takes as argument a cost `N` and returns a boolean indicating whether `N` is no more than $\alpha \cdot$`Nmax`. The intention is to allow sequentialization of computations whose execution is less than $\alpha \cdot \kappa$. Indeed, `Nmax` is associated with a previously-sampled computation that took less than $\kappa$ time. The multiplicative factor $\alpha$ involved here enables extrapolation. It allows to sequentialize computations whose abstract work is larger than any previously-seen computations, although not arbitrarily larger: at most by a factor $\alpha$.

The bottom of Figure 3 shows the functions `fork2join` and `spguard`. The `fork2join` function executes its two branches

```
1  const double κ    // parallelism unit
2  const double α    // growth factor
3
4  class estimator
5    int Nmax = 0    // max complexity measure
6    void report(int N, time T)
7      atomic { if T ≤ κ and N > Nmax
8                 then Nmax = N }
9
10   bool is_small(int N)
11     return N ≤ α · Nmax
12
13 template <Body_left, Body_right>
14 void fork2join(Body_left bl, Body_right br)
15   spawn bl()
16   br()
17   sync
18
19 template <Complexity, Par_body, Seq_body>
20 void spguard(estimator* es, Complexity c,
21               Par_body pb, Seq_body sb)
22   int N = c()
23   time work = if es.is_small(N)
24                 then measured_run(sb)
25                 else measured_run(pb)
26   es.report(N, work)
```

**Figure 3.** Pseudocode for the core algorithm.

in parallel and resumes after completion of both branches. The function `spguard` takes an estimator, a parallel body, a sequential body and a cost function that return the abstract work of the sequential body. It begins by computing the abstract cost `N` for the sequential body and consults the estimator. If the work is predicted to be small, it runs the sequential body, else the parallel body. It relies on a function called `measured_run` to measure the sum of the duration of the pieces of sequential code executed. This function is not entirely trivial to implement, but can be implemented efficiently with little code (see the technical appendix).

### 2.6 Robustness with respect to outliers

One crucial aspect of the design of our algorithm is its robustness. As explained in Section 2.3, a given run might be much slower than others, e.g., due to cache misses or OS interference. If such slow runs were taken into account for making future predictions, it could result in the sequentialization of tiny computations running in much less than $\kappa$ time (under favorable conditions). Our algorithm deals with outlier measurements by keeping track of `Nmax`, which denotes the maximal `N` (i.e., result of the complexity function) for which a sequential run took less than $\kappa$ time. Upon encountering a run that was much slower than previous runs, the corresponding time measure exceeds $\kappa$ time and thus gets discarded by our algorithm (Figure 3, Line 7).

Observe that our algorithm only ever increases `Nmax`. As we observed through experiments, there is an inherent asymmetry in the distribution of execution times, featuring a long tail. On the one hand, an execution time can be abnormally slow by a very large factor, due to, e.g., repeated OS interference or saturated memory bus. Our algorithm is carefully designed to be robust to such outliers, which we do observe in practice. On the other hand, an execution time can be abnormally fast only by some constant factor. It is possible that a warmed-up cache, or a data set unusually laid out in memory, could speed up the execution, but not arbitrarily. Our theoretical analysis accounts for that possibility through the error factor $E$ (Section 2.3). As we prove, in the worst case, a run that would be abnormally fast by a factor $E$ could only increase the total amount of overheads by a factor $E$.

We highlight two interesting features of our C++ implementation. First, to avoid always having to pass a sequential body explicitly, the spguard provides a default setting. If the spguard is called with such a default setting, then the parallel body is called whenever the spguard switches to sequential execution. However, in this scenario, when it sequentializes, the spguard executes the parallel body in an environment in which all `fork2join` calls are performed sequentially, using dynamic checks to selectively disable parallelism. Second, our implementation supports defining higher-level abstractions on top of `fork2join`, such as `parallel-for`, `map`, `reduce`, `map_reduce`, `scan`, `filter`, etc. For these abstractions, the programmer may either indicate a custom cost function or simply rely on the default one, which assumes a constant-time processing for each item or iteration. Because it applies often, the default cost function greatly reduces the number of hand-written cost functions.

### 2.7 Intuition for the proof

Without loss of expressiveness, we assume for the proof that the parallel body of a spguard consists of a sequence of (1) a sequential preprocessing, (2) a fork-join operation with left and right branches, and (3) a sequential postprocessing. (In practice, this amounts to surrounding every block of source code featuring a fork-join with a spguard.)

In that setting, we assume that the execution of a spguard (including the evaluation of its cost function) is assumed to incur a cost $\phi$, and that each execution of a spguard that evaluates in parallel (i.e., that triggers a fork-join) is assumed to incur an extra cost $\tau$. In other words, $\phi$ quantifies the overheads of granularity control, and $\tau$ quantifies the overheads of parallelism.

We define the *total work*, written $\mathbb{W}$, and the *total span*, written $\mathbb{S}$, to measure the actual work and span of an execution, i.e., accounting for the costs $\tau$ and $\phi$. To derive our final theorem, we bound the total work and span ($\mathbb{W}$ and $\mathbb{S}$) in terms of the *algorithmic* work and span ($w$ and $s$), which

excludes overheads. We then invoke Brent's theorem [Zadeh 2017]: $T_p \leq \frac{\mathbb{W}}{P} + \mathbb{S}$, to bound the parallel execution time.

For the span, we establish that the total span can only be a multiplicative factor larger than the algorithmic span, in the sense that $\mathbb{S}$ is $O(\kappa) \cdot s$. More precisely, we establish the following bound by induction on the program execution:

$$\mathbb{S} \leq (1 + \phi + \max(\tau, E\beta\kappa)) \cdot s.$$

Recall that $\kappa$ denotes the parallelism unit (Section 2.1), and that $E$ and $\beta$ are constant parameters introduced for the analysis (Section 2.3). The proof relies on a key lemma, which asserts that if a spguard executes its sequential body, then this body involves at most $E\beta\kappa$ sequential work. Intuitively, this lemma captures the fact that work predictions performed by our algorithm cannot be under-estimated by more than a factor $E\beta$. (We cannot expect tighter bounds in our model, for the reasons discussed in Section 2.3.)

The intuition for the bound $(1 + \phi + \max(\tau, E\beta\kappa)) \cdot s$ is as follows. At a spguard, the algorithmic span increases by one unit. If the corresponding spguard executes its parallel body, then its total span increases by $1 + \phi + \tau$, to account for overheads. If, however, the spguard executes its sequential body, then, because the work involved is at most $E\beta\kappa$, the total span increases at most by $1 + \phi + E\beta\kappa$. The bound is obtained by considering the maximum of these two cases.

For the work, we prove that the total work is only a fraction larger than the algorithmic work, plus a small constant:

$$\mathbb{W} \leq \left(1 + \frac{O(1)}{\kappa}\right) \cdot w + O(P \log^2 \kappa).$$

To establish this bound, we distinguish the spguard calls that involve $\Theta(\kappa)$ sequential work from those that involve less.

After the algorithm has converged, spguards are able to make predictions sufficiently accurate to ensure that the sequentialized spguards involve at least $\Theta(\kappa)$ sequential work.[1] When all sequentialized spguards involve $\Theta(\kappa)$ work, the constant-time overheads associated with each spguard only amount for a fraction $\frac{O(1)}{\kappa}$ of the algorithmic work involved. For example, consider a simple, parallel program performing binary division of its input. The execution of this program can be viewed as a binary tree. Let $n$ denote the number of leaves in the tree. Every non-leaf node induces an overhead $\phi + \tau$, to pay for the evaluation of the spguard and the fork. Every node (including leaves) induces an overhead $\phi$, to pay for the spguard. Because there are $n-1$ nodes in a binary tree with $n$ leaves, the overheads amount to $n \cdot \phi + (n-1) \cdot (\phi + \tau)$, which is $O(n \cdot (\tau + 2\phi))$. These overheads can be amortized over the sequential work performed at the $n$ leaves, each of which involves $\Theta(\kappa)$ work. More precisely, the ratio between the overheads and the sequential work is $O\left(\frac{n \cdot (\tau + 2\phi)}{n \cdot \kappa}\right)$, which simplifies to $O\left(\frac{\tau + 2\phi}{\kappa}\right)$. Because $\tau$ and $\phi$ are constants, the relative overheads are $\frac{O(1)}{\kappa}$, as claimed.

It remains to bound the overheads involved in the evaluation of the spguards that contain less than $\Theta(\kappa)$ sequential work. Each such evaluation induces an overhead $O(1)$, because it costs $\phi$, plus $\tau$ in case of a parallel execution. We show that the number of these evaluations is at most $O(P \log^2 \kappa)$. To that end, we study *parallel small calls*: evaluations of spguards that involve less than $\Theta(\kappa)$ sequential work and whose parallel body gets executed. Such parallel small calls can be nested, but, due to the balance condition (third assumption from Section 2.4), there can be no more than $O(\log \kappa)$ levels of nesting. As we argue in the technical appendix, the evaluation of an innermost parallel small call (i.e., one that features no nested parallel small call) necessarily multiplies the value of `Nmax` associated with the estimator by a factor $\alpha$. Thus, for each syntactic spguard and for each processor, the number of innermost parallel small calls on that estimator is $O(\log \kappa)$. Given that a program contains a constant number of syntactic spguards, we deduce that there are at most $O(P \log \kappa)$ innermost parallel small calls during an execution. Then, because each parallel small call contains at least one innermost parallel small call, and because an innermost parallel small call is nested in at most $O(\log \kappa)$ parallel small calls, we deduce that the total number of parallel small calls is at most $O(P \log^2 \kappa)$. Finally, we argue that the number of *sequential small calls* (i.e., evaluations of spguards that involve less than $\Theta(\kappa)$ sequential work and whose sequential body gets executed) is at most a constant factor larger than the number of *parallel small calls*. By summing up the bounds for parallel small calls and sequential ones, we obtain a $O(P \log^2 \kappa)$ bound for the overheads associated with all calls to spguard involving less than $\Theta(\kappa)$ sequential work. This concludes the proof summary.

## 3 Experimental evaluation

This section presents a study of eight benchmarks from The Problem Based Benchmark Suite (PBBS) [Blelloch et al. 2012]. These benchmarks consist of state-of-the-art solutions to problems on sequences, strings, graphs, and in geometry and graphics. The implementation of the benchmarks consists of a collection C++ codes that use the linguistic extensions of Cilk Plus to realize parallel execution. Parallelism is expressed across the source codes in the idiomatic style of Cilk, namely, nested parallelism, involving, in specific, parallel loops occurring inside parallel loops mixed with calls to recursive, parallel functions. As is common in nested parallelism, the applications generate irregular and, often, highly data-dependent parallelism, thereby making granularity control a particular challenge. Two of the benchmarks

[1] Technically, any spguard that involves less than $\kappa/(\gamma DE)$ work will necessarily execute its sequential body, where $E$ denotes the error factor, where $D$ bounds the ratio of the work of the sequential body and the sequential work of the parallel body of a spguard (first assumption from Section 2.4), and where $\gamma$ bounds the ratio between the sequential work of a spguard and that of the immediately nested spguard (second assumption from Section 2.4). Formal definitions for $D$ and $\gamma$ are given in the technical appendix.

we consider, namely ray-cast and delaunay, represent the largest and most complex codes in PBBS.

The original, authors' codes leverage a variety of manual techniques to control granularity. These techniques involve program instrumentation along with careful tuning to achieve efficiency. For our programs, we ported the original PBBS codes by inserting spguards where necessary, and, for the spguards, we wrote a total of 24 explicit cost functions. It was otherwise possible to use default cost functions, as described in Section 2.6. We summarize the changes:

1. A few divide-and-conquer algorithms involved a manually fixed grain to control granularity. We replaced each of them with a spguard, thereby eliminating the hardware-dependent magic numbers.
2. A number of loops were parallelized by splitting in fixed-size blocks of 2048 items each—a pattern widely used throughout the PBBS sequence library. We replaced all of them with our automatically controlled parallel-for loops.
3. Other loops exploited Cilk parallel for-loops, which essentially split the loop range in $8P$ blocks, where $P$ is the number of cores. We also replaced all of them with our automatically controlled loops.
4. A number of inner loops were forced to sequentialize, even though they could have been parallel. As we learned from private communication with the authors, the purpose was to tame the overheads. We restored parallelism using our automatically-controlled loops.
5. Two loops were forced to always make one spawn per iteration, using a Cilk for-loop annotated with `#pragma grainsize = 1`. Again, we replaced all such loops with our automatically-controlled loops, which properly support nested parallelism.

Overall, automatic granularity control enables replacing careful selection of techniques, such as those listed above, with a single, uniform technique for controlling granularity, able to handle nontrivial nesting of parallelism constructs. Furthermore, the technique automates away the labor-intensive tuning of grain sizes, and it automatically adapts to the hardware—performance is portable.

Yet, these benefits are not completely free. The programmer must take care to ensure that the algorithm in question meets the requirements of our approach, as described in Section 2. Owing to such requirements, we left out eight of the remaining PBBS benchmarks. Fortunately, for each such benchmark, the reason for the incompatibility is straightforward, and in the technical appendix, we explain the reasons. In spite of such cases, our approach nevertheless remains sound, even in applications where there are some functions that use our approach, and some that use, for example, manual granularity control, or perhaps some other method. The only caveat is that the guarantees associated with our approach do not apply for regions of the program in which there are calls to manually-controlled functions from inside spguards.

In our approach, there are potential overheads related to the need to infer granularity thresholds online, through a convergence phase. Nevertheless, as established by our analysis (and, as we confirm through our experimental results), the cost of the convergence phase generally accounts for at most a few percent of the parallel run time. Overheads associated with manual granularity control pose their own challenges, but are not backed by guarantees. We observed that, by removing the pragmas (fifth item in the list above), performance would sometimes degrade by up to 5%, and we note that the degradation observed by the PBBS authors may have been more pronounced. The overheads associated with the fourth item in the list above are analyzed in detail by our study of the parallel BFS benchmark, which shows that our automatic method achieves better performance across a wider range of inputs than any manual setting.

***Experimental setup.*** Our primary test harness is an Intel machine with 40 cores. We compiled the code using GCC (version 6.3) using the extensions for Cilk Plus (options `-O2 -march=native -fcilkplus`). Our 40-core machine has four 10-core Intel E7-4870 chips, at 2.4GHz, with 32Kb of L1 and 256Kb L2 cache per core, 30Mb of L3 cache per chip, and 32Gb RAM, and runs Ubuntu Linux kernel v3.13.0-66-generic. For each data point, we report the average running time over 30 runs. The variation in the running times is negligible: overall, we observed only a few cases where the standard deviation is 5%, but it was usually below 3%.

To pick settings for parameters $\kappa$ and $\alpha$, we developed a method that involves a one-time-per-machine, automatic tuning step. The starting point is a benchmark program that takes as input an array of 32-bit integers and computes the sum using a parallel reduction. The reduction code uses a spguard to control granularity. To pick $\kappa$, we perform a series of runs of the reduction benchmark using a single core, starting with a small setting of $\kappa = 1\mu$s and progressively trying larger settings. For these initial runs, we pick $\alpha = 1.3$, a small setting so that the estimator has plenty of slack to converge to reach the target $\kappa$. We stop this process upon reaching the first setting of $\kappa$ for which the program runs slightly above 5% slower than the *sequential elision* of the our reduction benchmark. The sequential elision is (in general) the version of the parallel program in which all fork-join and spguard constructs are erased, leaving only the bare sequential code.

Having picked $\kappa$, we then run the same benchmark, but this time using all available cores and trying a number of settings in $1.3 \leq \alpha \leq 5.0$. We then pick $\alpha$ to be the setting that gives the best running time. On our test machine, this process gave us $\kappa = 10.2\mu$s and $\alpha = 3.0$.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| Application/input | Sequential elision | | 1-core execution | | 40-core execution | | | |
| | PBBS | Oracle | PBBS | Oracle | PBBS | Oracle | Oracle / PBBS | |
| | (s) | | (relative to elision) | | (s) | | Idle time | Nb threads |
| **samplesort** | | | | | | | | |
| random | 21.765 | +7.1% | +15.6% | -1.9% | 0.834 | -7.0% | -7.1% | -18.1% |
| exponential | 15.682 | +5.3% | +13.1% | -1.4% | 0.626 | -4.4% | -4.4% | -15.7% |
| almost sorted | 6.799 | +19.3% | +26.4% | -4.3% | 0.423 | -7.8% | -7.8% | -3.0% |
| **radixsort** | | | | | | | | |
| random | 3.060 | +3.6% | +0.5% | -5.3% | 0.235 | +8.8% | +9.3% | -11.4% |
| random pair | 4.920 | -0.3% | +0.3% | -1.0% | 0.469 | +4.3% | +3.2% | +128.0% |
| exponential | 3.135 | +3.3% | +1.0% | -2.6% | 0.245 | -1.9% | -3.1% | +23.7% |
| **suffixarray** | | | | | | | | |
| dna | 19.276 | +1.4% | +2.2% | +3.5% | 1.494 | +3.8% | +1.4% | +28.2% |
| etext | 70.322 | +1.1% | +2.0% | +2.4% | 4.527 | -0.0% | -1.7% | +68.4% |
| wikisamp | 62.112 | -1.1% | +2.0% | +3.7% | 4.139 | +3.2% | +1.4% | +64.1% |
| **convexhull** | | | | | | | | |
| kuzmin | 7.040 | +24.5% | -0.8% | -15.1% | 0.517 | -0.1% | +6.0% | -81.6% |
| on circle | 130.835 | -6.1% | +104.8% | +6.7% | 9.871 | -31.5% | -31.5% | -23.4% |
| **nearestneighbors** | | | | | | | | |
| kuzmin | 19.712 | +1.8% | +21.7% | +13.2% | 1.308 | -1.8% | -3.5% | +20.0% |
| plummer | 27.173 | -3.4% | +6.2% | +3.4% | 2.444 | -4.6% | -10.5% | -0.3% |
| **delaunay** | | | | | | | | |
| in square | 63.967 | -0.1% | -0.6% | -0.4% | 3.150 | -0.6% | -0.8% | +3.2% |
| kuzmin | 69.428 | +0.7% | +0.5% | +1.7% | 3.802 | -4.0% | +0.9% | -21.4% |
| **raycast** | | | | | | | | |
| happy | 10.698 | +6.3% | +3.2% | +0.3% | 0.473 | +5.5% | +5.5% | -32.7% |
| xyzrgb | 338.920 | -1.9% | -2.5% | +1.2% | 9.677 | +1.5% | +2.0% | -67.3% |

**Table 1.** Benchmark results. Column 3 gives an estimate of the difference in performance between our implementation and the original PBBS version, on a single core, neglecting parallelism-related overheads. Column 4 gives a lower bound on the overheads of the original PBBS code, with figures relative to Column 2. Column 5 gives an estimate of the thread-creation and spguard overheads in our approach, with figures relative to Column 3. In the 40-core section, Column 6 gives PBBS execution time, and Column 7 gives the oracle-guided figure relative to Column 6. Negative figures indicate that our approach is performing better. Columns 8 and 9 give the ratios between our approach and that of PBBS for two measures: total idle time and number of threads created.

### 3.1 Main PBBS results

For input data sets, we reused much from the original PBBS study [Blelloch et al. 2012], but introduced newly acquired data in a few cases. The inputs to *comparison-* and *radix-sort* consist of sequences of length $10^8$. For *convex-hull*, the inputs consist of $10^8$ 2-d points. For *nearest neighbors*, the *kuzmin* input consists of $10^8$ 2-d points and *plummer*, $10^8$ 3-d points. For *delaunay*, the inputs consist of $10^7$ 2-d points. For *ray-cast*, we used a two non-synthetic inputs: *happy* consists of the Happy Buddha mesh from the Stanford 3D Scanning Repository, and it consists of 1 million triangles, and *xyz-rgb-manusript* comes from the same repository and consists of 4 million triangles. Finer details appear in the technical appendix.

The main performance results appear in Table 1. Columns 2 and 3 represent runs of the sequential elision of the original PBBS code and of our implementation, respectively. The results in column 3 in particular show the relative performance of our implementation versus the original PBBS one. A -5% value indicates that our version performs 5% faster. Although our code is almost always identical to the original PBBS code, there are some minor structural differences in the function bodies that affect performance slightly. Overall, however, the sequential elisions in both cases perform similarly.

We next evaluate the overheads of thread creation in the original PBBS codes and in our oracle-guided versions. To evaluate for original PBBS, we compare the execution time of the sequential elision of the PBBS code to that of the PBBS code compiled as a Cilk parallel binary. The estimation of overheads in Cilk may be incomplete because the Cilk system in some places detects at runtime that there is only one active worker thread in the system. Nevertheless, the comparison

should give us a lower bound on the parallelism overheads affecting Cilk programs. Column 4 in Table 1 shows that overheads are sometimes low, but sometimes over 15%, and in one case 104%.

To examine overheads of our approach, we compare in a similar fashion the execution time of our sequential elision to that of our parallel-ready binary, running on a single core. Our parallel-ready binary includes the overhead costs of both the Cilk parallel constructs and the estimation-related overheads of the granularity controller. Column 5 in Table 1 shows that the parallel-ready binary is in one case 13% slower, but is otherwise close to the overhead of 5% or smaller that we targeted. There are a number of cases where the parallel-ready binary is slightly faster than the sequential elision. The reason is that the sequential elision switches from recursive, parallel-ready to loop-based sequential code when the complexity function returns a value smaller than 10k, which is sometimes slightly less efficient than the threshold given by the parallel-ready binary.

We finally compare the execution time of original PBBS to that of our binary on 40 cores, using automatic granularity control. Columns 6 and 7 in Table 1 show the results. The results show that in all but two cases, our binary is faster or on par with the PBBS binary. When it is slower, our binary is slower by 8.8%, whereas when it is faster, it is in one case 31.5% and another 7.8% faster. These latter two cases correspond to worst-case inputs in samplesort and convex-hull, where the original PBBS binary is slower because of comparatively poor utilization of the cores.

To gain further insight, we included Columns 7 and 8 in Table 1. Column 8 shows the ratio between idle time (counting periods during which workers are out of work) in our approach and idle time in PBBS. The figures show that the idle time is of the same order of magnitude.[2] Column 9 shows the ratio between the number of threads created in our approach and the number of threads created in PBBS. Taken together, these last two columns indicate that our approach is able to achieve similar utilization despite having to modulate the creation of threads online.

### 3.2 Parallel BFS

In addition to the other PBBS programs, we considered the non-deterministic BFS benchmark, named ndBFS in the PBBS paper. We chose ndBFS because it is the fastest among the alternative BFS implementations in PBBS and, to the best of our knowledge, the fastest publicly available Cilk implementation of BFS. To test of the robustness of granularity control, we picked the input graphs to feature a wide range of graph structures. These graphs were used in a different performance study to test the granularity control of a DFS-like graph-traversal algorithm [Acar et al. 2015]. We picked a representative subset of the graphs, including small-world graphs, such as livejournal, twitter, and wikipedia, and high-diameter graphs, such as europe and cube-grid.

There are two versions of BFS: the *flat* and the *nested* version. In the flat version, the algorithm traverses over the list of neighbors of a vertex in the BFS frontier sequentially, whereas, in the nested version, the algorithm uses a parallel loop, thereby allowing to process in parallel the out-edges of vertices with high out-degree.

PBBS currently uses the flat version. Via personal communication, we learned that the authors sequentialized the inner loop because (1) the graphs they considered did not feature such high-degree vertices, and (2) they observed that performance improved for some of their test graphs when this loop was serialized. Columns 2 and 6 from Table 2 give the execution time for the flat PBBS BFS, and for its nested counterpart, using a parallel `cilk_for` loop over the edges. The figures from these two columns confirm that overheads of parallelization using Cilk-for are significant, sometimes above 50%.

In contrast, our algorithm supports nested parallelism and, even so, delivers significantly better results for nested PBBS, as reported in Column 7 from Table 2 ("nested - ours"). The last column from this figure shows our main result: it compares the performance of the original authors' flat BFS versus our nested BFS with automatic granularity control. Our version either performs about as well or up to 84% faster.

### 3.3 Summary

Overall, the results show that our technique enables a simple, uniform method for granularity control, whereas the original PBBS code involves several types of manual intervention. The main PBBS results show our method delivering similar or better performance overall, in spite of having to adaptively adjust grain settings on the fly. The BFS results show our technique delivering consistently better performance in the face of fine-grain, nested, and irregular parallelism. In the technical appendix, we present a portability study, showing that the technique yields similar results across various test machines with different core counts and architectures.

## 4 Related work

Controlling the overheads of parallelism has been an important open problem since the early 1980's, when it was first recognized that practical overheads can undo the benefits of parallelism [Halstead 1984; Mohr et al. 1990]. Since then, researchers have explored two separate approaches.

***Granularity Control.*** Perhaps the oldest technique for granularity control is to use manual, programmer-inserted "cut-off" conditions that switch from a parallel to a sequential

[2] Because utilization in these benchmarks is generally between 80% and 99%, the total idle time represents less than 20% of the total execution time, thus a 20% change in idle time would affect the execution time by less than 4%.

| | Flat | | | | Nested | | | | Ours nested vs. PBBS flat |
|---|---|---|---|---|---|---|---|---|---|
| Graph | PBBS | Ours | Oracle / PBBS | | PBBS | Ours | Oracle / PBBS | | |
| | (sec.) | | Idle time | Nb threads | (sec.) | | Idle time | Nb threads | |
| livejournal | 0.12 | +8.2% | −5.5% | −16.8% | 0.19 | −29.0% | −2.2% | −58.8% | +11.3% |
| twitter | 2.02 | −7.3% | +14.0% | −16.0% | 2.71 | −32.4% | −0.1% | −57.1% | −9.5% |
| wikipedia | 0.11 | +5.1% | −3.9% | −28.1% | 0.14 | −19.0% | −6.4% | −55.3% | +7.4% |
| europe | 4.28 | −38.3% | −50.4% | −24.8% | 4.28 | −37.9% | −51.3% | −27.6% | −37.9% |
| rand-arity-100 | 0.12 | +6.5% | −6.8% | +9.8% | 0.44 | −71.3% | −20.4% | −40.8% | +8.0% |
| rmat27 | 0.21 | +8.4% | −3.1% | −43.4% | 0.41 | −46.4% | −5.4% | −56.3% | +4.8% |
| rmat24 | 0.36 | +5.0% | −3.8% | +4.8% | 0.37 | +3.0% | −4.6% | +8.8% | +5.0% |
| cube-grid | 0.79 | +0.0% | −15.9% | +14.4% | 0.79 | −0.7% | −17.4% | +10.1% | −0.8% |
| square-grid | 4.51 | −38.9% | −24.5% | −31.5% | 4.47 | −38.1% | −25.5% | −32.7% | −38.7% |
| par-chains-100 | 67.7 | −71.0% | −71.4% | −90.9% | 69.2 | −72.6% | −67.7% | −79.6% | −72.0% |
| trunk-first | 13.1 | −47.4% | −2.1% | −97.4% | 13.4 | −48.5% | −2.1% | −97.6% | −47.2% |
| phases-10-d-2 | 1.40 | +8.1% | −8.9% | −0.4% | 0.51 | +3.5% | −2.6% | +10.5% | −62.0% |
| phases-50-d-5 | 0.67 | +5.8% | −5.0% | +6.9% | 0.66 | +4.7% | −3.6% | +16.6% | +4.3% |
| trees-524k | 12.9 | +21.0% | −14.7% | −8.2% | 1.80 | +15.7% | −2.5% | +23.5% | −83.8% |

**Table 2.** Parallel BFS experiment on 40 cores. Depending on the input graph, either Flat PBBS or Nested PBBS is faster. Our nested-parallel algorithm compares favorably to both of them, as reflected by the last column, and the column "Nested/Ours".

mode of execution. Researchers have addressed the limitations of such manual granularity control by using various forms of automation. Duran et al. propose a method for selecting among three parallelization options: outer-loop only, inner-loop only, or mixed mode, where inner and outer loops may be parallelized and granularity controlled by a heuristic [Duran et al. 2008]. The cut-off technique of Duran et al. makes its decisions online and, like our technique, requires neither tuning nor recompiling. An earlier approach by Huelsbergen, Larus, and Aiken uses list size to determine the grain size [Huelsbergen et al. 1994], however, their technique assumes linear work complexity for every parallel operation. Another approach uses the height and depth of the recursion tree [Pehoushek and Weening 1990; Weening 1989] to predict the execution time and the grain size, but lacks crucial information because depth and height are not a direct measure of execution time. As Iwasaki et al. point out, because techniques, such as those mentioned here, base decisions solely on certain dynamically collected data, such as recursion-tree depth, height, and dynamic load conditions, the granularity controller typically risks decreasing parallelism adversely [Iwasaki and Taura 2016].

Lopez et al. use a similar approach to ours, but in the context of logic programming [Lopez et al. 1996]. Their cost estimators, however, rely on the asymptotic cost annotations alone, but not on prediction of wall-clock running times. Using the asymptotic cost alone is an overly simplistic method, because, on modern processors, execution time depends heavily on effects of caching, pipelining, etc. Our technique improves on theirs by defining precise assumptions (Section 2) for correct use, backing with theoretical bounds and with a range of challenging benchmarks.

Iwasaki et al. propose a technique for synthesizing static cut-offs for divide-and-conquer functions in Cilk-style programs [Iwasaki and Taura 2016]. The determination of whether or not to cut to serial is made by using cost estimation from a compiler analysis. As such, this technique relies on having accurate static analysis and sophisticated compiler support. Our technique offers an alternative that can, in contrast to theirs, be implemented entirely as a library, using off-the-shelf tools, such as GCC, and can adapt its decisions to changing conditions rather than compile-time estimates. The main cost of which is sometimes having to write cost functions.

***Lazy task creation.*** Lazy task creation, lazy scheduling, and heartbeat scheduling are techniques that aim to reduce the number of tasks created by observing the load in the system and creating tasks only when necessary [Acar et al. 2018; Mohr et al. 1991; Tzannes et al. 2014]. The aim is to reduce the total overhead of thread creation of a parallel program, whereas granularity control aims to reduce the same kind of overhead by instead selectively cutting to serial code.

Lazy task creation has proved to be an indispensable technique for modern parallel programming systems, many of which could not perform well without it. Although it can reduce overheads of parallelism, lazy task creation has some important limitations that prevent it from being the silver bullet. First, there is no way in lazy task creation to account for

the fact that sequential algorithms are usually both asymptotically and practically more work-efficient than their parallel counterparts. At best, lazy task creation can reduce parallelism-related overheads by executing an already parallel algorithm sequentially. Even without parallelism-related overheads, using the stripped-down parallel algorithm (i.e., *sequential elision* [Frigo et al. 1998]) could lead to suboptimal performance, because the problem could have been solved much more efficiently by a faster sequential algorithm. It is thus reasonably common, even when using lazy task creation, to observe a 3-10 fold overhead compared to a piece of code that manually controls granularity of tasks.

Another issue relates to infrastructure requirements. Because it depends on polling on a regular basis for work requests, there is a need for sophisticated support from the compiler [Feeley 1993a]. A key component of such compiler support is the injection of polling checks [Feeley 1993b], which is generally necessary, unless there is support for a suitable inter-core interrupt mechanism a la ADM [Sanchez et al. 2010b], or a suitable interrupt-based alarm mechanism.

We believe, however, that lazy task creation and granularity control, such as our algorithm, are largely complementary: they could easily coexist and provide benefits to each other. Lazy task creation could likely improve performance in cases where a cost function is difficult to define, for example, and our oracle-guided algorithm could likely improve performance by enabling the program to cut in a safe manner to faster serial code. Additional analysis and experimentation will help to better understand the tradeoffs.

## 5 Conclusion

The problem of managing parallelism-related overheads effectively is an important problem facing parallel programming. The current state-of-the-art in granularity control places the burden of the tuning on the programmer. The tuning process is labor intensive and results in highly engineered, possibly performance-brittle code. In this paper, we show that it is possible to control granularity in a more principled fashion. The key to our approach is an online algorithm for efficiently and accurately estimating the work of parallel computations. We show that our algorithm can be integrated into a state-of-the-art, implicitly parallel language and can deliver executables that compete with and sometimes even beat expertly hand-tuned programs.

**Acknowledgments.** This research is partially supported by the European Research Council (ERC-2012-StG-308246).

## References

Umut A. Acar and Guy E. Blelloch. 2017. *Algorithm Design: Parallel and Sequential.* http:www.parallel-algorithms-book.com.

Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2002. The Data Locality of Work Stealing. *Theory of Computing Systems* 35, 3 (2002), 321–347.

Umut A. Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. 2018. Heartbeat Scheduling: Provable Efficiency for Nested Parallelism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 769–782. https://doi.org/10.1145/3192366.3192391

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling Parallel Programs by Work Stealing with Private Deques. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*.

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2015. A work-efficient algorithm for parallel unordered depth-first search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*. 67:1–67:12.

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2016. Oracle-guided scheduling for controlling granularity in implicitly parallel languages. *Journal of Functional Programming (JFP)* 26 (2016), e23.

Arvind and K. P. Gostelow. 1978. *The Id Report: An Asychronous Language and Computing Machine.* Technical Report TR-114. Department of Information and Computer Science, University of California, Irvine.

Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2012. Lazy Tree Splitting. *J. Funct. Program.* 22, 4-5 (Aug. 2012), 382–438.

Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *PPoPP '12*. 181–192.

Guy E. Blelloch and John Greiner. 1996. A provable time and space efficient implementation of NESL. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming*. ACM, 213–225.

Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. 1994. Implementation of a Portable Nested Data-Parallel Language. *J. Parallel Distrib. Comput.* 21, 1 (1994), 4–14.

Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46 (Sept. 1999), 720–748. Issue 5.

Richard P. Brent. 1974. The parallel evaluation of general arithmetic expressions. *J. ACM* 21, 2 (1974), 201–206.

Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data parallel Haskell: a status report. In *Proceedings of the POPL 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP 2007, Nice, France, January 16, 2007*. 10–18.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*. ACM, 519–538.

A. Duran, J. Corbalan, and E. Ayguade. 2008. An adaptive cut-off for task parallelism. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

Marc Feeley. 1992. A Message Passing Implementation of Lazy Task Creation. In *Parallel Symbolic Computing*. 94–107.

Marc Feeley. 1993a. *An efficient and general implementation of futures on large scale shared-memory multiprocessors.* Ph.D. Dissertation. Brandeis University, Waltham, MA, USA. UMI Order No. GAX93-22348.

Marc Feeley. 1993b. Polling efficiently on stock hardware. In *Proceedings of the conference on Functional programming languages and computer architecture (FPCA '93)*. 179–187.

Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2011. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming* 20, 5-6 (2011), 1–40.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*. 212–223.

Adrien Guatto, Sam Westrick, Ram Raghunathan, and Umut A. Acarand Matthew Fluet. 2018. Hierarchical Memory Management for

Mutable State. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*. ACM Press.

Robert H. Halstead. 1985. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems* 7 (1985), 501–538.

Robert H. Halstead, Jr. 1984. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming (LFP '84)*. ACM, 9–17.

Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. 2009. Backtracking-based load balancing. In *PPoPP '09*. ACM, 55–64. https://doi.org/10.1145/1504176.1504187

Lorenz Huelsbergen, James R. Larus, and Alexander Aiken. 1994. Using the run-time sizes of data structures to guide parallel-thread creation. In *Proceedings of the 1994 ACM conference on LISP and functional programming (LFP '94)*. 79–90.

Shams Mahmood Imam and Vivek Sarkar. 2014. Habanero-Java library: a Java 8 framework for multicore programming. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14*. 75–86.

Intel. 2011. Intel Threading Building Blocks. (2011). https://www.threadingbuildingblocks.org/.

Shintaro Iwasaki and Kenjiro Taura. 2016. A static cut-off for task parallel programs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM, 139–150.

Suresh Jagannathan, Armand Navabi, KC Sivaramakrishnan, and Lukasz Ziarek. 2010. The Design Rationale for Multi-MLton. In *ML '10: Proceedings of the ACM SIGPLAN Workshop on ML*. ACM.

Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. 2010. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming (ICFP '10)*. 261–272.

Doug Lea. 2000. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande (JAVA '00)*. 36–43.

Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. 2009. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. 227–242.

P. Lopez, M. Hermenegildo, and S. Debray. 1996. A methodology for granularity-based control of parallelism in logic programs. *Journal of Symbolic Computation* 21 (June 1996), 715–734. Issue 4-6. https://doi.org/10.1006/jsco.1996.0038

E. Mohr, D. A. Kranz, and R. H. Halstead. 1991. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems* 2, 3 (1991), 264–280.

Eric Mohr, David A. Kranz, and Robert H. Halstead Jr. 1990. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Conference record of the 1990 ACM Conference on Lisp and Functional Programming*. ACM Press, New York, New York, USA, 185–197.

OpenMP Architecture Review Board. 2008. OpenMP Application Program Interface. (2008). https://www.openmp.org/.

Joseph Pehoushek and Joseph Weening. 1990. Low-cost process creation and dynamic partitioning in Qlisp. In *Parallel Lisp: Languages and Systems*, Takayasu Ito and Robert Halstead (Eds.). Lecture Notes in Computer Science, Vol. 441. Springer Berlin / Heidelberg, 182–199.

Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. 2016. Hierarchical Memory Management for Parallel Programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 392–406.

Mike Rainey. 2010. *Effective Scheduling Techniques for High-Level Parallel Programming Languages*. Ph.D. Dissertation. University of Chicago.

Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. 2010a. Flexible architectural support for fine-grain scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS '10)*. ACM, New York, NY, USA, 311–322.

Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. 2010b. Flexible architectural support for fine-grain scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS '10)*. ACM, New York, NY, USA, 311–322. https://doi.org/10.1145/1736020.1736055

Bjarne Stroustrup. 2013. The C++ Programming Language. (2013).

Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. 2014. Lazy Scheduling: A Runtime Adaptive Scheduler for Declarative Parallelism. *TOPLAS* 36, 3, Article 10 (Sept. 2014), 51 pages. https://doi.org/10.1145/2629643

Joseph S. Weening. 1989. *Parallel Execution of Lisp Programs*. Ph.D. Dissertation. Stanford University. Computer Science Technical Report STAN-CS-89-1265.

Reza Zadeh. 2017. Overview, Models of Computation, Brent's Theorem. (2017). https://stanford.edu/~rezab/dao/notes/lecture01/cme323_lec1.pdf

# A Artifact Appendix

This artifact contains scripts for building the executables, downloading the input data, executing the benchmark programs, and output tables of results similar to Tables 1 and 2.

## A.1 Artifact check-list (meta-information)

- Algorithm: samplesort, radixsort, suffixarray, convexhull, nearestneighbors, delaunay, raycast, bfs
- Program: nix build scripts, C++ code
- Compilation: nix build scripts
- Data set: Problem Based Benchmark Suite
- Run-time environment: Linux
- Hardware: An x86-64 machine with multiple cores
- Output: Results tables comparing running times, core idle times, and number of green-thread creations
- How much disk space required (approximately)?: 300GB
- How much time is needed to prepare workflow (approximately)?: 5-10 hours
- How much time is needed to complete experiments (approximately)?: 4 hours
- Publicly available?: Yes
- Code/data licenses (if publicly available)?: Yes
- Workflow frameworks used?: nix package manager

## A.2 Description

***How delivered*** Our source code consists of a several of our open-source libraries, all of which are under the MIT license. The libraries are hosted along with documentation on Github. The two main repositories are available at https://github.com/deepsea-inria/sptl and https://github.com/deepsea-inria/pbbs-sptl.

***Hardware dependencies*** The experiments require a machine with multiple x86-64 cores and well-provisioned RAM. Of our test machines, the one with the minimum amount of RAM has 74GB, but this amount is likely more than is needed.

***Software dependencies*** Our build scripts depend on the nix package manager. https://nixos.org/nix/download.html

***Data sets*** The data sets take approximately 300GB on disk.

Although they are taken from the Problem Based Benchmark Suite, the data sets used in our experiments are represented in our custom serialized format. As such, the data sets can be accessed only via either the automatic downloading function of our benchmark scripts or the manual downloading process described in Section A.6.

## A.3 Installation

After you have the nix package manager, clone our repository and change to the script folder in the clone.

```
$ git clone https://github.com/deepsea-inria/pbbs-sptl.git
```

Then, change to the folder containing the nix build scripts.

```
$ cd pbbs-sptl/script
```

Because the current directory is where the input data is going to be stored, make sure that there is sufficient space on the file system.

Next, to build the benchmarks, run the following command (there are a number of alternative configuration parameters discussed in Section A.6).

```
$ nix-build
```

If the build succeeds, there will be a symlink named `result` in the current directory.

## A.4 Experiment workflow

The first step is to run the auto tuner, which picks (machine-specific) settings for $\kappa$ and $\alpha$, following the process described in Section 3.

```
$ ./result/bench/autotune
```

The result of this process is to write to `/var/tmp` a folder containing the settings data.

It is important to complete the auto tuning, because otherwise the default settings are used, and the default settings may lead to poor performance on the benchmarks.

To start the main benchmarks, run the following command.

```
$ ./result/bench/bench.pbench compare
```

The command will automatically download the input data sets, and will likely take several hours. We recommend first trying to complete this process on just one benchmark (defaultly, the script will try to run all to completion). See Section A.6 for instructions on how to run individually. If there is a problem downloading the input data automatically, see Section A.6 for instructions on how to obtain the input data manually.

After the previous command finishes successfully, there should appear in the `_results` folder a number of new text files of the form `results_`*benchmark*`.txt` and a PDF named `tables_compare.pdf`. The source for the table can be found in the same folder, in the file named `latex.tex`.

To start the BFS benchmarks, run the following command.

```
$ ./result/bench/bench.pbench bfs
```

This command should generate a text file named `results_bfs.txt` and a PDF named `tables_bfs.pdf`. Like before, the sources for the latex table can be found in the same folder, named `latex.tex`.

## A.5 Evaluation and expected result

The trends in `tables_compare.pdf` `tables_bfs.pdf` should bear resemblance to the trends in Tables 1 and 2, respectively. The raw results from benchmark runs should appear in text files in the `_results` folder. These results are human readable, but the more efficient way to interpret them is to look at the generated tables.

## A.6 Experiment customization

***Running benchmarks individually.*** The benchmarking script, namely `bench.pbench`, supports running one benchmark at a time. For example, the following command runs just convexhull.

```
$ ./result/bench/bench.pbench compare -benchmark
    convexhull
```

To run multiple, use a comma-separated list, such as the following.

```
$ ./result/bench/bench.pbench compare -benchmark
    convexhull,samplesort
```

***Manually downloading the input dataset.*** First, download the input data from http://mike-rainey.site/ppopp19. Then, use the following command to build the benchmarks.

```
$ nix-build --argstr pathToData /path/to/data/folder
```

***Building with custom modified sources.*** The interested experimenter may want to edit the source code and benchmark with their changes. Modifications to three critical repositories used by our benchmarks can be built by the following process. First, from the folder containing `pbbs-sptl`, clone the original Problem Based Benchmark Suite library code and our library code.

```
$ git clone https://github.com/deepsea-inria/pbbs-
    include.git
$ git clone https://github.com/deepsea-inria/sptl.git
```

Because the sptl repository may change over time, we recommend starting with the revision that is used by the default sources. The revision number can be found in the file at

`pbbs-sptl/script/default-sources.nix`

Next, build as follows.

```
$ cd pbbs-sptl/script
$ nix-build  --arg sources 'import ./local-sources.
    nix'
```

The library `sptl` contains our granularity-control algorithm along with critical data-parallel library functions used across the benchmarks. Documentation for `sptl` can be built by following the instructions in Section A.6.

***Building with library documentation.*** Documentation for `sptl` is available either in markdown format or in easier-to-read PDF and HTML formats. To render PDF and HTML documentation, pass to the build script the `buildDocs` flag as follows.

```
$ cd pbbs-sptl/script
$ nix-build --arg buildDocs true
```

After completion, the `sptl` documentation should appear along with the markdown source in the folder

`./result/sptl/share/doc`

***Running with custom $\kappa$, $\alpha$ settings*** The following command line overrides the settings generated by the auto tuner.

```
$ ./result/bench/bench.pbench compare -sptl_kappa
    12.0 -sptl_alpha 3.0
```

***Picking the number of processors manually*** The number of cores in the machine is detected by the autotuner, and that number is used by default for benchmarking. To override this number, say, to use 12 hardware threads for benchmarking, use the `-proc` flag, as follows.

```
$ ./result/bench/bench.pbench compare -proc 12
```

***Picking the number of benchmarking runs*** The default number of runs per input is 30, and the result reported for every such batch of runs is the average. To override this number to instead perform, say, five runs, use the `-runs` flag, as follows.

```
$ ./result/bench/bench.pbench compare -runs 5
```

Later, to add to the previous results an additional 25 runs per input, use the `-mode append` flag.

```
$ ./result/bench/bench.pbench compare -runs 25 -mode
    append
```

### A.7 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20180713.html
- http://cTuning.org/ae/reviewing-20180713.html
- https://www.acm.org/publications/policies/artifact-review-badging

# B Implementation of time measurements

In the description of the algorithm in Section 2, we abstracted over an important detail: measuring the work of a sequential or a parallel function. In this section, we present the version of the algorithm that performs such measurements. The basic strategy is to store in processor-local memory a few cells in which to store intermediate timing results, to accumulate some timing results in this processor-local memory as the program runs, and to combine timing results at join points in the computation. Figure 4 shows the full specification, which we describe below.

***Time measures.*** We assume a function called `now()` that returns the current time. Importantly, we never need to assume a global clock synchronized between the various processors, because all our time measurements are always local to a processor. We nevertheless need to assume that the clocks on the various processors deliver homogeneous results, that is, that the clocks tick (roughly) at the same pace. The precision of the clock should be sufficient to measure time interval one or two orders of magnitude smaller than $\kappa$, that is, roughly in the thousands of cycles. In practice, we rely on hardware cycle counters, which are both very precise and very cheap to query.

***Sequential work time.*** We define the *sequential work time* of a computation to be the sum of the durations of all the sequential subcomputations performed over the scope of that computation.

***Invariants.*** Our algorithm maintains some information as processor-local state. More precisely, each processor manipulates two variables, called `total` and `timer`. These two variables are used for evaluating the sequential work time associated with the innermost call to the `measured_run` function, according to the following two invariant. First, the `timer` variable stores either the time of beginning of the innermost `measured_run` call, or a point in time posterior to it. Second, the `total` variable stores the sequential work time that was performed between the timestamp associated with the beginning of the innermost `measured_run` call and the timestamp stored in the variable `timer`.

The auxiliary function `total_now` returns the sequential work time since the beginning of the innermost call to `measured_run`. It is implemented by computing the sum of the contents of the `total` variable and the width of the time interval between the `timer` and the current time.

Remark: when outside of the scope of any `measured_run` call, the two processor local variables keep track of the sequential work time since the beginning of the program. Our code never exploits such values.

***Transitions.*** When entering the scope of a new call to `measured_run` (Line 7), the variable `total` is set to zero, and the variable `timer` is set to the current time. When exiting the scope of a `measured_run` call (Line 11), the auxiliary function `total_now` is used to compute the total sequential work time over this call.

When entering the scope of a new call to `measured_run`, it is essential to save the relevant information associated with the current (immediately outer) call, otherwise this valuable information would get lost. The variable `t_before` serves this purpose, by saving the sequential work time performed so far in the current call, just before entering the scope of the new call (Lines 15 and 31). When subsequently leaving the scope of this new call, we restore the invariant by setting `total` to `t_before` (Lines 19 and 34), and by setting `timer` to the current time (Lines 20 and 35).

In case the innermost `measured_run` call executes a fork-join, the value of the local variable `t_before` is captured by the join continuation. In technical terms, the value of `t_before` is part of the call frame associated with the join thread that executes after the `sync`. This join thread could be executed on a different processor than the one that initiated the spawn, but such a possibility does not harm the correctness of our algorithm. Regardless of potential thread migration, we correctly set the `timer` and the `total` immediately after the `sync` (Lines 19 and 20). More precisely, the processor that executes the join continuation sets its `total` variable to be the sum of the sequential work time performed before the spawn, plus that performed in each of the two branches (which may execute on different processors), and it resets its `timer` variable to the current time.

In the simple case of a spguard executing its sequential body (Lines 26-29), our algorithm does not bother calling `measured_run`. Instead, it directly measures the time before and after the sequential body, and then computes the difference between the two values. This simpler scheme applies because the sequential body involves no spguard nor any fork-join call.

# C Analysis

## C.1 Definitions and assumptions

***Work and span.*** To take into account the actual overheads of parallelism that granularity control aims to amortize, our analysis accounts for the cost of parallel task creation, written by $\tau$, as well as the cost of evaluating cost functions, written by $\phi$. Although these costs may vary in practice, we assume $\tau$ and $\phi$ to be upper bounds for them. In particular, we assume cost functions to evaluate in constant time, and we exploit the fact that our algorithm makes predictions and handles time measurements in constant time for each spguard call. For the latter, we make the simplifying assumption that the resolution of data races during reports in an estimator incurs no more than a fixed cost. (In practice, CAS-conflicts are quite rare in our experiments; a more refined cost model taking contention into account would be required to account for the cost of CAS conflicts.)

Our analysis establishes bounds on the work and on the span, including the overheads of parallelism, of the execution of a program under the guidance of our automatic granularity control algorithm. We name these entities *total work* and *total span*. Our bounds on the total work and span are expressed with respect to the work and span of the *erasure* version of that program, in which all spguards are replaced with

```
core_local time total = 0
core_local time timer = 0

time total_now(time t)
  return total + (now() - t)

time measured_run(f)
  total = 0
  timer = now()
  f()
  return total_now(timer)

template <Body_left, Body_right>
void fork2join(Body_left bl, Body_right br)
  time t_before = total_now(timer)
  time t_left = spawn measured_run(bl)
  time t_right =      measured_run(br)
  sync
  total = t_before + t_left + t_right
  timer = now()

template <Complexity, Par_body, Seq_body>
void spguard(estimator* es, Complexity c,
             Par_body pb, Seq_body sb)
  int N = c()
  if es.is_small(N)
    time t = now()
    sb()
    es.report(N, now() - t)
  else
    time t_before = total_now(timer)
    t_body = measured_run(pb)
    es.report(N, t_body)
    total = t_before + t_body
    timer = now()
```

**Figure 4.** Implementation of time measurements.

their parallel bodies. The erasure of a program can be viewed as a program that is not granularity controlled at all, but instead exposes all available parallelism.

We define the work and span of the erasure program, written $w$ and $s$ respectively, as the work and the span of its erasure, where work and span are defined in the standard manner [Acar and Blelloch 2017]. In short, the work of two expressions composed sequentially is the sum of the work of the two expressions; the work of two expressions composed in parallel is the sum of the work of the two plus one. The span of two expressions composed sequentially is the sum of the spans of the two; the span of two expressions composed in parallel is the maximum of the spans of the two plus one. Note that to avoid ambiguity we call $w$ and $s$ the *raw work* and the *raw span*, respectively.

We define the *total work*, written $\mathbb{W}$, and the *total span*, written $\mathbb{S}$, to measure the actual work and span of an execution (rather than a program), accounting also for the cost of parallelism and granularity control. More specifically, total work and total span include the *overheads*: each `fork2join` is assumed to incur an extra cost $\tau$ (covering in particular the cost of spawning threads and dealing with their scheduling), and each estimator operation (including the evaluation of the cost function, the call to `is_small` and to `report`, and the time measurements) is assumed to incur an extra cost $\phi$. When computing total work and span, we compute work/span of a spguard based on the branch that it took: if the sequential branch is taken, then the total work/span of the spguard is the total work/span of the sequential branch plus $\phi$. Otherwise, if the parallel branch is taken, then the total work/span of the spguard is the total work/span of the parallel branch plus "$\tau + \phi$".

Finally, for the purpose of stating the assumptions of our theorems, we define the *sequential work*, written $W_s(t, I)$, as the work of a piece of code $t$ executed on input $I$ fully sequentially, that is, where all spguards are forced to execute the sequential body. For such a sequential

execution, no parallelism is created and no cost function is evaluated, thus the sequential work is equal both to the work and to the total work.

***Accuracy of time measurements.*** We assume that time measured for a sequential execution may diverge from the sequential work by up to some multiplicative factor $E$, in either direction. Formally, we assume the existence of constant $E$ such that, for any sequential execution of a purely sequential term $S$ on input $I$, its measured time, written $M(S, I)$, satisfies:

$$\frac{M(S,I)}{W_s(S,I)} \in [\frac{1}{E}, E].$$

***Well-defined spguards.*** We say that a spguard is *well-defined* if the relative sequential work cost of the sequential body and the parallel body of each spguard can be lower and upper bounded by a constant, i.e., are asymptotically the same. Specifically, consider a spguard $g$, with a cost function $F$, sequential body $S$, and parallel body $B$, executed on some input $I$. Let $W_s(S, I)$ denote the sequential work of the sequential body of this call. Let $W_s(B, I)$ denote the sequential work of parallel body of this call, that is, the sequential execution time of the parallel body when all spguards choose their sequential body. The spguard is well-defined if there exists a constant $D$ such that:

$$1 \leq \frac{W_s(B,I)}{W_s(S,I)} \leq D.$$

For the analysis, we consider programs where each spguard is well-defined with respect to the same constant $D$.

The lower bounds assert that the sequential body induces no more work than the parallel body. This is justified because the parallel body can always serve as a sequential body by replacing `fork2join` calls with function calls. Note that without this assumption, the sequentialization of subcomputations can increase the total work making it impossible to control granularity.

The upper bound asserts that the sequential body cannot be arbitrarily faster than executing the parallel body sequentially. This requires the parallel body to use spguards to fall back to a sequential body without doing too much work. This is not difficult to achieve for many nested-parallel algorithms, because they present an opportunity to use a sequential body at each nest level, which usually corresponds to a recursive call. Without this assumption, our algorithm is not guaranteed to converge to the desired constant factor due to the gap between the parallel and sequential bodies.

***Accurate cost functions.*** We make several accuracy assumptions on cost functions.

- As mentioned previously, we assume that cost functions evaluate in constant time, and that this amount of time does not exceed $\phi$.
- We assume that the work increases with the cost. Formally, we assume that, for any spguard $g$ and associated cost function $F$, and for any pair of input $I$ and $J$ such that $F(I) \leq F(J)$, we have: $W_s(g, I) \leq W_s(g, J)$.
- We furthermore assume that the work increases with the cost no faster than at some maximal rate (recall Section 2). Formally, we require that, for the program and the value considered for the parameter $\alpha$, there exists a value $\beta$ such that, for any guard $g$ and associated cost function $F$ involved in the program, and for any pair of inputs $I$ and $J$ such that $F(I) \leq \alpha \cdot F(J)$, we have: $W_s(g, I) \leq \beta \cdot W_s(g, J)$.

***Syntactic forms of spguards.*** For the analysis, we assume that we are given a program written by using the two parallelism primitives that we offer: `fork2join` and `spguards` (described in Section 2). To facilitate and simplify the analysis, we make two syntactic assumptions about programs. These assumptions are not necessary for the algorithm or our implementation but are used purely to facilitate analysis.

First, we assume that the parallel body of a spguard consists of fork-join call surrounded by two pieces of sequential code, one to split the input and one to merge the output. In `C++` syntax:

$$\texttt{spguard}(F,\ \texttt{[\&]\{}S_p\texttt{;fork2join(}L\texttt{,}R\texttt{);}S_m\ \texttt{\},}S\texttt{)}.$$

This assumption causes no loss of generality because more complex expressions can be guarded by sequencing and nesting `spguards`.

Second, we treat the body of spguards as functions that operate on some input. More precisely, an execution of the guard above requires some input $I$ and proceeds by first executing $F(I)$ to compute the cost, and then running with $I$ either the sequential body $S(I)$ or the parallel body based on the outcome of the estimator as described in Section 2. The parallel body is of the form:

$$S_p(I);\ \texttt{fork2join(}L(I_L)\texttt{,}R(I_R)\texttt{);}S_m(I_m)$$

where $I_L$ and $I_R$ are the inputs to branches and $I_m$ is the input to $S_m$. The inputs $I_L$ and $I_R$ are obtained after processing of the input $I$ by $S_p$, and $I_m$ is an output produced by branches $L$ and $R$.

***Regularity of forks.*** As explained in Section 2, to allow for efficient granularity control, we need to rule out ill-balanced programs. We assume the existence of a constant $\gamma$ (with $\gamma \geq 1$), called the *regularity factor*, satisfying the following requirements.

- When considering a fork-join, there must be at least some balance between the left and the right branch. Formally, for any execution on input $I$ of a spguard with sequential body $S$ and branches $L$ and $R$, we assume:

$$\frac{W_s(L,I)}{W_s(S,I)} \quad \text{and} \quad \frac{W_s(R,I)}{W_s(S,I)} \quad \in \left[\frac{1}{\gamma},\ 1 - \frac{1}{\gamma}\right].$$

  Without this assumption, the program can be a right-leaning tree, with all left branches containing only a tiny subcomputation; in such a program the overheads of forks cannot be amortized.

- spguards must be called sufficiently frequently in the call tree. Formally, for any call to a spguard that has sequential body $S$ and executes on input $I$, if the immediate outer spguard call has a sequential body $S'$ and executes on input $I'$, we assume:

$$\frac{W_s(S,I)}{W_s(S',I')} \geq \frac{1}{\gamma}.$$

  Without this assumption, the program can have a spguard called on an input that takes much longer than $\kappa$ to execute, with the immediate inner spguard called on an input that takes much less than $\kappa$, leaving no opportunity to sequentialize a subcomputation that takes approximately time $\kappa$ to execute.
- For an outermost call to a spguard, we need to assume that it involves a nontrivial amount of work. Formally, if an outermost call to a spguard with sequential body $S$ executes on input $I$, we assume $W_s(S,I) \geq \kappa$. Without this assumption, the program can consist of a sequential loop that iterates calls to a spguard on tiny input; and, again, the overheads would not be amortized. (Note that, technically, the requirement $W_s(S,I) \geq \frac{\kappa}{\gamma DE}$ would suffice.)

### C.2 Presentation of the results

***Bounds on total span, total work, and execution time.*** To establish a bound on the parallel execution time of our programs, we first bound the total span and total work. We then derive a bound on the parallel execution time under a greedy scheduler. For the bounds, we assume programs that are 1) $\gamma$-regular, 2) where spguards are well-defined, and 3) where all cost functions are accurate. We express our bounds with respect to work and span, which do not account for the overheads as well as the parameters of the analysis that account for various overheads and factors.

**Theorem C.1** (Bound on the total span)**.** $\mathbb{S} \leq (1 + \phi + \max(\tau,\ E\beta\kappa)) \cdot s$.

The span may grow by a multiplicative factor, but no more. As our final bound will show, for a program with sufficient parallelism (i.e., with $\frac{w}{s} \gg P$), this increase in the span has no visible impact on the parallel run time.

**Theorem C.2** (Bound on the total work)**.** *Let $\kappa' = \frac{\kappa}{DE\gamma}$, and $F = 1 + \log_\alpha \frac{\kappa}{DE}$, and $H = \log_{\gamma/(\gamma-1)} \frac{\kappa}{DE}$, and $P$ denote the number of processes, and $G$ denote the number of spguards occuring in the code. Then, we have*

$$\mathbb{W} \leq \left(1 + \frac{\tau + 2\phi}{\kappa'}\right) \cdot w + PFGH \cdot (\tau + 2\gamma\phi).$$

The first component of the left hand side asserts that the work grows by no more than a multiplicative factor $1 + \epsilon$, where $\epsilon$ may be tamed to a couple of percents by the choice of a sufficiently larger $\kappa$. The second component asserts that the total overheads associated with the sequentialization of tiny subcomputations during the convergence phase of the estimators are bounded by some constant cost, proportional to the number of estimators and to the product $P \cdot F \cdot H$, which is $P \cdot O(\log^2 \kappa)$.

Remark: in the bound on the work, the value of $\kappa$ occuring in the definition $F$ and $H$ is to be expressed in cycles; We here exploit the assumption that a computation with cost $N$ takes at least $N$ cycles to execute. If this was not the case, the bound could easily be adapted by adding a constant factor to $F$ and $H$.

To bound the parallel run time, we exploit Brent's theorem [Brent 1974], which asserts that, for any greedy scheduler, $T_p \leq \frac{\mathbb{W}}{P} + \mathbb{S}$, where $P$ stands for the number of processors. To simplify the final statement, we make some over-approximation, and also we exploit several inequalities that are always satisfied in practice. We assume $\kappa \geq \tau$ and $\kappa \geq 1 + \phi$, since in practice the user always sets $\kappa \gg \max(\tau, \phi)$ to ensure small overheads.

**Theorem C.3** (Bound on the parallel run time)**.** *Let $P$ denote the number of processors and $G$ denote the number of spguards.*

$$T_P \ \leq \ \left(1 + \frac{\gamma ED \cdot (\tau + 2\phi)}{\kappa}\right) \cdot \frac{w}{P} \ + \ (E\beta + 1) \cdot \kappa \cdot s \ + \ O(G \cdot \log^2 \kappa \cdot (\tau + 2\gamma\phi)).$$

**Theorem 2.1** (Simplified bound on the parallel run time)**.** *For fixed hardware and any program, all parameters of the analysis except for $\kappa$ (the unit of parallelism) can be replaced with constants, leaving us with the following bound:*

$$T_P \ \leq \ \left(1 + \frac{O(1)}{\kappa}\right) \frac{w}{P} \ + \ O(\kappa) \cdot s \ + \ O(\log^2 \kappa).$$

The most important term in this bound is the first term that says that the overhead of various practical factors impact the work term $w$ by only a small factor $O(1)/\kappa$, which can be reduced by controlling $\kappa$. The second term states that doing so increases the span by a small factor $O(\kappa)$ and that all of this comes at a small logarithmic overhead, which is due to our granularity control algorithm $O(\log^2 \kappa)$.

Our theorem establishes that a non-granularity controlled nested parallel program with work $w$ and span $s$ can be executed on a real machine to guarantee fast execution by using our granularity control algorithm. Our experiments (Section 3) show that the analysis appear to be valid in practice.

### C.3 Additional definitions

***Syntax of programs*** The BNF grammar for programs with spguards is thus as follows. For brevity, we focus on the language constructs that matter to the analysis: sequences, conditionals, and spguards combined with fork-join.

- B ::= *a boolean variable*
- S ::= *a purely sequential piece of code, without forks nor guards.*
- t ::= S | (t; t) | `if` B `then` t `else` t | `spguard`(S, p, S)
- p ::= S; `fork2join`(t, t); S

* * *

***Detailed definitions of work and span*** The table below shows the definition of *raw* work and span, which correspond to the standard definitions used in parallel program analysis, as well as the definition of *total* work and span, which include the overheads of thread creation, written $\tau$, and the overheads associated with the estimator, written $\phi$. Note that in the definitions below we are ultimately referring to the work of a sequential piece of code, written $W(S)$, which we assume to be defined as the number of instructions involved in the execution of $S$, or as the number of cycles involved if the considered execution model assigns a cost to each instruction.

**Definition C.4** (Work and span, raw and total, for each construct)**.**

| t: source expression | $W(t)$: raw work | $S(t)$: raw span | $\mathbb{W}(t)$: total work | $\mathbb{S}(t)$: total span |
|---|---|---|---|---|
| S | $W(S)$ | $W(S)$ | $W(S)$ | $W(S)$ |
| $(t_1; t_2)$ | $W(t_1) + W(t_2)$ | $S(t_1) + S(t_2)$ | $\mathbb{W}(t_1) + \mathbb{W}(t_2)$ | $\mathbb{S}(t_1) + \mathbb{S}(t_2)$ |
| `if` $B$ `then` $t_1$ `else` $t_2$ *when* $B$ *is* `true` | $1 + W(t_1)$ | $1 + S(t_1)$ | $1 + \mathbb{W}(t_1)$ | $1 + \mathbb{S}(t_1)$ |
| `if` $B$ `then` $t_1$ `else` $t_2$ *when* $B$ *is* `false` | $1 + W(t_2)$ | $1 + S(t_2)$ | $1 + \mathbb{W}(t_2)$ | $1 + \mathbb{S}(t_2)$ |
| `spguard`$(F, (S_p;$ `fork2join`$(L, R); S_m), S)$ *when parallel body is chosen* | $W(S_p) + W(L) + W(R) + 1 + W(S_m)$ | $S(S_p) + \max(S(L), S(R)) + 1 + S(S_m)$ | $\mathbb{W}(S_p) + \mathbb{W}(L) + \mathbb{W}(R) + \phi + \tau + \mathbb{W}(S_m)$ | $\mathbb{S}(S_p) + \max(\mathbb{S}(L), \mathbb{S}(R)) + \phi + \tau + \mathbb{S}(S_m)$ |
| `spguard`$(F, (S_p;$ `fork2join`$(L, R); S_m), S)$ *when sequential body is chosen* | $W(S)$ | $W(S)$ | $W(S) + \phi$ | $W(S) + \phi$ |

Due to the fact that spguard may dynamically select between the execution of the sequential or the parallel branch, it does not make sense to speak about the work and span of a program. We may only speak about the work and span of a particular execution of the program, that is, of the work and span of an execution trace describing which spguards have been sequentialized and which have not.

**Definition C.5** (Execution trace)**.** A particular execution of a source term $t$ on an input $I$ corresponds to a trace, written $X$, that describes, for each evaluation of a spguard during the program execution, whether the sequential body or the parallel body is selected by the spguard.

**Definition C.6** (Work and span, raw and total, of an execution trace)**.** For an execution of a source term $t$ on an input $I$ producing a trace $X$, we let:

- $W(t, I, X)$ denote the *raw work*,
- $S(t, I, X)$ denote the *raw span*,
- $\mathbb{W}(t, I, X)$ denote the *total work*,
- $\mathbb{S}(t, I, X)$ denote the *total span*.

The definitions are obtained by applying the appropriate rules from the previous table.

When the arguments $t$, $I$ and $X$ are obvious from the context, we write simply $W$, $S$, $\mathbb{W}$, and $\mathbb{S}$.

**Definition C.7** (Sequential work)**.** For a term $t$ and an input $I$, we define the *sequential work*, written $W_s(t, I)$, as the raw work $W(t, I, X)$, where $X$ is the trace that systematically selects the sequential bodies.

**Definition C.8** (Work and span)**.** For a term $t$ and an input $O$, we define:

- the *work* $w(t, I)$ as the raw work $W(t, I, X)$ where $X$ is the trace that systematically selects parallel bodies.
- the *span* $s(t, I)$ as the raw span $S(t, I, X)$ where $X$ is the trace that systematically selects parallel bodies.

For a purely sequential subcomputation, the work is equal to the raw work. (It also matches the span since there is no parallelism involved.) We use this fact implicitly in several places through the proofs.

* * *

***Time measurements*** In addition to the definition already given for the measurement of a sequential execution time, written $M(t, I)$, we need for the analysis to quantify the total sequential work time involved in a parallel execution, written $M(t, I, X)$. (Recall Appendix B.)

**Definition C.9** (Measured time)**.** We let:

- $M(t, I)$ denote the measured time of the sequential execution of the term $t$ on input $I$.

- $M(t, I, X)$ denote the sum of the measured time of all the pieces of sequential sub-computations involved in the evaluation of the term $t$ on input $I$ according to trace $X$. The measure thus ignores the overheads of fork-join operations and the overheads associated with our runtime decisions.

* * *

***Classification of spguard calls***

**Definition C.10** (Small call)**.** A call to a spguard $g$ on input $I$ is said to be *small* whenever $W_s(g, I) \leq \frac{\kappa}{DE}$. Otherwise, it is said to be *non-small*.

**Definition C.11** (Domination of a spguard call)**.**

- We say that a spguard call $a$ is *dominated* by a spguard call $b$ if $a$ is executed as part of the execution of the parallel body of $b$.
- We say that $a$ is *directly dominated* by $b$ if there are no spguard in-between, i.e. if there does not exist a spguard $c$ that dominates $a$ and at the same time is dominated by $b$.

**Definition C.12** (Covered sequential call)**.** A sequential call to a spguard is said to be *covered* if it is directly dominated by a parallel small call. Otherwise, it is *non-covered*.

**Definition C.13** (Classification of spguard calls)**.** Every spguard call falls in one of the four categories:

- *parallel non-small call* (when the spguard executes its parallel body on a non-small call).
- *parallel small call* (when the spguard executes its parallel body on a small call).
- *covered sequential call* (when the spguard executes its sequential body and is dominated by a parallel small call; note that a covered sequentiall call is a small call).
- *non-covered sequential call* (when the spguard executes its sequential body but is not directly dominated by a parallel small call; in this case, it can be dominated by a parallel non-small call, or not dominated at all).

*Proof.* A call is either parallel or sequential. If parallel, it is either small or non-small. If sequential, it is either covered or non-covered. □

**Definition C.14** (Critical parallel calls)**.** A small parallel call to a spguard is said to be a *critical* if all the calls that it dominates are sequential small calls.

This classification simplifies the presentation of the proof.

**Proof strategy.** Any non-small call amortizes the overheads in a standard manner, i.e., provides multiplicative factor to the work and span. So, our major task is to bound the overhead spend during small calls. Our bound is obtained in several steps.

1. We bound the number of critical parallel calls (that is, a parallel small call that dominates only sequential small calls) in Lemma C.26. To that end, we exploit the fact that, after each critical parallel call, the algorithm performs a report that increases the value of `Nmax` by at least some constant factor.
2. We show that the number of parallel small calls is bounded in terms of the number of critical parallel calls in Lemma C.28. To that end, we observe that each parallel small call features at least one nested critical parallel call, and that, reciprocally, each critical parallel call is nested in at most a logarithmic number of small parallel calls.
3. We bound the number of covered sequential calls in terms of the number of parallel small calls in Lemma C.29. We do so by arguing that each parallel small call can have at most a logarithmic number of nested sequential calls.
4. We independently show that the non-covered sequential calls correspond to non-small calls, so their overheads are properly amortized in Lemma C.30.

All together, we derive the bound on the overheads associated with all small calls in Lemma C.32.

* * *

***Parameters involved in the statement of the bounds***

**Definition C.15** (Bound on the number of processors)**.** Let $P$ denote the number of processors involved in the evaluation.

**Definition C.16** (Bound on the number of spguards)**.** Let $G$ denote the total number of different spguards occuring in the source code of the program.

**Definition C.17** (Auxiliary parameter $F$)**.** We define $F = 1 + \log_\alpha \frac{\kappa}{DE}$, where $\kappa$ is expressed in number of machine cycles.

**Definition C.18** (Auxiliary parameter $H$)**.** We let $H = \log_{\gamma/(\gamma-1)} \frac{\kappa}{DE}$, where $\kappa$ is expressed in number of machine cycles.

Remark: as explained previously, for simplicity, in the definitions of the auxiliary constants $F$ and $H$ introduced below, we assume that the number of cycles involved in the sequential execution of a spguard always exceeds the number returned by the cost function. The bounds could be easily adapted by adding a constant factors to $F$ and $H$ if this was not the case.

**Proof strategy.** To bound the total work $\mathbb{W}$, our proof first bounds the total work excluding overheads involved in parallel small calls and covered sequential calls, which we call $\mathbb{W}'$, then bounds the excluded overheads, that is, the value of $\mathbb{W} - \mathbb{W}'$.

**Definition C.19** (Total work excluding the overheads involved in parallel small calls)**.** We let $\mathbb{W}'$ denote the subset of the total work $\mathbb{W}$ obtained by excluding the overheads ($\tau$ and $\phi$) involved in small parallel calls, in the sense that when reaching such a call, we only count the raw work involved in this call, regardless of the decisions involved in the subcomputations.

* * *

### C.4 Basic auxiliary lemmas

**Lemma C.20.** *If term $S$ does not contain any spguard, then $w(S, I) = W_s(S, I)$.*

*Proof.* Immediate from the definitions. □

**Lemma C.21** (Sequential work associated with a spguard)**.** *In the particular case where the term $t$ corresponds to some spguard $g$, the sequential work is that of its sequential body $S$, which does not contain any spguard. Thus, we have:*

$$W_s(g, I) = W_s(S, I) = W(S, I, X) = \mathbb{W}(S, I, X) - \phi \qquad \text{for any trace } X.$$

**Lemma C.22** (Relationship between work and measured time of a computation)**.** *For any term $t$ executed on input $I$ according to trace $X$, we have:*

$$\frac{M(t, I, X)}{W(t, I, X)} \in [\frac{1}{E}, E].$$

*Proof.* Recall our hypothesis: $\frac{M(S,I)}{W(S,I)} \in [\frac{1}{E}, E]$ for all sequential computation $S$. First, we prove that $M(t, I, X) \le E \cdot W(t, I, X)$ by induction on the execution tree.

- Case t = S. By assumption on the variability of time measurements, we have: $M(t, I, X) = M(S, I, X) = M(S, I) \le E \cdot W_s(S, I) = E \cdot W_s(t, I)$. Using the fact $W_s(S, I) = W(S, I, X)$ from Lemma C.21 we get $M(t, I, X) \le E \cdot W(t, I, X)$.
- Case t = $(t_1, t_2)$. We have: $M(t, I, X) = M(t_1, I, X) + M(t_2, I, X) \le E \cdot W(t_1, I, X) + E \cdot W(t_2, I, X) = E \cdot W(t, I, X)$.
- Case t = `if` $B$ `then` $T_1$ `else` $T_2$. Similar to the previous case, after performing the case analysis.
- Case t = `spguard`$(F, (S_p;$ `fork2join`$(L, R); S_m), S)$.
  - First case: spguard chooses the parallel body. Then, $M(t, I, X) = M(S_p, I, X) + M(L, I, X) + M(R, I, X) + 1 + M(S_m, I, X) \le E \cdot W(S_p, I, X) + E \cdot W(L, I, X) + E \cdot W(R, I, X) + 1 + E \cdot W(S_m, I, X) \le E \cdot (W(S_p, I, X) + W(L, I, X) + W(R, I, X) + 1 + W(S_m, I, X)) = E \cdot W(t, I, X)$.
  - Second case: spguard chooses the sequential body. Similar to the case $t = S$.

The second inequality $W(t, I, X) \le E \cdot M(t, I, X)$ again can be proved by induction on the execution tree. The proof is identical to the proof of the inequality above with the only difference: $M$ and $W$ should be swapped. □

We assumed, for each spguard, that the execution of the sequential body is faster than the sequential execution of the parallel body, thus, when the trace chooses the parallel body the execution should be slower than the corresponding sequential execution.

**Lemma C.23.** *Consider a term $t$ with well-defined spguards. For any execution $t$ on input $I$ with trace $X$, we have: $W_s(t, I) \le W(t, I, X)$.*

*Proof.* We prove this by induction on the execution tree.

- Case t = S. We have: $W_s(S, I) = W(S, I, X)$.
- Case t = $(t_1, t_2)$. We have: $W_s(t, I) = W_s(t_1, I) + W_s(t_2, I) \le W(t_1, I, X) + W(t_2, I, X) = W(t, I, X)$.
- Case t = `if` $B$ `then` $t_1$ `else` $t_2$. Similar to the previous case, after performing case analysis.
- Case t = `spguard`$(F, (S_p;$ `fork2join`$(L, R); S_m), S)$. We distinguish two cases:
  - First case: spguard chooses the parallel body. Then, using the property of well-defined spguards, $W_s(t, I) \le W_s(S_p, I) + 1 + W_s(L, I) + W_s(R, I) + W_s(S_m, I) \le W(S_p, I, X) + 1 + W(L, I, X) + W(R, I, X) + W(S_m, I, X) = W(t, I, X)$.
  - Second case: spguard chooses the sequential body. Then, $W_s(t, I) = W_s(S, I) = W(S, I, X) = W(t, I, X)$.

□

Because work considers full parallelization, it never selects the sequential bodies of spguards, which are assumed to execute faster than parallel bodies on one process. Thus, the work always exceeds the raw work.

**Lemma C.24.** *Consider a term $t$ with well-defined spguards. For any execution of $t$ on input $I$ with trace $X$, we have: $w(t, I) \ge W(t, I, X)$.*

*Proof.* We prove this by induction on the execution tree.

- Case t = S. We have: $w(S, I) = W(S, I, X)$.
- Case t = $(t_1, t_2)$. We have: $w(t, I) = w(t_1, I) + w(t_2, I) \ge W(t_1, I, X) + W(t_2, I, X) = W(t, I, X)$
- Case t = `if` $B$ `then` $t_1$ `else` $t_2$. Similar to the previous case, after performing the case analysis.
- Case t = `spguard`$(F, (S_p;$ `fork2join`$(L, R); S_m), S)$. We distinguish two cases:
  - First case: spguard chooses the parallel body. Then, $w(t, I) = w(S_p, I) + 1 + w(L, I) + w(R, I) + w(S_m, I) \ge W(S_p, I, X) + 1 + W(L, I, X) + W(R, I, X) + W(S_m, I, X) = W(t, I, X)$.
  - Second case: spguard chooses the sequential body. Thus, the raw work corresponds to the sequential work, i.e. $W(t, I, X) = W_s(t, I)$. Besides, by definition of work, we know that $w(t, I) = W(t, I, X')$, where $X'$ is the trace that systematically selects parallel bodies. By Lemma C.23, $w(t, I) = W(t, I, X') \ge W_s(t, I) = W(t, I, X)$

□

### C.5 Proof of the main results

**Lemma C.25.** *If a spguard $g$ executes its sequential body, then its sequential work is bounded as follows:*

$$W_s(g, I) \leq E \cdot \beta \cdot \kappa.$$

*Proof.* At first, observe that $W_s(g, I) = W_s(S, I)$, where $S$ denotes the sequential body of the spguard.

Let $N$ denote the value of $F(I)$. According to the implementation of function `is_small`, a spguard selects the sequential body according to the boolean condition: ($\mathtt{N} \leq \mathtt{Nmax}$) or (($\mathtt{N} \leq \alpha \cdot \mathtt{Nmax}$) and ($\mathtt{N} \cdot \mathtt{C} \leq \alpha \cdot \kappa$)), where `Nmax` and `C` are the values stored in the estimator for this spguard.

Since $N$ is non-negative, the condition may only evaluate to true if `Nmax` is non-zero, indicating that at least one previous report has been stored for this spguard. Let $J$ be the input on which this previous report has been obtained, in other words $\mathtt{Nmax} = F(J)$, let $X$ denote the trace of spguards choices, and let $M(g, J, X)$ denote the measured time at this previous report. Since the report was stored, according to the implementation of function `report`, we know that $M(g, J, X) \leq \kappa$ and that $\mathtt{C} = \frac{M(g,J,X)}{F(J)}$.

By Lemma C.21 on sequential work we know that $W_s(S, J) = W_s(g, J)$. Lemma C.23 provides us with $W_s(g, J) \leq W(g, J, X)$. And Lemma C.22 on measured time in the execution on input $J$ gives $W(g, I, X) \leq E \cdot M(g, J, X)$. Thus, the raw work on input $J$ can be bounded as follows: $W_s(S, J) = W_s(g, J) \leq W(g, J, X) \leq E \cdot M(g, J, X)$.

In what follows, we establish the inequality $W_s(S, I) \leq \beta \cdot W_s(S, J)$. By exploiting that $W_s(S, J) \leq E \cdot M(g, J, X)$ and $M(g, J, X) \leq \kappa$, this inequality allows us to conclude as follows:

$$W_s(g, I) = W_s(S, I) \leq \beta \cdot W_s(S, J) \leq \beta \cdot E \cdot M(g, J, X) \leq E \cdot \beta \cdot \kappa.$$

The desired inequality is deduced from the fact that the boolean condition evaluates to true. Indeed, we know that $\mathtt{N} \leq \mathtt{Nmax}$ is true, or that $\mathtt{N} \leq \alpha \cdot \mathtt{Nmax}$ and $\mathtt{N} \cdot \mathtt{C} \leq \alpha \cdot \kappa$ are true.

- In the first case, the condition reformulates to $F(I) \leq F(J)$. By the property of well-defined spguards, we deduce $W_s(S, I) \leq W_s(S, J)$. By exploiting $\beta > 1$, we conclude $W_s(S, I) \leq \beta \cdot W_s(S, J)$, as desired.
- In the second case, the condition reformulates to: $F(I) \leq \alpha \cdot F(J)$ and $F(I) \cdot \frac{M}{F(J)} \leq \alpha \cdot \kappa$. By the property of well-defined spguards, exploiting the first inequality, we deduce: $W_s(S, I) \leq \beta \cdot W_s(S, J)$, the desired inequality.

□

**Theorem C.1** (Bound on the total span)**.** *For any execution of a program with well-defined spguards, we have:*

$$\mathbb{S} \leq (1 + \phi + \max(\tau, E\beta\kappa)) \cdot s$$

*Proof.* Let $\rho$ be a shorthand for $1 + \phi + \max(\tau, E\beta\kappa)$. We establish the inequality $\mathbb{S} \leq \rho \cdot s$ by induction on the execution tree.

- Case t = S. The program is sequential, so $\mathbb{S} = s \leq \rho \cdot s$.
- Case t = $(t_1; t_2)$. We have: $\mathbb{S} = \mathbb{S}(t_1) + \mathbb{S}(t_2) \leq \rho \cdot s(t_1) + \rho \cdot s(t_2) = \rho \cdot s$.
- Case t = `if` $B$ `then` $t_1$ `else` $t_2$. Similar to the previous case, after considering the two cases.
- Case t = `spguard`$(F, (S_p; \mathtt{fork2join}(L, R); S_m), S)$. We have two cases:
  - First case: the spguard chooses the parallel body. Then, $\mathbb{S} = \mathbb{S}(S_p) + \max(\mathbb{S}(L), \mathbb{S}(R)) + \phi + \tau + \mathbb{S}(S_m) \leq \rho \cdot s(S_p) + \max(\rho \cdot s(L), \rho \cdot s(R)) + \rho + \rho \cdot s(S_m) = \rho \cdot (s(S_m) + \max(s(L), s(R)) + s(S_p) + 1) = \rho \cdot s$.
  - Second case: the spguard chooses the sequential body. In this case by Lemma C.25 we know that the sequential work of this spguard call is bounded: $W_s(g, I) \leq E\beta\kappa$. Thus, using the fact that the span is nonnegative ($s \geq 1$), we have:

$$\mathbb{S} = W_s(S, I) + \phi \leq E\beta\kappa + \phi \leq \rho \leq \rho \cdot s.$$

□

**Lemma C.26** (Bound on the number of critical calls)**.** *A given estimator involves no more than $P \cdot F$ critical calls.*

*Proof.* Recall that $P$ denotes the number of processes and that $F$ is defined as $1 + \log_\alpha \frac{\kappa}{DE}$, which is $O(\log \kappa)$. In this proof, we show that each process can perform no more than $F$ critical calls on a given estimator. Multiplying by the number of processes yields the bound $P \cdot F$. We remind that our reports are performed by atomic updates (Figure 3 Lines 7-8). By that, the $P$ processes may concurrently interact over a same estimator and this can only speedup the convergence process.

Let us consider the critical call on input $I$ with measured time $M$ by process $p$.

At first, we prove that $M$ does not exceed $\kappa$. From Lemma C.22, we have $\frac{M}{W} \leq E$. Because the critical call is small, we know that $W_s(S) \leq \frac{\kappa}{DE}$ and that all directly dominated calls are sequentialized, thus $W = W_s(S_p) + W_s(L) + W_s(R) + 1 + W_s(S_m)$. By the property of well-defined spguards $W_s(S_p) + W_s(L) + W_s(R) + 1 + W_s(S_m) \leq D \cdot W_s(S)$. Combining all these facts, we get $M \leq E \cdot W = E \cdot (W_s(S_p) + W_s(L) + W_s(R) + 1 + W_s(S_m)) \leq E \cdot D \cdot W_s(S) \leq E \cdot D \cdot \frac{\kappa}{DE} = \kappa$.

Second, after the report `Nmax` is at least $F(I)$: either another process updated `Nmax` to be not less than $F(I)$, or this report successfully updates `Nmax` and sets it to $F(I)$. This report can be successful since the reported time $M$ does not exceed $\kappa$.

Next, we show that the cost of the next critical call by process $p$ increases at least by a factor $\alpha$. Suppose that the input of the next critical call is $J$. Our goal is to show $F(J) > \alpha \cdot F(I)$. For the spguard to choose parallel body on $J$, the boolean condition in `is_small` function

($F(J) \leq \alpha \cdot F(Y)$, where $Y$ is the latest reported input), needs to evaluate to `false`. This means that the following condition needs to be satisfied: $F(J) > \alpha \cdot F(Y)$. Note that $F(I) \leq F(Y)$, since after the last report by process $p$ the value of `Nmax` was at least $F(I)$. Thus, we get the desired inequality $F(J) > \alpha \cdot F(Y) \geq \alpha \cdot F(I)$.

Since critical calls are small calls, their work cannot exceeed $\frac{\kappa}{DE}$. By the assumption that the number of cycles exceeds the cost, the value of the cost function cannot exceed $\frac{\kappa}{DE}$, with $\kappa$ expressed in cycles. The cost associated with the first critical report is at least 1 unit, and since the value increases by a factor $\alpha$ at least between every two consecutive critical call by the same process, the number of such critical calls cannot exceed $P \cdot (1 + \log_\alpha \frac{\kappa}{DE})$.

□

**Lemma C.27** (Bound on the number of nested small calls)**.** *Small spguard calls may be nested at no more than on depth $H$.*

*Proof.* Consider a set of nested small calls. The outermost call is small, so it involves work at most $\frac{\kappa}{D \cdot E}$. As we argue next, the ratio between the work of a call directly nested into another one is at least $\frac{\gamma}{\gamma - 1}$. Combining the two, we can deduce that the number of nested small calls is bounded by $\log_{\gamma/(\gamma-1)} \frac{\kappa}{D \cdot E}$.

To bound the ratio between two directly nested calls, we proceed as follows. Consider a spguard with sequential body $S$ on input $I$, directly dominated by a call to a spguard with sequential body $S'$ on input $I'$. Assume, without loss of generality, that the inner spguard call occurs in the left branch, call it $L$, of the outer spguard call. From the $\gamma$-regularity assumption, we know that $\frac{W_s(L', I')}{W_s(S', I')} \leq 1 - \frac{1}{\gamma}$. Furthermore, since $S$ executes as a subcomputation of the sequential execution of $L'$, we have: $W_s(S, I) \leq W_s(L', I')$. Combining the two inequalities gives: $W_s(S, I) \leq (1 - \frac{1}{\gamma}) \cdot W_s(S', I')$, which can be reformulated as $\frac{W_s(S', I')}{W_s(S, I)} \geq \frac{\gamma}{\gamma - 1}$, meaning that the ratio between the two nested calls is at least $\frac{\gamma}{\gamma - 1}$. □

**Lemma C.28** (Bound on the number of parallel small calls)**.** *A given spguard involves no more than $P \cdot F \cdot H$ parallel small calls.*

*Proof.* First, we observe that each parallel small call must dominate at least one critical call. Indeed, when following the computation tree, there must be a moment at which we reach a spguard such that all dominated spguards choose sequential bodies, or such that there are no dominated spguards in the body.

Thus, we may bound the number of parallel small calls by multiplying the number of critical calls with the number of parallel small calls that dominate it (including the critical call, which is itself a parallel small call). Of course, we may be counting a same parallel call several times, but this over-approximation is good enough to achieve our bound. More precisely, we multiply the bound from Lemma C.27 with the bound from Lemma C.26. □

**Lemma C.29** (Bound on the number of covered sequential calls)**.** *A given spguard involves no more than $P \cdot F \cdot H \cdot (2\gamma - 2)$ covered sequential calls.*

*Proof.* A covered sequential call is dominated by a parallel small call. The claimed bound follows from the bound of Lemma C.28 and the fact that, for each parallel small call, we can have at most $2\gamma - 2$ directly dominated sequential small calls. We next prove this last claim.

Consider a parallel small call on input $I'$ to a spguard with sequential body $S'$ and branches $L'$ and $R'$. Consider a directly dominated call on input $I_c$ to a spguard with sequential body $S_c$. By $\gamma$-regularity, we have: $W_s(S_c, I_c) \geq \frac{1}{\gamma} \cdot W_s(S', I')$. From the structure of the program, we know: $W_s(L') + W_s(R') \geq \sum_{c \in C} W_s(S_c, I_c)$, where $C$ is the set of all directly dominated sequential small calls. Also, by the first property of $\gamma$-regular programs: $W_s(L'), W_s(R') \leq (1 - \frac{1}{\gamma}) \cdot W_s(S', I')$, giving us additional inequality $W_s(L') + W_s(R') \leq (2 - \frac{2}{\gamma}) \cdot W_s(S', I')$. By combining the facts, we get: $|C| \cdot \frac{1}{\gamma} \cdot W_s(S', I') \leq \sum_{c \in C} W_s(S_c, I_c) \leq W_s(L') + W_s(R') \leq (2 - \frac{2}{\gamma}) \cdot W_s(S', I')$. Thus, we deduce that the number of directly dominated sequential small calls $|C|$ does not exceed $2\gamma - 2$. □

**Lemma C.30** (Work involved in a non-covered sequential call)**.** *If a call to a spguard $g$ on input $I$ is a non-covered sequential call, then it involves at least some substantial amout of work, in the sense that:*

$$W_s(g, I) \geq \frac{\kappa}{\gamma DE}.$$

*Proof.* A call can be a non-covered sequential call for one of two reasons.

- First case: the call is directly dominated by another spguard call. This spguard call is necessarily parallel, and by assumption it is a non-small call (otherwise, the inner call would be covered). This non-small parallel call occurs on some spguard $g'$ with sequential body $S'$ executed on input $I'$. This call is not small, meaning that $W_s(S', I') > \frac{\kappa}{DE}$ holds. Besides, by the definition of $\gamma$-regularity, we have: $\frac{W_s(S, I)}{W_s(S', I')} \geq \frac{1}{\gamma}$. Combining the two inequalities gives: $W_s(S, I) > \frac{\kappa}{\gamma DE}$.
- Second case: the call is not dominated by any other spguard call. Then, by the last assumption from the definition of $\gamma$-regularity, we have $W_s(S) \geq \frac{\kappa}{\gamma DE}$.

□

**Lemma C.31** (Bound on the work excluding overheads during small calls)**.**

$$\mathbb{W}' \leq \left(1 + \frac{DE\gamma \cdot (\tau + 2\phi)}{\kappa}\right) \cdot w$$

*Proof.* Let us introduce the shorthand $\kappa' = \frac{\kappa}{\gamma DE}$. The bound is equivalent to $\mathbb{W}' \leq \left(1 + \frac{1}{\kappa'}\tau + \frac{2}{\kappa'}\phi\right) \cdot w$.

Let $B(w) = w + \frac{(w-\kappa')^+}{\kappa'}\tau + \frac{(2w-\kappa')^+}{\kappa'}\phi$, where $x^+$ is defined as $x$ if $x$ is non-negative, and 0, otherwise.
We establish the slightly tighter inequality $\mathbb{W}' \leq B(w)$, by induction on the execution tree.

- Case t = S. The program is sequential, so $\mathbb{W}' = \mathbb{W} = w \leq B(w)$.
- Case t = $(t_1; t_2)$. $\mathbb{W}' = \mathbb{W}'(t_1) + \mathbb{W}'(t_2) \leq B(w(t_1)) + B(w(t_2)) \leq B(w_1 + w_2)$. The last inequality holds, because $B(x) + B(y) \leq B(x + y)$ due to the fact $(x - k)^+ + (y - k)^+ \leq (x + y - k)^+$.
- Case t = `if` $B$ `then` $t_1$ `else` $t_2$. Considering two different cases, similar to the previous case.
- Case t = `spguard`$(F, (S_p; \texttt{fork2join}(L, R); S_m), S)$. Let $I$ be the input and $X$ be the trace for this call. Thereafter, we write $W_s(S)$ as short for $W_s(S, I)$ ad $w$ as short for $w(t, I)$.
  By Lemma C.23, we know that: $W_s(t, I) \leq W(t, I, X)$. By Lemma C.24, we have $W(t, I, X) \leq w$. Combining the two gives $W_s(t) \leq w$. We then distinguish four cases:
  - First case: the call is the parallel non-small call. By definition of *small*, this means: $W_s(S) > \frac{\kappa}{DE}$. Let us focus first on the left branch. By the definition of $\gamma$-regularity, we know $\frac{W_s(L)}{W_s(S)} \geq \frac{1}{\gamma}$. Besides, for the same reason as we have $W_s(t) \leq w$, we have $W_s(L) \leq w(L)$. Combining these results, we get: $w(L) \geq \frac{1}{\gamma} \cdot W_s(S) > \frac{\kappa}{\gamma DE} = \kappa'$. By symmetry, we have $w(R) > \kappa'$. Also, we know that $B(x) + B(y) \leq B(x + y)$. Putting everything together gives: $\mathbb{W}' = \mathbb{W}(S_p) + \mathbb{W}'(L) + \mathbb{W}'(R) + \tau + \phi + \mathbb{W}(S_m) \leq B(w(S_p)) + B(w(L)) + B(w(R)) + \tau + \phi + B(w(S_m)) = B(w(S_p)) + (w(L) + w(R)) + \frac{(w(L)-\kappa')+(w(R)-\kappa')+\kappa'}{\kappa'}\tau + \frac{(2w(L)-\kappa')+(2w(R)-\kappa')+\kappa'}{\kappa'}\phi + B(w(S_m)) \leq B(w(S_p)) + B(w(L) + w(R)) + B(w(S_m)) \leq B(w(S_p) + w(L) + w(R) + w(S_m)) = B(w - 1) \leq B(w)$.
  - Second case: the call is the parallel small call. By definition of $\mathbb{W}'$, we do not count the overheads within the scope of this call and only count the raw work, thus $\mathbb{W}' \leq W(t, I, X)$. Recall that $W(t, I, X) \leq w$. Besides, by definition of $B$, we have: $w \leq B(w)$. Combining these results gives: $\mathbb{W}' \leq B(w)$.
  - Third case: the call is the covered sequential call. In this case, the call is dominated by a parallel small call. Thus, such calls are never reached by our proof by induction, because to reach them one would necessarily first go through the case that treats parallel small calls, case which does not exploit an induction hypothesis. (Recall that the definition of $\mathbb{W}'$ excludes all the overheads involved throughout the execution of a parallel small call.)
  - Fourth case: the call is the non-covered sequential call. In this case, the work equals to the sequential work, $w = W_s(S)$, and $\mathbb{W}'$ does not exclude the overheads, so $\mathbb{W}' = W_s(S) + \phi$. By Lemma C.30, we have: $W_s(S) \geq \frac{\kappa}{\gamma DE}$. In other words, $W_s(S) \geq \kappa'$. This inequality may be reformulated as: $\frac{2W_s(S)-\kappa'}{\kappa'} \geq 1$. Recall that we have $W_s(S) \leq w$, since here $W_s(t) = W_s(S)$. Combining all these results yields:

$$\mathbb{W}' = W_s(S) + \phi \leq W_s(S) + \frac{2W_s(S) - \kappa'}{\kappa'}\phi \leq B(W_s(S)) \leq B(w).$$

□

**Lemma C.32** (Bound on the overheads associated with small calls)**.**

$$\mathbb{W} - \mathbb{W}' \;\leq\; PFGH \cdot (\tau + (2\gamma - 1) \cdot \phi)$$

*Proof.* This bound on the overheads associated with small calls is obtained as the sum of the overheads associated with covered sequential calls and the overheads associated with parallel small calls. For the former, each covered sequential call induces an overhead of $\phi$, and there are at most $PFH \cdot (2\gamma - 2)$ of them per spguard, by Lemma C.29. For the latter, each parallel small call induces an overhead of $\tau + \phi$, and there are at most $PFH$ of them per spguard, by Lemma C.28. Multiplying by $G$, the number of spguards, and factorizing the sum leads to the aforementioned bound. □

**Theorem C.2** (Bound on the total work using our algorithm)**.**

$$\mathbb{W} \;\leq\; \left(1 + \frac{DE\gamma \cdot (\tau + 2\phi)}{\kappa}\right) \cdot w \;+\; PFGH \cdot (\tau + 2\gamma \cdot \phi)$$

*Proof.* Obtained by summing the bound on $\mathbb{W}'$ obtained from Lemma C.31 and the bound on $\mathbb{W} - \mathbb{W}'$ obtained from Lemma C.32 with the fact that $2\gamma - 1 < 2\gamma$. □

**Theorem C.33** (Bound on the running time of $\gamma$-regular program)**.** *Consider the $\gamma$-regular program with well-defined spguards. Let $T_P$ be the running time of the program on a machine with $P$ processors and a greedy scheduler. Let $w$ and $s$ be the work and span of the program, correspondingly. $G$, $F$ and $H$ are as defined in C.16, C.17 and C.18. We have:*

$$T_P \;\leq\; \left(1 + \frac{\gamma ED \cdot (\tau + 2\phi)}{\kappa}\right) \cdot \frac{w}{P} \;+\; (1 + \phi + \max(\tau, E\beta\kappa)) \cdot s \;+\; FGH \cdot (\tau + 2\gamma\phi).$$

*Proof.* Combining Theorem C.1 about the total span and Theorem C.2 about the total work together with Brent's theorem [Brent 1974], we get our main theorem. □

**Theorem C.3** (Bound on the parallel run time). *Under two assumptions $\kappa \geq \tau$ and $\kappa \geq 1 + \phi$, the bound on the parallel running time from the previous theorem can be slightly simplified:*

$$T_P \;\leq\; \left(1 + \frac{\gamma ED \cdot (\tau + 2\phi)}{\kappa}\right) \cdot \frac{w}{P} \;+\; (E\beta + 1) \cdot \kappa \cdot s \;+\; O(G \cdot \log^2 \kappa \cdot (\tau + 2\gamma\phi)).$$

*Proof.* The proof is straightforward. At first, since $\kappa \geq \tau$, $\kappa \geq 1 + \phi$ and $E, \beta \geq 1$ we obtain: $(1 + \phi + \max(\tau, E\beta\kappa)) \leq (1 + E\beta) \cdot \kappa$. Secondly, $F = 1 + \log_\alpha \frac{\kappa}{DE}$ and $H = \log_{\gamma/(\gamma-1)} \frac{\kappa}{DE}$, thus $F \cdot H = O(\log^2 \kappa)$. □

**Theorem 2.1.** *For fixed hardware and any program, all parameters of the analysis except for $\kappa$ which represents the unit of parallelism can be replaced with constants, leaving us with the following bound:*

$$T_P \;\leq\; \left(1 + \frac{O(1)}{\kappa}\right) \frac{w}{P} \;+\; O(\kappa) \cdot s \;+\; O(\log^2 \kappa).$$

*Proof.* Immediate from the previous theorem. □

## D Input data

For *radix-sort*, we used a variety of inputs of $10^8$ items. The input *random kvp 256* consists of integer pairs $(k, v)$ such that $k \in [0, 2^{31})$ and $v \in [0, 256)$, and *random kvp* $10^8$ with $v \in [0, 10^8)$. The *exponential* input consists of 32-bit integers $\in [0, 2^{31})$ drawn from the exponential distribution.

For *comparison-sort*, we used a variety of inputs of $10^8$ items. The *random* input consists of 64-bit floats $\in [0, 1)$ from a uniform distribution, and *exponential* from an exponential distribution. The *almost sorted* input consists of a sorted sequence 64-bit floats $\in [0, 10^8)$ that is updated with $10^4$ random swaps. The *trigrams* input consists of strings generated using trigram distribution.

For *suffix-array*, we used three inputs. The *dna* input consists of a DNA sequence and has about 32 million charachters. The *etext* input consists of about 105 million characters drawn from Project Gutenberg. The *wikisamp* input consists of 100 million characters taken from wikipedia's xml source files.

For *convex-hull*, we used a variety of inputs of $10^8$ 2-d points. The *on circle* consists of points on the unit circle centered on the origin, and *kuzmin* consists of points from Kuzmin's distribution.

For *nearest neighbors*, we used a variety of inputs of $10^8$ 2-d and 3-d points. The input *kuzmin* consists of 2-d points drawn from the Kuzmin distribution. The input *plummer* consists of 3-d points drawn from the Plummer distribution.

For *ray-cast*, we used two non-synthetic inputs. The input *happy* consists of the Happy Buddha mesh from the Stanford 3D Scanning Repository, and it consists of 1087716 triangles. The input *xyz-rgb-manusript* comes from the same repository and consists of 4305818 triangles. For each of the mesh with $n$ triangles, $n$ rays were generated: the start of each ray is randomly drawn from the lowest side of the bounding box of the mesh and the end of each ray is randomly drawn from the upper side of the bounding box of the mesh.

For *delaunay*, we used two inputs consisting of $10^7$ 2-d points. The input *in square* consists of points in the unit square, and *kuzmin* consists of points drawn from the Kuzmin distribution.

# E Portability study

To test the portability of our approach, we ran the benchmarks on three additional machines:

- a 48-core AMD machine with four 12-core AMD Opteron 6172 processors, at 2.1GHz, with 64Kb of L1 and 512Kb L2 cache per core, 2x6Mb of L3 cache per processor, and 128Gb RAM in total ($\kappa = 12.2\mu$s, $\alpha = 1.4$), ;
- a 72-core Intel machine with four 18-core Intel Xeon E7-8867 chips, at 2.40GHz, with 32Kb of L1 and 256Kb L2 cache per core, 45Mb of L3 cache per chip, and 1Tb RAM in total ($\kappa = 8.2\mu$s, $\alpha = 1.4$); and
- a 12-core Intel machine with two 6-core Intel Xeon E5-2609 chips, at 1.9GHz, with 32Kb of L1 and 256Kb of L2 cache per core and 2x15Mb of L3 cache per chip, and 31Gb of RAM ($\kappa = 10.2\mu$s, $\alpha = 3.0$).

The results across these three machines are overall qualitatively similar to those of our main 40-core test harness.

| Application/input | Sequential elision | | 1-core execution | | 48-core execution | | | |
|---|---|---|---|---|---|---|---|---|
| | PBBS | Oracle | PBBS | Oracle | PBBS | Oracle | Oracle / PBBS | |
| | (s) | | (relative to elision) | | (s) | | Idle time | Nb threads |
| **samplesort** | | | | | | | | |
| random | 24.338 | +9.9% | +16.0% | -2.4% | 0.861 | -1.4% | -0.4% | -44.2% |
| exponential | 17.450 | +6.2% | +19.5% | -0.5% | 0.723 | -13.0% | -12.4% | -51.8% |
| almost sorted | 9.506 | +18.9% | +35.1% | +1.6% | 0.537 | +4.8% | +3.1% | +13.8% |
| **radixsort** | | | | | | | | |
| random | 4.927 | +4.4% | +0.6% | -4.0% | 0.328 | -0.6% | +0.1% | -16.9% |
| random pair | 8.193 | +0.6% | -0.8% | -1.1% | 0.711 | +2.1% | +2.1% | -0.3% |
| exponential | 4.916 | +3.8% | +1.8% | -3.5% | 0.332 | +1.9% | +0.7% | +7.1% |
| **suffixarray** | | | | | | | | |
| dna | 33.141 | -4.5% | -0.6% | +9.6% | 2.738 | -1.2% | +0.6% | -17.8% |
| etext | 127.117 | -1.3% | -0.9% | +5.5% | 7.495 | +0.3% | +0.7% | +2.3% |
| wikisamp | 102.287 | -2.1% | -0.7% | +6.6% | 6.757 | +0.6% | -0.9% | +23.1% |
| **convexhull** | | | | | | | | |
| kuzmin | 8.053 | +1.7% | -3.9% | +21.1% | 1.361 | +3.3% | +33.0% | -77.5% |
| on circle | 192.416 | +13.4% | +82.2% | +12.9% | 15.230 | -1.1% | -0.9% | -66.0% |
| **nearestneighbors** | | | | | | | | |
| kuzmin | 28.796 | +1.3% | +12.0% | +11.2% | 1.771 | -0.2% | +4.8% | -13.7% |
| plummer | 44.455 | -0.7% | +3.2% | +1.4% | 5.602 | -5.1% | +0.2% | -11.0% |
| **delaunay** | | | | | | | | |
| in square | 103.483 | -0.6% | -3.7% | +1.7% | 4.844 | -3.0% | -1.5% | -22.8% |
| kuzmin | 111.264 | +1.0% | +0.4% | +2.9% | 6.247 | -5.8% | +2.9% | -31.9% |
| **raycast** | | | | | | | | |
| happy | 18.379 | +5.3% | +2.1% | +3.1% | 0.747 | +4.8% | +6.9% | -45.3% |
| xyzrgb | 469.700 | +1.0% | +0.6% | +1.3% | 14.751 | +0.3% | +0.9% | -59.4% |

**Table 3.** Results from PBBS benchmarks, executed on the 48-core AMD machine.

| Application/input | Sequential elision | | 1-core execution | | 72-core execution | | | |
|---|---|---|---|---|---|---|---|---|
| | PBBS | Oracle | PBBS | Oracle | PBBS | Oracle | Oracle / PBBS | |
| | (s) | | (relative to elision) | | (s) | | Idle time | Nb threads |
| **samplesort** | | | | | | | | |
| random | 14.290 | +5.7% | +20.0% | +1.2% | 0.422 | -8.6% | -8.7% | -15.5% |
| exponential | 10.347 | +4.1% | +18.6% | +0.9% | 0.320 | -8.9% | -9.1% | -19.4% |
| almost sorted | 3.915 | +20.0% | +31.3% | -3.6% | 0.179 | -3.7% | -4.5% | -12.4% |
| **radixsort** | | | | | | | | |
| random | 2.149 | +1.5% | +1.0% | -7.8% | 0.107 | -3.3% | -3.6% | -15.3% |
| random pair | 3.428 | +1.2% | +0.2% | -8.6% | 0.209 | -5.3% | -5.7% | -7.6% |
| exponential | 2.148 | +3.2% | +1.4% | -6.6% | 0.102 | +4.6% | +3.8% | +7.1% |
| **suffixarray** | | | | | | | | |
| dna | 13.471 | +1.5% | +1.3% | +5.0% | 0.609 | -1.0% | -2.0% | +7.0% |
| etext | 51.637 | +1.1% | +0.7% | +3.6% | 2.008 | +3.3% | +2.1% | +27.8% |
| wikisamp | 45.232 | +0.9% | +0.8% | +3.3% | 1.886 | +0.2% | -1.0% | +30.5% |
| **convexhull** | | | | | | | | |
| kuzmin | 3.074 | -0.2% | +1.5% | +9.8% | 0.280 | -16.5% | -12.0% | -74.0% |
| on circle | 74.340 | -3.3% | +128.8% | +16.6% | 5.206 | -41.6% | -41.8% | -29.1% |
| **nearestneighbors** | | | | | | | | |
| kuzmin | 11.100 | -1.9% | +9.4% | +4.2% | 0.541 | +9.5% | +11.0% | -10.9% |
| plummer | 14.267 | -1.3% | +9.1% | +2.9% | 1.302 | +7.6% | +8.2% | -13.1% |
| **delaunay** | | | | | | | | |
| in square | 45.347 | -0.3% | -0.2% | -0.0% | 1.702 | +0.8% | +3.5% | -16.9% |
| kuzmin | 49.418 | -0.6% | -0.1% | +1.1% | 2.130 | -4.6% | +3.4% | -26.5% |
| **raycast** | | | | | | | | |
| happy | 6.304 | +5.7% | +2.7% | +0.4% | 0.226 | +3.8% | +6.8% | -56.8% |
| xyzrgb | 190.822 | +0.2% | -0.1% | +0.2% | 3.954 | -1.3% | -0.3% | -70.5% |

**Table 4.** Results from PBBS benchmarks, executed on the 72-core Intel machine.

| Application/input | Sequential elision | | 1-core execution | | 12-core execution | | | |
|---|---|---|---|---|---|---|---|---|
| | PBBS | Oracle | PBBS | Oracle | PBBS | Oracle | Oracle / PBBS | |
| | (s) | | (relative to elision) | | (s) | | Idle time | Nb threads |
| **samplesort** | | | | | | | | |
| random | 22.929 | +5.3% | +19.8% | +0.2% | 2.473 | -11.3% | -11.2% | -51.7% |
| exponential | 16.414 | +4.0% | +18.6% | -0.0% | 1.803 | -12.3% | -12.2% | -69.0% |
| almost sorted | 6.069 | +21.4% | +31.3% | -1.2% | 0.869 | -9.7% | -9.5% | -43.0% |
| **radixsort** | | | | | | | | |
| random | 3.130 | +0.1% | +0.2% | -3.0% | 0.482 | -6.1% | -6.1% | -57.4% |
| random pair | 5.085 | +0.5% | -0.6% | -6.3% | 1.040 | -6.2% | -6.4% | -4.6% |
| exponential | 3.346 | +0.6% | +0.0% | -2.9% | 0.435 | -0.1% | +0.0% | -18.9% |
| **suffixarray** | | | | | | | | |
| dna | 21.592 | +19.1% | +1.1% | -13.4% | 3.437 | +1.2% | +0.4% | +29.8% |
| etext | 82.698 | +1.4% | +0.3% | +2.0% | 11.370 | +0.5% | +0.0% | +26.9% |
| wikisamp | 67.672 | +0.2% | +1.1% | +2.5% | 10.540 | -0.3% | -0.7% | +34.2% |
| **convexhull** | | | | | | | | |
| kuzmin | 5.185 | -1.0% | -4.0% | +3.6% | 0.813 | +9.3% | +14.1% | -89.4% |
| on circle | 172.796 | -3.4% | +55.3% | +4.4% | 26.637 | -23.5% | -23.5% | -66.9% |
| **nearestneighbors** | | | | | | | | |
| kuzmin | 18.145 | +7.0% | +14.3% | +3.1% | 2.313 | -2.4% | -3.1% | +10.5% |
| plummer | 25.570 | +3.5% | +9.5% | +2.8% | 3.301 | +0.3% | -1.5% | +11.7% |
| **delaunay** | | | | | | | | |
| in square | 75.902 | +0.9% | -0.4% | -0.1% | 8.148 | -0.2% | +3.1% | -53.5% |
| kuzmin | 82.553 | +0.3% | +0.9% | +1.3% | 8.985 | -1.1% | +2.5% | -45.9% |
| **raycast** | | | | | | | | |
| happy | 13.574 | +3.7% | +1.1% | +6.8% | 1.303 | +3.3% | +3.5% | -38.7% |
| xyzrgb | 350.074 | +2.5% | +0.4% | +9.2% | 32.790 | +1.0% | +1.2% | -86.3% |

**Table 5.** Results from PBBS benchmarks, executed on the 12-core Intel machine.

| Graph | Flat | | | | Nested | | | | Ours nested vs. PBBS flat |
|---|---|---|---|---|---|---|---|---|---|
| | PBBS | Ours | Oracle / PBBS | | PBBS | Ours | Oracle / PBBS | | |
| | (sec.) | | Idle time | Nb threads | (sec.) | | Idle time | Nb threads | |
| livejournal | 0.25 | +31.1% | −0.3% | +4.3% | 0.32 | +5.2% | −7.6% | −8.0% | +35.1% |
| twitter | 5.94 | −10.5% | +7.1% | −15.6% | 5.67 | −10.0% | +1.2% | −68.5% | −14.2% |
| wikipedia | 0.21 | +42.6% | −0.3% | −12.5% | 0.25 | +14.7% | −7.2% | −25.9% | +42.0% |
| europe | 6.27 | +0.7% | −28.6% | +19.2% | 6.21 | +0.4% | −28.6% | +19.0% | −0.5% |
| rand-arity-100 | 0.21 | +44.9% | −5.8% | +1.1% | 0.54 | −39.1% | −28.5% | −5.3% | +55.0% |
| rmat27 | 0.50 | +22.2% | +4.4% | −20.7% | 0.64 | −10.6% | −1.5% | −40.7% | +15.5% |
| rmat24 | 0.70 | +16.2% | −2.7% | +3.0% | 0.70 | +20.9% | −3.2% | +14.3% | +19.7% |
| cube-grid | 1.38 | +11.9% | +0.5% | +15.7% | 1.37 | +14.6% | −1.5% | +15.3% | +13.5% |
| square-grid | 6.13 | −8.6% | −6.2% | −2.0% | 6.17 | −7.3% | −9.7% | +0.2% | −6.6% |
| par-chains-100 | 74.9 | −59.7% | −46.3% | −55.0% | 74.9 | −59.6% | −46.8% | −55.0% | −59.6% |
| trunk-first | 10.5 | +0.0% | +22.2% | +0.3% | 10.5 | +0.0% | +24.5% | −0.5% | +0.4% |
| phases-10-d-2 | 3.07 | +8.5% | −5.2% | +3.1% | 1.14 | −5.6% | −0.1% | −45.0% | −64.8% |
| phases-50-d-5 | 1.57 | +15.2% | +1.8% | −19.5% | 1.41 | +14.5% | −1.4% | −23.3% | +3.3% |
| trees-524k | 73.6 | +6.0% | +4.2% | +5.0% | 6.30 | −1.2% | +0.4% | −14.6% | −91.6% |

**Table 6.** Results from the BFS experiment, executed on the 48-core AMD machine.

| | Flat | | | | Nested | | | | Ours nested vs. PBBS flat |
|---|---|---|---|---|---|---|---|---|---|
| Graph | PBBS | Ours | Oracle / PBBS | | PBBS | Ours | Oracle / PBBS | | |
| | (sec.) | | Idle time | Nb threads | (sec.) | | Idle time | Nb threads | |
| livejournal | 0.06 | +49.2% | −6.3% | −3.3% | 0.12 | −36.0% | +1.2% | −71.5% | +17.3% |
| twitter | 1.19 | −19.6% | +30.7% | −15.2% | 1.51 | −42.3% | −0.8% | −68.2% | −27.0% |
| wikipedia | 0.06 | +4.6% | +4.9% | −45.9% | 0.10 | −41.8% | +1.6% | −68.7% | +2.9% |
| europe | 3.11 | −42.2% | −54.0% | −32.0% | 3.09 | −41.9% | −54.3% | −33.5% | −42.3% |
| rand-arity-100 | 0.06 | +14.2% | −4.7% | +1.9% | 0.22 | −68.0% | −22.4% | −42.7% | +8.7% |
| rmat27 | 0.11 | +12.4% | −1.7% | −57.8% | 0.22 | −46.0% | −7.4% | −69.9% | +12.3% |
| rmat24 | 0.16 | +9.4% | −5.1% | −20.8% | 0.18 | +0.7% | −3.9% | −39.7% | +9.6% |
| cube-grid | 0.49 | −6.7% | −13.3% | +2.2% | 0.51 | −11.0% | −13.7% | +3.3% | −7.2% |
| square-grid | 2.98 | −40.4% | −44.2% | −31.2% | 3.12 | −42.4% | −43.5% | −31.0% | −39.5% |
| par-chains-100 | 49.5 | −86.3% | −72.3% | −94.2% | 50.1 | −85.2% | −71.4% | −91.2% | −85.1% |
| trunk-first | 7.46 | −15.3% | −0.3% | −39.6% | 7.89 | −20.6% | −0.8% | −39.7% | −16.0% |
| phases-10-d-2 | 0.99 | +9.8% | +6.0% | −3.3% | 0.20 | +2.7% | −2.6% | −24.0% | −79.4% |
| phases-50-d-5 | 0.30 | +22.8% | −5.4% | −1.6% | 0.30 | +7.6% | −6.4% | +7.4% | +11.0% |
| trees-524k | 12.9 | +7.1% | +7.1% | +4.2% | 0.93 | +9.0% | −4.4% | +23.9% | −92.2% |

**Table 7.** Results from the BFS experiment, executed on the 72-core Intel machine.

| | Flat | | | | Nested | | | | Ours nested vs. PBBS flat |
|---|---|---|---|---|---|---|---|---|---|
| Graph | PBBS | Ours | Oracle / PBBS | | PBBS | Ours | Oracle / PBBS | | |
| | (sec.) | | Idle time | Nb threads | (sec.) | | Idle time | Nb threads | |
| livejournal | 0.22 | +6.1% | −1.9% | −28.6% | 0.29 | −19.6% | −0.2% | −68.2% | +7.5% |
| twitter | 4.24 | +16.3% | +0.7% | −3.1% | 5.49 | −9.6% | −0.1% | −7.6% | +17.0% |
| wikipedia | 0.15 | +3.9% | +0.6% | −59.4% | 0.20 | −21.1% | +1.2% | −71.3% | +3.8% |
| europe | 3.35 | −4.2% | −16.3% | +15.3% | 3.36 | −4.4% | −16.3% | +14.7% | −4.2% |
| rand-arity-100 | 0.16 | +8.8% | −2.2% | +17.2% | 0.72 | −75.6% | −3.3% | −58.9% | +8.5% |
| rmat27 | 0.53 | +3.9% | +1.8% | −61.8% | 0.80 | −29.9% | +0.6% | −66.5% | +5.1% |
| rmat24 | 0.93 | +4.4% | +0.1% | −36.9% | 0.94 | +3.9% | −0.1% | −31.5% | +5.4% |
| cube-grid | 2.02 | −0.4% | +1.5% | −11.4% | 2.02 | −0.0% | +1.4% | −13.1% | −0.4% |
| square-grid | 4.76 | −10.3% | +5.4% | −15.5% | 4.75 | −10.4% | +5.7% | −15.8% | −10.5% |
| par-chains-100 | 37.8 | −47.2% | −41.4% | −36.8% | 37.9 | −47.2% | −41.4% | −36.6% | −47.1% |
| trunk-first | 10.8 | −33.4% | +0.1% | −90.2% | 10.6 | −32.0% | +0.1% | −88.3% | −33.5% |
| phases-10-d-2 | 2.19 | +7.0% | −1.0% | +0.5% | 1.58 | +9.9% | +0.7% | −70.2% | −20.9% |
| phases-50-d-5 | 2.20 | +21.4% | +0.3% | −14.6% | 2.43 | +8.7% | −0.0% | −34.8% | +20.2% |
| trees-524k | 13.2 | +4.7% | −1.4% | +5.3% | 3.90 | +4.3% | +2.1% | −27.5% | −69.2% |

**Table 8.** Results from the BFS experiment, executed on the 12-core Intel machine.

## F Other PBBS benchmarks

- The *n-body* and *delaunay-refine* benchmarks have critical code regions that do not readily admit a cost function.
- The *remove-duplicates* and *dictionary* benchmarks shows high variability in the running time of their critical loops, the cause of which is the heavy use of atomic operations that, in turn, triggers massive bus contention.
- The deterministic-reservation benchmarks consist of *maximal independent set*, *spanning forest*, and *minimal spanning tree*. These benchmarks make heavy use of atomic compare-and-exchange operations on a shared array, and as such show high variablility in their execution times. Even so, we found through experimentation that minimal spanning tree and spanning forest performed reasonably well on our test machines.

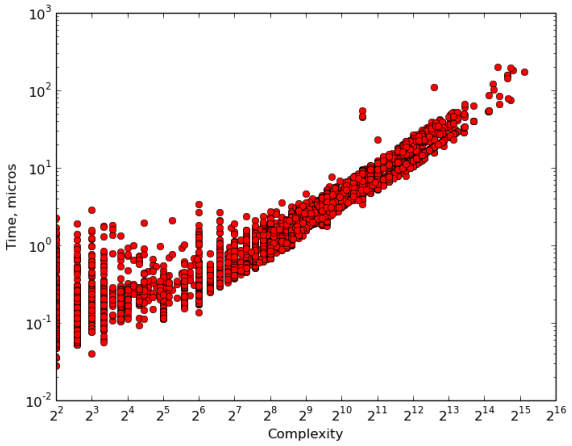


**Figure 5.** Example taken from a collection of measured runs for a given spguard. The x-axis corresponds to the asymptotic complexity of a sequentialized computation, while the y-axis corresponds to the sequential execution time measured for that computation. The value of $E$ corresponds to the maximal (vertical) deviation from the line going through the mean of the execution times. The ratio $\beta/\alpha$ corresponds to the maximal slope of that line.

## G Estimating $E$ and $\beta$

Estimating $E$ and $\beta$ on the target hardware enables one to estimate the worst-case bounds predicted by our theoretical analysis, either on the overheads or on the increase in the span. To estimate these two parameters, we generate, for each spguard from each of our benchmark programs, a plot summarizing the distribution of the execution time of the sequentialized computation. One representative plot is shown in Figure 5. The x-axis indicates the asymptotic complexity of computations sequentialized by the spguard considered. The y-axis corresponds to the measure of the sequential execution time. Note that both axes are on a logarithmic scale. In particular, moving by one unit to the right on the x-axis corresponds to doubling the input size. To evaluate $E$ and $\beta$, we draw an interpolation line going through the mean of the measured execution times. Then, the maximal vertical distance between the line and a point gives the value of $E$, and the maximal slope of that line gives bounds the value of the ratio $\beta/\alpha$. Multiplying this ratio by the value set for $\alpha$ yields the value of $\beta$. To evaluate the worst-case bound from the theoretical analysis for a given benchmark program, we instantiate $E$ and $\beta$ with the maximal values observed for all spguards involved in that program.

| Type T | Size | grain | Time | Comment |
|---|---|---|---|---|
| char | 800M | 1 | 1.963 | 100x slower |
| | | 10 | 0.330 | 17x slower |
| | | 5000 (TBB-rec.) | 0.020 | optimal |
| | | auto (ours) | 0.020 | optimal |
| char[64] | 200M | 1 | 0.129 | 78% slower |
| | | 10 (TBB-rec.) | 0.077 | 6% slower |
| | | 5000 | 0.072 | optimal |
| | | auto (ours) | 0.073 | optimal |
| char[2048] | 0.4M | 1 (TBB-rec.) | 0.049 | optimal |
| | | 10 | 0.050 | optimal |
| | | 5000 | 0.057 | 16% slower |
| | | auto (ours) | 0.050 | optimal |
| char[131072] | 0.01M | 1 (TBB-rec.) | 0.075 | optimal |
| | | 10 | 0.075 | optimal |
| | | 5000 | 1.419 | 19x slower |
| | | auto (ours) | 0.075 | optimal |

**Table 9.** Running times on 40 cores for inputs of various sizes, for manually-fixed grain sizes, including the one obtained following Intel's TBB manual, and for our algorithm.

# H The granularity-control problem and our solution, by example

We present an overview of the challenges of the granularity problem and a detailed description of our proposed solution. As an example, we use an instance of the generic `map` function from the introduction and make it concrete by using our implementation.

## H.1 Example program

Consider a simple, data-mining problem: given an array of elements of type `T`, find the number of elements that satisfy a given predicate `p`. Using C++ templates, we specify such a generic function as follows:

```
template<T,P>
int match(T* lo, T* hi, P p) { ... }
```

Because `match` is generic, we can set `T` to be `char`, and define the predicate to be a function (a C++ lambda function [Stroustrup 2013, Sec.11.4]) that tests equality of the character to '#' as follows.

```
p = [&] (char* c) { return *c == '#' }
```

Similarly, we can perform matches over arrays whose elements are 1024-character strings, by setting `T` to be `char[1024]`. For example, we may count the strings whose hash code matches a particular value, say `2017`, by instantiating the predicate as follows.

```
p = [&] (T* x) { return hash(x, x+sizeof(T)) == 2017 }
```

A classic way to implement a parallel `match` is to divide the input array into two halves, recur on each half, and compute the sum of the number of occurrences from both halves. Figure 6 shows the code for such an implementation, where `match` takes as arguments the input string, specified by `lo` and `hi`, and a predicate function, `p`. To control granularity, we stop the recursion when the input contains fewer than `grain` elements and switch to a fast sequential algorithm, `match_seq`. When the input string is large, it is divided in half and solved recursively in parallel using `fork2join`.

## H.2 Choice of the grain size

To ensure good performance, the programmer must choose the setting for `grain`; but what should this setting be? The challenge is that there is no a priori suitable setting for the `grain`, because the suitability depends on the particular hardware and software environment and, in fact, on the inputs to the function `match`. Applying the established practice of manual granularity control leads to poor results even on the same machine and with the same software environment.

Table 9 illustrates the issue. It shows the 40-core run times for different types `T` and different `grain` settings. (The experiment is run on an Intel machine described in Section 3.) We picked the input size (total number of characters) to ensure a sequential execution time of a few seconds. When `T` is the type `char`, we use character equality as predicate. When `T` is an array of characters, we compare the hash values, for a simple hashing function. For each setting of `T`, we consider various values of `grain`, including the "recommended" grain value determined by following the process described in Intel's TBB manual [Intel 2011]: start by setting the `grain` to the value 10,000 and halve it until the 1-processor run-time stops decreasing by more than 10%. Such tuning maximizes the exposed parallelism by considering the smallest grain value for which the overheads are not prohibitive.

```
int grain = ... // determined by tuning
template <T, P>
int match(T* lo, T* hi, P p)
  int result
  int n = hi - lo
    if n ≤ grain
      result = match_seq(lo, hi, p)
    else
      T* mid = lo + (n / 2)
      int result1, result2
      fork2join([&] {
        result1 = match(lo, mid, p)
      }, [&] {
        result2 = match(mid, hi, p)
      })
      result = result1 + result2
  return result
```

**Figure 6.** Match, with manual granularity control.

```
// no grain size needed
template <T, P>
int match(T* lo, T* hi, P p)
  int result
  int n = hi - lo
  spguard([&] { // complexity function
    return n
  }, [&] { // parallel body
    if n ≤ 1
      result = match_seq(lo, hi, p)
    else
      T* mid = lo + (n / 2)
      int result1, result2
      fork2join([&] {
        result1 = match(lo, mid, p)
      }, [&] {
        result2 = match(mid, hi, p)
      })
      result = result1 + result2
  }, [&] { // sequential body
    result = match_seq(lo, hi, p)
  })
  return result
```

**Figure 7.** Match, with automatic granularity control.

First, observe that the TBB-recommended value of `grain` changes for different settings of `T`. For `char`, it is `5000`; for `char[64]`, it is `10`; for `char[2048]`, it is `1`. Second, observe that a grain optimal in one setting may induce a very significant slowdown in a different setting. For example, when `T` is `char`, setting the grain to `1` instead of `5000` results in a 100-fold slowdown. We thus conclude that, there is no single value of `grain` that works well for all instances of `match`.

One might attempt to select the `grain` size based on the arguments provided to the `match` function. For example, the `grain` could be set to `C/sizeof(T)`, for some constant `C`, to ensure use of a smaller grain size when processing bigger elements. This approach helps in some cases, but it does not solve the problem in general: note that in `match`, the grain depends not only on the type `T`, but also on the predicate passed as second argument. If `T` is set to `char[64]` and the predicate is instantiated as a simple hash function, the optimal grain size is `10`; however, providing a different, more computationally expensive predicate function causes the optimal grain size to be `1`. Selecting the right `grain` for different predicates would require the ability to predict the run-time behavior of a function, an intractable problem. To control granularity in cases of nested-parallel programs, the programmer will likely have to specialize the code for each predicate and apply granularity control to each such specialization, thus losing the key benefits of generic functions.

The problems illustrated by the simple example above are neither carefully chosen ones nor isolated cases. They are common; more realistic benchmarks exhibit even more complex behavior. For example, arguments to parallel functions can be parallel, leading to nested parallelism, making granularity control more difficult. In fact, as discussed in more detail in Section 3, in the state-of-the-art PBBS benchmarking [Blelloch et al. 2012], nearly every benchmark relies on carefully written, custom granularity control techniques.

### H.3 Our approach

Our goal is to delegate the task of granularity control to a smart, library implementation. To this end, we ask the programmer to provide for each parallel function a *series-parallel guard*, by using the keyword `spguard`. A spguard consists of: a *parallel body*, which is a lambda function that performs a programmer-specified parallel computation; a *sequential body*, which is a lambda function that performs a purely sequential computation equivalent to the parallel body, i.e. performing the same side-effects and delivering the same result. a *cost function*, which gives an abstract measure, as a positive number, of the work (run-time cost) that would be performed by the sequential body.

At a high level, a spguard exploits the result of the cost function to determine whether the computation involved is small enough to be executed sequentially, i.e. without attempting to spawn any subcomputation. If so, the spguard executes the sequential body. Otherwise, it executes the parallel body, which would typically spawn smaller subcomputations, each of them being similarly guarded by a spguard.

The cost function may be any programmer-specified piece of code that, given the context, computes a value in proportion to the one-processor execution time of the sequential body. Typically, the cost function depends on the arguments provided to the current function call. A good choice for the cost function is the average asymptotic complexity of the sequential body, e.g., $n \lg n$, or $n$, or $\sqrt{n}$, where $n$ denotes the size of the input. The programmer need not worry about constant factors because spguards are able to infer them on-line, with sufficient accuracy.

```
template <T, P>
int match(T* lo, T* hi, P p)
  return map_reduce(lo, hi, 0, [&] (int x, int y) {
    return x + y // associative combining operator
  }, [&] (T* i) {
    if p(*i) return 1 else return 0 // leaf–level operation
  })
```

**Figure 8.** Match, using `map_reduce` with our automatic granularity control.

In a real implementation, the sequential body can be left implicit in many cases, because it can be inferred automatically. For example, the sequential body for a parallel-for loop can be obtained by replacing the parallel-for primitive with a sequential for. Likewise, in many instances, the complexity function is linear, allowing us to set it to the default when not specified. In our library and experiments, we use this approach to reduce dramatically the annotations needed.

Figure 7 shows the code for our example `match` function using spguards. Compared with the original code from Figure 6, the only difference is the code being structured as a spguard with three arguments: cost function, parallel body, and sequential body. There, the cost function simply returns the input size, written `n`, because the sequential body (`match_seq`) uses a linear-time, sequential matching algorithm.

As we show in this paper, once a parallel algorithm is modified with the insertion of spguards like in Figure 7, the information provided by the cost function suffices for our run-time system to control granularity effectively for all settings of the parameters. As shown in Table 9, our oracle-guided version matches the performance achieved by the `grain` settings recommended by the TBB method, but without any of the manual tuning effort and code modifications.

In Section 2.6, we described the higher-level interface provided by our implementation. The code below shows how we can use the higher-level interface provided by our library to implement a more concise version of the `match` function in Figure 7. The solution in Figure 8 uses a call to the `map_reduce` provided by our sequence library. The `map_reduce` implementation uses the default sequential body and default cost function, but can optionally be called with a custom sequential body or custom complexity function.

Having the ability to write such high-level code in our approach turned out to be a crucial piece of the implementation. It allowed us to write code that is almost always as concise as original PBBS code, is sometimes even more concise, and offers automatic granularity control. Thanks to the so-called zero-cost abstraction features of C++ (e.g., cheap higher order functions) codes, such as the code in Figure 8, perform as well as the lower-level counter parts, such as the code shown in Figure 7.