Poster: Performance Challenges in Modular Parallel Programs

Umut A. Acar
Carnegie Mellon University, USA
Inria, France

Vitaly Aksenov
Inria, France
ITMO University, Russia

Arthur Charguéraud
Inria, France
Université de Strasbourg, CNRS, ICube, France

Mike Rainey
Inria, France

Abstract

Over the past decade, many programming languages and systems for parallel-computing have been developed, including Cilk, Fork/Join Java, Habanero Java, Parallel Haskell, Parallel ML, and X10. Although these systems raise the level of abstraction at which parallel code are written, performance continues to require the programmer to perform extensive optimizations and tuning, often by taking various architectural details into account. One such key optimization is granularity control, which requires the programmer to determine when and how parallel tasks should be sequentialized.

In this paper, we briefly describe some of the challenges associated with automatic granularity control when trying to achieve portable performance for parallel programs with arbitrary nesting of parallel constructs. We consider a result from the functional-programming community, whose starting point is to consider an "oracle" that can predict the work of parallel codes, and thereby control granularity. We discuss the challenges in implementing such an oracle and proving that it has the desired theoretical properties under the nested-parallel programming model.

Context 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*, a.k.a. *implicit parallelism*, which 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. Implementations include: OpenMP, Cilk, TBB, X10, parallel ML. Keywords, such as Cilk's **spawn** and **sync**, suffice to express many common parallel patterns, including parallel loops and *nested parallel* computations, wherein parallel computations may themselves start and synchronize with other parallel computations.

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 [3]. 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 alternative is executed for small (and only for small) computations. For example, consecutive iterations of a parallel loop can be "bunched" into sequential blocks. The number of iterations bunched together is called the *cut-off* or the *grain* size. Crucially, the grain size needs to be large enough to amortize the cost of parallelism, but small enough to enable the creation of sufficiently many parallel tasks.

Selecting the grain size is challenging because: (1) the optimal grain size depends on the hardware; (2) the optimal grain size depends on the instantiation of the code, in the case of modular (templated) code; (3) for a fixed program, the optimal grain size may depend on the input data, e.g., when the processing time varies for each item; (4) in the case of nesting, the grain sizes are interdependent, e.g., the grain size of an outer loop depends on the work load involved in the inner loop, which might itself be data dependent.

Related work Intel's TBB manual [3] describes the following process for determining the grain size: 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. Duran et al. [2] 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 an online heuristic. Another approach uses the height and depth of the recursion tree [6, 8] 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, basing

PPoPP '18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-4982-6/18/02…$15.00
https://doi.org/10.1145/3178487.3178516

decisions solely on such dynamically collected data may end up decreasing parallelism adversely [4]. Iwasaki et al propose a technique for synthesizing static cut-offs for divide-and-conquer functions in Cilk-style programs [4]. They rely on static analysis to determine whether or not to switch to sequential execution. Yet, static analysis cannot cope well with complex code. Lazy task creation (or, sometimes, lazy scheduling) [5, 7] is a strategy to circumvent the problem of selecting a grain size. Processors observe the load in the system and create tasks only when necessary, by splitting their currently-running task. Lazy task creation has proved to be an indispensable technique for reducing overheads of parallelism in many systems. Yet, it does not provide support for deciding that a subtask is small enough that it may be executed, without harming parallelism, using a sequential algorithm more efficient than its parallel counterpart.

Algorithmic granularity control Prior work [1] proves that, if it is possible to approximately predict the sequential execution time associated with every computation involved in the execution of a parallel program, then there exists a scheduling strategy that ensures bounded parallelism overheads, while preserving the asymptotic amount of parallelism being exposed. The proof of this theorem, which applies to a relatively large class of parallel programs, is carried out in the work-span computation model, which is well suited for analyzing nested parallel programs.

In this work, we investigate the possibility to implement execution-time predictions in practice, by combining asymptotic cost functions provided by the programmer with runtime measurements. There are three key challenges. First, predictions need to be sufficiently precise to benefit from the bounds associated with the aforementioned theorem. Second, the algorithm needs to be robust to the large variations in the execution times typical for modern hardware (constant factors increase noticeably when the data stops fitting into the cache). Third, the algorithm needs to infer the constant factors *online* during the execution of the parallel program. Doing so involves a circularity problem, illustrated next.

```
let f(x) =
  spguard(fun () → c(x), // cost function
          fun () →       // parallel body
             if |x| = 1 then .. else
                let (x1,x2) = divide(x)
                spawn let r1 = f(x1)
                      let r2 = f(x2) sync
                conquer(r1,r2),
          fun () → g(x)) // sequential body
```

The body of the above function involves a *spguard*, which is a combinator expecting 3 arguments. First, `c(x)` describes the asymptotic cost function. Second comes the parallel body. It implements a divide-and-conquer approach; splitting the input data in halves, making two recursive calls in parallel, then combining the output results. Third, `g(x)` describes the sequential body.

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 some constant threshold κ, then `g(x)` should be used. Under a small number of assumptions, this policy leads to provably efficient granularity control.

The circularity problem is the following. To decide which computations are safe to execute using the sequential body, `g(·)`, our algorithm needs to first know the constant factor that applies. Indeed, without an accurate estimate of the constant, the algorithm might end up invoking `g(·)` on a large input and potentially destroy all available parallelism. Yet, in order to estimate the constant factor, the algorithm needs to measure the execution time of a call to `g(·)`.

To resolve this critical circular dependency, we designed an algorithm that progressively sequentializes larger and larger computations. It begins by sequentializing only the base case, and ultimately converges to computations of duration close to κ. Each call to `g(·)` leads to a time measurement, which may be subsequently used to predict that another, slightly larger input may be processed sequentially. We increase input sizes progressively, in order to always remain on the safe side, making sure that our algorithm never executes sequentially a computation significantly longer than κ.

We prove, under a few assumptions typically met by many efficient parallel programs, end-to-end bounds on the execution time, including the overheads of parallelism and the cost of the aforementioned convergence phase. Benchmarks suggest that our automatic granularity control approach enables replacing careful selection of techniques with a single, uniform technique for controlling granularity, that automatically adapts to the hardware, i.e., the technique is portable.

Acknowledgments This research is partially supported by the National Science Foundation (CCF-1408940 and CCF-1629444), European Research Council (ERC-2012-StG-308246).

References

[1] U. A. Acar, A. Charguéraud, and M. Rainey. 2016. Oracle-guided scheduling for controlling granularity in implicitly parallel languages. *JFP* 26 (2016).

[2] A. Duran, J. Corbalan, and E. Ayguade. 2008. An adaptive cut-off for task parallelism. In *SC*. 1–11.

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

[4] S. Iwasaki and K. Taura. 2016. A static cut-off for task parallel programs. In *PACT*. 139–150.

[5] E. Mohr, D. A. Kranz, and R. H. Halstead. 1991. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE TPDS* 2, 3 (1991), 264–280.

[6] J. Pehoushek and J. Weening. 1990. Low-cost process creation and dynamic partitioning in Qlisp. In *LNCS*. Vol. 441. 182–199.

[7] A. Tzannes, G. C. Caragea, U. Vishkin, and R. Barua. 2014. Lazy Scheduling: A Runtime Adaptive Scheduler for Declarative Parallelism. *ACM TOPLAS* 36, 3 (2014), 10:1–10:51.

[8] J. S. Weening. 1989. *Parallel Execution of Lisp Programs*. Ph.D. Dissertation.