Analysis of Overheads in Parallel Adaptive Algorithms

Stefano Mor, Nicolas Maillard
GPPD/UFRGS
{Stefano.Mor, Nicolas}@inf.ufrgs.br

Jean-Louis Roch
Laboratoire d'Informatique de Grenoble
{Jean-Louis.Roch}@imag.fr

Abstract

This paper shows new upper-bounds for overheads on adaptive algorithms (parallel algorithms that select different task implementations accordingly with the input and independently from the underlying hardware). The targets are shared memory parallel computations where parallel tasks are scheduled by work stealing with random victim selection. New bounds are more tight than previous ones and improve current state-of-the-art by being independent from the work and depth of a parallel execution. The main result is that the number of successful steals (which introduce overhead in adaptive computations) of independent parallel task sets is lesser in expectation than $2.2(P-1)$*, where* P *is the number of workers (*e.g.*, processors, nodes, threads,* etc.*). Experiments with adaptive polynomial evaluation by Horner's Scheme match the theory.*

1. Introduction

We start the discussion by defining adaptive algorithms. The base is the taxonomy for parallel algorithms stated by Cung *et al.* [4].

Foremost, *Hybrid* algorithms are informally defined:

> "An algorithm is hybrid when there is a choice at a high level between at least two distinct algorithms, each of which could solve the same problem."

Hence, *Adaptive* algorithms are informally defined in terms of hybrid algorithms:

> "A hybrid algorithm is adaptive if it avoids any machine or memory-specific parameterization. Strategic decisions are made based on resource availability or input data properties, both discovered at runtime (such as idle processors)."

Motivation. Adaptive algorithms are hardware oblivious. Its efficiency is ubiquitous whenever the underlying hardware follows the designed abstraction.

Problem and Proposed Solution. Traore *et al.* efficiently parallelizes several STL algorithms using random victim work stealing in an adaptive fashion [9], where the parallel overhead is "paid" only at each successful steal, but tight upper bounds are hard to obtain in a general fashion. This paper presents an analysis over general adaptive algorithms and provides new upper-bounds for its overhead through a new mechanism named *overhead counters*.

Tool . The analysis is general. We apply it to the Cilk Plus parallel framework to guide and illustrate the exposed principles. Cilk Plus implements the runtime described by Frigo *et al.* [6]. Coding in Cilk Plus is simple, yet the framework has solid theoretical foundations and optimal performance [3]. The framework schedules threads over multi-core processors with shared memory communication.

Roadmap. The setting – the underlying runtime and sandbox example (Horner's Scheme) – is reviewed at Section 2. Definitions for overhead counters, the upper-bounds over them and its proof sketches are found on Section 3. A performance evaluation and the comparison of the results with theory is at Section 4. Conclusions over the paper with related works and planned future contributions lie on Section 5.

2. Setting

A polynomial evaluation by Horner's Scheme is the referential parallel algorithm over which the general theory is presented and evaluated. Before describing the algorithm itself, we review the underlying parallel model.

2.1. Underlying Model

We examine adaptive algorithms within the context of parallel tasks model with the following informal invariants:

- Ready tasks are tasks whose all sequential dependencies with other tasks are satisfied. A task is written/described as a procedure, dependencies are the sequential constraints between these procedures.
- Workers are entities that execute a ready task (*e.g.*, a processor, a process, a thread, *etc.*).

- States are discrete, totally ordered time stamps regarding the execution of a parallel program. Current state is denoted by s, previous state by $s-$ and next by $s+$. The state before execution is 0 and first state of execution is 1.
- The dynamic scheduler decides which worker executes which ready task at a given state s.
- Ready tasks are executed with either best sequential code or parallel code (hybrid), based on the actions performed by the dynamic scheduler (adaptive).

The following description by Arora *et al.* [1] express the scheduler in terms of runtime actions performed in response to specific operations that may be performed by ready tasks:

1. Task α Spawns task β. In this case the worker pushes α on (the bottom of) the ready deque, and starts work on task β.
2. Task α Returns. There are two cases:
   (a) If the deque is nonempty, the worker pops a task (from the bottom) and begins working on it.
   (b) If the deque is empty, work steal: the worker selects a victim uniformly at random. If its deque is non-empty, its top-most task is stolen and executed. Otherwise, performs work steal again.
3. Task α Syncs. If there are no outstanding children, then continue: we're properly synced. Otherwise (when there are no outstanding children), work steal. Note that deque is empty in this case.

2.2. Sandbox: Horner's Scheme

Horner's Scheme is the algorithm that performs the smallest number of arithmetic operations to evaluate a polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$, as reviewed by Knuth [7, pp. 486–488]. Looking at the polynomial in its unfolded form

$$p(x) = ((\ldots((a_n)x + a_{n-1})x + \cdots + a_2)x + a_1)x + a_0$$

makes the problem straightforward solvable by setting an initial value $v = a_n$ and repeating the procedure $v \leftarrow vx + a_i$ for i from $n-1$ to 0.

Parallel Implementation. At any time, an active processor holds in its local deque an array of some polynomial coefficients $[a_{k+d}, \ldots, a_k]$, sorted from highest to lowest degree. Whenever a steal occurs, the deque is split by half, the theft stealing the coefficients of lowest degree — the smallest part, if $k + d$ is odd. Locally, an active processor continually subtracts a fixed-size chunk of elements from the array and performs sequential Horner over it, accumulating the result to serve as the initial value of the next iteration. When a processor empties its deque it enables the local result to be subject of a joining operation with the partial computations calculated by that successfully stolen the local deque, if any. This algorithm is hybrid (both sequential and parallel versions may run over the same input) and adaptive (the decision is performed by the scheduler, independently from underlying hardware – at least in a direct fashion). Figure 1 shows an implementation using Cilk Plus parallel for and reducers.

```
#include <cilk/reducer_horner.h>

template <typename T> class Horner {
  public:
    Horner (T x = 1, size_t ldc = 0, T res = 0)
      : x (x) , ldc (ldc) , res (res) { } ;

    void acc_horner (T res, T x, size_t ldc) {
      this->x (x) ; this->res *= this->x ;
      this->res += res ; this->ldc ++ ;
    }

    void join (Horner* a) {
      T diff (a->ldc - this->ldc) ;
      this->res *= (T) pow (this->x , diff)  ;
      this->res += a->res ;
      this->ldc = a->ldc ;
    }

  private: T res ; T x ; size_t ldc ;
} ;
...
cilk::reducer_horner<Horner<double> > result ;
cilk_for (size_t i = 0 ; i < SIZE ; ++ i)
    result.acc_horner (input[i], x, (double) i) ;
```

Figure 1: Parallel Horner's Scheme using Cilk Plus' reducers and parallel for. The array is segmented by two at each steal and `acc_horner` is performed locally. Method `join` is called whenever a theft finishes its current portion of the interval.

Overhead. Joining is performed over two chunks $left = [a_{i+d}, \ldots, a_i]$ and $right = [a_{i-1}, \ldots, a_{i-m}]$ that have been already processed and had generated sub results R_{left} and R_{right}. It is a simple attribution $R \leftarrow R_{left} \cdot x^m + R_{right}$. This multiplication by x^m on join operations (which may be implemented in time $\log_2 m$) is interpreted as the overhead — additional arithmetical instructions — added by breaking the "previous iteration dependency" from sequential Horner. An adaptive implementation only pay join costs if a steal operation occurs, implying that the overhead is mitigated online.

The overhead is directly proportional to the number of successful steals, which we prove at the next section to be no more, in expectation, than $2.2(P-1)$, whenever one uses P workers to run the algorithm in parallel.

3. Synchronization Counters

The main idea of synchronization counters is to store the accumulated number of synchronizations performed throughout the execution of a parallel program over P workers.

Each worker $1 \leq i \leq P$ has associated a local counter ϕ_i:

Definition 1 (Local Counter)**.** Let $\mathcal{S}$ be the poset of all reachable states during a parallel execution. A local counter is any function $\phi_i : \mathcal{S} \to \mathbb{N}$ where:

1. If i is inactive at $s \in \mathcal{S}$, then $\phi_i(s) = 0$.
2. If i is active at $s \in \mathcal{S}$, then $\phi_i(s) > \phi(s-)$.

Remark. A local counter is strictly increasing while i is active.

A global counter is the gathering of all local counters with specific constraints:

Definition 2 (Global Counter)**.** Let Σ be a (possibly non maximal) subset of $\mathcal{S}$ containing only synchronization operations. A global counter is any function

$$\begin{array}{rccc} \phi: & \mathcal{S} & \to & {\mathbb{R}_{\geq 0}}^P \\ & s & \mapsto & [\phi_1(s), \cdots, \phi_P(s)] \end{array}$$

where:

1. Function ϕ_i is a local counter for worker i.
2. If $s(i,j) \in \Sigma$, then $\min(\phi_i(s+), \phi_j(s+)) \geq \min(\phi_i(s-), \phi_j(s-)) + 1$
3. If $s(i,j) \notin \Sigma$, then $\min(\phi_i(s+), \phi_j(s+)) > \min(\phi_i(s-), \phi_j(s-))$

Henceforward we consider all successful steals to be the interesting synchronizations, *i.e.*, the ones in Σ. The local counter is the number of successful steals in which the worker participated as either theft or victim. The global counter is the total number of successful steals.

An upper-bound for all local counters bounds the global counter at the worst case:

Theorem 1. *During a randomized work stealing execution over P workers, let Σ subset of steal operations and let ϕ be a global counter over Σ. If there is a constant M such that for all $1 \leq i \leq P$ active at s always $\phi_i(s) \leq M$, then $\mathbf{E}(u) \leq 2.2M(P-1)$.*

Proof Sketch. Call any synchronization operation a "local step". Define a high-order function $\phi_{\min}(\phi, s)$ which returns the value of the minimal non-zero local counter at time s. Prove that its value is strictly increasing until the computation ends (see the Remark after Def. 2). Prove that at the worst case it always increases after a given idle worker tries to steal from every other worker. Call this round of steal attempts a "global step". Use as corollary the fact that the maximum number of global steps is M. Show that in expectation (*e.g.*, by coupon collector's problem [5]) each global step takes no more than $2.2(P-1)$ local steps to complete. □

We use Cilk Plus runtime as the randomized work stealing scheduler. As the local counter $\phi_i(s)$ we chose the size of worker i's deque at s. Thus, M is the maximum size that any deque may reach during computation.

Synchronization counters are *compositional*. One may select different synchronization subsets Σ and use set addition to compose its upper-bounds.

We analyze the overhead on adaptive computations by composing several global counters, one per steal size. We call it overhead synchronization counters:

Definition 3 (Overhead Synchronization Counters)**.** Given an adaptive computation with input size n, an overhead synchronization counter is a collection of $1 \leq i \leq n$ global counters, each one over a set Σ_i, which contains all the steals of size i.

Theorem 2. *Local overhead synchronization counters are binary,* i.e.*, M is either 0 or 1.*

Proof Sketch. Value 0 is trivially shown by Def. 1. Prove that once a steal of size k is suffered, it cannot be suffered again until processor becomes idle (size is strictly decreasing). Thus, $\max(M(i)) = 1$. □

Corollary 1. *For a given adaptive computation, for each possible steal size of k,*

$$\mathbf{E}_k(u) \leq \min(2.2(P-1), n/k)$$

Proof. Follows directly from Theorems 1 and 2. □

Remark. The n/k min term is introduced to tightly upper-bound large values of k. We bound large values of k because there are fewer than P chunks of this size. This is not strictly necessary, but increases the tightness of the limit. Nevertheless, since generally $n \gg P$, this is not the dominant factor in the bound.

4. Experiments

A practical experiment is compared against theory. The code in Figure 1 was executed for $2 \leq P \leq 32$, one hundred times for each value of P and over an input of size 10^8. At each execution, the total number of steals of chunks of size k is counted and arithmetical mean is taken for each value of P. The standard deviation is discussed beyond. Next, we compare the obtained values for any k with the theory. The experiment was performed over the Turing machine of the Parallel and Distributed Processing Group at UFRGS:

- Linux turing 3.2.0-40-generic #64-Ubuntu SMP x86_64 GNU/Linux. CPU(s): 64. Thread(s) per core: 2. NUMA node(s): 4. Vendor ID: GenuineIntel. CPU family: 6. Model: 46. CPU MHz: 1994.971. L1d cache: 32K. L1i cache: 32K. L2 cache: 256K. L3 cache: 18432K.
- Mem. Total: 132,018,988 kB

Figure 2 shows the results. Just the number of steals of each size k is showed as a black ball for each value of number of workers P, without distinguishing the values of k, because theory upper-bounds the number of steals for *any* value of k. The standard deviation ranges from

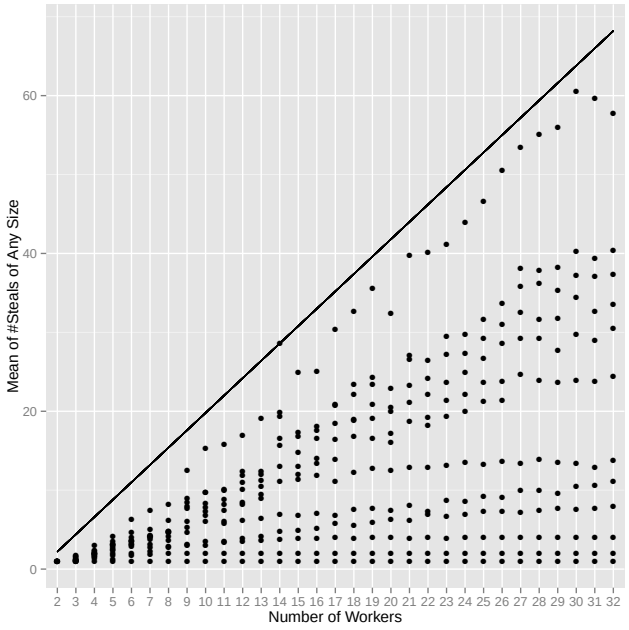

Figure 2: Results from the experiment. Each black ball is the number of steals of some size k. The solid black line is the theoretical limit of $2.2(P-1)$.

values around 0.01 (2 workers) to 0.25 (19 workers), but it is not necessary to our analysis for the same reason; Corollary 1 bounds the number of steals in expectation, for any standard deviation.

The number of points for a given number of workers is the total overhead paid by the parallelization of the Horner scheme. All the points are within the area predicted by the theory, no matter the value of k.

5. Conclusions

As overall result, Theorem 1 and Corollary 1 hold tightly over the performed experiment.

Significant related works are (non-exhaustive list):

- The already mentioned paper on deque-free work-stealing by Traore *et al.*. [9].
- A great number of papers extending the original analysis of Cilk's work stealing algorithm by Blumofe *et al.* [3]. Also, Tchiboukdjian's one [8], with the tightest bounds (as far as we know).
- Several work stealing variations analysis, such as Barik's one [2] over help-first and work-first (the later used at Cilk) strategies.

Yet, slow and fast clone compile strategy by Cilk [6] produce hybrids parallel programs obliviously to the user, although not adaptive.

The subject is not exhausted. Ongoing works by the authors explore the distribution of binary synchronization counters:

- The distribution of the steals as bursts analyzed in the prism of the Remark after Theorem 1. If we color each black ball according with its value of k a cumulative pattern for each k seems to emerge, which flattens as it approaches to $\log_2 k$.
- The generalization from different-sized to independent tasks: steals of different sizes are always independent (may be executed concurrently). Fundamentally, our upper bound relies on independency, which is stronger and more general than the condition of having different sizes. It changes from algorithm to algorithm.

The writing of efficient programs is also a concern. Authors works over random number generators are towards this goals.

References

[1] N. Arora, R. Blumofe, and C. Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. *Proceedings of the tenth annual ...*, 1998.

[2] R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, May 2009.

[3] R. D. Blumofe and P. A. Lisiecki. Adaptive and Reliable Parallel Computing on Networks of Workstations. In *Proceedings of the USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133–147, Anaheim, California, 1997.

[4] V.-D. Cung, V. Danjean, J.-G. Dumas, T. Gautier, G. Huard, B. Raffin, C. Rapine, J.-L. Roch, and D. Trystram. Adaptive and hybrid algorithms: classification and illustration on triangular system solving. In J.-G. Dumas, editor, *Transgressive Computing, April, 2006*, pages 131–148, Grenade, Espagne, 2006.

[5] P. Erdős and A. Rényi. On a classical problem of probability theory. *Magyar Tud. Akad. Mat. Kutató Int. Közl.*, 6:215–220, 1961.

[6] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation - PLDI '98*, pages 212–223, 1998.

[7] D. E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[8] M. Tchiboukdjian, N. Gast, and D. Trystram. A Tighter Analysis of Work Stealing. *Algorithms and ...*, 2010.

[9] D. Traoré, J. Roch, and N. Maillard. Deque-free work-optimal parallel STL algorithms. *Euro-Par 2008âĂŞParallel ...*, 2008.