Information and Computation 254 (2017) 1–40

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Doing-it-All with bounded work and communication ☆

Bogdan S. Chlebus [a,*], Leszek Gąsieniec [b], Dariusz R. Kowalski [b], Alexander A. Schwarzmann [c]

[a] Department of Computer Science and Engineering, University of Colorado Denver, Denver, CO 80217, USA
[b] Department of Computer Science, University of Liverpool, Liverpool L69 3BX, UK
[c] Department of Computer Science and Engineering, University of Connecticut, Storrs, CT 06269, USA

ARTICLE INFO

Article history:
Received 20 September 2002
Received in revised form 20 March 2012
Available online 11 March 2017

Keywords:
Distributed algorithm
Message passing
Crash failures
Scheduling tasks
Load balancing
Ramanujan graphs

ABSTRACT

We consider the *Do-All* problem, where p cooperating processors need to complete t similar and independent tasks in an adversarial setting. Here we deal with a synchronous message passing system with processors that are subject to crash failures. Efficiency of algorithms in this setting is measured in terms of *work*complexity and *communication* complexity. When work and communication are considered to be comparable resources, then the overall efficiency is meaningfully expressed in terms of *effort* defined as *work* + *communication*. We develop and analyze a constructive algorithm that has work $\mathcal{O}(t + p \log p\,(\sqrt{p \log p} + \sqrt{t \log t}))$ and a nonconstructive algorithm that has work $\mathcal{O}(t + p \log^2 p)$. The latter result is close to the lower bound $\Omega(t + p \log p / \log\log p)$ on work. The effort of each of these algorithms is proportional to its work when the number of crashes is bounded above by $c\,p$, for some positive constant $c < 1$. We also present a nonconstructive algorithm that has effort $\mathcal{O}(t + p^{1.77})$.

1. Introduction

Performing a collection of tasks by processors prone to failures is among the fundamental problems in fault-tolerant distributed computing. We consider the problem called *Do-All*, where the processors cooperate on tasks that are similar, independent, and idempotent. Simple instances of this include checking all the points in a large solution space, attempting either to generate a witness or to refute its existence, and scheduling collections of tasks admitting "at least once" execution semantics. We consider the synchronous setting with crash-prone processors that communicate via point-to-point messages, where t tasks need to be performed by p processors subject to f crash failures, provided at least one processor does not crash (i.e., $f \leq p - 1$).

Synchronous message-passing solutions for *Do-All* were first developed by Dwork et al. [27] who estimated the performance of their algorithms in terms of *effort* defined as the sum of task-oriented work and communication. *Task-oriented work* counts only the processing steps expended on performing tasks and it discounts any steps spent idling or waiting for messages. *Communication* costs are measured as the message complexity, calculated as the total number of point-to-point

☆ The results of this paper were announced in a preliminary form in [10]. The work of the first author was supported by NSF Grants 0310503 and 1016847. The work of the third author was supported by Polish National Science Center UMO-2015/17/B/ST6/01897. The work of the fourth author was supported by NSF Grants 0311368 and 1017232.

* Corresponding author.
E-mail address: bogdan.chlebus@ucdenver.edu (B.S. Chlebus).

http://dx.doi.org/10.1016/j.ic.2017.02.003
0890-5401/

messages. Note that the time of algorithm executions may be very large, for example, when f can be $p-1$, time may be at least linear in t for the cases where one remaining processor must perform all tasks. Thus time is not normally used to describe the efficiency of *Do-All* algorithms when large number of failures is allowed. We also measure algorithm performance in terms of *effort*, except that we use a more conservative approach. Instead of task-oriented work, we include *available processor steps* complexity (or *total work*) defined by Kanellakis and Shvartsman [38] that accounts for all steps taken by each processor, including idling, until the processor either terminates the computation or crashes. Thus we define the *effort* of an algorithm to be $\mathcal{W}+\mathcal{M}$, where $\mathcal{W}$ is its total work complexity and $\mathcal{M}$ is its message complexity. Other prior research focused on developing *Do-All* algorithms that are efficient in terms of work, then dealing with communication efficiency as a secondary goal, e.g., using the *lexicographic complexity* [24].

Trade-offs between work and communication in solutions of *Do-All* are to be expected, indeed, communication improves coordination among processors with the potential of reducing redundant work in unreliable systems. There are two direct ways in which a processor can get to know that a certain task is complete in a message-passing system: the processor can either perform the task, or it can receive a message that the task was completed by another processor. Note that *Do-All* can be solved without any communication: simply have each processor perform each task. The work of such an algorithm is $\mathcal{O}(p \cdot t)$. On the other hand, an algorithm may cause each processor to always share its knowledge with all other processors in an attempt to reduce work. This may result in efficient work, but the communication complexity of such an algorithm is $\Omega(p \cdot t)$: each time a processor completes a task, it sends a message to all other processors. Thus it is highly desirable to develop algorithms for which both work is $o(p \cdot t)$ and communication is $o(p \cdot t)$. This makes it meaningful to balance work and communication, and to consider them as comparable resources. These observations motivate the use of the quantity $\mathcal{W}+\mathcal{M}$ as a unifying performance metric.

The performance bounds of our algorithms are expressed in terms of three parameters: the number of processors p, the number of tasks t, and the number of crashes f that may occur in the course of an execution. Our algorithms place no constraints on the relationship between t and p; these parameters are independent in our algorithms and the complexity bounds. The only restriction on f is that at least one processor does not crash, i.e., $f < p$.

We say that a parameter is *known* when it can be used in the code of an algorithm. The parameters p and t are always known. When the parameter f appears in performance bounds, then this indicates that the algorithm is designed to optimize performance for the number of crashes that is at most f. When $f < p$ is used as a parameter then f is known. When f is not used in the code of an algorithm then it is only known that $f \leq p-1$. It so happens that if f appears in an algorithm in this paper, then the corresponding communication complexity depends on f, whereas the bounds on the work complexity involve only p and t.

We consider an adversarial setting, in which a nefarious adversary causes processors to crash. An adversary is *f-bounded* if $f < p$ is an upper bound on the number of crashes in any execution. When stating a performance bound for a known $f < p$ we normally state that the bound holds "for an f-bounded adversary." The $(p-1)$-bounded adversary is called *unbounded*. An f-bounded adversary is called *linearly bounded* if $f \leq c\,p$, for some positive constant $c < 1$. Our algorithms always solve *Do-All* when exposed to the unbounded adversary, but their message complexity may be especially efficient when the adversary is linearly-bounded.

We call a deterministic algorithm *constructive* if its code can be produced by a deterministic sequential algorithm in time polynomial in t and p. This is in contrast with *nonconstructive* algorithms that may rely on combinatorial objects that are only known to exist. Methodologically, starting with a constructive algorithm, we trade constructiveness for better effort bounds in producing nonconstructive algorithms.

We aim for algorithmic solutions for *Do-All* that attain good effort complexity, rather than seeking just work-efficient solutions. Here a key challenge, besides tolerating crashes and controlling work, is to ensure that communication costs do not exceed work complexity. Whereas any two processors can communicate in any step of computation, in our algorithms we limit communication by allowing messaging to take place over certain constant-degree subnetworks. To this end, we use constructive graphs with good "expansion" properties, and this contributes to the emerging understanding of how expansion-related properties of the underlying communication schemes can be used to improve fault tolerance and efficiency.

Our results. We present a new way of structuring algorithms for the p-processor, t-task *Do-All* problem that allows for both work *and* communication to be controlled in the presence of adaptive adversaries. We give a generic algorithm for performing work in systems with crash-prone processors, and we parameterize it by (*i*) task-assignment rules and (*ii*) virtual overlay graphs superimposed on the underlying communication medium. We now detail our contributions.

I. We present a deterministic constructive algorithm, called Balance-Load, that uses a balancing task allocation policy (Section 5). This algorithm solves *Do-All* in any execution with at least one non-crashed processor, and its performance is tuned to a known upper bound f on the number of crashes, where $f < p$. The algorithm's work is $\mathcal{W} = \mathcal{O}(t + p \log p\,(\sqrt{p \log p} + \sqrt{t \log t}))$, which does not depend on f, while the message complexity does depend on f. When the adversary is additionally constrained to be linearly-bounded, the message complexity of the algorithm is $\mathcal{M} = \mathcal{O}(\mathcal{W})$.

No prior algorithms using point-to-point messaging attained total work (available processor steps) that is both $o(p^2)$ and $o(t^2)$ against the f-bounded adversary, for any known $f < p$. By using embedded graphs whose properties depend on f,