Analysis of Synchronizations In Greedy-Scheduled Executions - Application to Efficient Generation of Pseudorandom Numbers in Parallel

Stefano Drimon Kurz Mor

UFRGS
UNIVERSIDADE FEDERAL
DO RIO GRANDE DO SUL

UNIVERSITÉ DE
GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L’UNIVERSITÉ DE GRENOBLE

prépareé dans le cadre d’une cotutelle entre l’Université Grenoble Alpes et l’Universidade Federal do Rio Grande do Sul

Spécialité : **Informatique**

Arrêté ministériel : le 6 janvier 2005 -7 août 2006

Présentée par

Stéfano Drimon KURZ MÓR

Thèse dirigée par **Bruno RAFFIN**
codirigée par **Jean-Louis ROCH et Nicolas MAILLARD**

préparée au Laboratoire d’Informatique de Grenoble dans le cadre de l’**Ecole Doctorale Mathématiques, Sciences et Technologies de l’Information, Informatique** et au Laboratoire de Parallelisme et Distribution dans le cadre du **Programme de Doctorat en Informatique**

Analyse des synchronisations dans un programme parallèle ordonnancé par vol de travail. Applications à la génération déterministe de nombres pseudo-aléatoires.

*Université Joseph Fourier / Université Pierre Mendès France /
Université Stendhal / Université de Savoie / Grenoble INP*

Thèse soutenue publiquement le **26 Octobre 2015**,
devant le jury composé de :

M. Philippe O. A. NAVAUX
Professeur, Universidade Federal do rio Grande do Sul, Président
M. Avelino Francisco ZORZO
Professeur, Pontifícia Universidade Católica RS, Rapporteur
M. Emmanuel JEANNOT
Directeur de Recherche, INRIA, Rapporteur
M. Gérson Geraldo H. CAVALHEIRO
Maître de Conférences, Universidade Federal de Pelotas, Examinateur
M. Bruno RAFFIN
Chargé de Recherche, Université de Grenoble, Directeur de Thèse
M. Jean-Louis ROCH
Maître de Conférences, Université de Grenoble, Directeur de Thèse
M. Nicolas Bruno MAILLARD
Maître de Conférences, Universidade Federal do Rio Grande do Sul, Directeur de Thèse

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

STÉFANO DRIMON KURZ MÓR

Analysis of Synchronizations in Greedy-Scheduled Executions and Applications to Efficient Generation of Pseudorandom Numbers in Parallel

Thesis prepared in a co-tutelle agreement
and presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dr. Jean-Louis ROCH
Coadvisor: Prof. Dr. Nicolas MAILLARD

Porto Alegre
November 2015

CIP – CATALOGING-IN-PUBLICATION

KURZ MÓR, Stéfano Drimon

Analysis of Synchronizations in Greedy-Scheduled Executions and Applications to Efficient Generation of Pseudorandom Numbers in Parallel / Stéfano Drimon KURZ MÓR. – Porto Alegre: PPGC da UFRGS, 2015.

184 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2015. Advisor: Jean-Louis ROCH; Coadvisor: Nicolas MAILLARD.

1. Parallel Algorithms. 2. Work-Stealing. 3. Logical Clocks. 4. Pseudorandom Numbers. 5. Nondeterministic Executions. I. ROCH, Jean-Louis. II. MAILLARD, Nicolas. III. Título.

"There exists, in Math, only one situation in which two plus two is not four.

It is when the mathematician has made a mistake."

— Edgar de Souza Mór

CONTENTS

LIST OF FIGURES

ACRONYMS

SPAA	ACM Symposium on Parallelism in Algorithms and Architectures	61
PRNG	Pseudorandom Number Generator	8
BBS	Blum Blum Shub	13
SIPS	Strictly Increasing Per Synchronization	7
CSP	Concurrent Sequential Processess	42
CPU	Central Processing Unit	4
GPU	Graphical Processing Unit	23
GPGPU	General Purpose Graphical Processing Unit	159
PRAM	Parallel Random-Access Machine	25
DSM	Distributed Shared Memory	24
SMP	Simultaneous Multi-Processors	4
DAG	Directed Acyclic Graph	6
SIMD	Single Instruction Multiple Data	67
SFMT	SIMD-oriented Fast Mersenne Twister	117
GPPD	Grupo de Processamento Paralelo e Distribuído	14
ENSIMAG	École d'Ingénieurs Mathemátiques Appliquées Télécommunications	18
Grenoble-INP	Institut Polytechnique de Grenoble	18
MOAIS	Parallel Algorithms, Programming Models, Scheduling and Interactive Computing	18
LICIA	International Laboratory on High Performance Computing and Environmental Informatics	18
MIT	Massachusetts Institute of Technology	39
CAPES	Coordenação de Aperfeiçoamento de Pessoal de Ensino Superior	18
CNPQ	Conselho Nacional de Desenvolvimento Científico e Tecnológico	18
UFRGS	Universidade Federal do Rio Grande do Sul	4
PPGC	Programa de Pós-Graduação em Ciência da Computação	18
LIG	Laboratoire d'Informatique de Grenoble	18

UGA	Université Grenoble Alpes	4
deque	double-ended queue	9
ABI	Application Binary Interface	163
API	Application Program Interface	9
NIST	The United States' National Institute of Standards and Technologies	12
LAN	Local Area Network	23
WAN	Wide Area Network	23
OS	Operating System	6
OpenMP	Open Multiprocessing	38
MPI	Message-Passing Interface	10
TBB	Threading Building Blocks	9
Kaapi	Kernel for Adaptative, Asynchronous Parallel and Interactive programming	9
GPGPU	General-Pourpose Graphichs Processing Unit	159
SPMD	Single Program, Multiple Data	174
IO	Input/Output	174
UMA	Uniform Memory Access	14
NUMA	Non-Uniform Memory Access	23
PBBS	Problem-Based Benchmark Suite	138
MIS	Maximum Independene Set	136
STL	C++'s Standard Template Library	107
TLMM	Thread-Local Memory Mapping	57

ABSTRACT

We present two contributions to the field of parallel programming.

The first contribution is theoretical: we introduce *SIPS* analysis, a novel approach to estimate the number of synchronizations performed during the execution of a parallel algorithm. Based on the concept of logical clocks, it allows us: on one hand, to deliver new bounds for the number of synchronizations, in expectation; on the other hand, to design more efficient parallel programs by dynamic adaptation of the granularity.

The second contribution is pragmatic: we present an efficient parallelization strategy for pseudorandom number generation, independent of the number of concurrent processes participating in a computation. As an alternative to the use of one sequential generator per process, we introduce a generic API called `Par-R`, which is designed and analyzed using SIPS. Its main characteristic is the use of a sequential generator that can perform a "jump-ahead" directly from one number to another on an arbitrary distance within the pseudorandom sequence. Thanks to SIPS, we show that, in expectation, within an execution scheduled by work stealing of a "very parallel" program (whose depth or critical path is subtle when compared to the work or number of operations), these operations are rare. `Par-R` is compared with the parallel pseudorandom number generator DotMix, written for the Cilk Plus dynamic multithreading platform. The theoretical overhead of `Par-R` compares favorably to DotMix's overhead, what is confirmed experimentally, while not requiring a fixed generator underneath.

Keywords: Parallel Algorithms. Work-Stealing. Logical Clocks. Pseudorandom Numbers. Nondeterministic Executions.

RESUMO

Análise de Sincronizações em Execuções por Escalonamento Guloso e Aplicações para Geração Eficiente de Números Pseudoaleatórios em Paralelo

Nós apresentamos duas contribuições para a área de programação paralela.

A primeira contribuição é teórica: nós introduzimos a análise *SIPS*, uma nova abordagem para a estimar o número de sincronizações realizadas durante a execução de um algoritmo paralelo. SIPS generaliza o conceito de relógios lógicos para contar o número de sincronizações realizadas por um algoritmo paralelo e é capaz de calcular limites do pior caso mesmo na presença de execuções paralelas não-determinísticas, as quais não são geralmente cobertas por análises no estado-da-arte. Nossa análise nos permite estimar novos limites de pior caso para computações escalonadas pelo popular algoritmo de roubo de tarefas e também projetar programas paralelos e adaptáveis que são mais eficientes.

A segunda contribuição é pragmática: nós apresentamos uma estratégia de paralelização eficiente para a geração de números pseudoaleatórios. Como uma alternativa para implementações fixas de componentes de geração aleatória nós introduzimos uma *API* chamada `Par-R`, projetada e analisada utilizando-se SIPS. Sua principal idea é o uso da capacidade de um gerador sequencial `R` de realizar um “pulo” eficiente dentro do fluxo de números gerados; nós os associamos a operações realizadas pelo escalonador por roubo de tarefas, o qual nossa análise baseada em SIPS demonstra ocorrer raramente em média. `Par-R` é comparado com o gerador paralelo de números pseudoaleatórios DotMix, escrito para a plataforma de multithreading dinâmico Cilk Plus. A latência de `Par-R` tem comparação favorável à latência do DotMix, o que é confirmado experimentalmente, mas não requer o uso subjacente fixado de um dado gerador aleatório.

Palavras-chave: Algoritmos paralelos, roubo de tarefas, relógios lógicos, números pseudoaleatórios, execuções não-determinísticas.

RÉSUMÉ

Analyse des synchronisations dans un programme parallèle ordonnancé par vol de travail. Applications à la génération déterministe de nombres pseudo-aléatoires.

Nous présentons deux contributions dans le domaine de la programmation parallèle.

La première est théorique : nous introduisons l'analyse *SIPS*, une approche nouvelle pour dénombrer le nombre d'opérations de synchronisation durant l'exécution d'un algorithme parallèle ordonnancé par vol de travail. Basée sur le concept d'horloges logiques, elle nous permet : d'une part de donner de nouvelles majorations de coût en moyenne; d'autre part de concevoir des programmes parallèles plus efficaces par adaptation dynamique de la granularité.

La seconde contribution est pragmatique : nous présentons une parallélisation générique d'algorithmes pour la génération déterministe de nombres pseudo-aléatoires, indépendamment du nombre de processus concurrents lors de l'exécution. Alternative à l'utilisation d'un générateur pseudo-aléatoire séquentiel par processus, nous introduisons une API générique, appelée `Par-R` qui est conçue et analysée grâce à SIPS. Sa caractéristique principale est d'exploiter un générateur séquentiel qui peut "sauter" directement d'un nombre à un autre situé à une distance arbitraire dans la séquence pseudo-aléatoire. Grâce à l'analyse SIPS, nous montrons qu'en moyenne, lors d'une exécution par vol de travail d'un programme très parallèle (dont la profondeur ou chemin critique est très petite devant le travail ou nombre d'opérations), ces opérations de saut sont rares. `Par-R` est comparé au générateur pseudo-aléatoire DotMix écrit pour Cilk Plus, une extension de C/C++ pour la programmation parallèle par vol de travail. Le surcout théorique de `Par-R` se compare favorablement au surcoput de DotMix, ce qui apparait aussi expériemntalement. De plus, étant générique, `Par-R` est indépendant du générateur séquentiel sous-jacent.

Mots-clef: Algorithmes parallèle, vol de travail, horloges logiques, nombres pseudo-aléatoire, exécutions non-déterministes.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

STÉFANO DRIMON KURZ MÓR

Analysis of Synchronizations in Greedy-Scheduled Executions and Applications to Efficient Generation of Pseudorandom Numbers in Parallel

Thesis prepared in a co-tutelle agreement
and presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dr. Jean-Louis ROCH
Coadvisor: Prof. Dr. Nicolas MAILLARD

Porto Alegre
November 2015

CIP – CATALOGING-IN-PUBLICATION

KURZ MÓR, Stéfano Drimon

Analysis of Synchronizations in Greedy-Scheduled Executions and Applications to Efficient Generation of Pseudorandom Numbers in Parallel / Stéfano Drimon KURZ MÓR. – Porto Alegre: PPGC da UFRGS, 2015.

184 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2015. Advisor: Jean-Louis ROCH; Coadvisor: Nicolas MAILLARD.

1. Parallel Algorithms. 2. Work-Stealing. 3. Logical Clocks. 4. Pseudorandom Numbers. 5. Nondeterministic Executions. I. ROCH, Jean-Louis. II. MAILLARD, Nicolas. III. Título.

1 INTRODUCTION

This thesis is concerned with the performance of computer algorithms written to run on current parallel hardware. In what follows, we expect the reader to have at least a bachelor degree's knowledge level in Computer Science. Although no further knowledge is mandatory, a reader with a consistent background in parallel programming and high-performance computation is more likely to profit from the whole material.

This work is centered around *concurrent algorithms*, informally defined to be any algorithm whose set of instructions or its subsets may execute concurrently. We define two specializations of it: *parallel algorithms*, informally defined to be any concurrent algorithm with a corresponding meaningful sequential version, and *distributed algorithms*, defined to be any concurrent algorithm without a corresponding meaningful sequential version. Examples of parallel algorithms are reduction, sorting, search, and transform. Examples of distributed algorithms are snapshot, consensus, leader election, and decentralized scheduling. Since there is no unison position in the literature (see, for instance, (JAJA, 1992; KUMAR, 2002; CASANOVA; LEGRAND; ROBERT, 2008; HERLIHY; SHAVIT, 2008; FORUM, 2012)) about this taxonomy, we chose to define one that suits our work and is coherent with the problems we investigate.

The discussion that follows is over the discipline of parallel programming in the sense that the "front-end" algorithms we analyze, the applications of our principles, are parallel. However, we also discuss distributed algorithms that manage the parallel execution — *e.g.*, decentralized scheduling of parallel programs — and its analysis — *e.g.*, logical clocks.

Given that discussion is meaningless without context — and its associated motivation —, we examine the scenario in breadth through a brief survey on parallel programming trends (Section 1.1). After that, we present this thesis' two main contributions.

The first contribution is theoretical (Section 1.2): we introduce a novel approach to analyze the number of synchronizations performed during a given execution of a concurrent algorithm named *SIPS*. The number of synchronizations is essentially the communication cost for a parallel algorithm, what allows one to estimate the overhead introduced by the parallelization. Through the generalization of the concept of logical clocks in asynchronous systems, we can deliver new worst-case bounds on these operations.

The second contribution is pragmatic (Section 1.3): we present a parallelization of generic algorithms for pseudorandom number generation in current hardware. We explore the fact that we can "jump-ahead" on the generated sequence faster than a serial generation and propose synchronization techniques to produce deterministic sequences in parallel. Both design and analysis of these algorithms are performed using SIPS.

We introduce each topic separately, providing individual motivation and description of the contribution. Nonetheless, the parts are abridged into a derivative sequence. The link between the two parts is the design of the algorithms, whose efficiency and generalization are only possible through the estimation of the synchronization bounds.

The present chapter intends to give an accurate scope for the ensuing discussion and rationale. This scope is both semantical — regarding motivation, enlisting of problems, and proposal of solutions — and syntactic — regarding structural premises and outlining of contents (Section 1.4). At the end of the chapter, contextual information on the institutional relationship is provided, scoping the work in a co-direction agreement between Universidade Federal do Rio Grande do Sul (UFRGS) in Brazil and Université Grenoble Alpes (UGA) in France (Section 1.5).

The chapter finishes with closing remarks (Section 1.6).

1.1 A Brief Survey on Parallel Programming Trends

Programming courses first teach a student to program sequentially. First and foremost, it is easier to write and to maintain sequential code. Besides, until *circa* 2000, sequential programs could extract the best the hardware had to offer: deep pipelines, out of order dispatch, speculative executions, *etc.* With the advent of *superscalar architectures* performance profited from parallelism in an oblivious fashion. (ASANOVIC et al., 2009)

Explicit parallelism became mainstream with the arrival of multi and many-core hardware architectures. The primary resource is the core, a processing unit. It varies in quantity and type. As for quantity, it increases every day, and several resources are grouped in processors. As for type, it makes available different sets of operations (specialized) and speed (clock frequency or microinstructions). Resources evolve fast in shape and comprehension: Simultaneous Multi-Processors (SMP) are the current standard paradigm in the implementation of a general-purpose Central Processing Unit (CPU); accelerators are a larger set of specialized cores; vectorial machines offer single instructions over whole vectors; Clusters (PFISTER, 1998) (resp. Grids (FOSTER; KESSELMAN, 1999)) are a grouping of homogeneous (resp. heterogenous) machines whose resources are parallel.

Parallelism is also omnipresent in computational devices, being the default for standard personal computers (including laptop, desktops, workstations, *etc*) and in portable computing devices, mainly smartphones and, more recently, smart watches. It is also available in large-scale items like cars and refrigerators.

Moore's Law (SCHALLER, 1997), *viz.*

> "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will remain nearly constant for at least 10 years."

still dictates the growth of hardware, transistor-wise. His reasoning was based on an empirical log-linear relationship between device complexity and time, observed over three data points. He revised his rate of circuit complexity doubling to 18 months and projected from 1975 onwards at this reduced rate. Today, parallel hardware conforms almost linearly to Moore's Law. To obtain nearly linear performance increase implies to profit from the linear growth in the number of transistors.

Parallel hardware requires adequate software. Concurrency of resources enables simultaneous execution of programs and their parts. (Simultaneous executions of programs are approached in the implementation of operating systems.) Concurrent decomposition of single programs have several pre-requisites: suitable parallel algorithms; proper resource management (*e.g.*, scheduling); hardware-based scalability (performant in changing the number of processors, cost of communication, *etc*); and software-based scalability (performant in changing of input's size, out-of-order execution), *etc.*

The overall objective of parallel software is to minimize the *makespan*, *i.e.*, total execution time. Further examination requires a differentiation between performance and throughput (PATTERSON; HENNESSY, 2008):

Performance. Execution of a fixed set of tasks in less time. Example applications: physical simulations, weather forecast, DNA sequencing, gaming, audio and video decomposition.

Throughput. Execution of a larger set of tasks in same time. Example applications: traffic analysis, swarm algorithms (bioinformatics), genetic algorithms.

The concepts correlate. (This implies correlation of the given examples as well.) Faster algorithms allow a larger set of their kindred to run simultaneously in fixed time. Conversely, decomposing a program in parallel tasks (an atomic sequence of operations) allows a program to run faster on larger throughput. *Ergo*, performance and throughput are synergic.

Expressing a sequential program as a system of disjoint components allows one to increase both performance and throughput. Here we consider the following decomposition outline:

1. State the problem in terms of concurrent tasks. A task is an indivisible procedure, although it may spawn other tasks.
2. Execute the program on the given hardware, allocating ready tasks to idle workers as the execution goes on. This is known as *dynamic scheduling*.

To identify sequential dependencies among tasks is much of a parallel programmer's work — correct parallelization must overall ensure the same semantics of the parallel version, what is determined by respecting the dependencies. It personifies the importance of this particular topic; the programmer is unable to profit from the performance offered by the hardware unless he knows to decompose its programs in tasks and allocate them appropriately. The details of steps (1) and (2) are the objects of discussion at this thesis. The ultimate goal of our contribution is to provide new insights in the analysis of schedulers and draw performance through it in important fields, like pseudorandom number generation.

Next, we discuss motivation and contributions of the central topics of this thesis. The work is structured in two intersectional parts, and so we provide motivation and contribution for each separately.

1.2 Part I - The Tools of Analysis: Synchronizations in Greedy Scheduled and Work-Stealing Scheduled Parallel Algorithms

1.2.1 Motivation

Scenario. Many parallel algorithms that are described by a set of tasks rely on a greedy scheduler to maximize processor utilization and throughput: at any time each resource is performing an operation, either one from the parallel algorithm or one from the scheduling algorithm. Among greedy schedulers, a non-preemptive task scheduler (not to be confused with an Operating System (OS) process/thread scheduler) maintains a pool of tasks; when a processor is idle it extracts some task from the pool and executes it. This distributed algorithm is described by a Directed Acyclic Graph (DAG) (CASANOVA;

LEGRAND; ROBERT, 2008), where every node represents some task, and a directed edge represents some precedence. A task is ready when all its predecessors in the DAG are complete. The DAG begins with all tasks in it. When the program executes a task the corresponding action on the DAG is to remove it and all its outgoing edges. The implementation has to specify the scheduling operations a processor performs when it completes its current task or creates a new ready task.

Scientific Gap. Classical analysis considers a fixed DAG for the same algorithm and input (LEISERSON, 2009; FRIGO; LEISERSON; RANDALL, 1998; FRIGO et al., 1999; ARORA; BLUMOFE; PLAXTON, 1998; BLUMOFE, 1994; BLUMOFE; LEISERSON, 1999; TCHIBOUKDJIAN et al., 2010). However, algorithms may be random or work on the top of random parts and certain events can occur in an unpredictable order within concurrent algorithms. Besides, the number of processes in the concurrent algorithm is fixed to the number of computing resources (or cores), based on a one-to-one mapping of processes to resources. The Classical analysis provides loose bounds in these cases. Our work aims to fill this gap.

1.2.2 Contributions

We present Strictly Increasing Per Synchronization (SIPS) analysis. It is used to analyze the number of given specific operations performed by a concurrent algorithm. The idea of SIPS already existed prior to this thesis by the unpublished works of Jean-Louis Roch and the MOAIS Team (ROCH, 2012). Besides organizing and presenting the concept of SIPS, this thesis delivers a new analysis of expectation and the associated upper-bounds for the number of synchronisations on parallel computations through SIPS clocks. It also validates these and previous bounds over the parallel middleware Cilk Plus and a discrete event simulator written by ourselves to emulate other middlewares' behavior.

Contrary to classical analysis, SIPS assumes a parallel architecture with an unbounded number of workers and resources (eventually heterogeneous) and a non-deterministic DAG. Classical measurement on the field is based solely on the *work* (total number of operations) and the *depth* (number of operations that must run sequentially) of a parallel algorithm. This limits a given DAG to be dependent only on the input and not on execution parameters. Thus, the graph must be fixed for a particular input, neither allowing random components nor dynamic, adaptable — architecture-wise — algorithms.

SIPS is based on the idea of using upper-bounds on logical clocks to deliver an upper-bound on the number of operations performed either by the parallel algorithm or by the scheduler. In fact, it is especially useful in upper-bounding the synchronization operations between workers. This allows us to deliver new bounds for algorithms scheduled by the randomized work-stealing scheduling algorithm. It is also especially useful in the analysis of *adaptive algorithms*, which may dynamically change based on execution parameters.

As a guiding example, we propose an adaptive implementation of polynomial evaluation by Horner's Method (KNUTH, 1997b). Parallel implementations of polynomial evaluation are usually based on a less efficient method named Estrin's (ESTRIN, 1960). We use SIPS do demonstrate that our proposed implementation is more efficient in both total number of operations performed and parallel overhead introduced. Finally, we show how the analysis can be used to *design* more efficient algorithms, which is used to write efficient random number generators on the second part of the thesis.

1.3 Part II - The Product of Practice: Applications to Parallel Pseudorandom Number Generation

1.3.1 Motivation

Scenario. Dynamic multithreading is a parallel programming model that provides a (thread-based) *processor-oblivious* framework, where keywords enable parallelism on the serial code without any reference to the number of available processors. A non-blocking randomized work-stealing scheduler manages the execution. Dynamic-scheduled multithread platforms guarantee deterministic computations, despite the intrinsic non-determinism introduced by the scheduler. Nonetheless, this guarantee is not extended to the parallel execution, what breaks determinism in state-based components. Such is the case of Pseudorandom Number Generators (PRNGs) (BARKER; KELSEY, 2012). Sometimes called "quasi-random number generator" or "random bit generator", a PRNG is a stream-based structure that provides random numbers deterministically from a given initial seed. It is useful to ensure reproducibility to random experiments and also in the debugging of randomized algorithms.

Scientific Gap. State-of-the-art PRNGs for dynamic multithreaded environments overcomes the lack of determinism guarantees in work-stealing schedulers by fixing a tailored generation algorithm, trading-off particular generator properties (*e.g.*, randomness, cryptography, regularity, *etc.*) for performance.

1.3.2 Contributions

As an alternative to fixed implementations for parallel PRNGs, we propose a generic parallel Application Program Interface (API) called `Par-R`. It is designed with the upper-bounds on work-efficient algorithms obtained by SIPS in mind and ensures deterministic parallel executions on dynamic multithreading platforms. `Par-R` uses as underlying engine a sequential PRNG named `R`, state or counter-based, and inherits its qualities without compromising parallel efficiency. This is done through the discipline of generic programming (STEPANOV; MCJONES, 2009), which allows us to write one algorithm that works for a family of types. Its main insight is the use of `R`'s capability of "jumping-ahead" in the generated stream to ensure determinism; the application partitions the random sequence on-the-fly among the parallel tasks, and each task re-seeds its PRNG through a jump-ahead to generate only random numbers belonging to its subsequence. To ensure efficiency, these re-seeds occur only when triggered by a steal operation performed by the work-stealing scheduler. We prove through SIPS analysis that this method introduces an overhead upper-bounded by the parallel work (*work-efficiency*) even when efficient jump-ahead is absent, and that the theoretical re-seeding overhead is polylogarithmic (*work-optimality*) whenever `R` provides at least logarithmic jump operations in the random sequence.

The core benefit of `Par-R` is its performance. Because the determinism overhead is only "paid" at steal operations, our SIPS-based theoretical analysis shows that in expectation it does not occur many times. Otherwise, the introduced overhead is adaptively mitigated by the available parallelism. `Par-R` can be used as a component in parallel libraries both because of its generic requirements and because its operations do not produce side-effects, by design. Benchmarks are taken over classical random algorithms like Musser's Introsort (MUSSER, 1997) or Luby's Maximum Independent Set (FERREIRA; SCHABANEL, 1999).

`Par-R` is compared with the stateful PRNG DotMix, written for the Cilk Plus dynamic multithreading platform. DotMix supports infinite simulations, but requires any execution to match the same DAGs and produces side-effects. `Par-R` does support non-deterministic DAGs, but only finite computations. Also contrary to DotMix — whose implementation is fixed —, our approach can be made secure by using underlying cryptographic generators.

The polylog overhead of `Par-R` compares favorably with the linear cost of DotMix reseedings.

The core limitation of `Par-R` is its dependency on an estimation of how many random numbers are to be generated. This imposes a narrower range of useful algorithms benefiting from it, although the range is not narrow itself. Examples go from generating algorithms — *e.g.*, static generation, stream-based generation — and finite step algorithms — *e.g.*, genetic algorithms for crossing over — to selection-based algorithms — *e.g.*, randomized quicksort, maximal independent set, Monte-Carlo. Graph algorithms are especially suitable for this kind of approach. As we will see, one can frequently rely on *overestimation* of the generated numbers to benefit from `Par-R`. The performance gain usually surpasses the introduced overhead.

1.4 Outline, Conventions, and Principles

Present work is a thesis by monograph, in opposition to a thesis by publications. Notwithstanding, there are two relevant publications associated with it:

2011. In "International Journal on High Performance Systems Architecture": *Dynamic workload balancing deques for branch and bound algorithms in the message passing interface* (MOR; MAILLARD, 2011). This is a paper that follows the preliminary thesis plan delivered to UFRGS — the Ph.D. studies begun during the second semester of 2010. The proposal's theme is the development of an "algebra of tasks" whose sequential dependencies are given by the semantics of classical data structures like lists, sets, and priority queues — a "container" in our terminology. We present in the paper the first container implementation, a library where the container of tasks defined by the programmer is a queue. The balancing of work between each process queue without the programmer's intervention is discussed. Profiting from a ring structure offered by the distributed memory runtime (an implementation of the Message-Passing Interface (MPI)), the analysis of the steals is somewhat easier than the general case, but yet it provides useful bounds and guarantees on the expected number of retries. It shows, in this context, that if any steal of workload performed by the scheduler is unsuccessful, then the computation will end in finite time proportional to the number of processors.

2014. In "Euro-Par 2014 Parallel Processing - 20th International Conference": *Generic Deterministic Random Number Generation in Dynamic-Multithreaded Platforms* (MOR;

ROCH; MAILLARD, 2014). This is a condensed version of this thesis. The focus is to present the results obtained in Part II. However, a good part of it is dedicated to introducing a narrower version of SIPS named "synchronization counters". This summarized content is employed to justify the design of the algorithms and their analysis since it is built on the top of Part I.

The 2011 paper was planned as the first one in a series, each covering a different data structure (container). However, this work was not feasible with the bounds for synchronization operations the literature provided at the time. A more general framework was needed to encompass the analysis. In this sense, a more ambitious goal was traced to the advent of international co-direction, shifting the focus to the analysis of synchronizations in parallel executions and the applicability of those bounds to the design of algorithms. Once this work reaches full maturity, we will be able to review and analyze the algebra of tasks concept once again.

1.4.1 Outline

Besides this introductory chapter, two other ones compose our core concepts. They discuss fundamental concepts — along with their associated vocabulary — and current literature on the subject. A chapter-by-chapter description follows.

Chapter 2. *Background.* A chapter that discuss parallel programming concepts. Contents are arranged in a stacked sequence ranging from parallel machine architectures at the bottom to parallelization of algorithms on the top. The chapter approaches topics such as parallel machine models, programming artifacts, DAG, and scheduling.

Chapter 3. *State Of The Art.* Lists and discusses related works both in the analysis of parallel algorithms (theme of Part I) and advances on pseudorandom number generators (as seen in Part II). We use this discussion to establish comparison criteria between our methods and up-to-date literature. Implementation factors are discussed whenever possible.

Two self-contained chapters compose the first part. First, at the central chapter of the thesis, we develop our solution to a class of generalized problems. Then, we present the concept of adaptive algorithms and show how SIPS analysis can be used to estimate their overhead:

Chapter 4. *SIPS: A Technique to Analyze Synchronizations in Greedy Scheduled Algorithms.* We introduce SIPS, an analysis framework that aims to bound different aspects of a parallel program's execution through an upper-bound on the number of synchronizations. The focus is computations scheduled by the work-stealing algorithm. Distinct modalities on victim selection are analyzed, such as global minimum/maximum SIPS value and random choice. Also, the bounds bridge diverse extraction methods from victim's work list, like top-most task, half of tasks, and any k tasks. Finally, two classes of adaptive algorithms are introduced in respect to the inserted overhead, work-efficient and work-optimal algorithms.

Chapter 5. *Case Study: Adaptive Algorithms and Polynomial Evaluation Schemes.* We discuss adaptive algorithms, capable of changing themselves dynamically, providing efficiency by combining parallel and sequential implementations over the available resources. The rationale is traced over Horner's Method, the most efficient known algorithm for polynomial evaluation. We show that an adaptive version of it is usually more efficient than classical implementation for parallel evaluation.

Part II is composed of three chapters that, although coherent, are not self-contained, since they depend on the analysis methods discussed in Part I. Its primary concern is to establish the basis and develop algorithms for parallel generation and the family of algorithms built on the top of it:

Chapter 6. *A Parallel API for Sequential Pseudorandom Number Generators -* `Par-R`. We introduce an API for state-based generators named `Par-R`. This embodies the basis for the programmable artifacts used to provide a stream of random numbers in parallel. Each design is modelled for maximum flexibility and mirrors current standards in diverse areas such as pragmatical implementation — *e.g.*, C++ standard library (PLAUGER et al., 2000) — and specifications — *e.g.*, The United States' National Institute of Standards and Technologies (NIST) directives (BARKER; KELSEY, 2012). By the advent of generic programming and adaptor interfaces, we are able to provide compile-time dispatch for a vast myriad of state of the art generators orthogonally to our algorithms. Considerations about the asymptotic complexity of the primitives are also displayed, being essential for the design of algorithms presented in the next chapters. An interface for the sequential generating algorithm is also shown.

Chapter 7. *Design and Analysis of an Adaptive Generation Algorithm.* We use SIPS to design and analyze an adaptive generation algorithm. The algorithm relies on an API named `Par-R` to use any sequential PRNG implementing `R` as the underlying generator. The main feature we explore is the ability of a given sequential pseudo-random number generator to jump ahead n terms on the generated stream of random numbers to directly produce term $n + 1$ at least as fast as the successive generation of n numbers before generating term $n + 1$. Whenever the adaptive algorithm changes from sequential to parallel implementation, what occurs on successful steals performed by the scheduler, a jump is performed for pairing the disjoint sequences between workers. Thanks to bounds provided by SIPS we are able to build two fast algorithms: one *work-efficient*, when the provided jump complexity is linear, and one *work-optimal*, when the jump complexity is at least logarithmic.

Chapter 8. *Algorithms & Benchmarks.* This chapter provides experimental evidence that the asymptotic limits shown previously do not excessively penalize the execution with their hidden constants and if they are competitive with state of the art parallel PRNG DotMix. DotMix relies on *pedigrees*, thread-unique numerical labels, a feature its authors persuaded Intel to include in its Cilk Plus parallel framework implementation. We compare our generic solution with a tailored design and reason about the abstraction penalty of using generic PRNGs. Along with parallel generation we implement other adaptive algorithms: introspective sort (MUSSER, 1997), randomized Fibonacci (LEISERSON; SCHARDL; SUKHA, 2012), and a maximal independent set of vertices in a graph (FERREIRA; SCHABANEL, 1999). As sequential underlying generators we use Boost Library versions for classical Mersenne Twister (HARAMOTO; MATSUMOTO; L'ECUYER, 2008), Tausworth (L'ECUYER, 1996), and Linear Congruential (KNUTH, 1997b), along with our own implementation for the crypto generator Blum Blum Shub (BBS) (BLUM; BLUM; SHUB, 1986). SIPS is used to analyse the algorithms in work-efficient and work-optimal versions (when the generator is suitable) and a direct makespan comparison is made against DotMix, with a competitive performance.

The thesis ends with a conclusions chapter:

Chapter 9. *Conclusions.* We summarize the thesis and trace correlations between its topics and our current research. Points for improvement are delimited and, from those, future works are established, like unification of work-efficient and work-optimal algorithms into a single procedure, the development of a theory for semiassociative operations, or combining our solutions with the use of pedigrees from DotMix. Finally, we take a position about future trends in the area.

Finally, there is an appendix:

Chapter A. *Expanded Background.* This appendix expands Chapter 2. The informed reader may skip its parts in conformance to his previous knowledge on the topic. Its contents are aimed at the reader not familiarized with multithreaded parallel programming and scheduling theory. It details concepts that while not wholly pertinent to the contents in depth, may be useful when analyzing the topics in breadth.

All chapters present a brief introduction to their theme and outline at the beginning and a section called "Closing Remarks", summarizing its contents, at the end. Some chapters will have an extended closing remarks section, which expands and comments the bibliographical references on that chapter on a historical and interconnected perspective. This optional extension is aimed at giving the reader contextual inspection of the chapter's content.

1.4.2 Conventions

All experiments in the chapters ahead are performed on a machine called "Turing". It is an Uniform Memory Access (UMA) machine owned by the Grupo de Processamento Paralelo e Distribuído (GPPD) (UFRGS, Brazil):

- Operating System: Linux 3.2.0-40-generic #64-Ubuntu SMP x86_64.
- CPUs: Intel Xeon X7550 2GHz ×32 (2 thread per core), data cache of 32K, instruction cache/levels of 32 KB/256 KB/18,432 KB.
- Memory: 132,018,988 KB.

Benchmarks are written in C++11. The language is widely used to write multicore/multithreaded middleware, like Cilk, OpenMP and Threading Building Blocks, all discussed ahead. The concepts are not directly presented in C++11 to spare the reader from unimportant implementation details. Instead, we use a dialect derived directly from it. This dialect will be presented gradually, guided by use. Nonetheless, all language-specific resources are also readily available in C++11, and the code runs over an updated C++ compiler as long as adequate macros are provided. Deriving C++ allows us to implement efficient yet abstract algorithms. This way, it enables us to reason accurately about the impact of the algorithms in actual machines, compiler-wise and architecture-wise, without losing abstraction facilities.

We employ generic algorithms, procedures around a family of types with standard mathematical properties, to glue together different components of the program. Through generic programming, efficiency and further desired properties are kept whenever the middleware fulfills a (rather small) set of requirements. It provides orthogonality between data types and algorithms by removing type constraints in favor of "concepts", a family of types. A concept is defined as a set of requirements to a type. Fundamentally, concepts are to types what types are to values. By correctly specifying a concept — *e.g.*, "binary integer" — we make our algorithms support a myriad of types like **`int`**, **`long int`**, **`gmp_int`**, *etc.* (This is the base rationale to define a generic API at Chapter 6).

As example of code writing, let us examine the implementation of the calculus of the n-th term on Fibonacci's sequence, $\{0, 1, 1, 2, 3, 5, 8, \ldots, \}$. In it, each term is the sum of the two previous, except for the two first ones; the zeroth term is zero, and the first term is one. A recursive naïve implementation:

```
1   concepts <Integer N>
2   fib (N n) -> N
3   // precondition : n >= N (0)
4   {
5       if (n < N (2)) return n ;
6       N x = fib (n - N (1)) ;
7       N y = fib (n - N (2)) ;
8       return x + y ;
9   }
10
```

This listing:

1. On line 1 declares a concept called `Integer`, modeling integer numbers, and states `N` as the concrete type with operations defined by integer arithmetic. That means the user can invoke the function with *any* type for `N` as long as it has defined:

   - assignment (in terms of copy construction),
   - total ordering,
   - integer arithmetics.

   It is valid for types like **`int`**, **`short int`**, and **`char`** and even for objects like `mpz_t`, the integer type of large number library GNU GMP. The return type of the function is deduced at compile time because it is the same of at least one of the parameters. A user simply do something like `{ int a = 5 ; int b = fib (a) ; }` and the compiler generates code identical to the one we wrote, but without the first line and with all occurrences of `N` replaced by **`int`**. (This mechanism is the same as C++'s

`template` for the familiarized reader.)

2. On line 2 declares the function signature itself. Its name is `fib`, receives as parameter (by value) a value named `n` of integer type `N` and returns an unnamed value also of integer type `N`. In mathematical notation, $\mathit{fib} : \mathbb{N} \to \mathbb{N}$.
3. On line 3 declares its preconditions, *i.e.* the facts over the inputs that must be true in order to allow the algorithm to work properly. (It is possible also to declare postconditions — that always hold after the algorithm ends — and invariants — that are always true during the execution of the algorithm.) The declarations are made in form of comments because even though some conditions are easy to assert on runtime (*e.g.*, "is positive"'), others are impossible (*e.g.*, "has inverse") or could demand exponential time (*e.g.*, "is prime"). All preconditions must be true for the algorithm to work properly (it is assumed a logical conjunction between preconditions). Whenever a program violates a precondition, the correspondent behavior is unpredictable. At this instance, it would return the value without any calculation — whether this is useful or not is at judgment of the programmer. The example at hand could demand a `NonNegativeInteger` instead of `Integer` as a concept, but it would not allow the program to use types that allow negative values — like **`int`** — even if we used only its positive values. In this case we exchange a small degree of safety for a large amount of flexibility. This is a general rule we follow to accept as many types as possible.
4. On lines 4 to 9 lies the *de facto* implementation. Line 5 returns the index itself if the input is zero or one, because the index and value coincide in this case. The `N(2)` is a type casting (written in a function call style) that will convert the input from whatever type the compiler assumes for the literal `2` and returns the correspondent value on the integer type `N` in compile time. We avoid implicit casting both because its rules are complex — sometimes inconsistent — and because this enables us to describe algebraic structures — *e.g.*, we may write the neutral element of monoid type `T` as `T(0)`. Lines 6 and 7 declare two variables of integer type `N` and initializes it with the correspondent recursive Fibonacci calculations. The compiler does not perform an assignment in this case (unless the programmer re-defines operator `=`) but uses *copy construction* to calculate the value "in-place". Finally, on line 8 the value is returned by copy, unless the compiler finds out it is safe to return the value directly on an eventual target variable — *e.g.*, when the function appears on the right side of an assignment.

We count on abstract code to be converted into concrete, parametrized code at compile time. This implies in no overhead to the execution.

The presented algorithms and data structures are independent of the multicore middleware. Few underlying primitives are necessary, all present in current libraries. Nonetheless, those primitives can be easily abstracted.

1.4.3 Principles

This work follows a set of principles, sorted by order of precedence:

Reproducibility. All results and methodologies must be reproducible by the reader. All source code, libraries, binaries and manuscripts are readily available to download at <http://www.inf.ufrgs.br/~sdkmor/Thesis>. Besides the program, a set of correctness tests and benchmarks is also provided. In addition, third-party libraries for statistical evaluation of results provides transparency; *e.g.* the GNU R scripts are also available.

Focus on performance. Execution time (in number of operations or physical time) is the primary criteria for comparison. We compare absolute time against relative time whenever possible.

Faithfulness. A parallel program should respect the semantics of the sequential version whenever possible. It is allowed to deviate in exceptional cases, *e.g.*, associativity for parallel reduction may not be required in sequential. This includes (but not restricts to): side-effects or their absence; interfaces; parameters and output format; determinism from outputs and execution. Parallel code shall be made as simple as possible, introducing the smallest possible set of changes to the sequential programming source code. The notion of *elision* code is useful; if one removes parallelism directives what remains is valid sequential code. It is implemented in multithreaded middlewares OpenMP and Cilk Plus through compiling directives.

Abstractness. The algorithms should perform over a set of similar types whenever those types provide the required mathematical properties. For instance, integer-based algorithms should run over any types defined over integers' arithmetic, like **`short int`** or `size_t` of C or `Integer` of Java.

Efficiency-wise, we follow the sub-principle that one shall not pay for resources it does not use. This guides our designs in various ways that will be detailed at the proper time. For instance, no parallelism overhead on sequential execution, even for parallel code (in implementation of adaptive algorithms); overhead is moved to operations on the critical path and not on parallel work (the work-first principle); and lock-free (HERLIHY; SHAVIT, 2008) implementations are always preferable.

1.5 Institutional

This thesis is directed by joint supervision between the Brazilian university UFRGS and the French university UGA. It is directed by

- Nicolas Maillard, PhD., academic and administrative director at the Brazilian side. Associate Professor/Researcher at Programa de Pós-Graduação em Ciência da Computação (PPGC) / UFRGS. Head of International Relations Office at UFRGS and co-director of the International Laboratory on High Performance Computing and Environmental Informatics (LICIA).
- Jean-Louis Roch, PhD., academic director at the French side. Associate Professor/Researcher at École d'Ingénieurs Mathemátiques Appliquées Télécommunications (ENSIMAG) / Institut Polytechnique de Grenoble (Grenoble-INP). Project Leader of the Parallel Algorithms, Programming Models, Scheduling and Interactive Computing (MOAIS) Team, associated to Laboratoire d'Informatique de Grenoble (LIG), INRIA and UGA.
- Bruno Raffin, PhD., administrative director at the French side. Research Scientist (CR1-HDR) at MOAIS Team, associated to INRIA.

The thesis is written in the context of a long-lasting relationship between French and Brazilian laboratories in Porto Alegre/Brazil and Grenoble/France. This partnership was consolidated in 2010 with the creation of LICIA, a joint laboratory between the two countries for qualification and mutual scientific cooperation between researchers from both sides.

The work was funded by three distinct scholarships throughout its development, from several agencies:

2010-2011. Doctoral scholarship from Coordenação de Aperfeiçoamento de Pessoal de Ensino Superior (CAPES)/Brazil.

2011-2012. Doctoral scholarship from Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPQ)/Brazil.

2012-2013. Doctoral scholarship from The French Ministry of Foreign Affairs and International Development/France (Eiffel Excellence Scholarship Programme).

2013-2014. Doctoral scholarship from CNPQ/Brazil.

We thank Brazilian agencies CAPES and CNPQ for the support. It is highlighted that the student was selected as an *Eiffel Laureatte* from the Eiffel Excellence Scholarship Program, awarded by the French government, that we also thank.

Finally, several research missions Brazil-France were taken during the Ph.D. thanks to funding of two projects:

- *Équipes Associées.* From INRIA, provides funding to INRIA teams in order to interact with high-level laboratories around the world.
- *High-Performance Computing for Geophysics Applications (HPC-GA).* The HPC-GA project gathers an international, multidisciplinary consortium of leading European and South American researchers featuring complementary expertise to face the challenge of designing high-performance geophysics simulations for parallel architectures.

1.6 Closing Remarks

Most of our assertions over the evolving of parallel computing until the beginning of the decade is corroborated by a famous six-year-old paper from Asanovic *et al.* (ASANOVIC et al., 2009). It states, in its introduction, that prior to 2009 an implicit contract existed in which programmers would keep sequential semantics in exchange for an increase in the performance guaranteed by hardware people. Transistor number and power consumption would be neglected in favor of this “gentlemen’s agreement”. This paper not only gave an accurate view of parallel programming up until 2009 but also assembled a forecast of what would come for the next five to ten years after its publishing.

This chapter restates the current hardware trend to be still following Moore’s Law. It was stated in the 19 April 1965 issue of Electronics magazine, within the article “Cramming more components onto integrated circuits.” by Gordon E. Moore, director at Fairchild Semiconductors. He was asked to predict what would happen over the next 10 years in the semiconductor components industry. His article speculated that circa 1975 it would be possible to cram as many as 65.000 components. Robert Schallar performs an in-depth analysis of the impact of Moore’s Law in a 1997 paper (SCHALLER, 1997).

At *p.* 58 of this document there is a discussion about Moore's Law uniqueness and possible applications to other fields or knowledge, with an insightful comparison to aerial transportation. The paper also explains, in *p.* 54, the log-linear extrapolation done by Moore.

The discussion issued at the end of the parallel computing survey section about synergy between performance and throughput is discussed in the book by Patterson and Hannesy (PATTERSON; HENNESSY, 2008) when arguing about processor pipelines on Section 4.5 — where the term "speed-up" (not to be confused with the speedup concept defined on Chapter 2) is used interchangeably with the term "performance". Their "laundry" analogy for pipelining is insightful in illustrating the differences between the concepts. The third edition of the book took the question in depth, proposing an elucidating exercise on the differentiation between performance and throughput and showing their synergy.

Generic Programming is a facility approached in details by Stepanov and McJones in their 2009 book, *Elements of Programming* (STEPANOV; MCJONES, 2009). The most influential chapters in this thesis are the first, *Foundations*, because of the notion of Regularity; the second, *Transformations and Their Orbits*, because of the unexpected link between fixed-point functions and PRNGs; and the sixth, *Iterators*, because it is how we describe a polynomial on Chapter 5 and a list of random numbers in Chapter 7. We do not employ its theory in full range, but we too use a simplified version of a real programming language to expose implementations and on the writing of algorithms in terms of conceptual types. Not only it abstracts the reader from details like the number of bytes of a floating point type, but it also exposes the mathematical underlying concepts that the algorithm requires to work properly. Nevertheless, we must state that abstracting these concepts is not at the price of losing efficiency. On the contrary, since type instantiation is made at runtime, the implementations have equivalent performance to hand-written code with a given target type. Also, by using a subset of C++ we can discuss architectural influences on implementations and necessary optimizations a compiler is supposed to apply.

2 BACKGROUND

This chapter provides mandatory background concepts on concurrent programming for the topics discussed next. Here we discuss the scenario on which we operate, introducing topics like parallel machine models, programming artifacts, DAG, and scheduling. At each section, we begin by picking the interesting concepts the reader should have in mind to progress in the thesis and expand it in a comprehensive yet trimmed fashion.

A stack-based approach walks from parallel machine architectures at the bottom to parallelization of algorithms on the top. The concepts are layered as displayed in Figure 2.1. Each layer is connected to the other by an abstraction level, where the concept at the bottom provides primitives and hide implementation details from the top. On the right side, we have domains, displayed as stacked boxes, each corresponding to a subsection of this chapter. The domains are grouped in sections, two-by-two, represented by linked edges named accordingly.

The first part (Section 2.1), Underlying Machines, describes the parallel hardware, both physically (Parallel Machine Architectures) and logically (Parallel Machine Models). We focus on streamlined concepts to the standard reader of concurrent programming-focused works. This topic is expanded in further details to the less familiarized reader on Appendix A.

The second part (Section 2.2), Foundations of Parallel Programming, provides uniformity of notation for the execution of parallel programs (Parallel Execution Models) and the central problem (Scheduling) for the reasoning ahead. Since the definitions in it correspond to the axioms employed on Chapter 4, the respective blocks are marked with bold lines.

The third part (Section 2.3), The Art of Writing Parallel Programs, contains classical parallelization techniques for sequential algorithms (Parallelization) and a description of the selected parallel runtime to display our rationale, Cilk Plus (Middlewares and Runtimes). Both topics are not stacked because there is no hierarchical coupling; it is fundamental for a programmer to combine parallelization techniques and a chosen middleware in harmony. A brief description of the most popular parallel programming frameworks (middlewares) and further information on parallelization is available on Appendix A.

Although the Algorithms domain should also be described in the third section, being the central part of the thesis, it is progressively developed in the remaining chapters — for this reason, it is marked with dashed lines.

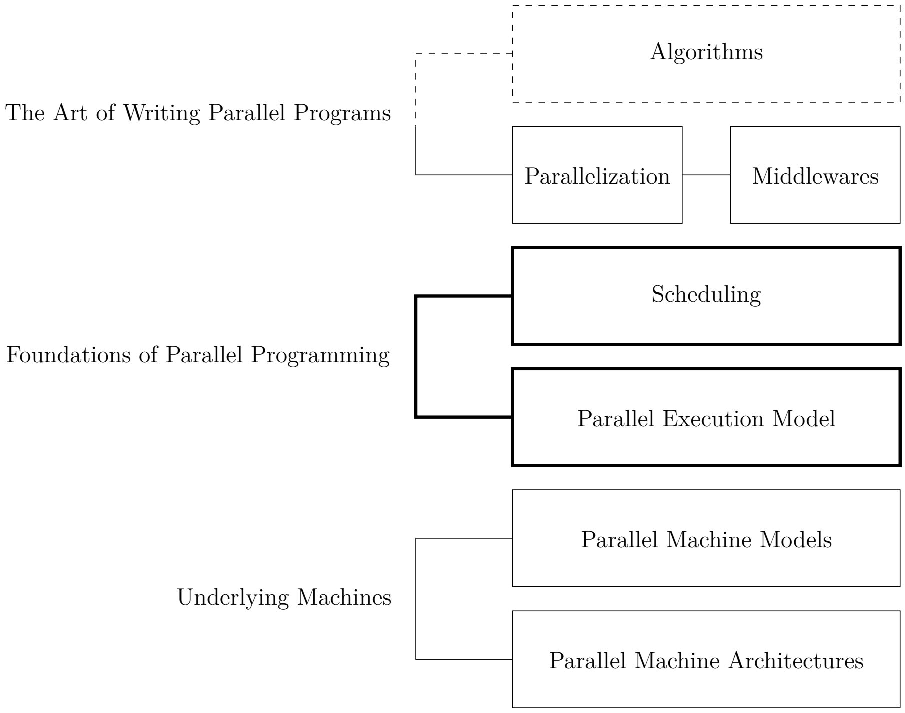


Figure 2.1: Stack of background requirements for the thesis. Bold lines denote mandatory reading. Dashed lines denote concepts expanded in further chapters.

The chapter ends with a more detailed bibliographical analysis of its contents (Section 2.4).

Henceforward the domains are discussed in bottom-up order.

2.1 Underlying Machines

The fast evolution of hardware requires abstractions to be used in scalable software. First, one elaborates ideas over theoretical virtual machines that should hold properly in real machines. Second, one shows empirically that its solution indeed keeps desired properties, like correctness and performance. Algorithms to be decoupled from the underlying hardware results in facilities for abstraction. Nevertheless, software does not run in thin air. Abstraction and performance are not by all means dichotomic, even if not always simultaneously optimal. To achieve efficient abstractions, however, implies knowing implementation details. Thus, even abstracted parallel algorithms need to consider elementary aspects of their underlying structure. For this reason, we start background analysis by a taxonomy of parallel hardware. We then generalize the device concepts in an (implied) theoretical machine that allows efficient abstraction.

2.1.1 Parallel Machine Architectures

Parallel machine architectures are the lowest level of our hierarchy, being the hardware on which we operate.

We work over *shared-memory MIMD*, following Flynn's taxonomy (FLYNN, 1972). General-purpose machines are usually built around one or more multicore processors. (Nowadays, not only the CPU but also the Graphical Processing Unit (GPU) and accelerators fall in this category.) Shared-memory communication subtracts much concern about remote memory access, latency, and protocols that are not related to the techniques we want to display.

MIMD machines specialize in two types (PACHECO, 2011):

Shared Memory. There is one global memory shared by all processors. In the case of SMP, identical interconnected processors share the same memory, although each one may possess its own private cache. Access time subdivides the class. UMA machines deliver the same access time for any memory location. It is the case of Multiprocessor Systems (MPSoC) and Multicores. Non-Uniform Memory Access (NUMA) machines offer different access times for different memory locations. Access to local memory is faster. The access time differs among different remote parts.

Distributed Memory. Each processor has its own private memory. Machines have the notion of local and remote memory. There is no guarantee on memory access time. Communication latency and dedication of resources subdivide the class. Clusters are dedicated homogeneous machines interconnected by a fast Local Area Network (LAN) (*e.g.*, Myrinet). Grids are part-time heterogeneous machines interconnected by an Wide Area Network (WAN). The original idea was to incorporate an abstract processing power on volunteer computation from idle resources. However, the *de facto* implementation of Grids nowadays is a group of heterogeneous clusters interconnected. This concept eventually evolved as a part of the concept of Cloud Computing, defined by Google *circa* 2000.

More recently, machines incorporated some hybridism in this schema. For instance, there are clusters whose nodes are a composition of MIMD CPUs and SIMD GPUs. The European project Grid-5000 is such an example.

The physical design of these systems may not match the logical layout. Parallel programming models and their proper runtime make the bridge.

2.1.2 Parallel Machine Models

A model express the underlying logical machine the programmer sees when programming. It defines data partition, execution flow, synchronization, communication, *etc.*

A hardware model usually induces a logical model, due to negligent overhead for converting logical commands into physical commands. However, there are exceptions like Distributed Shared Memory (DSM), the Go programming language, and remote memory access.

Models based on hardware constraints are similar to the ones shown in the previous section:

Shared Memory. General purpose machines with shared address space in memory. All processors may write and read all memory words, independently and asynchronously. Particular executions flows are defined by processes — mainly their lightweight descriptor, the thread — a descriptor for a flow inside a process. It holds a computation's state, but not shared raw data. The state is defined in low-level abstractions by concepts like the program counter, registers, and flags. Communication are performed through shared variables in the global memory space. The runtime may guarantee the existence of private local variable to each thread, either explicitly or through programming language mechanisms. A process can usually wait for other processes to progress through synchronization directives. Punctual variations on the order of access to shared variables may result in execution inconsistencies — a situation known as "race condition".

Distributed Memory. General purpose machines with distinct address space in memory. The communications system unfolds parallelism, used to share data (input, partial computations, or output). Communications are peer-based, although the middleware may, oblivious to the programmer, use circuit pathways to both overcome interconnection limitations and increase performance. Two major types families of communication primitives:

- **Point-to-point.** One processor communicates with another processor directly. Central primitives are in send/receive style. There are others, however, like set/get or read/subscribe.
- **Collective.** Generalizes many-to-many communication types. Primitives are frequently tied to conventional concurrent algorithms procedures like broadcast

(one process sends message to all others), all-to-all (all processes send data to all others), scatter/gather (shares and groups data), and reduction (one process combines sub computations made by others).

By adding constraints to the general models above we deliver frameworks for theoretical validation of parallel algorithms. The interested reader will find in Appendix A a discussion over two of such models: Parallel Random-Access Machine (PRAM) (KRUSKAL; RUDOLPH; SNIR, 1990) and LogP (CULLER et al., 1993).

We work over *shared-memory parallel machines*. By choosing a logical model identical to the hardware we extract maximum efficiency without loss of abstraction to the algorithms we are interested. The usual approach for modeling theoretical machines on multicores is PRAM since it is useful to demonstrate bounds on algorithms. However, the kernel of our work is a theoretical device that enables us to deliver precise bounds *even* when the workers are completely asynchronous. In this sense, we opted for the least restraining model constraint to serve as our abstracted machine.

2.2 Foundations of Parallel Programming

We discuss parallel programming “in a nutshell”. Contents are organized and distributed to provide to the reader a minimal common ground. Next we discuss the representation and execution of parallel programs over an abstracted hardware.

2.2.1 Parallel Execution Model

Sequential algorithms express their execution time through a function on input’s length. Although there are other executed-related factors as caches, OSs, and memory access, this is the dominant cost.

Simplified execution models exist for accounting of fundamental operations performed by parallel algorithms. Ideally this execution models are simple enough to be analyzed but sophisticated enough to model relevant aspects of the computation. They provide a comparison tool for some selected criteria.

Execution models are not only useful to compare parallel and sequential implementations, but also distinct parallel ones. They also provide a proof system for complexity statements like optimality, lower/upper bounds, and asymptotic growth.

We use definitions from a shared memory machine model to model an asynchronous system. Our results do not depend on the guarantees offered by models like PRAM. (Nevertheless they hold since those are more constrained models.)

We begin by defining a glossary of definitions that serve as the building blocks for all thesis. First, the most fundamental entities:

Task. A task is an indivisible set of machine instructions. Consumes an input and generates an output. Two tasks can be executed in parallel unless related by a sequential dependency. Tasks are performed by *workers*.

Worker. An entity that executes/consumes a task. A synonym for execution unit: resource/processor/thread/*etc* (it is context-dependent). A worker is *inactive* when it is idle and *active* otherwise.

Top. A top is a totally ordered time stamp – represented by an integer — regarding the execution of a parallel program. Current top is denoted s, previous top $s-$ and next top $s+$. The top before the execution is 0, and the first top is 1.

Now we define classical parallel programming notation in terms of those entities (JAJA, 1992; CASANOVA; LEGRAND; ROBERT, 2008):

Sequential Work. We denote it W_{seq}. It is the number of tasks of the best sequential algorithm. We also use the notation $W_{\text{seq}}(n)$ to make the relation between the input length n explicit. Also referred as "work" of the sequential algorithm or "sequential work" of the parallel algorithm.

Sequential Time. We denote it T_{seq}. It is the execution time of W_{seq} expressed in number of tops.

Parallel Work. We denote it W. It is the total number of tasks of the parallel algorithm. Also referred as "work" of the parallel algorithm or "parallel work" of the algorithm. It accounts extra operations from straightforward parallelization. We also use the notation $W(n)$ to make the relation between the input length n explicit. For instance, initialization of middleware, conditional branching, indexed access, *etc*.

Parallel Overhead. We denote it V. It is the sequential overhead introduced by synchronizations on the parallel algorithm. It represents the extra operations accounted for the parallel work. Also referred as "overhead" of the parallel algorithm or "synchronization overhead" of the algorithm. We also use the notation $V(n)$ to make the relation between the input length n explicit.

Parallel Time. We denote it T_P. It is the execution time of W with P workers.

The parallel execution is represented as an abstract container of tasks in the task-based parallel programming model. By abstract container we describe a data structure that represents the parallel computation but it is not necessarily addressable by the program. It can be generated *a priori* (through code analysis) or *a posteriori* (through trace). A representation of it may be kept by the runtime during the execution to make decisions about operations that may impact performance. *e.g.*, to resolve scheduling dependencies.

The dependencies between tasks are made explicit. There is a twofold implication. First, one can reason efficiently about the proposed parallelization. Second, one can derive notions not self-evident without a representation of dependencies, such as the critical path (which we explain more ahead). Thus, it allows one to reason about central topics, such as scheduling, synchronizations and overall parallel multicore executions.

We use a precedence DAG (PACHECO, 1996; PACHECO, 2011) as the logical representation of the execution where:

$$\begin{aligned}\text{Nodes} &= \text{tasks},\\ \text{Edges} &= \text{sequential dependencies between tasks.}\end{aligned}$$

To exemplify, consider the calculus of the distance between points $p_1 = (a_1, b_1, c_1, d_1)$ and $p_2 = (a_2, b_2, c_2, d_2)$ on a $\mathbb{R}^4$ Euclidean Space:

$$x = \sqrt{(a_2 - a_1)^2 + (b_2 - b_1)^2 + (c_2 - c_1)^2 + (d_2 - d_1)^2}$$

On Figure 2.2 we display one *possible* precedence DAG describing the computation of this formula. In order to hold temporary values we use four auxiliary variables: a, b, c, and d. For illustrative reasons the tasks are numerically labeled accordingly to a valid sequential execution, but this is not mandatory. An edge connecting nodes i and j means “task i must execute before task j”, or, conversely, $s(i) < s(j)$. The constraint may be given by factors as the correctness of the algorithm, memory synchronization (as in the lock-free algorithms), *etc.* In this case, the dependency is established due to a necessary order in reading and writing of variables. For instance, values of variables c and d in task 5 must first be squared on tasks 2 and 3, respectively.

A precedence DAG generalizes the intuition of sequential dependency. Therefore, it delivers the notion of *ready task*, a task whose all precedent tasks were already executed. If execution of i makes j ready, we say “j is enabled by i” or “i enables j”. In Figure 2.2 tasks 4 and 5 enable task 6.

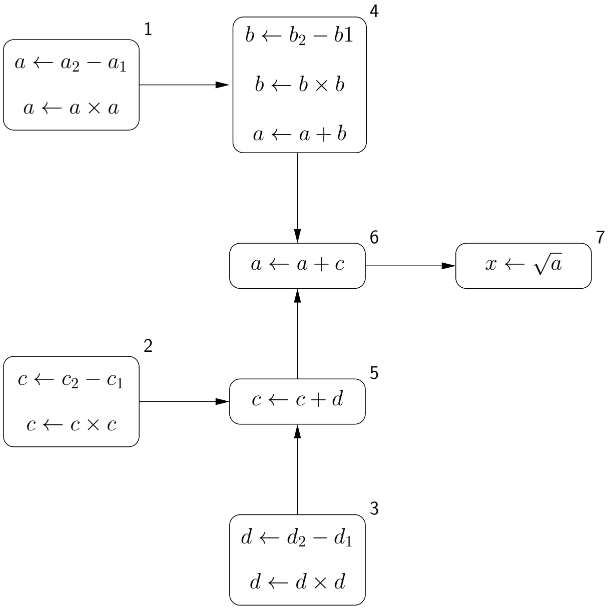


Figure 2.2: Possible execution DAG for the distance between two points on a $\mathbb{R}^4$ Euclidean Space. Nodes are the tasks and edges the sequential dependencies between them.

We call *sources* of the DAG all nodes without an ongoing edge. We call *sinks* of the DAG all nodes without an outgoing edge. The sources begin the computation readies. The sinks are the last tasks to be executed and deliver the output of the computation. In Figure 2.2 tasks 1, 2, and 3 are sources, and task 7 is the only sink.

Execution DAGs enables us to define the depth of a parallel computation (JAJA, 1992):

Depth. We denote it D. It is the longest path in the DAG from any source to any sink. This longest path is named the “critical path”. It is equivalent to the execution time with an unbounded number of workers, and can be also denoted by T_∞.

We highlight that tasks do not all have the same size or take the same time to execute. For instance, tasks 5, 6, and 7 each performs two high-level programming instructions (one arithmetic and one assignment). Tasks 1, 2, and 3 all perform four instructions. Task 4 performs three programming instructions. Nonetheless, even the number of instructions is not a precise estimate of time a task takes to execute. Although tasks 5, 6, and 7 all have the same number of instructions, tasks 5 and 6 perform an addition and then an assignment, while task 7 performs a square root operation, what is typically more expensive than additions on modern machines.

The notion of determinism may also be defined in terms of DAGs as well. A parallel computation is: *deterministic* if its DAG is fixed for a given input and *non-deterministic* otherwise. A DAG may change for the same input, for instance, when the algorithm spawn tasks obeying random criteria.

We now define the notion of speedup, enabling us to compare different parallel algorithms (CASANOVA; LEGRAND; ROBERT, 2008). Speedup is the ratio between best sequential time and the obtained parallel time when solving the same problem:

$$\text{Speedup} = T_{\text{seq}}/T_P. \tag{2.1}$$

Equation 2.1 measures parallelism gain. The alternative ratio Speedup $= T_1/T_P$, where T_1 is the time of the parallel version with one worker, may be used to measure scalability. If Speedup $= P$, we name it *linear* speedup. If Speedup $> P$, we name it *superlinear* speedup. Superlinearity is achievable, for instance, by using algorithms with some randomness in either its code or in its input (*e.g.*, find the first occurrence of an element of an array). Hardware optimizations like cache also allow this type of observation.

The speedup is a useful tool to compare parallelization of the same sequential algorithm. The reader must be aware, however, that different parallel/sequential ratios are not comparable. The sequential algorithm can be, even unintentionally, penalized and, thus, the comparison would be artificially improved.

Unfortunately, linearity is not a lower bound for speedup. Amdahl's Law (AMDAHL, 1967) makes a correlation between the portion of a program that can be parallelized and the maximum speedup obtained. For instance, if a program needs 20 hours using a single worker, and a particular portion of the program that takes 1 hour to execute cannot be parallelized, while the remaining 19 hours (95%) of execution time can be parallelized, then regardless of how many workers are devoted to a parallelized execution of this program, the minimum execution time cannot be less than 1 hour. Hence, the speedup is limited to at most 20×. Formally, considering $B \in [0,1]$ the fraction of the algorithm that is strictly sequential, the time an algorithm takes to execute using P workers over an input of size n is

$$T_P \leq T_1 \left(B + \frac{1-B}{P} \right)$$

and the maximum theoretical speedup would be, therefore,

$$\text{Speedup} \leq \frac{T_{\text{seq}}}{T_P} = \frac{T_{\text{seq}}}{T_{\text{seq}} \left(B + \frac{1-B}{P} \right)} = \frac{1}{B + \frac{1-B}{P}}$$

2.2.2 Scheduling

A step-by-step precedence DAG execution may be obtained by updating the DAG at each top. A ready task without in-edges that is executed is taken out of the graph along with its out-edges (KUMAR, 2002; HERLIHY; SHAVIT, 2008).

Let us consider two workers. At top $s = 0$ the execution has not begun yet, and the workers are idle, what is denoted by "$-$". Thus, a possible assignment in Figure 2.2 top-by-top is:

Top s	0	1	2	3	4	5
Worker A	-	1	2	5	6	-
Worker B	-	3	4	-	-	7.

In this small example, we assigned tasks to workers by convenience. However, a computational system must follow established criteria and follow an algorithm for scheduling task to workers on each top. Thus, we define:

Scheduler. Is an algorithm that assigns tasks to workers on each top.

A scheduler is also known as "workload balancer".

For P workers and n tasks, there are two possible scenarios during a parallel execution:

1. $n \leq P$, trivial scheduling. Single programs with fewer tasks will share the workers with other programs for maximum resource profiting.
2. $n > P$, non-trivial scheduling. Programs with a large number of tasks, on a dedicated environment. usually, $n \gg P$, known as "parallel slackness".

If the number of tasks is fixed, a *static* scheduler operates *a priori.* Nevertheless, the runtime may be able to create new tasks as the computation progresses. In this case, a *dynamic* scheduler operates. If a dynamic scheduler has to manage a large number of tasks at once, it may impact in non-negligible management overhead due to, *e.g.*, data copy, synchronization, manipulation of meta-task data structures, *etc.* The scheduler is usually provided by the runtime, although implementations may embed it inside their algorithms.

Next we introduce an important class of schedulers, *greedy schedulers.*

Definition 1 (Greedy Scheduler.) A greedy scheduler is any scheduler where the following holds: if there are n *ready* tasks and P *idle* workers in top s,

1. If $n > P$ at s, then exactly P are executed at $s+$.
2. If $n \leq P$ at s, then exactly n are executed at $s+$. □

Graham (GRAHAM, 1969) and Brent (BRENT, 1974) provide a lower-bound for any greedy scheduler, whose proof derives from Definition 1:

Theorem 1 (Graham and Brent's Theorem) *For any parallel computation with work W and depth D, and for any number of workers P, any greedy P-worker execution achieves*

$$T_P \leq \frac{W}{P} + D. \tag{2.2}$$

Proof In Definition 1, condition (1) may happen at most W/P times. Conversely, condition (2) may happen at most D times. Thus, $\frac{W}{P} + D$ is an upper-bound to execution time. ■

Theorem 1 directly derives the centralized list scheduling algorithm, also known as "busy-leaves" (BLUMOFE; LEISERSON, 1999). In that, all source tasks are initially placed on one worker. All workers share a list of tasks. Active workers that produce tasks put them on the list's front. Idle workers obtain tasks from the list's back. Computation ends when all workers are idle. This cycle of running, pushing, and popping is called "micro-loop" and runs inside each worker throughout the computation. The programmer writes the tasks, and the middleware deploy them to the micro-loop inside each worker. Figure 2.3 shows a flowchart for the micro-loop busy-leaves algorithm operating in each worker. Since one worker starts with the initial task, we added a dashed line from the "start" to the "run task" block. All the remaining workers start testing if the execution is over, *i.e.* if all other workers are idle. The only task that runs without being popped first is the initial task. Also, once popped, a task is immediately executed.

Centralized list scheduling is optimal within Graham and Brent's, but does not scale well. There is contention due to mutual exclusion on list's top and bottom. Distributed list scheduling mitigates this cost and stays asymptotically optimal.

Work-stealing is the distributed generalization of busy-leaves. It provides efficient scheduling for irregular parallel problems within an optimal asymptotic bound of Theorem 1. By irregular, we mean dynamic scheduled problems with no previously known pattern of task spawning. The optimal solution to this problem is in the NP-Hard class.

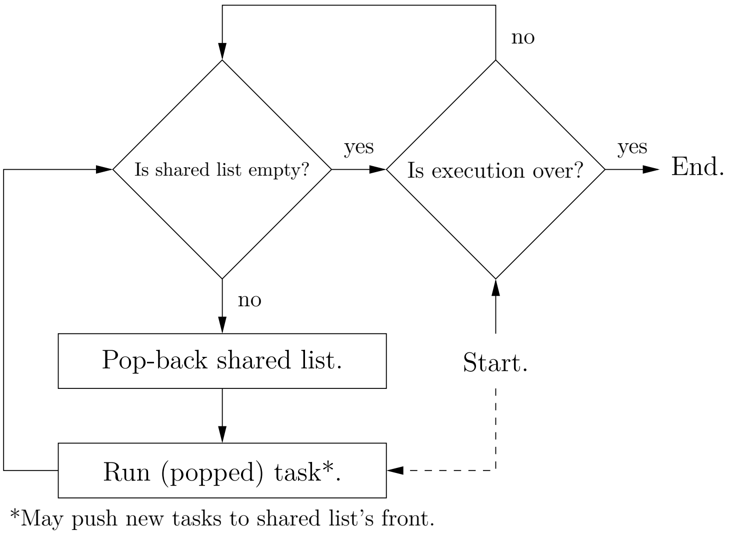


Figure 2.3: Flowchart for the micro-loop on the busy-leaves algorithm.

The work-stealing algorithm is based on the existence of local and remote task lists. Each worker owns a local list. An idle worker gets new tasks from its local list. If the local list is empty, the worker becomes a thief and selects a non-idle victim worker to get tasks from its remote list.

The local/remote list is implemented as a deque. Henceforward its ends are named front and back. Active workers push ready tasks to local deque's front. Inactive workers pop ready tasks from local deque's front as well. Thieves pop tasks from a remote deque's back. Tasks in any deque are stacked in sequential order. *Ergo*, if no steals occur, then one has strict sequential execution. The computation begins with one worker holding the initial task and $P - 1$ idle workers. The computation ends when all workers are idle.

In work-stealing, if the thief chooses an idle victim it chooses another one until the selected victim has tasks on its deque or the computation is terminated by one of the workers. Eventually, a given idle worker may become active because it performs a successful steal. In this case, it can be selected again to be a victim. This process of re-choosing, running for all workers, is named "nano-loop" and is inside the micro-loop. Figure 2.4 shows the micro and nano-loops algorithms that run on each worker, with the nano-loop components highlighted in bold lines.

Let us discuss ABP (ARORA; BLUMOFE; PLAXTON, 1998), the *de-facto* algorithm in parallel multicore task scheduling (the name comes from its authors; Arora, Blumofe, and Plaxton). We highlight three key elements of their approach: the random selection of victim, work-first principle, and non-blocking implementation.

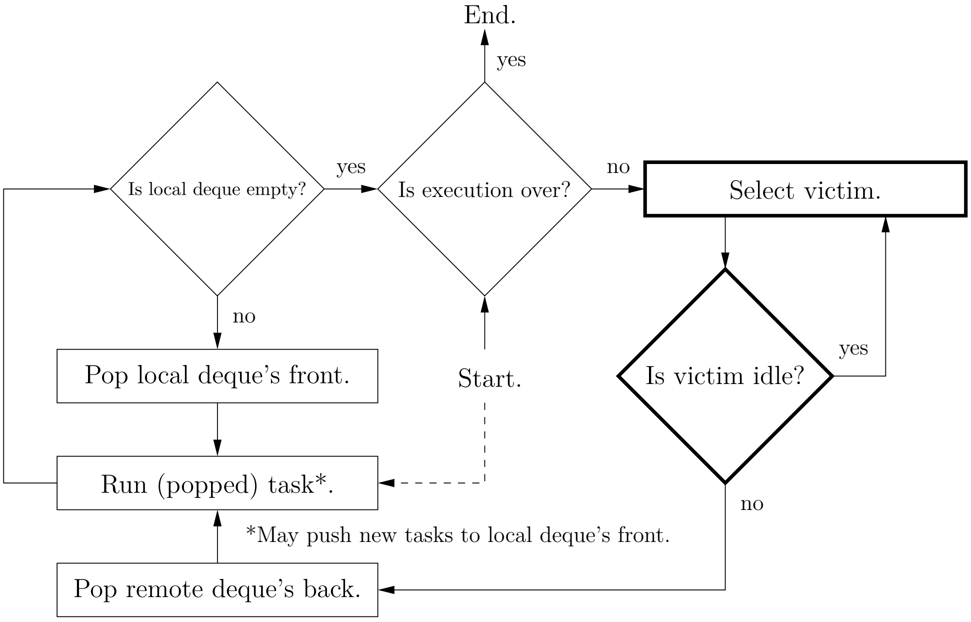


Figure 2.4: Flowchart for the micro-loop and nano-loop on the work-stealing algorithm. Bold lines indicates the boxes belonging to the nano-loop.

Random selection of victim. A thief selects its victim randomly. If the victim is also idle, retry until success or computation ends. This eliminates victim selection overhead. The expected number of steals attempts is $O(PD)$. Makespan is

$$T_P \leq \frac{W}{P} + O(D)$$

which holds resemblance with Theorem 1, except for the Big-O hiding an associated contant on the depth D.

The Work-First Principle. In work-stealing, the idle workers are the agents of load balancing. Active workers contribute to the computation itself by decreasing the workload and should not deviate from it under performance loss penalty. Therefore, the scheduling overhead should be paid by the inactive workers, contributing to the critical path overhead. This is, in general lines, the work-first principle, a heuristic for scheduling algorithms. Assuming parallel slackness and since the number of steals is proven small — in the order of PD — scheduling overhead is only paid when actual parallelism unfolds. Whenever parallel tasks are executed sequentially, they should not pay any additional costs. A concrete example of an application of this heuristic is the slow and fast clone strategy. Tasks pushed into a deque's front are replaced by a fast clone with no synchronization operations. If it executes locally, by being popped from the front, a minimum overhead is paid. However, whenever a task is stolen — popped from a deque's back — the thief acquires a slow clone of it, embedded with full synchronization and task management overhead. This lazy strategy delays effective full task creation procedure until a steal occurs and only move pointers in local execution. It implies that parallel execution with one worker is almost equal to sequential execution for large inputs.

Non-blocking implementation. The work-first principle is based on the fact that there are much more pops on a given deque's front than on its back. The parallel overhead (task creation, task management, and mutual exclusion, *etc.*) is moved to those operations. They may (in fact, should) occur concurrently and, by no means, are allowed to "lock" the computation due to mutual exclusion. Thus, it is imperative that a non-blocking protocol is followed by concurrent thieves trying to steal the same victim.

Regarding the structural dependency relation supported by ABP, its authors enumerate their implementation as fully-strict and bounded fan-out. Fully-strict communication means direct spawner-spawned ("parent-child") communication. (When communication is between indirect spawner-spawned ("ancestor-descendant"), it is called *strict.*) Bounded fan-out means that if at most k workers trying to steal a non-idle victim at the same time will get tasks; the remaining ones will have to proceed on the nano-loop.

Taking into account the limitations of the original paper, an array of extensions emerged. For instance,

GPU. Extensions of ABP to work on GPUs (CHATTERJEE et al., 2013).

Idempotent. New semantics guaranting that each inserted task is eventually extracted *at least* once-instead of *exactly* once (MICHAEL; VECHEV; SARASWAT, 2009). It is used for applications that allow for relaxed semantics, because either the application already explicitly checks that no work is repeated, or the application can tolerate repeated work.

Help-first It inverts the execution order, proceeding to run the spawner and running the spawnies after it ends (GUO et al., 2009).

We work over *greedy and work-stealing schedulers.* Greedy schedulers are optimal. Work stealing schedulers are asymptotically optimal and popular in middleware for multicore parallel programming. Since our theory is aimed at both models, they are the natural choice to go.

2.3 The Art of Writing Parallel Programs

2.3.1 Parallelization

To design a parallel solution to a given problem involves roughly two non-dichotomic options:

1. to parallelize existent sequential algorithms by removing sequential constraints where there are no actual sequential dependencies; or
2. to create a new algorithm that operates in parallel, sometimes less efficient than the sequential one for a small number of workers.

Parallelization of sequential algorithms often adds constraints to implementations, such as work grouping, data distribution, workload balancing, communication overhead, and parallelism overhead. To partition a sequential program in terms of its parallel tasks is straightforward for *simple* problems. It may be hard for complex algorithms, though.

We describe parallel programs in terms of parallel tasks, to provide uniformity to the reader. What changes between approaches is how tasks are defined and how sequential dependencies are introduced.

There are plenty of parallelization strategies regarding the identification of sequential dependencies. This is a somewhat more specific question than the task mapping schema described above. We highlight three common partition arrangements: bag-of-tasks, communicating (discussed on Appendix A) and recursion-based (KUMAR, 2002), which we approach next. There are more. In fact, within a single program one can usually find a combination of these and other paradigms.

The recursion approach uses the natural division of a program in its procedural components. If function calls are not nested, they may naturally be executed in parallel since there are no dependencies. In this case a "function" shall be implemented according to its mathematical meaning (a functional relation between two sets). A function must be preferably pure in the sense it is deterministic and produces no *collateral* effects. Some flexibility is allowed, but there must be no overlaps that produce race conditions. Synchronizations may be needed when combining the outputs of these functions. As presented, recursion-based approaches introduce sequential dependencies in a "parent-child" style. Therefore, they unfold a tree-shaped representational structure where predecessor nodes depend on the conclusion of successor nodes. However, there are systems that allow arbitrary graph dependencies to be stated.

Recursion is the paradigm we use to illustrate all the techniques we discuss in the thesis, since it allows: a simple yet expressive way to consider ordering of operations on source code; short writing of complex parallel algorithms by encapsulation; straightforward elision version of parallel source code by omitting the parallelism-unfold keywords; and correctness proofs easier to write and read. Our algorithms are simple enough to be implemented recursively. This will provide us higher-level, shorter code. Discussion benefits, since we can model our underlying interface as an algebra of functors, *i.e.*, higher-order function programming. We do not loose efficiency when doing so. Optimizations like the elimination of tail recursion at compile-time are discussed, and the middleware usually encapsulates function calls. The solutions, however, are designed to hold on other paradigms as well, with small adaptations.

Our rationale follows the simple yet powerful fork-join operations of UNIX processes:

Fork. Receives as argument a function and its parameters. Tells the runtime that this procedure should be executed in parallel along with the caller and other forked functions whenever possible.

Join. Receives as arguments references to forked functions. (May be implicit; *e.g.*, all the functions forked at the given scope.) Tells the runtime to stop the execution flow until all referenced functions have returned.

Each time we write "fork/join" we refer to this behavior, but with a specific ordering in mind: if a forked function is not immediately scheduled, the *forker* function continues execution on its worker, while the *forked* function waits until scheduled.

Nevertheless, we seldom speak of fork/join in this thesis. usually we will refer to a similar paradigm, spawn/sync. We differentiate spawn from fork by their ordering: if a spawned function is not immediately scheduled, then the *spawned* function continues execution on the current worker while the *spawner* function waits to be executed. Thus, spawn is a *preemptive* operation, because the spawner is preempted in favor of the spawned within its executing worker. Join and sync are exactly the same operation; we use one or another to be consistent with the correspondent fork or spawn.

Granularity control is perhaps the most important technique to mitigate parallelism overhead. Parallelism introduces overhead in various levels — *e.g.*, middleware management, extra tests, communication, *etc.* This cost is "paid" by the performance gain of parallelizing the program. From this, a threshold between the overhead cost and parallelization gain emerges. To achieve nearly optimal overhead mitigation, a parallel program may choose to execute a given task using the sequential algorithm if its computational cost is small enough. This cuts off the overhead for small tasks. This rationale can be applied to distinct instances for some definition of “computational cost”, “small enough” and “task”. For instance, when sorting recursively in parallel, one may call sequential quicksort over chunks smaller than the square roof of the original input size. Or, since network communication is usually the dominating overhead in multicomputers, one may choose to execute small tasks locally instead of sending it to another worker.

When speaking of granularity we will by default mean the minimal input size (acknowledging pseudo-polynomial complexity) a parallel algorithm should observe to be efficiently executed. We call a “grain” to be a task within this minimal size (the task is indivisible). We call “granularity” the measuring of grains in quantity; it has “high granularity” when the grain is small and, therefore, there are more grains. The converse adjective is being “granular”, a small number of large grains. (Here again, “small” and “large” are about the problem at hand and the gained performance, not a global evaluation of optimality.) The size of the grain is sometimes referred as the “threshold”, the bound separating sequential from parallel execution. Threshold definition may be static or dynamic. There is usually a conditional structure inside the parallel program deciding which algorithm to perform at a given step. However, to reach an optimal value for it is usually a process of trial-and-error, since it depends on the actual machine, algorithm behavior, input, *etc*. The decision-making process in each step regarding threshold between different algorithms is generalized into the notion of *Adaptive Algorithms*, discussed in depth in Chapter 5. The theory behind adaptive algorithms allows one to leave the granularity adjustments to the algorithm itself, with little to no guessing.

Simultaneous access to data may change its value non-deterministically (race conditions). To avoid it, synchronizations are used. Synchronization is a broad concept usually meaning the “sequentialization” of some parts of a parallel program to assure semantic correctness. The idea is to create mutually exclusive chunks of monoprocessed code. There are mechanisms provided by the middleware that use atomic hardware operations or software protocols (proven) to ensure mutual exclusion. For instance, there are mutexes (based on atomic locks and unlocks on a boolean variable), semaphores (based on a queue of workers trying to access a mutual-exclusion region), monitors (mutual exclusive access of procedures provided by the language/framework). Those mechanisms should be used with caution, since undisciplined use may lead to deadlocks (all process are locked in a circular way), performance issues (blocking implementations), starvation (a process waits for an unconstrained amount of time), *etc.* Distinct algorithms may not need mutual exclusion in different degrees, being lock-free (no blocking synchronizations), wait-free (no worker waits a resource) (HERLIHY; SHAVIT, 2008). Synchronizations between workers are central to our contribution. We see the implications more ahead.

2.3.2 Middlewares: Libraries and Runtimes

A runtime and the associated middleware encompass the parallel program. It provides yet another level of abstraction, creating a "virtual environment" that provides guarantees and resource management to the programmer. For instance, a runtime may be in charge of: managing distributed memory between workers; making standard parallel algorithm available for a given worker, such as "obtain my id" or "store how many workers participate in this computation"; hiding implementation issues not related to the algorithm at hand, in things like scheduling, initialization, conversions, communication. A runtime may be fine — *e.g.*, system-level threads use the actual OS to perform simple operations like interruption handling or mutual exclusion — or coarse — *e.g.*, a distributed memory process manager has to manage all execution and raw communication underneath.

Runtimes coexist and are usually accessed through dedicated software libraries. Conversely, there are libraries that rely on a companion runtime to work.

We work over Intel's *Cilk Plus* as the underlying middleware. It fits with all the previous selections we have made, is widespread in the scientific community, and allows the simpler parallelization among all alternatives. We describe other middlewares in detail at Appendix A, such as Open Multiprocessing (OpenMP) and TBB.

"Cilk" (FRIGO; LEISERSON; RANDALL, 1998) refers to a family of multithreaded runtimes whose main idea is to schedule user threads (a task) using the ABP work-stealing. The runtime works as a small extension to C and C++, adding three keywords that unfold parallelism:

1. **`cilk_spawn`**. The "spawn" primitive defined in Subsection 2.3.1.
2. **`cilk_sync`**. The "sync" primitive defined in Subsection 2.3.1.
3. `cilk_for`. Replace the traditional loop construct **`for`** by the analogous parallel one, distributing iterations over workers.

It started as a source-to-source (Cilk to C/C++) compiler. The generated program has added a runtime module (including a scheduler) that manages user-level threads on the top of OS level threads on shared memory machines. The keywords, if removed, leave valid sequential code, being also an elision framework.

The name “Cilk” alone refers to versions one to five, all hosted within the Massachusetts Institute of Technology (MIT) by Charles Leiserson’s team. In 2008, MIT’s Cilk was stalled in favor of a new version called “Cilk++”, maintained by Leiserson’s new company Cilk Arts and introducing features as automatic parallel loops and the notion of hyper-objects. Finally, in 2009 Cilk Arts was acquired by Intel and a new version, “Cilk Plus” was made available, both in closed and open source forms. While MIT’s and Cilk Art’s versions where source to source compilers, Cilk Plus is an extension built-in in Intel’s Compiler and is available as a branch of GNU’s compiler, working in a more close fashion to OpenMP.

Cilk was born as a proof of concept for ABP work-stealing. Thus, it is common to mix details of its implementation with the scheduling algorithms requisites.

Cilk is built around the work-first principle as discussed earlier in this chapter. The premise remains: most of work is sequential work, there are few steals if the work is large enough. Consequences also hold; runtime overhead are moved to steals, what occurs in O(PD) (ARORA; BLUMOFE; PLAXTON, 1998). We highlight two strategies of Cilk to conform to the work-first principle:

Non-blocking steals. It is achieved through a simplified version of Dijkstra’s THE multiprogramming system protocol for synchronization (DIJKSTRA, 1965). The deque has three pointers, T (current), H (front) and E (back). Concurrent thieves dispute E using try-lock. Only when $E = H$ no thief can steal because it remains only one element. Besides, during a pop-front operation, where $T \leftarrow H$, it is guaranteed that if $H \neq E$ then the pop will never fail.

Fast/slow clone. A spawned function executes a “fast” (no synchronizations, no copying, no context-saving nor jumping) when executed by the same worker that spawned it. Otherwise, execute a “slow” version that is meant to be performed remotely, with all due overhead. No dynamic creation is needed in both cases since both versions are created at compile time, and the framework just handle the pointers.

We show an example of the calculus of the n-th Fibonacci term on Figure 2.5 (*cf.* the Fibonacci example in Subsection 1.4.2). This code is written in Cilk Plus, what we already stated to be a small extension of C/C++. In this listing,

1. On line 1 we have the function signature. It receives and returns a C integer.

```
 1  int fib (int n)
 2  {
 3      int a, b ;
 4      if (n < 2) return n ;
 5      a = cilk_spawn fib (n - 1) ;
 6      b = cilk_spawn fib (n - 2) ;
 7      cilk_sync ;
 8      return a + b ;
 9  }
10
```

Figure 2.5: Fibonacci in Cilk Plus.

2. On line 3 we declare the variables that will serve as the output of the recursive call, `a` and `b`.
3. On line 4 we test the recursion limit and return the parameter if nothing has to be done.
4. On lines 5 and 6 we spawn child user-level threads (tasks) through the **`cilk_spawn`** keyword as recursive invocations of `fib`.
5. On line 7 we perform a join through a **`cilk_sync`** keyword, where a given user-level thread waits for the completion of all spawned threads on the current scope.
6. On line 8 we sum the outputs of the recursive calls and return it.

If the **`cilk_spawn`** and **`cilk_sync`** keywords are removed from Figure 2.5 (on lines 5, 6, and 7) the resulting code is a valid sequential code.

We highlight that code on Figure 2.5 has no control of granularity and is implemented naïvely, performing too many redundant calculations, and spawning one OS thread per recursive call. It is just for syntax demonstration, not meant for performance.

2.4 Closing Remarks

All the content in this chapter is explained in details in the textbook by Joseph Jaja, "An Introduction to Parallel Algorithms" (JAJA, 1992). It is a repository for the parallel programmer even today. In the same sense, "The Art of Multiprocessors Programming" by Herlihy and Shavit (HERLIHY; SHAVIT, 2008) is a more modern book with diverse content about parallel programming but focused on the multicore paradigm.

As nearly all books on parallel machines, we use Michael Flynn's classical taxonomy from 1972, which he introduced in his paper "Some Computer Organizations and Their Effectiveness" (FLYNN, 1972). The taxonomy itself appears briefly only on the second section and only to situate the reader in respect to its streams and other jargon. In fact, surprisingly for a hardware-based work, the whitepaper has a strong abstract mathematical setting where it relies upon. For instance, still in the second section we find an accurate modelling of the producer-consumer problem adapted to its multi-stream approach. It models requests as a pair of functions, the requester and server and differentiates the hardware and logical processing and give an order relation on it.

Also, many other classifications in Computer Science take inspiration on this combinatorial arrangement of two factors like Flynn's. In PRAM, for instance, there are also four strategies based on read and write combinations, and the third is also empty, like MISD machines! Patterson and Hanessy (PATTERSON; HENNESSY, 2008), *p.* 197, and its appendixes not only mention Flynn but also enumerate a careful list of examples in each category. Even if not specialists in parallel hardware, this book is a good reference, since it does not dissociate those advances from other optimizations on processors. They also state that there is no MISD machine. This is also the case of Peter Pacheco's textbook “An Introduction to Parallel Programming” (PACHECO, 2011), which even asks the reader about it in its Exercise 2.9 (*p.* 78).

In the case of distributed memory MIMDs, we briefly differentiated Clusters and Grids. Now we expand on their historic definitions.

The history of term “Cluster” is approached in details in a 1998 book by Gregory Pfister, then an IBM engineer, called “In Search of Clusters” (PFISTER, 1998). Pfister claims that despite DEC and IBM claims throughout the years to have invented the term and concept, neither were true. Customers, Pfister says, invented clusters, in order to gain in processing power and/or memory space, as needed at the time. The first Clusters in this spirit begun to appear by the 60's, in different places and contexts. Yet, the engineering of cluster computing as a parallel machine was probably introduced by Gene Amdahl of IBM, who in 1967 published his paper on parallel processing, Amdahl's Law, in his paper “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities” (AMDAHL, 1967). This article defined the engineering basis for both multiprocessor computing and cluster computing.

Grids, in their turn, were first defined by Ian Foster and Carl Kesselamn in their 1999's work “The Grid: Blueprint for a new computing infrastructure” (FOSTER; KESSELMAN, 1999). It is a hole book covering all aspects of Grid implementations, hardware and software-wise. It is also, however, a too abstract and comprehensive definition. While it would, arguably, be an entity where computers come and go to provide processing power world-wide, in practice implementations diverted from this concept. While there are examples of attempts at implementing the complete blueprint we only have partial models operating. Perhaps the most popular current variation is a cluster-of-clusters, where one can allocate a sharing on a cluster of machines that may enter or leave the Grid, although sparingly. Such is the case of the European Grid 5,000 project, described in a 2005 paper (CAPPELLO et al., 2005).

The fact we used the same underlying hardware model as the logical machine model (both shared memory MIMDs) may mislead the reader about the lack of differentiation at this level. This, however, is not true. Consider, for instance, the abstract C language machine model as presented in a textbook by Kernighan and Ritchie (KERNIGHAN, 1988). Even if C is considered middle level, its abstract machine is perhaps the greatest responsible for its success, since it abstracts complex hardware structures and software constructs as arrays of bytes. It also provides guarantees over this logical model, such as a valid past last position to describes its arrays (modelling a semi-open interval) in a fashion advocated by Dijkstra's work, like his book "A Discipline of Programming" (DIJKSTRA, 1997).

A standard reference for formal computation models like PRAM and models of complexity is Kruskal *et al.*'s 1990 paper (KRUSKAL; RUDOLPH; SNIR, 1990). (This Kruskal shall not be confused with Joseph Kruskal, responsible for the namesake algorithm.) On its *p.* 96, it elaborates what would be six classes of parallel algorithms in respect to its speedup and efficiency. Although those classes have never gained mainstream, they are the conducting wire that holds together the overall paper. LogP, published in 1993, is newer than Kruskal's book. For a complete reference on it, we recommend the original paper, by Culler *et al.* (CULLER et al., 1993).

The notion of explicit task parallelism is not new, although its crescent gain in popularity. We trace back to the work by C. A. R. Hoare, Concurrent Sequential Processess (CSP) (HOARE, 1978) a formal language for describing patterns of interaction in concurrent systems. (Hoare is also famous for inventing the "burning a candle from both sides" Quicksort algorithm.) It is a *process algebra* — although the original paper presented a parallel programming language rather than an algebra —, based on message passing via channels. These channels, known as Hoare's Channels, are the basis of multithreaded communication of the Go, Occam, and several other programming languages. (We highlight the difference between Hoare's approach and what we described as the preliminary draft of this thesis when discussing the relevant publications on the beginning of

Section 1.4. There we discussed an algebra where the operands were containers of tasks. Hoare's approach, on the other hand, treats messages between workers as the operands.)

When we presented the concept of speedup in Subsection 2.2.1, we differentiate the use of T_{seq} and T_1 on the formula. The first one is used for a demonstration of the quality of the parallelization, the former on how well an algorithm scales. Although this is a known fact, there are few books that acknowledge the differentiation. One of these few examples is the book by Casanova *et al.* (CASANOVA; LEGRAND; ROBERT, 2008, *p.* 10).

On Subsection 2.2.2 we show briefly a famous theorem by R. L. Graham and Richard Brent that states the upper-bound of $T_1/P + D$ to greedy schedulers. Although named after both researchers, the results were obtained independently, in distinct works. Graham first published his result in 1969 on a paper entitled "Bounds on multiprocessing timing anomalies" (GRAHAM, 1969). Later Brent showed the same bounds in a 1974 paper entitled "The Parallel Evaluation of General Arithmetic Expressions" (BRENT, 1974). Both papers do not have the same subject or goals, yet they achieve an important common bound.

Still treating of greedy schedulers, the term "busy leaves" was used in a seminal paper by Arora *et al.* (ARORA; BLUMOFE; PLAXTON, 1998), which introduced the ABP work-stealing algorithm. The same paper brings proofs about the asymptotic limits on the particular implementation of work-stealing, overall an illustrative proof on the expected number of synchronizations $\mathrm{O}(PD)$ using the probability of a given critical steal happening in a burst of attempts. This paper was later republished with a different, simpler proof for the same result that employs a potential function in order to deliver an amortized value. The paper, proofs, and derivations are analyzed in the next chapter. In it, we will also discuss the paper on how to implement ABP work-stealing efficiently (BLUMOFE; LEISERSON, 1999). For instance, being implemented non-blocking guarantees the progress of computation. (For a taxonomy of wait-free, lock-free, and starvation-free algorithms, we recommend Herlihy and Shavit (HERLIHY; SHAVIT, 2008).) Finally, Berenbrink *et al.* (BERENBRINK; FRIEDETZKY; GOLDBERG, 2003) shows that any generalized version of work-stealing (including, thus, ABP) is stable, *i.e.*, it does not degenerate over long periods of time.

There are several resources on parallelization techniques. On this chapter, we delivered a brief survey and adapted it to our context. Three books we already cited contain a detailed explanation of these techniques, Casanova *et al.*'s, Jaja's, and Herlihy and Shavit's. The three examples we showed were adapted from Kumar (KUMAR, 2002) in its section

3.2, “Decomposition Techniques”. Another source of detailed examples on the subject is Pacheco (PACHECO, 2011), although he divides, still in the first chapter, parallelism types in task-based and data-based, which to us are not dichotomic.

3 STATE OF THE ART

This chapter lists and discusses related works both in the analysis of parallel algorithms (Section 3.1) — theme of Part I — and advances on pseudorandom number generators (Section 3.2) — as seen on Part II. We establish comparison criteria among our methods and up-to-date literature. Implementation factors are discussed whenever possible.

On the analysis of parallel algorithms, we approach an early analysis of work-stealing schedulers (Subsection 3.1.1) and current analysis by potential functions (Subsection 3.1.2). We also enumerate papers about the implementation of work-stealing schedulers (Subsection 3.1.3) because these implementations hold important principles for designing algorithms within. Since SIPS has some inspiration on logical clocks we also discuss Lamport's Clocks (Subsection 3.1.4). Finally, we review some current trends on the topics of analysis (Subsection 3.1.5).

On the topic of parallel generation of pseudorandom numbers we overview works on classical state-based generators (Subsection 3.2.1), current state-of-the-art counter-based generators (Subsection 3.2.2), and generation tied to the parallel runtime, which we compare with our results (Subsection 3.2.3). We also review current trends on the subject (Subsection 3.2.4).

The chapter ends with (brief) closing remarks on the abstractness of common parts of the presented works (Section 3.3).

3.1 Analysis of Parallel Algorithms

3.1.1 The Analysis of Work-Stealing Schedulers

Until a seminal paper by Blumofe and Leiserson (BLUMOFE, 1994), work-stealing was established to be more efficient than its converse, work pushing, only in folk wisdom. It was this paper that delivered the first optimal asymptotic bounds that would be optimized to become the ones we have today — both in time and space. In it, the authors analyze the scheduling of fine-grained tasks expressed as threads in multithreaded environments. They made the critical observation — and based their analysis on it — that work-stealing should be more efficient than work pushing schedulers because the schedule operations are performed by idle workers, *i.e.*, no active worker stops useful work to schedule tasks. This leads to less parallelism overhead because thing only run in parallel in there is room for it. (This paper was first published at the FOCS conference in 1994. It was reviewed and re-published later in the Journal of ACM (BLUMOFE; LEISERSON, 1999).)

The analysis focus on *fully-strict* parallel computations, where all child workers (the spawned) complete before their parent (the spawnie). It is a constraint on the concept of *strict* parallel computations, where all children synchronize only with their parents. Although work-stealing does not require these models, they express well recursive parallelism, which simplified code writing with no lesser expression power. The proofs on the paper, however, rely on fully-strict computations.

The first important result it provides is the space complexity of an algorithm scheduled by work-stealing. For any multithreaded computation with stack depth S_1, any P-processor execution by work-stealing uses space S bounded by $S < P \cdot S_1$. When sketching the proof, the important factor is that going down on the function call chain, only the leaves are active. Since the number of leaves is equal to the number of processors, and each leaf occupies at most the space required by sequential execution with one processor, the bound is obtained.

The most interesting result on the paper, however, is the time bound. Consider the execution of any fully strict multithreaded computation with work W and depth D scheduled by work-stealing on a P-processor execution. The expected running time, including scheduling, is, then,

$$\frac{W}{P} + \mathrm{O}(D)$$

Moreover, for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the execution time on P processors is

$$\frac{W}{P} + \mathrm{O}\left(D + \log_2 P + \log_2\left(\frac{1}{\epsilon}\right)\right)$$

This time and variance is a provably upper bound that hides a multiplicative constant within the Big-O notation on the critical path term. Empirically, however, since there are few steals, the performance is close to $W/P + D$.

As a pseudo-proof, consider that each worker is either working or stealing. The total time all workers spend working is W. Each steal has a $1/P$ chance of reducing the critical path length in the DAG by 1. Thus, the expected number of steal attempts is $\mathrm{O}(PD)$. Since there are P workers, the expected time is

$$\frac{W + \mathrm{O}(PD)}{P} = \frac{W}{P} + \mathrm{O}(D)$$

The most important piece of the proof is the argument that each successful steal has a chance of $1/P$ of decreasing the critical path length by 1.

The idea is that if we partition the execution in rounds, each round composed of successive steal attempts, and if the number and size of rounds is large enough then with high probability one “critical task” will be stolen. A critical task is a task that once stolen and executed will result in a DAG with a strict smaller critical path. Since the critical path can decrease at most D times, and a sufficient large number of attempts will decrease D, it is an upper bound to the number of steal attempts.

The bounds we present to the number of synchronizations with SIPS have the same spirit as this delay sequence technique. As shown in Chapter 4, our main proof is based on the fact that if a sufficient every worker synchronizes at least once, then the clock with the minimum value at that top must have been increased. Nevertheless, our SIPS analysis improves these earlier results em several ways. First, we do not depend on a fixed critical-path length D. This means that our DAG does not need to be fixed for a given input, nor does our proofs rely on a fixed graph. Being able to handle no-fixed DAGs allows the algorithms to change their behavior throughout the computation, adapting to execution constraints. Also, our analysis models effectively successful steals, not steal attempts, allowing one to estimate bounds on overheads that only happen when a steal is performed (examples will be given in the chapters ahead). Not only that, but we are also able to bound subsets of the total number of synchronizations, allowing, among other things, to deliver a more tight bound when the overhead depends on the “size” of the steal. Finally, we neither need a fixed number of workers P, nor are we tied to fully-strict computations. It may vary during the execution, admitting workers that come and go dynamically.

These results are old right now, and several improvements were made throughout the years. For instance, there is a paper by Bender and Rabin (BENDER; RABIN, 2000) where the results are generalized to processors with different speeds and preempting features. Nowadays, the limits are tighter. We compare ourselves to these more modern works. The tightest bounds one have as of today — as far as we know — come from a proof technique based on potential functions. We discuss the advent of these proofs next. On the next section, we present the modern approaches and compare ourselves to them.

Handling a variable number of processors, however, is not a feature we claim exclusivity. One paper by Arora *et al.* (ARORA; BLUMOFE; PLAXTON, 1998) proposes an extension to the work-stealing scenario, sketching a user-level thread scheduler suitable to multiprogramming. The authors model multiprogramming with two scheduling levels: a user-level work-stealing scheduler, that maps threads onto a fixed set of processes, and the kernel scheduler, that maps the processes to processors or cores. In it they consider the kernel to be an adversary and aim to efficient execution whatever the resources provided by the kernel. The kernel scheduling gives the variability of workers. The paper shows that in this scenario the work-stealing scheduler executes the computation in expected time

$$\mathrm{O}\left(\frac{W}{P_A} + \frac{DP}{P_A}\right),$$

where P_A is the average number of processors allocated to the computation by the kernel. There is in it a proof that the primitive "yield", common to most kernels, is a powerful primitive to scheduling systems, being the tool that constraints the adversary kernel and guarantees the asymptotic bound.

This paper improves the previous results in two ways: first, arbitrary multithreaded computations are considered, not only fully-strict, like ourselves. Second, the environment is shared with other programs and is not necessarily dedicated — also like SIPS. Its findings are implemented on the top of a library named Hood (BLUMOFE; PAPADOPOULOS, 1998), which we discuss later still in this chapter.

One interesting advancement the paper by Arora *et al.* brings along is a different proof for the earlier work-stealing bounds. It uses an amortization argument based on a potential function that decreases as the algorithm progresses. (The interested reader in the potential method may consult the textbook by Cormen *et al.* (CORMEN et al., 2009, *p* 459).)

Let each node u on the DAG, *i.e.*, a task, to have an associated weight $w(u) = D - d(u)$ where $d(u)$ is the depth of node u in the enabling tree. Also, let R_s denote the set of ready nodes at top s. A task is either assigned to a worker or in the deque of some worker. Thus, for each ready node u in R_s, the potential function is defined as:

$$\phi_s(u) = \begin{cases} 3^{2w(u)-1} & \text{if } u \text{ is assigned} \\ 3^{2w(u)} & \text{otherwise} \end{cases}$$

then, the potential at top s is

$$\Phi = \sum_{u \in R_s} \phi_s(u).$$

Two actions change the potential. The first one is the steal of a task u from the back of a given worker's deque (inside the nano-loop). In this case, the potential decreases by $\phi_s(u) - \phi_{s+}(u) = 3^{2w(u)} - 3^{2w(u)-1} = (2/3)\phi_s(u)$, which is positive. The second case where the potential changes is when the worker executes the task at its deque's front (outside the nano-loop). There, if the execution enables two children, then the spawner, labelled x, is placed on the deque and the spawned, labelled y, is executed. Thus, the potential decreases by

$$\begin{aligned}
&\phi_s(u) - \phi_{s+}(x) + \phi_{s+}(y) \\
&= 3^{2w(u)-1} - 3^{2w(x)} - 3^{2w(y)-1} \\
&= 3^{2w(u)-1} - 3^{2(w(u)-1)} - 3^{2(w(u)-1)-1} \\
&= 3^{2w(u)-1}\left(1 - \frac{1}{3} - \frac{1}{9}\right) \\
&= \frac{5}{9}\phi_s(u),
\end{aligned}$$

which is positive. If the execution of u enables fewer than two children, the potential decreases even more. (The spawn/sync semantics we described earlier in Chapter 2 in fact enables 0, 1, or 2 children only at a time. Further enabling is inside the children, even if the programmer writes a succession of spawns.)

The analysis proceeds by partitioning the potential into two parts, A_s, the set of workers whose deque is empty in top s, and D_s, the set of all other workers. Thus, the potential is written

$$\Phi_s = \Phi_s(A_s) + \Phi_s(D_s),$$

where

$$\Phi_s(A_s) = \sum_{q \in A_s} \phi_s(q) \quad \text{and} \quad \Phi_s(D_s) = \sum_{q \in D_s} \phi_s(q),$$

and the analysis follow separated. The authors proceed to show that whenever P or more steal attempts take place over a sequence of rounds, the potential decreases by a constant fraction with constant probability. First, it is demonstrated that 3/4 of the potential $\Phi_s(D_s)$ is sitting "exposed" at the back of deques where it is accessible to steal attempts. Second, they use a "balls and weighted bins" argument to show that 1/2 of this exposed potential is stolen with 1/4 probability. By dividing the execution in $\Theta(P)$ phases and letting s be the beginning of the current phase and s' be the start of the next phase, they proceed to show that summing each worker in A_s delivers $\Phi_s - \Phi_{s'} \geq (5/9)\Phi_s(A_s)$. Thus, no matter how Φ_s is partitioned between $\Phi_s(A_s)$ and $\Phi_s(D_s)$, the probability of decreasing is still larger than 1/4.

With this high probability of largely decreasing the potential, the authors arrive again at the $O(PD)$ expected upper bound on the number of steal attempts.

As with the previous "delay sequence" analysis, the potential method also relies on the top-most task on each deque being stolen. SIPS does not require so. In fact we shall see how different workload partition at steals is handled seamlessly by SIPS theorems by only change the value of one parameter.

Although not the focus of their published work, the proof through potential functions is the basis for modern bounds on dynamic scheduling. More flexible scenarios are abridged by the same family of proofs by selecting suitable functions. In next section, we discuss two recent papers that use it to model distributed list scheduling algorithms in general, including work-stealing. They also do not rely on top-most steal and model a myriad of distributed list scheduling algorithms.

Before moving forward, we highlight that the work-stealing algorithm as described in Subsection 2.2.2 and its variations are stable, despite its random nature. A system is said to be *unstable* if the system load (the sum of the load of all workers) grows unboundedly with time. A system is *stable* otherwise. A 2003 paper by Berebrink *et al.* (BERENBRINK; FRIEDETZKY; GOLDBERG, 2003) proves the assertion. They consider a fixed, but arbitrary, distribution G over generator-allocation functions that map producer workers to consumer workers. During each top, a generator-allocation function h is chosen from G, and the generators are allocated to the processors according to h. Each generator may then generate a unit-time task that inserts it into the deque of its host processor. It produces such a task independently with probability λ. After the new tasks are created, each processor removes one task from its deque and services it. For many choices of G, the work-generation model allows the load to become arbitrarily imbalanced, even when $\lambda < 1$. The authors consider the work-stealing algorithm as we presented on Subsection 2.2.2. Any non-empty worker having received at least one steal attempt in turn decides (again randomly) in favour of one of the requests. The number of tasks that are transferred from the non-empty processor to the empty one is determined by the so-called work stealing function f. In particular, if a processor that accepts a request has l tasks stored in its queue, then $f(l)$ tasks are transferred to the currently empty one. The authors analyse the long-term behaviour of the system as a function of λ and f and show that the system is stable for any constant generation rate $\lambda < 1$ and a broad class of functions f.

In the same sense as Berebrink *et al.*, SIPS also model the work-stealing execution as a composition of functions to generalize the algorithm and its analysis. Like them, we establish a general, fixed model and manipulate its variables according to the behavior of its functor components. In their case, they play between the balance of function h, the victim selection strategy, and f, the workload partition strategy at each steal. In our case, we model execution through clock functions private to each worker, both schemes being indirectly handled; the victim selection strategy impacts on the increase rate of a global clock — a function composed by all worker-private clocks — while the workload partition strategy changes the upper-bound for any local clock M.

3.1.2 Potential Function Analysis

There are two papers by Tchiboukdjian *et al.* that employ a variation of the potential method we described earlier to achieve tighter and more flexible bounds. The first one, from 2010, is named *A Tighter Analysis of Work-Stealing* (TCHIBOUKDJIAN et al., 2010) and displays usage of this new potential method for the scheduling of unit independent tasks and the ABP work-stealing algorithm. The second one, from 2013, is named *Decentralized List Scheduling* (TCHIBOUKDJIAN; GAST; TRYSTRAM, 2013) and expands the methods found in the previous paper to weighted independent tasks, tasks with precedence constraints, and cooperative stealing.

Like ours, the methods displayed on those papers improve previous analysis by modelling DAGs whose nodes' out-degree may be larger than two and algorithms whose workload partition strategy may differ from the top-most strategy employed by ABP. The authors also provide tighter limits on the makespan $C_{\max}$ of computations. While ABP delivered big constant factors

$$\mathbb{E}[C_{\max}] \leq \frac{W}{P} + 32 \cdot D \quad \text{and} \quad \mathbb{P}\left\{C_{\max} \geq \frac{W}{P} + 64 \cdot D + 16 \cdot \log_2 \frac{1}{\epsilon}\right\} \leq \epsilon,$$

the analysis found on the paper delivers

$$\mathbb{E}[C_{\max}] \leq \frac{W}{P} + 5.5 \cdot D + 1 \quad \text{and} \quad \mathbb{P}\left\{C_{\max} \geq \frac{W}{P} + \frac{3}{1 - \log_2\left(1 + \frac{1}{e}\right)} \cdot \left(D + \log_2 \frac{1}{\epsilon}\right)\right\} \leq \epsilon.$$

The number of steal requests provided by the authors compares to our SIPS based analysis. Before discussing the subject, we present a sketch of the proof method used in both papers.

First, as motivation, let us look to Figure 3.1, from the Distributed List Scheduling paper. It was obtained from a discrete step simulator and reveals acknowledgeable in-

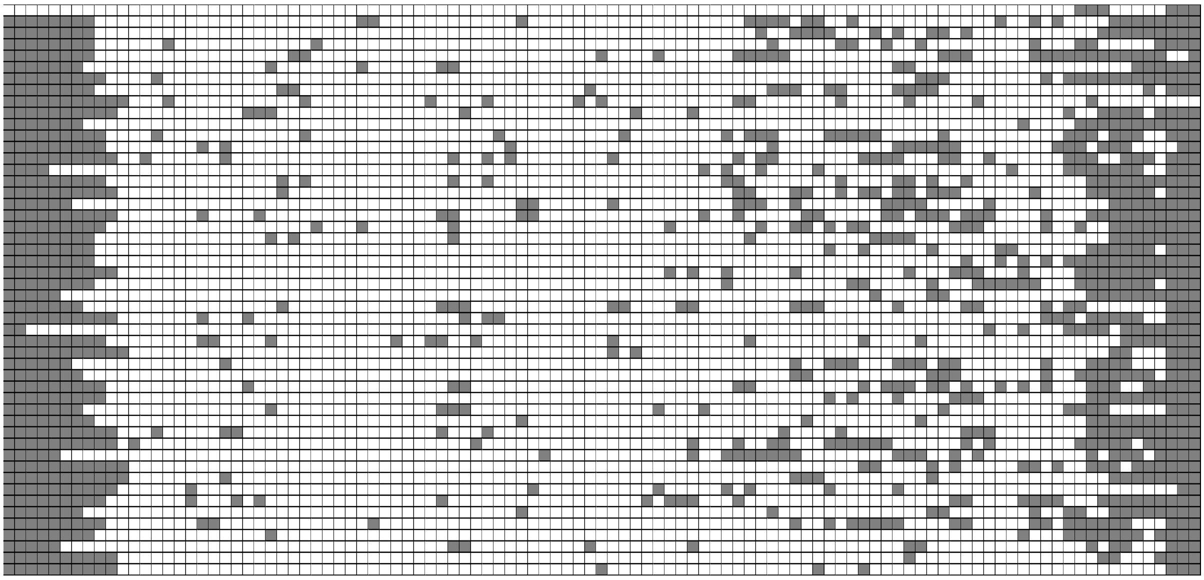


Figure 3.1: Gantt chart for work-stealing (TCHIBOUKDJIAN; GAST; TRYSTRAM, 2013). There are 25 workers and 2,000 unit time tasks. Each column is a top. White squares represent the execution of a unit time task. Grey squares represent steals.

formation. First, the two main steal phases are at the beginning and the end of the computation. At the start, only one processor has a task. Thus, the others will enter into a try-and-retry of steals inside the nano-loop (see Subsection 2.2.2). (Both our analysis and the potential method predict a time proportional to $\log_2 n$ in expectation before all workers are not idle.) In the end, few processors have few tasks; the retrying sequence occurs once again. Throughout the computation few steal occurs, because when all workers are active, and one of them becomes idle it will steal a task on the first try — or at least within few tries. (Indeed our strategy of moving overhead to steal operations presented on part II takes advantage of this fact.) Seeing the steals as a potential that decays during computation is the key insight of the potential model employment.

A potential function $\phi : \mathbb{N} \rightarrow \mathbb{N}$ represents how well the load is balanced between the deques

$$\phi(s) = \sum_{1 \leq i \leq P} \left(w_i(s) - \frac{w(s)}{P} \right),$$

where s is the current top, $w_i : \mathbb{N} \rightarrow \mathbb{N}$ is a function that receives a top s and returns the total amount of work at worker i on top s, and $w : \mathbb{N} \rightarrow \mathbb{N}$ is a function given a top returns the sum of $w_i(s)$ for each worker $1 \leq i \leq P$.

The potential function decreases at each steal. One can, thus, bound the number of steals a to bound $C_{\max}$: $P \cdot C_{\max} = W + a$. After a steal operation from worker i to worker j, some work is transferred from i to j. The quantity of work is determined by the workload partition strategy — *e.g.* half of tasks, top-most task. (Let us assume, without loss of generality, the strategy to be steal half of tasks in a list and that if several thieves try to steal the same active victim only one gets a share of its workload; which one is chosen randomly.) Thus, the larger potential decreases strictly:

$$\max(w_j(s+), w_i(s+)) \leq \rho \cdot w_i(s),$$

where $\rho < 1$ is the decay factor. This implies the following properties:

1. If $\phi = 0$ then there are no more steals and all lists are either empty or with one task.
2. For all workers i,
   $$w_i \to w_i - \mathrm{O}(1) \Rightarrow \Delta\phi = 0,$$
   i.e., if all workers descrease their lists by the same constant, then the potential function variation is zero.
3. If idle worker i steals half of the work of active worker j, then
   $$\Delta\phi = \frac{w_j^2}{2},$$

The proof methodology is then

1. Compute the expected decrease of the potential in one step when α_s workers are active, and $P - \alpha_s$ are stealing:
   $$\mathbb{E}[\phi(s) - \phi(s+) \mid \phi(s)] \geq h(\alpha(s)) \cdot \phi(s),$$
   where $h : \{0, \ldots, P\} \to [0, 1]$ is the ratio function modelling the decrease of the potential function and assumed to exist.
2. Solve the equation to bound the number of steal attempts a:
   $$\mathbb{E}[a] \leq \lambda \cdot P \cdot \log_2 \phi(0)$$
   $$\mathbb{P}\left\{a \geq \lambda \cdot P \cdot \left(\log_2 \phi(0) + \log_2 \tfrac{1}{\epsilon}\right)\right\} \leq \epsilon.$$
   where
   $$\lambda = \max_{1 \leq \alpha \leq P} \left(\frac{P - \alpha}{-P \log_2 (1 - h(\alpha))} \right)$$

3. Deduce a bound on the execution time:

$$\mathbb{E}[C_{\max}] \leq \frac{W}{P} + \lambda \cdot \log_2 \phi(0).$$

The details of each step and the extension to other scenarios escapes the scope of this thesis.

Our work improves the results found by the potential method by addressing a wider scenario. The potential function method is effective in providing sharp bounds to classical distributed analysis, but, unlike us, is constrained to fixed DAGs. Like more classical works, they rely on work and depth of parallel algorithms, which may change on varying the DAG. Besides, their bounds differ semantically from ours. We deliver an expectation to the worst-case number of effective synchronizations (*e.g.*, successful steals) u over an input size n and P workers. For top-most steals with random victim choice this number of successful steals is: $\mathbb{E}[u] = \mathrm{O}(P \log_2 P) M$, where M is an upper-bound on the number of successive synchronization one active processor may engage in before becoming idle. For the steal-half strategy the bound becomes

$$\mathbb{E}[u] = \mathrm{O}(P \log_2 P) \log_2 n$$

Tchiboukdjian *et al.* work delivers the expected total number of steal attempts a over an input size n, P workers, and assuming unit-time steal operations. Again, for top-most steals with random victim choice, this number is

$$\mathbb{E}[a] \leq 5.5PD + P - 1$$

This improves the constant factors in Blumofe *et al.*'s work, but assuming that steals are performed in time $\mathrm{O}(1)$ instead of 1 delivers $\mathbb{E}[a] \leq 5.5 \cdot \mathrm{O}(PD) + P - 1$, what converges asymptotically to Blumofe *et al.*'s bound of $\mathrm{O}(PD)$.

3.1.3 Implementation of Work-Stealing Schedulers

The group of Charles Leiserson on the MIT and its collaborators are the leading references on the implementation of work-stealing schedulers for shared memory machines. Now we examine some of their work and discuss the techniques bound to it, relating it to some design decisions of our work.

Implementations of schedulers play a significant role in real-world performance. It brings along several heuristical optimizations that are important in practice, like non-blocking steal protocols, the work-first principle, and elision code, *etc.* Although implementation optimizations usually do not impact the asymptotic complexity, it is essential to mitigate the constant factors associated. This is what makes the difference in everyday use and should not be neglected.

We begin by discussing Hood (BLUMOFE; PAPADOPOULOS, 1998). Older than Cilk-5, Hood was a user-level threads library, whose primary concern was to provide an efficient performance under multiprogramming without the need for support in the OS kernel. It already used a non-blocking implementation of the ABP work-stealing, what would later be the basis of Cilk-5. Contrary to it, however, and like other previous versions of Cilk, it was a library, requiring a fine-grained control by the programmer, without any support from the compiler. The main advent was the idea that the execution time of a program running with arbitrarily many processes on arbitrarily many processors is a function of sequential work and depth. Even in an early stage, the companion analysis of Hood considered the hypothetic scenario where the set of workers grows and shrinks over time, this time the kernel being the entity that allocates or deallocates the physical processors to the runtime. The paper showed that Hood applications behave well and achieve linear speedup regardless of the behavior of the kernel scheduler. The proof supposes an adversary kernel and considers the bounds of Arora *et al.* (ARORA; BLUMOFE; PLAXTON, 1998).

Contemporary to Hood, we overview the Cilk implementation paper from 1998 by Frigo, Leiserson, and Randall (FRIGO; LEISERSON; RANDALL, 1998), “The implementation of the Cilk-5 multithreaded language”. There the authors present the implementation decisions behind the fifth version of the original Cilk programming language (an extension to C). Cilk uses the provably good ABP work-stealing algorithm to manage user-level threads on the top of OS processes. The fifth version of Cilk was re-designed to move all the scheduling logic to the compiler, relieving the programmer from handling its data structures. It was this paper that brought along the work-first principle, *i.e.*, based on the analysis of ABP the authors argument that minimizing the overheads that contribute to work, even on the expense of overheads that contribute to the critical path, results in performance gain. Although counter-intuitive, this approach leads to a portable version of Cilk where the typical cost of spawning a task is only between 2 and 6 times the cost of a C function call on the machines of that time. Also, it only occurs at calls that will run in parallel; thanks to a “two clones” compilation strategy most of the calls are standard C function calls. The use of a Dijkstra-like mutual exclusion protocol to implement the deques ensures non-blocking steals, allowing the computation to progress without locks and starving.

The compiler employs the two clone strategy on behalf of the programmer. Cilk compiler, a type-checking, source-to-source translator, transforms a Cilk source code into C post source and run on GCC. To every Cilk procedure (a standard C function declaration preceded, at the time, by keyword `cilk`), the compiler produces two corresponding C functions, a “slow” and a “fast” one. The fast clone is almost identical to the elision version (without keywords **`spawn`** and **`sync`**) and executes when sequential semantics suffices (when there is no steal). The slow clone is executed in the infrequent case when parallel semantics is necessary, upon a successful steal. All communication generated by the scheduler occurs in the slow clone and contributes to the critical-path overhead, following the work-first principle.

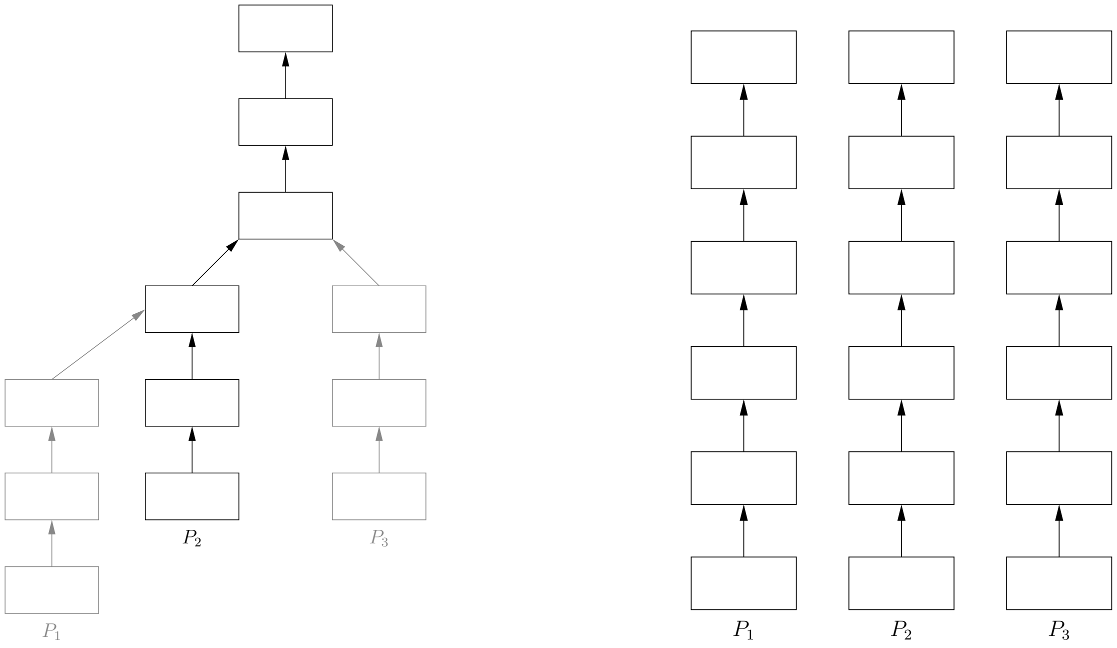
To minimize waiting times in order to avoid starvation and slow progression, the paper presents an implementation of a shared-memory, mutual exclusion protocol on the deques. This protocol, named THE, is inspired by Dijkstra’s mutual exclusion protocol (DIJKSTRA, 1965). Using THE protocol the scheduler guarantees that steal overheads contribute only to the critical path overhead, respecting the work-first principle. THE also allows an exception to be signaled to the working processor with no additional work overhead, a feature used on Cilk’s abort mechanism.

The next version of Cilk is reviewed in a brief paper also by Leiserson *et al.* (LEISERSON, 2009) entitled “The Cilk++ Concurrency Platform.” (As a “bonus”, the paper also brings a short, yet instructive review of the previous works on *p* 6 and *p* 7.) In addition to what was implemented earlier, it introduced a race condition detection tool. Since backwards compatibility was critical, the paper introduced the concept of “hyper object”, which is examined in details on a separate paper, “Reducers and Other Cilk++ Hyperobjects” (FRIGO et al., 2009). These hyper-objects are a mechanism tied to the Cilk language that allow different workers to maintain coordinated local views of non-local variables. The authors identify three “useful” kinds of hyper-objects, reducers, holders, and splitters. Reducers are described prominently, and the scheduler supports a randomized locking methodology for them without significant overhead. Each strand (a branch of the execution tree) has a view of the hyper object, powered by the runtime system. This view is a stateful object with a memory address. The strand accesses its view’s state independently, with no need for synchronization, being private and isolated of other strands. When two or more strands join, the views are combined by a function specified either by the programmer or by the runtime system — the views can be destroyed or carried on the resulting strand. Reducers hyper-objects, for instance, rely on algebraic structures called monoids, which are a set with an associative operation and identity element. The presented theorem and lemmas suppose reduce operations have time $\Theta(1)$ and Lemma 1 has the proof on probabilistic locking. (On Chapter 9, we list as future work exploiting the use of less general structures than monoids we discovered while on the making of this thesis.) The underlying multithreaded framework handles the view. The goal of a hyper object is to facilitate the parallelization of programs using non-local variables without much effort of the programmer.

Earlier versions of multithreaded schedulers, like the ones in Cilk, used a "cactus" activation stack to process function calls. A cactus stack is a stack with a common bottom but multiple ends that depart of the original structure through a branch. Figure 3.2 illustrates the difference between a traditional linear stack (Figure 3.2b) and a cactus stack (Figure 3.2a). On a cactus stack, each worker "owns" an end. Through it, the parallel function access to stack variables properly respects the function's calling ancestry, even when many of the function operate in parallel. Nevertheless, many of those frameworks fail to respect one of the following criteria:

1. complete interoperability with third-party serial binaries compiled to use an ordinary linear stack;
2. bounded, efficient use of memory for the cactus stack.

For this reason, earlier Cilk-5 forbade parallel function to call sequential functions within. Even Cilk on its most recent incarnation, Cilk Plus, chose to change its model back again to a linear stack, even in exchange for performance. In order to solve this *de facto* dichotomy, Lee *et al.* (LEE et al., 2010) proposes a modification to the Linux kernel to provide support for Thread-Local Memory Mapping (TLMM). Since the latest Cilk uses a linear stack, the authors have chosen to modify Cilk-5's cactus stack to be implemented in terms TLMM. With that they produced Cilk-M, a version of Cilk, eliminates the parallel/sequential call constraint from Cilk-5 while providing full compatibility with legacy binaries. Cilk-M is comparable to Cilk-5 in terms of performance and occupies small stack space.



(a) Cactus stack.

(b) Linear stack.

Figure 3.2: Linear and cactus stacks. The rectangles represent memory locations, specifically the function call stack. The arrows represent links between memory locations in the form of pointers (deferenceable addresses).

3.1.4 Lamport's Logical Clocks

One inspiration for SIPS is the classical paper by Lamport (LAMPORT, 1978). We, however, apply it in an unusual way: the analysis of overheads in parallel computations. Knuth's definition of time is counting of specific operations (KNUTH, 1997a); comparison and swapping for sorting, sums and multiplications for polynomial evaluation, *etc.* Our idea is to use logical clocks to count synchronizations.

Lamport examines the concept of the "happened before" relation on a distributed system and how it defines a partial order of events. Then, he proposes a distributed algorithm for synchronizing a system of logical clocks that can be used to totally order the events. At the end of the paper, the algorithm is generalized to physical clocks, but this variation is of no interest to our analysis — the user's perception of real versus logical clocks is of no importance to this thesis. The most significant presented idea, for our purposes, is the bound it derives on how far out of synchrony the clocks can become.

Consider three events a, b, and c. The "happened before" relation, denoted "$\to$" (its negation is noted "$\not\to$"), on the set of events on a system, is the smallest relation satisfying the following conditions:

1. If a and b are events on the same process and a comes before b, then $a \rightarrow b$.
2. If a is the event of sending of a message to another process and b is the event of receiving such message, then $a \rightarrow b$.
3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Two distinct events a and b are *concurrent* if, and only if, $a \not\rightarrow b$ and $b \not\rightarrow a$. It is assumed that, for any event a, $a \not\rightarrow a$, since an event happening before itself is neither feasible nor seems useful. Thus, $\rightarrow$ is an irreflexive partial ordering on the set of all events on the system.

A clock is defined as a function C over some event a, such that $C(a)$ corresponds to its time stamp. The clock condition is

$$\text{if } a \rightarrow b, \text{ then } C(a) < C(b).$$

Each process P running on the distributed system has its own private C. The clock condition is satisfied if the following conditions hold:

1. If a and b are events in process P_i and $a \rightarrow b$, then $C_i(a) < C_i(b)$.
2. If a is the event of sending an message by process P_i and b is the receipt of that message by process P_j, then $C_i(a) < C_j(b)$.

A system of clocks that satisfies the clock condition places a total ordering on the set of all system events. The events are ordered by the timestamps on which they occur. To break "ties", an arbitrary total order "$\prec$" is used (for instance, the id of the process). And, thus, one can define a relation "$\Rightarrow$" as follows: if a is an event in process P_i and b is an event in process P_j, then $a \Rightarrow b$ if and only if either

$$C_i(a) < C_j(b) \quad \text{or} \quad C_i(a) = C_j(b) \text{ and } P_i \prec P_j.$$

The clock condition implies that if $a \rightarrow b$ then $a \Rightarrow b$. In other words, relation $\Rightarrow$ complements and transforms relation $\rightarrow$ from a partial order in a total order.

The clock definition has the same structure of our Definition 2, on Chapter 4. Since we do not exchange messages, condition (2.) applies to synchronization between workers. The set of system events is all task creations and synchronization or a subset of interest from it. Lamport's processes correspond to our definition for workers. By analyzing the impact of the scheduler's victim selection strategy on clock progression and obtaining bounds on the local clocks from the program's work partition strategy, we extend logical clocks to serve as worst-case parameter of the computation's evolution.

3.1.5 Current Trends on Analysis

A MIT master's thesis from July 2014, written by Warut Suksompong and directed by Charles Leiserson (SUKSOMPONG, 2014), approaches one of our central issues: how to estimate the number of successful steals on multithreaded computations scheduled by work-stealing. Suksompong's analysis works under the argument that an upper-bound on the total number of steal attempts is not relevant for the worst-case scenario, a claim we share. The authors consider a tree-shaped DAG. If the computation starts with a complete k-ary tree of height h, the maximum number of successful steals is $\sum_{i=1}^{n}(k-1)^i\binom{h}{i}$. Also, the thesis proposes a work-stealing algorithm called "localized work stealing". The intuition behind it is that because of locality workers benefit from working on its own work. So, when a worker is free, it makes a "steal-back" operation, a particular type of steal that tries to retrieve some of its own work. Assuming an "even distribution of free agents", the expected running time of the algorithm is $W/P + \mathrm{O}(D \log_2 P)$.

The proof on successful steals is based on a recurrence of a potential function (which is slightly different from the potential method described on Subsection 3.1.2). Let $n \geq 0$ be an integer and T a binary tree. The nth potential of T, noted by $\Phi(T, n)$, is defined as the maximum number of steals that can be obtained from a configuration of $n + 1$ workers, one of which has the tree T and the remaining n, which have empty trees. With only one processor one cannot perform any steal, hence $\Phi(T, 0) = 0$. For a binary tree with right subtree T_r and left subtree T_l, the paper shows that the following recurrence holds:

$$\Phi(T, n) = 1 + \max\left(\Phi(T_l, n-1) + \Phi(T_r, n), \Phi(T_r, n-1) + \Phi(T_l, n)\right).$$

By using Pascal's identity this recurrence is generalized to k-ary trees that achieve the bound of $\sum_{i=1}^{n}(k-1)^i\binom{h}{i}$.

We improve this result. First, we make no assumption about the shape of a given DAG. Second, since D is an upper bound on the tree height h, this bound is dependent on the critical path and, since the number of leaves is precisely W, this bound is also dependent on the work. We explicitly deliver a bound that does not use those parameters to support randomized algorithms. Also, our worst-case bound is sharper for trees with large arity and overall size.

Another analysis trends are early considerations about the advent of "space-bounded schedulers". Since a large amount of the performance depends on how well the programs are scheduled regarding processors and cache hierarchy. Space-bounded schedulers, thus, schedule parallel programs on the multi-level cache hierarchies of current machines. Its primary benefit would be, allegedly, the preservation of locality at every level in the hierarchy, resulting in fewer cache-misses and better use of bandwidth that the work-stealing schedulers of nowadays.

In the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA) 2014 conference, Simhadri *et al.* (SIMHADRI et al., 2014) proposed the first analysis on the difference of such schedulers. The authors built an experimental framework for the analysis of separate interfaces for the programs and schedulers to allow comparison in terms of cache-misses and performance across a set of different benchmarks. The variants compared are the Cilk Plus work-stealing scheduler, an hierarchy-minded work-stealing algorithm, and two variants of space-bounded schedulers. The benchmarks vary from divide-and-conquer micro-benchmarks (alike the benchmarks we used in this thesis) and traditional algorithmic kernels. Results indicate that space-bounded schedulers reduce the number of L3 cache-misses compared to work-stealing by 25-65% for most of the benchmarks but incur up to 7% additional scheduler and load imbalance overhead. Only for benchmarks intensive in memory can the reduction in cache-misses overcome the added overhead, resulting in 25% improvement in running time for synthetic benchmarks and about 20% for algorithmic kernels.

Our SIPS analysis is entirely compatible with the analysis of space-bounded schedulers. To add the analysis of cache-misses, tied to certain operations, is a future branch to be explored in our work. The same holds for the mentioned hierarchy-minded work-stealing algorithms.

As will be shown in Chapter 4, Section 4.2 and Section 4.3, the advantage of random strategy over minimum clock strategy is the lack of contention in the first. We highlight some contours over it on Chapter 9, including mixed strategies. It is crucial for a strategy requiring the fast selection of a minimum value to be able to rely on heap-based primitives. We, however, have not found suitable hardware instructions that allowed us to accelerate the insert and extract operations significantly. This is also one of the points of a 2013 paper entitled "Reducing Contention Through Priority Updates" by Shun *et al.* (SHUN et al., 2013). They study the "priority update" operation as a primitive for limiting write contention in parallel programs. This primitive takes as argument a memory location, a new value, and a comparison function $>_P$ that enforces partial order over values. The operation atomically compares the new value with the current value in the memory location, and writes the new value only if it has higher priority according to $>_P$. This is an extension of wide-spread atomic primitives like compare-and-swap and test-and-set — priority updates are described, in fact, in terms of it. The authors proceed to show several algorithms and data structures that benefit from it. The experimental results demonstrate this approach excels on high-sharing algorithms like "remove duplicates".

The above paper proposes, among the algorithms, a union-find data structure that could be useful to the mixed strategies we propose more ahead.

3.2 Parallel Pseudorandom Number Generation

3.2.1 State-based PRNGs

State-base PRNGs are inherently sequential. A successive application of a transformation function $f : U \to U$ (U is a state space) over current state to obtain the next element:

$$u_{n+1} = f(u_n).$$

Thus, size of U is the period of the generator. This application is fundamentally serial since each value depends on the previous one. A paper by Paul Coddington (CODDINGTON, 1997) enumerates the two main approaches to parallelize a PRNG:

Multistream. The PRNG algorithm is instantiated in parallel with different parameters so that each instance produces a distinct stream of numbers.

Substream. A single logical sequence of random numbers is subdivided into disjoint substreams that can be accessed in parallel.

The paper also enumerates a useful array of techniques to parallelize PRNGs, like "leapfrog" (cyclic partition among processors) and "sequence splitting" (block partition among processors) but these are not processor-oblivious. In a perspective, our technique can be applied to both variants, since the decision between the substream or the multistream approach is performed at each steal. We presented a version based on jump operations, which generates the substreams, but we could have used a reseed operation in its place.

Haramoto *et al.* (HARAMOTO; MATSUMOTO; L'ECUYER, 2008) also argued in favor of parallel programs to build a fast jump-ahead algorithm over their PRNG Mersenne Twister, resulting in the implementation of SIMD-oriented Fast Mersenne Twister (SFMT). This is also the case of L'Ecuyer's RNGStreams library (build on the top of its MRG32k3a generator (FISCHER et al., 1999)). These optimizations are applied in the literature due to the programmer knowing the number of workers previously to partition the generate space per resource. (Not possible for `Par-R`, since it is processor-oblivious, as discussed in Chapter 7.) Both approaches deliver a jump with high constant cost, compensated by the large range skipped — which, contrary to `Par-R`, is defined at compile time. In the same sense, there is the popular pthread implementation of SPRNG by Mascagni and Srinivasan (MASCAGNI; SRINIVASAN, 2000) that creates several PRNG streams through parametrization. In general, this type of domain partitioning requires a maximum subspace interval in function of the number of workers, which is analogous to `Par-R` overestimation.

3.2.2 Counter-based PRNGs

Counter-based PRNGs are a novel approach to the traditional state-based one of earlier. They were first discussed in the 2011 paper "Parallel random numbers: as easy as 1, 2, 3" by Salmon *et al.* (SALMON et al., 2011). The authors argue that the state-based approach scales poorly on parallel high-performance architectures, which we corroborate. The proposal is to use independent, keyed transformations of counters to produce a class of PRNGs with practical statistical properties (long period, no discernable structure or correlation). Besides proposing the paradigm, the paper introduces proof-of-concept implementations over cryptographic standards (named ARS and Threefish) and based on new paradigms (named Philox). These PRNGs pass statistical tests (including the well-regarded TestU01 (L'ECUYER; SIMARD, 2007)) and produce at least 264 unique parallel streams, each with a period of 2128 or more.

Counter-based PRNGs are suited to parallel computation because they break the sequential dependence among output values.

Contrary to the state-based case, here each number in the sequence is obtained by a function b, where the n-th random number x_n is obtained by applying b to n:

$$x_n = b(n).$$

In the simplest case, n is a p-bit integer counter, deriving the name “counter-based”. Each computation takes the same constant-time, independently of the value of n. Furthermore, if b is a bijection on the set of p-bit integer onto itself, then the period of the generator is 2^p.

The authors use as the primary source of functions that satisfy the requisites of b cryptographic block cyphers, on the form

$$x_n = b_k(n),$$

where b_k is a keyed bijection, where k is a cryptographic key. A counter-based PRNG constructed from a keyed bijection can be easily parallelized using either the multistream approach over the key space or the substream approach over the counter space. Applications can choose to derive k and n on the fly from either machine parameters or application variables. Generating random numbers from state associated with application variables allows for machine-independent streams of random numbers. This approach permits deterministic results across different computing platforms.

The author introduced three families of bijections with periods of at least 2.128 and parametrized by a key that allows at least 264 or more parallel streams. The first family are *bona fide* cryptographic block ciphers, derived from AES since there are now specialized AES instructions on commodity x86 processors. The second family is based on simplified cryptographic cyphers, being the fastest implementation on all three families in CPUs. Finally, the third family consists of non-cryptographic bijections. The authors introduce Philox, a counter-based PRNG that uses multiplication instructions that compute the high and low halves of the product of word-sized operands. It is the fastest implementation on all three families in GPUs.

Salmon *et al.* (SALMON et al., 2011) argues that the use of a technique like `Par-R`, with conventional PRNGs is impractical due to the requirement of maintaining billions of PRNGs in memory. Our work partially refutes this, since once our conditions are met, one does not need more than one PRNG per thread per recursive call (roughly) to provide deterministic random number generation, for both fast and crypto-secure generators. Counter-based PRNGs have excellent statistical properties and can be used in deterministic parallel executions by either sub-stream or multi-stream approaches. However, considering performance, each random generation from the clock requires an operation equivalent to re-seeding, and thus a linear overhead. The polylog overhead of

Par-R compares favourably to this overhead. Moreover, R can itself use counter-based generators.

3.2.3 Deterministic Parallel Runtime

A 2012 paper by Leiserson *et al.* (LEISERSON; SCHARDL; SUKHA, 2012) entitled "Deterministic parallel random number generation for dynamic-multithreading platforms" proposes a third way to approach the multistream/substream duality. It offers a mechanism called pedigrees, built into the runtime system, to enable efficient deterministic parallel random number generation. The responsibility of generating and seeding the copies on the multistream approach is passed on to the underlying runtime. Experiments with the open-source MIT Cilk runtime system show that the overhead of maintaining pedigrees on a suite of 10 benchmarks, the relative overhead of Cilk with pedigrees to the original Cilk has a geometric mean of less than 1%. The authors persuaded Intel to modify its commercial C/C++ compiler, which provides the Cilk Plus concurrency platform, to include pedigrees, and built a library implementation of a deterministic parallel random number generator called DotMix that compresses the pedigree and then hashes the result.

The paper reports that the statistical quality of DotMix is comparable to that of the famous Mersenne Twister (MATSUMOTO; NISHIMURA, 1998) but somewhat slower than a nondeterministic parallel version of the later.

The cost of calling DotMix depends on the depth D of the invocation. For a naïve Fibonacci calculation with $n = 40$ that calls DotMix in every node of the computation, the overhead is about a factor of 2.3 in running time over the nondeterministic Mersenne Twister. For other applications that use random numbers — such as a Maximal Independent Set algorithm, a practical Sample Sort program, and a Monte Carlo discrete-hedging application from QuantLib — the observed overhead was at most 21%.

Their research provides fast parallel PRNGs through a jump-ahead function, much like our work does. The main difference is the usage of this mechanism; while we use jump to compensate parallel non-determinism in the usage of sequential PRNGs.

This thesis references DotMix several times, since it is the base of comparison of the benchmarks on Chapter 8.

3.2.4 Current Trends

A 2014 paper by the well-known PRNG specialist Pierre L'Ecuyer and others examine the requirements of random number generators for current parallel machines, emphasizing the advent of GPUs. This paper, entitled "Random Numbers for Parallel Computers: Requirements and Methods With Emphasis on GPUs", is currently on submitted status to the Mathematics and Computers in Simulation journal and can be found on the authors' website <http://www.iro.umontreal.ca/~lecuyer/myftp/papers/parallel-rng-imacs.pdf>. In it, there is an examination of the requirements and the available methods and software to provide (or imitate) uniform random numbers in parallel computing environments. The authors state that for the vast majority of applications, independent streams of random numbers are required, each being computed on a single processing element at a time. They aim to explain how they can be produced and managed and devote particular attention to multiple streams for GPU devices. The observations contained in this paper matches our own in many aspects. The premise is that in highly-parallel systems, one may need thousands or even millions of virtual PRNGs (this corroborates the paper by Salmon *et al.* (SALMON et al., 2011)). They can be either different PRNGs or copies of the same PRNGs starting from different states that run in parallel without exchanging data between one another, and behave from the user's viewpoint just like independent PRNGs. In our work, however, we need only one active PRNG per worker at each time.

As in this thesis, the use of deterministic PRNGs in the process of debugging parallel software is emphasized. It is often required that simulations must be exactly replicable and produce exactly the same results on different computers and architectures, either parallel or purely sequential, and when running the program again on the same machine. The latter is necessary for debugging and in the situation where one wants to simulate a complex system with slightly different configurations or decision-making rules. This is to make sure that exactly the same random numbers are used at exactly the same places in all configurations of the system and repeat this n times independently.

In single monitor tools, all the new streams are created and managed by a central monitor. The streams are defined so they are all distinct, long enough to make sure they cannot overlap, and they behave as statistically independent. For reproducibility, the user must make sure that they are created in the same order and used for the same purpose in different configurations. This single-monitor design means that all streams must be passed or copied from the single location where they are created to all other places where they are to be used. For most parallel applications, this is acceptable and sufficient. In multi-monitors environments, each creator will create exactly the same sequence of streams in exactly the same order, provided that the creators are created themselves in the same order. Once created, the creators no longer have to interact with each other and can be distributed in loosely connected groups of nodes. This is the case of the counter-based generators described earlier.

Functional programming languages like Haskell follow the multistream approach as `Par-R`, offering their own *splittable generators* to the programmer and a corresponding split function. In addition to the traditional operation of state-based generators **`next`** — that generates a new number and updates the state and is detailed in Chapter 6 — it also offers an operation named split, which replaces the original PRNG object with two (seemingly) independent PRNG objects, by creating and returning a new such object and updating the state of the original object. Splittable PRNG objects make it easy to organize the use of pseudorandom numbers in multithreaded programs structured using recursive parallelism. However, these implementations are bounded to a `R` provided by the runtime, unlike the generic model of `Par-R`, which accepts any generator with a given interface. This idea of splittable generators is taken in depth in a 2014 paper by Guy Steel, Doug Lea, and Christine Flood named "Fast Splittable Pseudorandom Number Generators". The paper was published in both a conference (STEELE JR.; LEA; FLOOD, 2014a) and, later, in a journal (STEELE JR.; LEA; FLOOD, 2014b). In that, the authors describe a new algorithm, SPLITMIX, for an object-oriented and splittable pseudorandom number generator.

SPLITMIX uses 9 64-bit arithmetic/logical operations per 64 bits generated and has a Single Instruction Multiple Data (SIMD) and GPU implementation. The authors derive SPLITMIX from the DotMix algorithm of Leiserson *et al.* (LEISERSON; SCHARDL; SUKHA, 2012). SPLITMIX is faster and produces pseudorandom sequences of higher quality than Haskell's or Java 8's. The generated sequences produced by SPLITMIX were tested using two standard statistical test suites (DieHarder and TestU01) and its results are inferior to serial generator Mersenne Twister, although the performance may compensate for it.

Since `Par-R` may use Mersenne Twister — and others — underneath, the quality of our method may be as good as any sequential generator used as a reference. Comparisons in performance are enlisted for future work.

3.3 Closing Remarks

In this chapter, we overviewed the state-of-the-art on analysis of synchronizations in parallel programs scheduled by a distributed algorithm and generation of pseudorandom numbers in parallel.

By analyzing works in a deep, structured way, we hope the reader sees more clearly why we chose the notation and definitions the thesis uses. Much like finding axioms through proofs of related theorems, a common underlying working set of definitions arises from related or derived works. The result is not intended to produce a "fit it all perfectly" relation with the enlisted works. Rather, we aim at a "fits most of it well" relation that applies itself between the studied works and this thesis.

Part I

The Tools of Analysis: Synchronizations in Greedy Scheduled and Work-Stealing Scheduled Parallel Algorithms

4 SIPS: A TECHNIQUE TO ANALYZE SYNCHRONIZATIONS IN GREEDY SCHEDULED ALGORITHMS

We present SIPS, an analysis framework that allows us to estimate the parallel overhead introduced by synchronizations. This is the central chapter of the thesis.

Upon definition of local and global clocks and the relevant functions over them (Section 4.1), we build the rationale by providing a method to estimate the number of successful steals in a computation scheduled by work-stealing (Chapter 2). We are able to deliver bounds on the number of synchronizations on such computations for a variety of victim selection strategies, exemplifying with the choice by minimum clock (Section 4.2) and the random selection (Section 4.3). Then we show that this limit is flexible by changing parameters on the execution, such as the workload partition strategy (Section 4.4), which allows us to model general synchronizations.

Through SIPS clocks, we are able to introduce the notion of asymmetrical parallelism (Section 4.5) and show how classical analysis based on work and depth does not encompass this scenario. We later use the presented concepts as the basis to define work-efficient and work-optimal algorithms (Section 4.6).

The chapter ends with closing remarks (Section 4.7), abridging this more abstract content to the next chapter, which is more practical.

4.1 Definitions

Parallel executions are examined through the model of task-based computations as established on Subsection 2.2.1 and Subsection 2.2.2.

As demonstrated by Blumofe *et al.* (BLUMOFE; LEISERSON, 1999), the expected number of total steal attempts for a parallel execution over P workers with depth D and scheduled by randomized work-stealing is $\mathrm{O}(PD)$. Nevertheless, the performance of our method is bounded by the number of successful steals, *i.e.*, the steal attempts over non-empty deques. Next we employ a counting technique that estimates the size of a particular subset of the performed steal attempts (*e.g.*, successful ones) and does not depend on execution's depth. This generalizes the bound to a non-deterministic DAG.

First, let each worker $1 \leq i \leq P$ to have associated a *local clock* ϕ_i, and a set of local clocks to be the *global clock* (or just "SIPS clock"):

Definition 2 Let S be the poset of all events during a parallel execution (identified by the respective tops). A *local clock* is any function $\phi_i : S \to \mathbb{N}$ where:

1. If i becomes inactive at $s \in S$, then $\phi_i(s+) = 0$.
2. If i is inactive at $s \in S$, then $\phi_i(s) = 0$.
3. If i becomes active at $s \in S$, then $\phi_i(s+) > 0$.
4. If i is active at $s \in S$, then $\phi_i(s) \geq \phi_i(s-)$. □

Definition 3 Let Σ be a (possibly non-maximal) subset of S containing only synchronization operations. A *global clock* is any function $\phi : S \to \mathbb{N}^P$ with $s \mapsto (\phi_1(s), \ldots, \phi_P(s))$ where:

1. Function ϕ_i is a local clock for worker i.
2. If $s(i,j) \in \Sigma$, then $\max(\phi_i(s-), \phi_j(s-)) < \min(\phi_i(s+), \phi_j(s+))$ □

Henceforward all successful steals are considered to be the "interesting" synchronizations, *i.e.*, the ones in Σ. The local clock $\phi_i(s)$ is the number of times a worker had tasks stolen from its **deque!** (**deque!**) since it is active until it becomes idle. The global clock is the total number of *successful* steals. The local clocks' upper bound M is defined as the maximum size of any deque during computation.

To develop our theorems we also need the notion of the minimum clock. It serves as the "weakest node" in the chain of synchronizations.

Definition 4 Let Σ be a (possibly non maximal) subset of S containing only synchronization operations. Also, let $\min^+ : \mathbb{N}^P \to \mathbb{N}^+$ be a function that returns the smallest natural greater than zero from a set of P naturals. Also, let Φ be the set of all possible global clocks. The minimum clock is a function

$$\phi_{min} : \Phi \times \Sigma \to \mathbb{N} \quad \text{with} \quad (\phi, s) \mapsto \min^+(\phi_1(s), \ldots, \phi_P(s)),$$

i.e., that returns the value of the minimum non-zero local SIPS clock at s. We denote it interchangeably in the following way:

$$\phi_{min}(s) = \min^+(\phi(s)) = \min^+(\phi_1(s), \ldots, \phi_P(s)).$$ □

Figure 4.1 shows a snapshot at top s of a global clock from Definition 3 (bounded by M) and function ϕ_{min}.

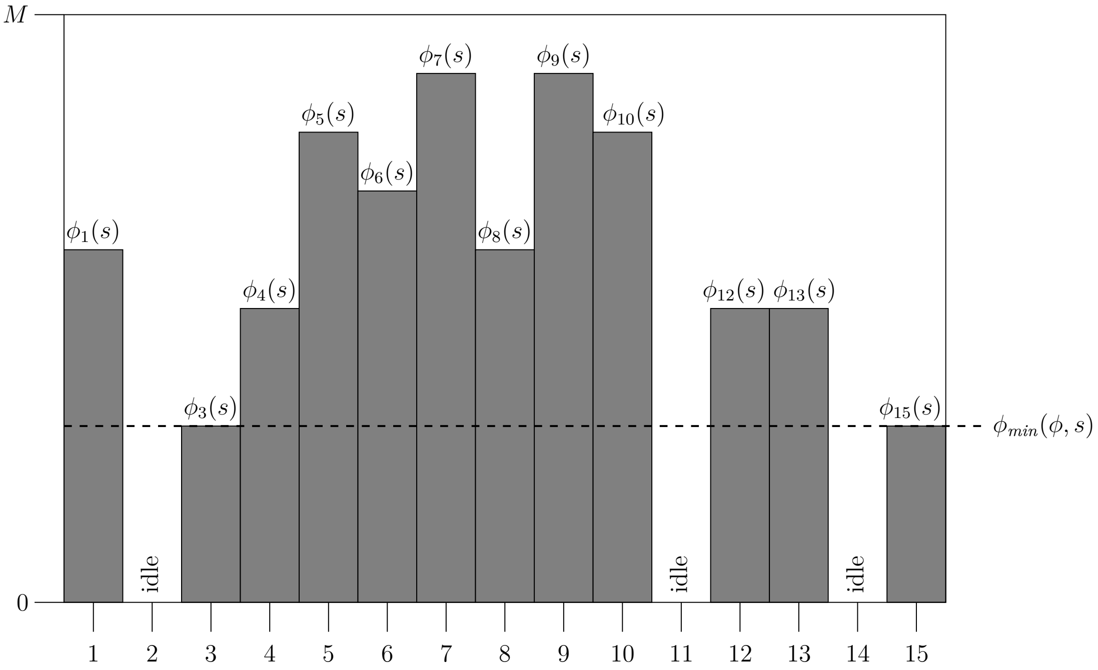


Figure 4.1: Example of a global clock at top s. Here, $P = 15$ and each active worker i has an value $\phi_i(s)$. In this example, $\phi_{min}(\phi, s) = \phi_3(s) = \phi_{15}(s)$. Workers 2, 11, and 14 are inactive (idle). They are not accounted in the calculus of $\phi_{min}(s)$, and have $\phi(s) = 0$.

By analyzing the different values ϕ_{min} may assume, we bound the worst-case computation since it is the value of the "most delayed" clock. The following lemmas apply. The first one shows that ϕ_{min} is an increasing function:

Lemma 1 *For all $s \in \Sigma$, $\phi_{min}(s-) \leq \phi_{min}(s+)$.*

Proof We show that at any given top it is impossible for the minimum clock to decrease, since all actions the runtime can undertake increase or keep its value. Suppose, without loss of generality, that worker i has the minimum clock, *i.e.*, $\phi_{min}(s-) = \phi_i(s-)$. By Definition 2, on top s it can either become idle — items (1.) and (2.) —, increase, or keep — items (3.) and (4.) — its clock value. Also, by Definition 3 it can participate in a synchronization and increase — item (2.) — its clock value. If it keeps its clock value, it remains the minimum clock, which remains the same. If it increases its clock value but is still lesser than the others, it remains the minimum clock, which increases. If it increases its clock value and it is larger than some other non-idle worker j or it is zeroed, than another worker j has now the minimum clock value at s. But, since by Definition 4 i has the minimum clock, we have $\phi_i(s-) \leq \phi_j(s-)$ and j has also increased or kept its clock value, than the minimum clock value has also remained the same or increased. In all cases, the value of the minimum clock never decreases. ■

We can also state that if all workers synchronized within a range of tops, then the minimum clock is strictly incremented after the last top on the range:

Lemma 2 *Let $s_0, s' \in \Sigma$ such that $s_0 \leq s'$ and the computation has not ended between s_0 and s' —* i.e.*, there was at least once clock value greater than zero. If $\forall i \exists j \in [1, P]$ such that $s(i,j) \in [s_0, s']$ or $s(j,i) \in [s_0, s']$, then $\phi_{min}(s_0) < \phi_{min}(s')$.*

Proof Suppose an adversary works to not increase the minimum clock value by choosing a new worker once the current owner of the minimum clock value has increased. Thus, at each time the current minimum clock value increases or is zeroed, the adversary has to select another worker with a clock value equal to the previous worker's clock value by Lemma 1. By Definition 3, any worker that synchronizes strictly increases its clock value and since we assumed all workers synchronized, after s' there is no worker whose clock value has not increased or is zeroed. In both cases, the adversary cannot choose any remaining worker of equal value and thus the minimum clock has increased. ■

A common upper-bound for all local clocks also bounds the global clock according to the *synchronization strategy*, *i.e.*, the function used by the thief — an idle worker — to choose its victim — an active worker. These are discussed next. We begin by proving the optimality of a simple scheduler where the thief always selects the victim with minimum clock value. Then, we prove the asymptotic optimality of a scheduler with random victim selection.

4.2 The Minimum Clock Strategy

An idle worker selects the active worker with the minimum clock. If more than one worker met this condition, then the victim is chosen randomly. If the selected worker is also idle or is being stolen by a third worker, then the thief picks the second minimum clock, and so on (in the case that there is no available worker to be stolen, it steals from the first one that becomes available and has the minimum clock).

Let us now see a tight upper bound on the number of steals in this case:

Theorem 2 *During a randomized work-stealing execution over P workers, let Σ be the subset of steal operations and ϕ be a local clock over Σ. Also let u be a random variable whose value is the number of occurrences of the steals in Σ and $\mathbb{E}[u]$ be its expected value. If there is a constant M such that $\phi_i(s) \leq M$ for all $1 \leq i \leq P$ active at s and all thieves* *follow the minimum clock strategy for victim selection, then*

$$\mathbb{E}[u] = M \cdot (P - 1).$$

Proof Since the local clocks are assumed to be upper-bounded by M, in the worst case ϕ_{min} goes from 1 to M before the computation ends according to Lemma 1. On this strategy, we select the worker with the minimum clock value to be synchronized at each time, and thus every worker has synchronized once after exactly $P-1$ tops. By Lemma 2, after every worker synchronizes at least once, ϕ_{min} necessarily increases. Thus, to each unit we increase ϕ_{min} from 1 to M we perform at most $P-1$ synchronizations. Therefore, the worst case number of synchronizations is $M(P-1)$. ■

This simple scheduler based on the SIPS clock value has a small overhead since the number of steals is strictly lesser than the number of workers. However, it suffers from contention, being on the family of work-pushing scheduling algorithm. Next, we use SIPS clocks to analyze the randomized victim selection case (as in ABP work stealing) in order to mitigate this contention.

4.3 The Random Selection Strategy

An idle worker selects an active worker randomly. If the selected worker is also idle or is being stolen by a third worker, then the framework retries, as discussed in Chapter 2, Figure 2.4.

Before proving the bound, we present the Coupon Collector's Problem and its solution, which is used as a component of the proof of the random case. We use it later to show, in expectation, how many steal attempts occur before every worker participates in at least one synchronization as a victim or thief.

Suppose that there is an urn, from which n different coupons are being collected, equally likely, with replacement. In probability theory, the coupon collector's problem asks the following question: How many coupons we expect one needs to draw with replacement before drawing each distinct coupon at least once? The answer is the following Lemma:

Lemma 3 (Coupon Collector's Problem) *Let T be the number of draws to collect all n distinct coupons at least once. Also, let T_i be the number of draws to collect the i-th* *different coupon after $i-1$ distinct coupons are already collected at least once. Thus, the expected total number of draws is*

$$\mathbb{E}[T] = n\sum_{i=1}^{n} 1/i = nH_n \tag{4.1}$$

where $H_n = \sum_{i=1}^{n} 1/i$ is the harmonic number. Moreover, the variance is Var$[T] < \frac{\pi^2}{6}n^2$.

Proof Consider T and each T_i as random variables. The probability P_i of drawing the i-th distinct coupon after $i-1$ distinct coupons were already collected is trivially $(n-(i-1))/n$. Therefore, T_i has geometric distribution with expectation $1/P_i$. Thus, as by linearity of expectation:

$$\begin{aligned}
\mathbb{E}[T] &= \mathbb{E}[T_1] + \mathbb{E}[T_2] + \cdots + \mathbb{E}[T_n] \\
&= \frac{1}{P_1} + \frac{1}{P_2} + \cdots + \frac{1}{P_n} \\
&= \frac{n}{n} + \frac{n}{n-1} + \cdots + \frac{n}{1} \\
&= n \cdot \left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}\right) \\
&= n \cdot \sum_{i=1}^{n} \frac{1}{i} \\
&= n \cdot H_n
\end{aligned}$$

Its variance is calculated also by linearity:

$$\begin{aligned}
\mathrm{Var}[T] &= \mathrm{Var}[T_1] + \mathrm{Var}[T_2] + \cdots + \mathrm{Var}[T_n] \\
&= \frac{1-P_1}{P_1^2} + \frac{1-P_2}{P_2^2} + \cdots + \frac{1-P_n}{P_n^2} \\
&< \frac{n^2}{n^2} + \frac{n^2}{(n-1)^2} + \cdots + \frac{n^2}{1^2} \\
&< n^2 \cdot \left(\frac{1}{1^2} + \frac{1}{2^2} + \cdots + \frac{1}{n^2}\right) \\
&< n^2 \cdot \left(\frac{\pi^2}{6} - \frac{1}{n} + \frac{1}{2n^2}\right) \\
&< \frac{\pi^2}{6}n^2
\end{aligned}$$

■

By mapping a steal attempt onto a coupon draw, we find how many steal attempts are required in expectation by one worker to query all other workers. Assuming that each global clock is upper-bounded by M synchronizations, we deliver a general worst-case expected upper-bound:

Theorem 3 *During a randomized work-stealing execution over P workers, let Σ be a subset of steal operations and ϕ be a local clock over Σ. Also let u be a random variable whose value is the number of occurrences of the steals in Σ and $\mathbb{E}[u]$ its expected value. If there is a constant M such that $\phi_i(s) \leq M$ for all $1 \leq i \leq P$ active at s, then*

$$\mathbb{E}[u] \leq M(P-1)H_P$$

Moreover, $\pi^2(P-1)^2/6$ is the expected variance.

Proof Since the local clocks are assumed to be upper-bounded by M, in the worst case ϕ_{min} goes from 1 to M before the computation ends according to Lemma 1. On this strategy, we select the worker randomly. By Lemma 3, in expectation we need $(P-1)H_P$ synchronizations before all workers synchronize at least once. By Lemma 2, after every worker synchronizes at least once, ϕ_{min} necessarily increases. Thus, to each unit we increase ϕ_{min} from 1 to M we have performed at most $(P-1)H_P$ synchronizations in expectation. Therefore, the worst case number of synchronizations is $M(P-1)H_P$. ■

Remark 1 The proof considers a loose bound of one idle worker per top motivated by worst-case analysis. Nevertheless, the local steps are usually performed in parallel, mitigating the $P-1$ factor. □

As a short example of SIPS analysis, consider a recursive computation of the n-th term on Fibonacci's series in parallel:

```
concepts <Integer N>
fib (N n) -> N
// precondition : n >= N (0)
{
    if (n < N (2)) return n ;
    N x = spawn fib (n - N (1)) ;
    N y = spawn fib (n - N (2)) ;
    sync ;
    return x + y ;
}

```

This code is exactly like on Subsection 1.4.2, but with keywords **`spawn`** and **`sync`** to unfold parallelism in a style close to Cilk, as displayed in Chapter 2. In it, maximum deque size $M = n-1$ is given by execution tree and Theorem 3 upper-bounds total number of steals to be $H_{P-1}(n-1)$. We, however, are not only interested in a bound for the total number of steals, but in bounds for some "kinds" of steals. How do we calculate, for instance, how many times a recursive call whose n is multiple of 3 is stolen? Theorem 3 allows us to define Σ to be all steals in the form $\texttt{fib}(3k)$ for any natural k. Thus, M is reduced to $\lfloor (n-1)/3 \rfloor$ and Theorem 3 delivers bound $(P-1)H_{P-1}(n-1)/3$.

4.4 Workload Partition Schemes

Since we elected, in Section 4.1, the steals to be the interesting synchronizations, let us consider a scenario where the clock increases only upon a successful steal, *i.e.*, only by the means of Definition 3 (2.). This implies that while a worker is active its clock will increase only when the worker is a victim of a successful steal. The maximum number of successful steals any computation can achieve has a loose upper-bound on the number of tasks. However, not all tasks may be subject to stealing, only the ones the algorithm puts on the deque. (This algorithmic decision is approached in details in the next section.) On the worst case, this is equal to the maximum possible size of a deque during the execution. Besides, the maximum number of steals a worker can suffer is also a function of *how many* tasks are taken at each steal. The later is what we refer to as the workload partition scheme. In this section, we discuss the impact of defining a workload partition scheme on M. (Here, workload means "tasks on the deque", while partition means "how many tasks leave and how many tasks stay on the deque".)

The workload partition scheme performed by the scheduler specifies which fraction of the deque will be taken by the thief from the victim on a successful synchronization. Since SIPS bounds orbit around M, this impacts on the global clock bounds calculated using SIPS.

From now on let us consider the maximum length of any deque as n.

The first scheme we examine is "back-most task" partition. Since we take sequentially, from back to the front, one task, we have $M = n$. This is the default policy on ABP work-stealing. With minimum clock victim selection strategy we expect at most $(P-1)n$ successful steals (Theorem 2). With the randomized victim selection strategy, we expect at most $(P-1)nH_P$ successful steals (Theorem 3).

The second scheme we examine is "half of tasks" partition. If the maximum size of the deque is n tasks, and we can only take half of the tasks each has at a given time — rounded-up — a worker can at most suffer $\log_2 n$ steals. This is usually used on adaptive algorithms, as exemplified on Chapter 5. With minimum clock victim selection strategy we expect at most $(P-1)\log_2 n$ successful steals (Theorem 2). With the randomized victim selection strategy we expect at most $(P-1)\log_2(n)H_P$ successful steals (Theorem 3).

The two schemes above are the most popular on runtimes scheduled by work-stealing. The first one, "back-most task" is used on the runtime of various implementations of Intel's Cilk (where it is referred as "top-most task"). The second one, "half of tasks" is the default policy on the Kaapi framework, which also supports adaptive parallel algorithms in an "out-of-the-box" fashion.

An hybrid strategy is also considered, where a thief steals "some k tasks" from the deque (where "some" may mean "smaller", "any", *etc.*). In this case, the maximum number of successive steals a worker may suffer is n/k. With minimum clock victim selection strategy we expect at most $(P-1)\log_2 n$ successful steals (Theorem 2). With the randomized victim selection strategy we expect at most $(P-1)\log_2(n)H_P$ successful steals (Theorem 3).

Several works (*e.g.*, (MICHAEL; VECHEV; SARASWAT, 2009; LIMA et al., 2013)) change the workload partition policy to favor some criteria. For instance, one might choose to steal the task with more affinity or closer (in terms of the locality) to the thief. With a given selection criteria, our work supports the analysis of those cases.

We highlight that, since one task may spawn a variable number of other tasks, stealing a task is not the same as stealing one n-th of the work W. Let us take as example Cilk's scheduler, which implements ABP work-stealing. Cilk's DAG is necessarily a fully-strict complete tree where each parent node has σ more work than its children, where σ is the degree of the tree. Thus, one does not steal an equal share of the workload by stealing top-most, but a major part of the work. If one spawns two tasks per node, for instance (say, the naïve Fibonacci example of earlier), each steal will take half of the work, despite stealing one n-th of the tasks. Therefore, our analysis is not directly dependent on W.

At first sight one may argue that the "largest possible size of any deque" is tied to the work W, since a loose bound to it is the total number of tasks, which is W by definition. Thus, our claim does not take W into account would be false. However, as we stated, W would be a loose bound, since the algorithm may not place all tasks on its deque — in fact, it frequently does not. This placement, affected by the order on which the tasks are spawned, are the subject of discussion in the next section. We also show that our analysis is not also directly dependent on D.

4.5 Asymmetrical Parallelism

As we indicated at the beginning of the previous section, the workload partition scheme is not the only factor contributing to the bound of the local clocks, since the way algorithms allocate tasks on the deque is a factor of equal importance. Now we proceed to show an example of a hidden constraint on parallel programs that classical analysis is unable to find but is directly approached by SIPS. We call it "asymmetrical parallelism". The work-stealing scheduling mechanism is clearly meant to be symmetric, *i.e.*, the order of parallel calls should not impact on the algorithm, since they will be executed potentially in parallel. However, although oblivious to the programmer, the work-stealing policy impacts, and largely, on the worst case number of synchronizations, as SIPS analysis shows below. Work-stealing schedulers have to choose which task will progress on sequential execution and which one goes to the local repository. We discussed this briefly on Chapter 2 when debating fork/join *vs.* spawn/sync.

Consider four generalized versions of the `fib` algorithm in Figure 4.2 named `fa`, `fb`, `fc`, and `fd`, all displayed on Figure 4.2. They all implement a variation of an algorithm that we name f_k. Each one calculates $f_k(n) = f_k(n-1)+f_k(n-k)$ with $f_k(0) = f_k(1) = 1$. For $k = 2$, f_k describes the Fibonacci's series. Each version varies on the order of the spawns and the use of the keyword **`spawn`** on the second recursive call.

For each one of the four variations on Figure 4.2, the work W and D verify the following equations for $n \geq k$:

$$W(n) = W(n-1)+W(n-k)+\Theta(1) \quad \text{and} \quad D(n) = \max\left(W(n-1), D(n-k)\right)+\Theta(1)$$

where $D(n) = \Theta(n)$. Therefore, the work and depth do not allow one to distinguish the four variants in terms of performance. For instance, the limits found in the paper by Arora *et al.* (ARORA; BLUMOFE; PLAXTON, 1998) for ABP deliver an expected number of steal requests bounded by $\mathrm{O}(PD) = \mathrm{O}(n)$ for the four variants.

Let us now analyze the four variants using SIPS clocks on the randomized victim selection strategy following ABP work-stealing.

In fact, the four programs do not create the same tasks and do not have the same critical path length in terms of the number of stolen tasks. Since we are on the spawn/sync model, at each spawn, the worker starts the execution of the newly created task and enqueues the remaining of the spawner task on its local deque. It is this task that can be

```
concepts <Integer N>
fa (N n, N k) -> N
// precondition : n >= N (0)
// precondition : k >  n
{
    if (n < k) return n ;
    N x = spawn fa (n - N (1)) ;
    N y = spawn fa (n - k) ;
    sync ;
    return x + y ;
}
```

```
concepts <Integer N>
fb (N n, N k) -> N
// precondition : n >= N (0)
// precondition : k >  n
{
    if (n < k) return n ;
    N y = spawn fb (n - k) ;
    N x = spawn fb (n - N (1)) ;
    sync ;
    return x + y ;
}
```

```
concepts <Integer N>
fc (N n, N k) -> N
// precondition : n >= N (0)
// precondition : k >  n
{
    if (n < k) return n ;
    N x = spawn fc (n - N (1)) ;
    N y =       fc (n - k) ;
    sync ;
    return x + y ;
}
```

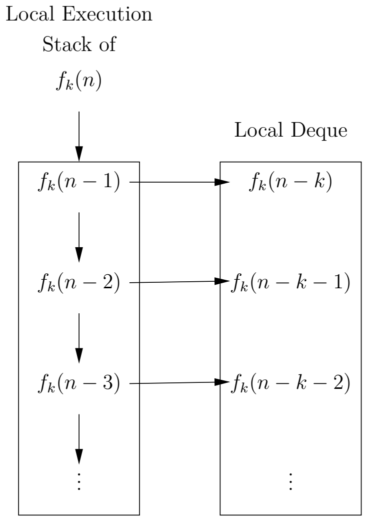
```
concepts <Integer N>
fd (N n, N k) -> N
// precondition : n >= N (0)
// precondition : k >  n
{
    if (n < k) return n ;
    N y = spawn fd (n - k) ;
    N x =       fd (n - N (1)) ;
    sync ;
    return x + y ;
}
```

Figure 4.2: Four parallel programs, `fa`, `fb`, `fc`, and `fd`.

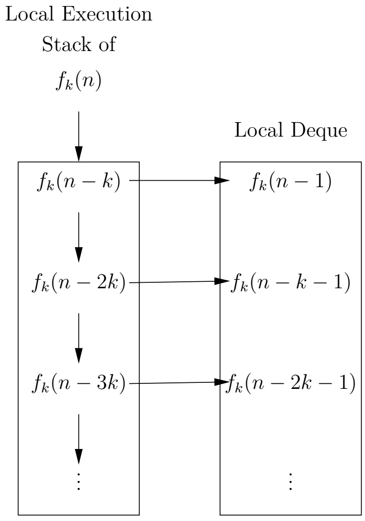
eventually stolen by a thief. Contrary to intuition, thus, the programs `fa` and `fc` (resp. `fb` and `fd`) are almost equivalents. The only difference is that `fa` (resp. `fb`) generate on the task path one extra task in addition to the tasks generated by `fc` (resp. `fd`). This extra task is the continuation of the second spawn and does not contribute to the work of the parallel program.

Let M_a (resp. M_b, M_c, M_d) the worst case maximum number of successively stolen tasks on the worker that calculates `fa (n)` (resp. `fb (n)`, `fc (n)`, and `fd (n)`) — and, thus, the maximum value their SIPS clock can achieve on top-most task workload partition strategy. This way, as illustrated by Figure 4.3, the size of the deque of ready tasks contains at most $n - 2k$ tasks for `fc` and `fa` and n/k tasks for `fb` and `fd`. For all under threshold values of n named as n', such that $n' < k$ we have $M_a = M_b = M_c = M_d = 0$ and, for $n' \geq k$,

- $M_c(n) = \max\left(M_c(n-1), 1 + M_c(n-k)\right) = \left\lfloor \frac{n}{k} \right\rfloor$,
- $M_d(n) = \max\left(M_d(n-k), 1 + M_d(n-1)\right) = n - k + 1$,
- $M_a(n) = M_c(n) + 1 = \left\lfloor \frac{n}{k} + 1 \right\rfloor$,
- $M_b(n) = M_d(n) + 1 = n - k + 2$.



(a) Case `fa` and `fc`.



(b) Case `fb` and `fd`.

Figure 4.3: Execution stack and deque for f_k variations.

As shown, each one of the four variations has a different upper-bound for their local clocks and, therefore, their worst-case number of steals will be significantly different when we apply this bound to Theorems 2 and 3. If each steal introduces parallel overhead — such as the algorithms we discuss ahead — their work would be different as well. Classical analysis that consider the spawning of tasks to be symmetrical are unable to catch these differences. Also, if the spawns are not always performed in some quantity and/or order (like on randomized algorithms), the classical analysis is unable to deliver reliable bounds.

4.6 Work-Efficiency and Work-Optimality

As recurrent notation, a parallel algorithm operates over P workers and input size n and has work

$$W(n) = W_{\text{seq}}(n) + V(n),$$

where $W_{\text{seq}}(n)$ is the sequential work and $V(n)$ is the parallelism overhead due to synchronizations.

Now, provided with a framework to estimate tight bounds on synchronizations in parallel computations, we may classify algorithms coherently with respect to overhead introduced by synchronizations performed by the scheduler.

A parallel algorithm is now defined to be work-efficient *iff* its synchronization overhead is not asymptotically larger than the work parallelized, *i.e.*,

$$W(n) = W_{\text{seq}}(n) + V(n) = \mathrm{O}(W_{\text{seq}}(n)) \tag{4.2}$$

A parallel algorithm is hereby defined to be work-optimal *iff* its synchronization overhead is polylogarithmic, *i.e.*,

$$V(n) = \mathrm{O}({\log_2}^{\mathrm{O}(1)} n) \tag{4.3}$$

This definition implies that the synchronization overhead cannot be asymptotically mitigated by parallelism any further. It belongs to the same category of parallel binary search and exponentiation; these problems have polylog time complexity even for $P = 1$.

4.7 Closing Remarks

This chapter introduced SIPS clocks, theoretical devices that allow one to analyze the number of synchronizations on concurrent algorithms. SIPS does not rely directly neither at work nor depth of a parallel computation and thus is more flexible than classical analysis. Some of its concepts are drawn from Lamport's logical clocks (LAMPORT, 1978).

Bounding the number of synchronizations on a given computation also allows us to bound the overhead introduced by communication on the parallelization of algorithms. In order to provide a taxonomy and parameter of efficiency we define parallel algorithms to be work-efficient when the parallel overhead is not asymptotically greater than the sequential work, and work-optimal algorithms, where the parallel overhead is asymptotically polylogarithmic. Applications of these concepts of efficiency and optimality will be shown in the next chapter and mainly on the analysis performed on the second part of this thesis.

On the next chapter, we examine adaptive parallel algorithms, whose primary trait is to adapt to the runtime without being parametrized. The degree of guaranteed transparent efficiency is essential to the construction of the generic algorithms we present on Part II. However, since the number and the relative order of synchronizations on adaptive algorithms may change accordingly to the runtime. Classical analysis is unable to handle this category of programs, since it accounts only for Work and Depth. In that, SIPS comes at hand, providing a very fit analysis framework to this kind of algorithms.

5 CASE STUDY: ADAPTIVE ALGORITHMS AND POLYNOMIAL EVALUATION SCHEMES

In this chapter we overview the concept of adaptive algorithms (Section 5.1) and their components (Section 5.2), showing how SIPS is useful to analyze their behavior over a simplified model more likely to be implemented by a wider range of middlewares (Section 5.3).

As a case study, we design and analyze an adaptive algorithm for polynomial evaluation (Section 5.4). We discuss two schemes within it: Horner's Method, the best solution possible (in number of additions and multiplications), programmed in sequential, and the state of the art parallel solution, Estrin's Method, which introduces parallel overhead in the form of extra multiplications. Finally we propose an adaptive implementation for Horner's Method in parallel, moving extra multiplications to successful steals and eliminating them when the program is run in sequential. We show our implementation to be more efficient than Estrin's Method in expectation through SIPS analysis and measure the execution round by round using a discrete event simulator (Section 5.5).

Adaptive algorithms are the primary tool we use to implement efficient algorithms in the chapters to come. By using this kind of algorithm we are able to write code that adapts to the heterogeneous parallel execution environments without being parametrized (*e.g.*, not using a threshold value guessed to amortize the parallel/sequential ratio on a given machine).

The chapter ends with closing remarks on the family of works that introduced the notion of adaptive algorithms and related content (Section 5.6).

5.1 Definition of Adaptive Algorithms

We start the discussion by defining adaptive algorithms. The base is the taxonomy for parallel algorithms stated by Cung *et al.* (CUNG et al., 2006). 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)."

An adaptive algorithm, thus, is any computational procedure capable of changing its behavior automatically according of its execution context — manipulated data, runtime configuration parameters, the load of resources — to reach optimal performance. The type of decision it makes induce classes:

Resource-Aware. An adaptive algorithm whose decision strategy is based on the configuration of its execution environment — *e.g.* number of workers, the size of caches, bandwidth, *etc.*

Resource-Oblivious. An adaptive algorithm whose decision strategy does not depend on any execution parameter, but only on actions taken by the runtime — *e.g.* data moving, scheduling of tasks.

An example of resource-aware algorithms is cache-oblivious (FRIGO et al., 2009). It explores the memory hierarchy efficiently without information about its structure or size. There are algorithms in this category that are asymptotically optimal, like Fast Fourier Transform and Matrix Multiplication, *etc.* These are frequently based on divide and conquer techniques and require some re-writing of the previous code. The division strategy considers the memory hierarchy.

A *processor-oblivious* algorithm (BERNARD; ROCH; TRAORÉ, 2008) is a parallel algorithm that does not know neither the number of workers participating in the computation nor their speed. This is valid for any instant during execution. It also holds if the number of workers is fixed or changes dynamically. Ideally, no information about the workers is needed, although implementation constraints may require some underlying data. In Chapter 7 we use the information about a worker's unique identification to execute a callback procedure when a steal occurs; if the framework implemented steal callbacks, it would not be necessary.

Using processor-oblivious algorithms does not only provide more power in form of abstraction but also enables programming resources such as:

Parallel overhead management. The parallel overhead is "paid" only when parts of the algorithm run in parallel. In other words, even if P workers are available during execution the overhead is only paid at each time a parallel task is scheduled to one of those in P. *Ergo*, if the parallel algorithm runs over only one worker, then no parallel overhead is “paid”.

Automatic granularity control. The control between sequential and parallel variations of the algorithms is moved from the program’s logic to the scheduler’s. Besides, the threshold definition is placed inside a re-usable data structure rather than on algorithm code, allowing one to vary it for a family of algorithms orthogonally.

5.2 Components and Organization of Adaptive Algorithms

Let us now examine parallel list-processing algorithms implemented in a processor-oblivious fashion (TRAORÉ, 2008). We then generalize it to the artifacts the scheduler shall provide to ensure proper execution of adaptive algorithms.

List-processing is a higher-order algorithm that will receive a function to be applied to each element of a list. Non-exhaustive examples are:

Reduce. Uses the list elements as operands in an associative binary operation given as parameter.

Prefix. As reduce, but returns another list, containing the sub-results of applying the associative binary operation “left-to-right”.

Map/Transform. Outputs another list where each element is the result of applying an unary function passed as a parameter to the respective (position-wise) element on the input list.

Filter/Selective Copy. Copies all elements satisfying an unary predicate function passed as a parameter to another list.

Linear Search. Finds any occurrence of a given element and returns its position on the list if any.

As demonstrated by Traore *et al.* (TRAORÉ et al., 2008), an adaptive version of list processing family of algorithms may be implemented by modifying the ABP work-stealing algorithm into a deque-free work-stealing scheduling. The deque-free algorithm replaces the deque on each worker by a range on the input double-linked list. This range is defined as a pair of indexes (the first element and one past the last element) that describes a task. The input list is shared among all workers and each worker is responsible for processing *part* of it and combining it with other sub-results as needed. Splitting and merging the list among workers is a constant-time operation that only returns the respective resulting ranges. Since a task is a range described as a pair of indexes representing an half-open interval, task stealing performed by the scheduler is also performed in constant time — there is *no data copying.* In fact, this implementation model enables theoretical bounds that supposes constant-time on steal operations, like the ones discussed in Subsection 3.1.2.

The algorithm requires the data structure describing the different ranges to have at least two splitting methods. Here, we consider an initial range $[f_0, l_0)$ of size m and each worker at a given top may own a subrange $[f, l)$, being active, or is trying to steal from other workers, being idle. (The same considerations about the atomicity of pop-front and pop-back from Chapter 2 hold.) The splitting procedures are:

Extract Seq. Atomically split elements from the range's front. It is the equivalent of a "pop-front" in the deque. Function `extract_seq` consists in extracting a range of elements of size $\alpha \log_2 m$. It is used by the worker to extract sequential work from the range it owns to execute locally. The constant α, whose default value is 1, is used to fine-tune the algorithm in taking smaller or larger fractions, multiple of $\log_2 m$ The fraction is logarithmic in order to allow a frequent and fast use of the operation when compared to the parallel extraction we detail next.

Extract Par. Atomically split elements from the range's back. It is the equivalent of a "pop-back" in the deque. Considering that m' elements were already sequentially processed, function `extract_seq` consists in extracting a range of elements $[f', l')$ of size $\lfloor m - m'/2 \rfloor$ (half of what remains). Idle workers perform it over active ones within steal operations. The minimum size we allow the algorithm to steal is $\sqrt{m}$ since the increasing of m will increase at the same time the size of the grain and number of grains. It is used by a thief to steal parallel work from a victim in the form of a sub-range.

Figure 5.1 shows, top-down,

1. An initial state after $k - 1$ operations `extract_seq` were performed, whose covered range is marked in black.
2. The performing of an `extract_seq` operation taking a subrange of size $\alpha \log_2 m$, marked in gray, and advancing f accordingly.
3. The performing of an `extract_par` operation taking a subrange $[f', l')$ of size $\max\left(\lfloor (m - m')/2 \rfloor, \sqrt{m}\right)$, marked in gray, and regress l accordingly. Here, $m' = k\alpha \log_2 m$, since until that moment we suppose k operations `extract_seq` were performed.

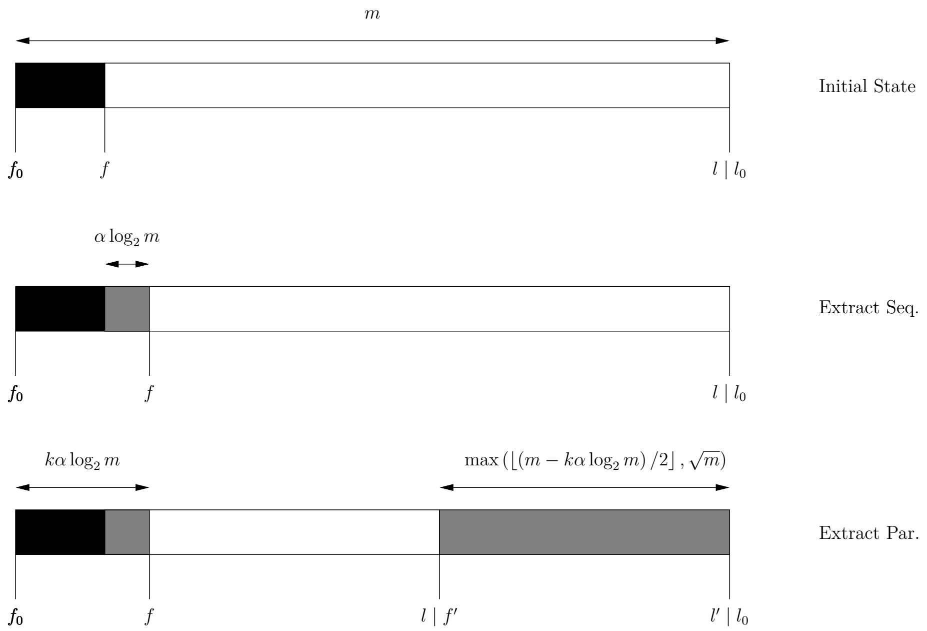


Figure 5.1: Procedures `extract_seq` and `extract_par`. The whole block is the processed range, with the relevant intervals $[f_0, l_0)$, $[f, l)$, and $[f', l')$ indicated on the ticks below. Black intervals represent already processed elements. Gray intervals show elements encompassed at the current or previous steps by one of the primitives, Extract Seq. or Extract Par. The doubly pointed arrows indicate an interval's size.

The number of times operation `extract_par` is executed determines the parallel overhead the algorithm will introduce. The analysis of the number of times a given synchronization operation occurs is precisely the situation SIPS is designed to handle. We show, as an example of application, the analysis of a processor-oblivious polynomial evaluation algorithm on Section 5.4 over a simplified adaptive algorithm implementation described in next section.

Besides Extract Seq. and Extract Par. there are other procedures employed by the scheduler in the execution of adaptive algorithms. They are:

Local Run. Executes the *best sequential algorithm* over the range extracted by `extract_seq` without *any* synchronization or arithmetic overhead. A non-idle worker executes it through primitive `local_run`.

Merge. It merges the results from both the work performed locally, and the work performed remotely upon an extraction, combining the partial calculations performed in parallel. The victim performs a merge through an operation called `join` while the thief uses its counterpart called `finalize`.

Jump. It allows a worker to skip the work extracted by an eventual thief and proceed to process the remaining work and execute it sequentially. The respective primitive is `jump`.

The programmer/middleware is in charge of defining data structures supporting these operations. Composed types such as `Work`, `WorkAdapt`, `JumpWork`, and `FinalizeWork` have to offer both synchronization and performance guarantees as the ones described by the list processing methods we discussed, mainly `extract_seq` and `extract_par`. These user-defined data structures are the straightforward way to generalize the considerations we traced for list processing algorithms into an encapsulated work data type. Changing the list-based approach where elements are the processed entities to other linear data structures (sets, hashes, vectors, *etc*) with pointers/iterators to tasks is a common trait on the implementation of adaptive algorithms.

The scheduling algorithm work as the work-stealing algorithm from Chapter 2, with few modifications, mainly on the non-blocking synchronizations. The synchronization protocol between concurrent thieves and remote ranges is more complex than ABP's. The later uses a simplified version of THE protocol by Dijkstra, to use a lock only on an improbable/infrequent last-element competition between local worker and thief. In deque-free, however, the steal is not atomic, and the protocol has to manage the situation where `extract_seq` and `extract_par` try to get overlapping data. Kaapi, for instance, implements a protocol where the range is first copied (through its indexes, in constant time) and only then overlap is tested. If there is no overlap the copied range is "cut-out" from the original range; otherwise there is a retry.

The complete protocol for the equivalent of micro and nano-loops are described thoroughly in works by Daoda Traoré (TRAORÉ et al., 2008) (TRAORÉ, 2008). However, since only Kaapi has full built-in support for it, we will introduce a simplified version of this algorithm that fits better the limitations of other multithreaded middlewares.

5.3 A Simplified Approach

We focus on and introduce a simplified version of the adaptive algorithm scheduling designed to work along with Cilk Plus, our middleware of choice (see Chapter 2), which does not support the required primitives directly in its spawn/sync scenario. Other middlewares, however, offer distinct degrees of support. Our version, based on nested parallel recursive calls, allows one to implement a semantically equivalent algorithm with same asymptotics over any middleware supporting fork/join or spawn/sync styles. We derive it from the Cilk Plus version, explained after brief considerations over the Kaapi and TBB versions.

Kaapi offers to the programmer a partitionable list of work to be used as its local deque within the runtime. This allows one to use deque-free work-stealing on its top. The work list is implemented as a double linked list and implements a fast synchronization protocol between thieves and the local worker. Moreover, thanks to its cooperative mode, concurrent thieves may all acquire some portion of the work-list, speeding up synchronizations.

In TBB, adaptive algorithms are supported through structures called auto partitioners. It is a mechanism that chooses granularity dynamically in function of worker's activity. As a side effect, it limits the number of steals of the scheduling algorithm. Auto partitioners work along data range structures. They are responsible for dividing data ranges in block obeying a threshold proportional to the number of workers. When a steal occurs, divide a block in two and give half away to the stealer. The source code is written closer to what is proposed as the processor-oblivious algorithm, but this "half" of the block is static, as in Cilk Plus.

In Cilk Plus, a reminiscence of processor-adaptive algorithms is implemented through a combination of the primitive `cilk_for` and hyper-objects, detailed below. The `cilk_for` uses internally the spawn-sync schema, by dividing the range in two recursively as tasks on the deque. The compiler translates a structure in the form

```
cilk_for (N low = 0 ; low < high ; ++ low) { body (low, high, data) ; }
```

to the listing

```
1        concepts<Integer N, Function F, Type T>
2        cilk_for (N low, N high, F body, T data, int grain) -> void
3        // invariant : low        >=   N (0)
4        // invariant : high       >=   N (0)
5        // invariant : high - low >=   N (2)
6        // invariant : grain      >= int (1)
7        {
8            tail :
9            N count = high - low ;
10           if (count > grain) {
11               N mid = low + count / 2 ;
12               spawn cilk_for (low, mid, body, data, grain) ;
13               low = mid ;
14               goto tail ;
15           }
16           body (low, high, data) ;
17       }
18
```

In this listing:

1. On line 1 the concepts Integer, Function and Type are declared. Integer concept is already discussed on Chapter 1. The concept `Function` designates a callable

type, *i.e.*, a type whose instances have defined the operator `()` with any arity. The concept `Type` designates *any* type of the variables containing the data of the loop body.

2. On lines 3 to 5 we show the invariants: `N` holds values greater than zero, the range's length shall be greater than two, and the grain should be larger than one.
3. On line 7 the beginning is marked with the label `tail` because tail recursion optimization is employed to cut the rather expensive recursive calls by half.
4. On lines 9 to 13 the calculus is performed; if the range is larger than the grain size it calls the function recursively in parallel — using **`spawn`** — for the first half of the range and update `low` to simulate the recursion on the next loop.
5. On line 15 the under threshold sequential call, without overhead, is called.

In fact, function `body` and structure `data` usually do not exist in the code; the compiler synthesizes it from the loop's body and variables used within it and rewrites the code in terms of an anonymous function and shared data structure. The grain size is also determined by the middleware translation upon analysis of the hardware.

A `cilk_for` loop is usually used along with a Cilk hyper object. Introduced in a 2009 paper (FRIGO et al., 2009), hyper-objects are data structures that represent "mergeable" objects. These data structures, implemented through a pre-defined interface, are handled my the Cilk Plus framework, allowing the programmer to use them inside a parallel loop without worrying about concurrent accesses. The framework builds copies of the object seamlessly and merges then through an operation defined by the user on the hyper object implementation. Critical sections and concurrent write accesses are decided by underlying synchronization protocols and the paper show to be highly improbable to occur any waiting.

Remains the question: is this approach capable of emulating the adaptive behavior? The answer is *yes*, although not perfectly. In this case, the deque will hold the correspondent range indexes. Each task, in its turn, will be *half* of the currently available range, the other being processed recursively. Thus, top-most task holds 1/2 of the elements, second top-most, 1/4, third top-most, 1/8 and so on, until the threshold of a virtual `extract_seq` is reached. The emulation is not perfect because this method has a fixed partition for any steal, instead of a dynamic one. Nonetheless, this is sufficient for the needs of our algorithms. And, as a bonus side-effect, there is a smaller overhead than on systems like Kaapi that use copy-guessing mechanism for synchronizations. Figure 5.2

shows a modified version of Figure 5.1, marking where the fixed partitions will occur at a first moment, keeping in mind that each sub-interval will call the process itself recursively; the equivalent of `extract_seq` will always take a sub-interval on the size of the grain.

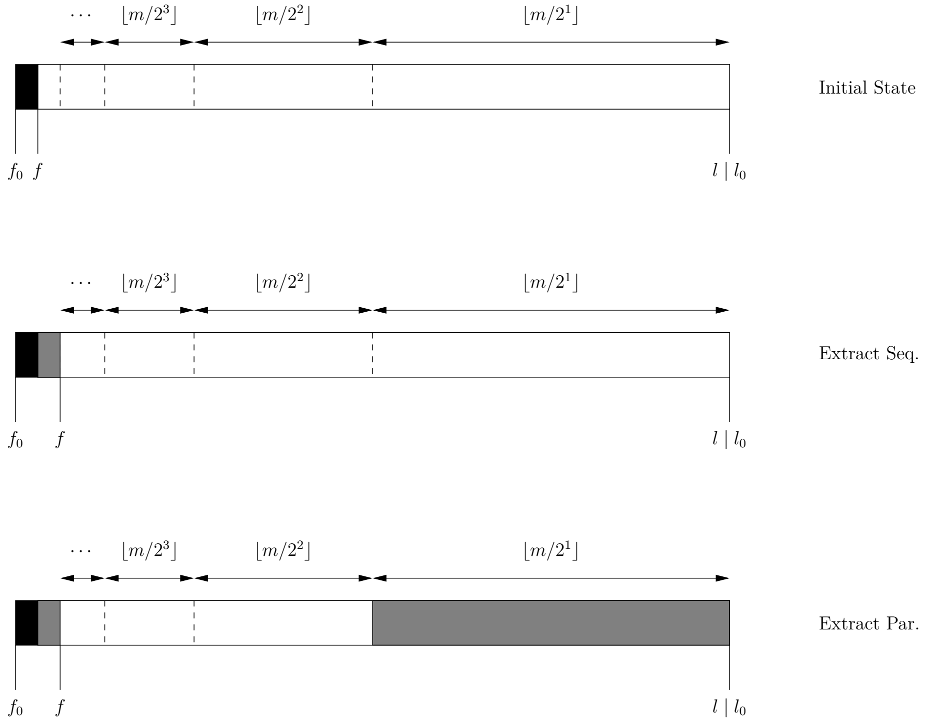


Figure 5.2: Procedures `extract_seq` and `extract_par`, Cilk Plus version. Elements are like on Figure 5.1. Dashed lines are marks for half, one-quarter, one-eight, *etc* from current data block.

Our approach is an optimization of the Cilk Plus translated code. We generalize the translation performed by the Cilk compiler in a more efficient approach to each algorithm on the top of any spawn-sync scheduler. Before attempting to write it, we describe the list representation based on concepts called iterators. Iterators are an abstract generalization of pointers, the address of a variable in memory. Although arrays and vectors usually instantiate iterators as pointers — because, since there is a direct mapping to modern architectures, it is fast —, some other sequential access data structures may use non-linear underlying organization. For instance, C++'s sets are implemented with red-black trees (PLAUGER et al., 2000). Iterators, provided by the container, work as a contract between the container implementer and the algorithm, where linear access interface is provided, and underlying work is embedded.

Although we refer to iterators throughout the thesis we frequently use forward iterators, a subset of the concept, delivering types where at least deference and increment operations are available. For a variable `i` of regular type `I` over the forward iterator concept, the operation is defined by:

`*i` Returns the contents of variable pointed by `i`.

`++ i` Increments `i`, pointing to position `i + 1` in the sequence.

(Since all forward iterators are iterators, we refer to the whole class as iterators from now on.)

Our approach uses a type function to express the distance between iterators. Type functions are functions that receive a type and return a type as well. Its calculus is performed in compile-time. To differentiate a type function from a standard function, we enclose its arguments with "`<`" and "`>`" instead of "`(`" and "`)`". We use it here to extract auxiliary types that will enable us to write generic code fitting an abstract structure. For instance, as an auxiliary type-function we use `Dist<I>`, which returns an integer type able to address the number of elements on the range. In the absence of an explicit implementation, `Dist<I>` returns type `size_t` a macro to a type capable of representing directly the largest positive integer on the machine. Finally, one important type function over iterators is `Val<I>`, which returns the type of the variable pointed by `i`.

Some iterator types offer random access operations in constant time. These are called indexed iterators, meaning that for one iterator `i` one can access its n-th successor with three equivalent operations,

`*successor (i, n)`, `*(i + n)`, and `i[n]`

in the same constant time as `*i`. Function `successor` returns an iterator whose content is the n-th element after `i`. To calculate the number of elements between `first` and `last` iterators make available the primitive `distance (first, last)`. While forward iterators implement it through a loop on operator `++` in linear time, indexed iterators perform it in constant time.

With what we have now at hand one may write a myriad of algorithms, like, for instance, linear search:

```
concepts <Iterator I>
find (I first, I last, Val<I> value) -> I
// precondition : distance (first, last) >= Dist<I> (0)
{
    while (first != last && *first != value) ++ first ;
    return first ;
}

```

This code iterates over a semi-open range [`first`, `last`), where `last` points to the past-end position of the vector. (For an insightful discussion on semi-open intervals to describe lists, see the seminal book by Dijkstra (DIJKSTRA, 1997)). One clear benefit of using a semi-open interval is that when `first == last` an empty list is described. We use iterators extensively in the next section and the second part of this thesis.

We are now apt to write the tail recursive code of `cilk_for` in a more generic and efficient way. We replace `cilk_for` for `parallel_list`, an generic scheme to process algorithms represented as lists:

```
1        concepts <Iterator I, Function F>
2        parallel_list (I first, I last, Dist<I> grain, F local_run) -> void
3        {
4            Dist<I> count = distance (first, last) ;
5            while (! (count < grain)) {
6                halve (count) ;
7                I mid = successor (first, count) ;
8                spawn parallel_list (first, mid, body, data, grain) ;
9                first = mid ;
10           }
11           local_run (first, last, data) ;
12       }
13
```

This listing:

1. On line 2, the indexes are replaced by the iterators and body function is replaced by the `local_run` function described before. We also omit `data` because it is contained within the iterators through operator `*`.
2. On line 4 replaces the subtraction of indexes by a call to `distance`. This enables the code to act on indexed iterators within constant time but also supports non-constant time iterators. While the subtraction on the Cilk version takes constant time, it is inside the loop, thus being executed in linear time. Anyhow the function call was moved to outside the main loop; even when using forward iterators, the constant time would be required just once.
3. On line 6 and 7 we replace the previous sum and division by two by an action named `halve` and a call to the successor. We do it first because halve is able to perform a faster split in half than dividing by two, since division is an expensive constant time operation and halve can take profit from a binary integer representation and perform a constant time bitwise operation, *e.g.*, shift-right. (Compilers may replace the “`/2`” by a shift-right nowadays, but there are no guarantees.) We rely on smart inlining optimization from the compiler to eliminate the function call or implement `halve` as a macro.

We also eliminated the **`goto`**-based loop in exchange for a **`while`** constructor in order to enable the processor and compiler to use branch prediction more efficiently.

The generic scheme presented above is used as a model to alike algorithms rather than a component. It allows us to dismiss the need for hyper-objects and apply it on the top of any spawn/sync based parallel middleware. In the next section, we show an adaptive version of polynomial evaluation using this approach. We analyze it with SIPS, then.

5.4 An Adaptive Polynomial Evaluation Scheme and Its Analysis

We present now algorithms to evaluate a polynomial in parallel. Three implementations are inspected: a naïve, the summation; Horner's Scheme (KNUTH, 1997b), the optimal sequential version; and Estrin's Scheme (ESTRIN, 1960), the state-of-the-art parallel version. The algorithms are refined until obtaining a work-optimal adaptive version that combines the best of all worlds.

First, a naïve algorithm for polynomial evaluation, *i.e.*, the result of $p(x) = \sum_{i=0}^{n} a_i x^i$. The polynomial is a list described by an iterator range. The evaluation is performed "left to right" with ascending degrees:

```
concepts <Iterator I, Semiring T>
polyeval (I first, I last, T x) -> T
// precondition : distance (first, last) > Dist<I> (0)
{
    T z = T *(first ++) ;
    T i = x ;
    while (first != last) {
        z += T (*first) * i ;
        i *= x ;
        ++ first ;
    }
    return z ;
}

```

We only multiply each element by x to the power corresponding to the position of the element, keeping the result to not perform redundant multiplications. Concept `Semiring` represents an algebraic structure with two binary operations defined, each one a monoid. It is employed to designate any type with addition and multiplication, with its proper identity elements, zero and one (or analogous). For a polynomial of size/degree n, this code performs $2n - 1$ multiplications and $n - 1$ additions:

$$W(n) = 3n - 2.$$

Horner's Scheme is the algorithm that performs the smallest (optimal) number of arithmetic operations to evaluate a polynomial, as reviewed by Knuth (KNUTH, 1997b, *p.* 486–488). Looking at the polynomial in its unfolded form

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

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. Noticing that it evaluates the polynomial "from right to left", we write

```
1         concepts <Iterator I, Semiring T>
2         horner (I first, I last, T x) -> T
3         // precondition : distance (first, last) > Dist<I> (0)
4         {
5             T z = T *(first ++) ;
6             while (first != last) {
7                 z = z * x + T (*first) ;
8                 ++ first ;
9             }
10            return z ;
11        }
12
```

For a polynomial of size/degree n, this code performs $n-1$ multiplications and $n-1$ additions:

$$W(n) = 2n - 2.$$

In current hardware it may be implemented even more efficiently, since a multiplication followed by an addition – the right side on line 7's assignment – is frequently replaced by compilers for a primitive that corresponds to a single processor instruction, usually labelled `fma` (fused multiply-add).

Since each iteration is dependent from the previous one, we cannot just divide the range and compute both halves in parallel. (Addition and multiplication are associative, but `fma` is not.) To profit from parallelism, another evaluation scheme is usually used, Estrin's (ESTRIN, 1960). It uses the fact that we can write

$$p(x) = (a_0 + a_1x) + (a_2 + a_3x)x^2 + ((a_4 + a_5x) + (a_6 + a_7x)x^2))x^4 + \cdots,$$

breaking the dependency between the two halves of a polynomial recursively.

A possible parallel implementation of Estrin's Scheme follows. We highlight that the powers of x are always powers of two. (One possible contour is to fill the polynomial with zeroed coefficients until it reaches the next power of two.)

```
1       concepts <Iterator I, Semiring T>
2       estrin (I first, I last, T x) -> T
3       {
4           Dist<I> n = distance (first, last) ;
5           if (n == Dist<I> (1)) return T (*first) ;
6           halve (n) ;
7           I middle = successor (first, n) ;
8           T a = spawn estrin (first , middle, x) ;
9           T b = spawn estrin (middle, last  , x) ;
10          raise (x, n) ;
11          sync ;
12          return a * x + b ;
13      }
14
```

We highlight that, as in Horner's, the multiply-add instruction on line 11 can also be replaced by a `fma` call.

Function `raise` multiplies the base to a given exponent using the Russian Peasant Algorithm in logarithmic time — since we always raise to a power of two it is used as efficiently as possible. This algorithm performs $n - 1$ additions. The number of multiplications is taken summing the multiplications performed recursively on lines 8 and 9 plus $\log_2 n/2$ (since n is halved on line 6), given by logarithmic `raise` on line 10, plus one multiplication from line 11. The result is the following recurrence equation, where $R(n)$ is the number of multiplications over an input size n:

$$R(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2R(\frac{n}{2}) + \log_2(\frac{n}{2}) + 1 & \text{if } n > 1 \end{cases} \quad = \quad 2n - \log_2 n - 2.$$

and thus, summing the multiplications and additions we would have

$$\begin{aligned} W(n) = &\quad 2n - \log_2(n) - 2 \quad + \quad n - 1 \\ = &\quad 3n - \log_2(n) - 3 \end{aligned}$$

One possible optimization is to implement a *memoized* version of `raise` that caches the powers it has already calculated for a given base. We introduce type `MemoizedFunction` that is compatible with concept `Function` and the procedure

```
concepts <Function F> memoize (F f) -> MemoizedFunction
```

that given a function f returns another function f' with same inputs and return value of f but that returns values it already calculated in constant time. Now, we can re-write the code to call a memoized version of `raise` before the recursive calls, recording all needed powers for a particular base, eliminating the logarithmic cost:

```
concepts <Iterator I, Semiring T>
estrin (I first, I last, T x) -> T
{
    Dist<I>  n = distance (first, last) ;
    MemoizedFunction memraise = memoize (raise) ;
    halve (n) ;
    memraise (x, n) ;
    return estrin (first, last, x, memraise) ;
}

concepts <Iterator I, Semiring T, Function F>
estrin (I first, I last, T x, F raise) -> T
{
    Dist<I> n = distance (first, last) ;
    if (n == Dist<I> (1)) return T (*first) ;
    halve (n) ;
    I middle = successor (first, n) ;
    T a = spawn estrin (first , middle, x) ;
    T b = spawn estrin (middle, last  , x) ;
    raise (x, n) ;
    sync ;
    return a * x + b ;
}
```

Although the number of additions remains the same, we take out the log term:

$$R(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2R(\frac{n}{2}) + 1 & \text{if } n > 1 \end{cases} \quad = \quad n - 1.$$

and we sum the $\log_2(n/2)$ from the pre-computation of raise, what delivers

$$\log_2(n/2) \quad + \quad n - 1 \quad = \quad n + \log_2 n - 1 \qquad \text{multiplications.}$$

Summing the $n - 1$ additions we have work

$$\begin{aligned} W(n) = &\quad n + \log_2 n - 1 \quad + \quad n - 1 \\ = &\quad 2n + \log_2 n - 3, \end{aligned}$$

close to Horner.

While this solution is work-optimal since the introduced overhead is $V(n) = \log_2 n$, it only works correctly while the number of terms is a power of two. The pre-computation will calculate all powers of two indexes of base x, but for a polynomial of size, say, 28, it will have to calculate the remaining powers from 8 to 14, the closest power of two. For a generic implementation we need to diminish the number of times `raise`, both because of these differences (that have linear cost) and because the pre-computation of powers may be not viable because of memory/cache issues for large polynomials or when the elements are “infinite” precision numbers (remember, our concept for x is a Semiring).

We circumvent the absence of function calls triggered by steals by asserting the id of the worker that called the function in the first place against the id of the worker currently running it. The algorithm stores the current worker's id and then test to verify if the worker is the same.

This overhead could be eliminated by enabling user callbacks at each steal, a feature we argue would be useful. In fact, not only Cilk — that we use on our benchmarks —, but also major parallel runtime systems such as OpenMP and TBB do not offer this kind of resource. The closest match is the reducers on TBB and hyper-reducers in Cilk Plus, which provide callbacks upon successful spawn (TBB) and sync (TBB and Cilk Plus) operations. This implementation works for a set of algorithms, but it is not as general as the steal callback. (Computations may lack sync, for instance.) Our method, however, has the advantage of relying on one thing common to major parallel middlewares, the capacity a worker has to discover its own id. An adaptive implementation is:

```
1      concepts <Iterator I, Semiring T, Function F>
2      poladapt (I first, I last, T x, F raise) -> T
3      {
4          T a = T (0) ;
5          T b = T (0) ;
6          Auto me = id () ;
7          Dist<I> n = distance (first, last) ;
8          while (! (n < grain)) {
9              halve (n) ;
10             I mid = successor (first, n) ;
11             a += spawn poladapt (first, mid, x) ;
12             if (me != id ()) { // stolen ?
13                 b += spawn poladapt (mid, last, x) ;
14                 raise (x, distance (start, mid)) ;
15                 sync ;
16                 return x * b + a ;
17             }
18             first = mid ;
19         }
20         return horner (first, last, x) ;
21     }
22
```

We have a new artifact and technique on this code. On line 6 we use a type named `Auto`. This is a unique construct that will declare any initialized variable with the same type resulting from the evaluation of the right side of an assignment. Since each middleware may implement the notion of worker id differently, we use `Auto` to declare a variable of whatever type the runtime elects to represent the id. Our only requirement is that a worker id type is comparable for equality. Here, we used `Auto` to deduce the return type of adaptor function `id`, which returns the current worker's id. We compare the values of `me` and `id` on line 12, after the parallel recursive call, in order to discover if the continuation is being executed locally by the spawner worker or remotely by some thief (*i.e.*, a successful steal happened). The programmer has to, thus, write an inlined version of `id` that returns a call to the proper function on the middleware (*e.g.*, `__cilkrts_get_worker_id` in Cilk).

On line 14 we call raise from an iterator unseen until now, `start`. It is a constant copy of initial position pointed by `first` on the first call to the function. It is read-only and visible to all scopes by all workers but only in the context of this algorithm — *i.e.*, it is not a global variable. Iterator `start` is used to determine the exponent used in raise, since we have to compensate for partition.

We highlight that the main expression on line 16 is still in the `fma` format, being susceptible to the optimizations discussed before.

Function `raise` can now pre-compute the powers or calculate them on the fly. Since we are using the simplified model, it is optimal when using a polynomial whose size is a power of two, since we use the Russian Peasant Algorithm. We next proceed to show that we expect to call raise few times even in the worst case.

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 thief 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 the 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). Joining is performed over two chunks $\mathit{left} = [a_{i+d}, \ldots, a_i]$ and $\mathit{right} = [a_{i-1}, \ldots, a_{i-m}]$ that have been already processed and had generated sub results R_{left} and R_{right}. It is a simple attribution $R \leftarrow R_{\mathit{left}} \cdot x^m + R_{\mathit{right}}$. This multiplication by x^m on join operations (that 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 pays 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 estimate through SIPS.

The number of additions remains the same: $n-1$. Now let us determine the number of multiplications accounting for `raise` considering a work-stealing scheduler whose victim selection is random. (The easier case of minimal clock victim selection is discussed more ahead.) The first step in SIPS is to determine this to determine the value of M. Since we are using the simplified model of Section 5.3, taking at most half of the deque at each steal, we have $M = \log_2 n$. Let us consider the following approximation to the Harmonic number

$$H_n \approx \log_e n + \frac{\pi}{2e} + \frac{1}{2n} - \frac{1}{12n^2}.$$

Applying Theorem 3 directly gives us

$$\begin{array}{rclclcl}
\mathbb{E}[\text{calls to } \texttt{raise}] & < & M & \cdot & (P-1) & \cdot & H_{P-1} \\
& < & \log_2 n & \cdot & (P-1) & \cdot & \left(\log_e(P-1) + \frac{\pi}{2e} + \frac{1}{2(P-1)} - \frac{1}{12(P-1)^2}\right) \\
& < & \log_2 n & \cdot & (P-1) & \cdot & \left(\frac{\log_2 P-1}{\log_2 e} + \frac{6\pi P^2+6eP-e}{12e(P-1)^2}\right) \\
& < & \log_2 n & \cdot & (P-1) & \cdot & \left(\frac{\log_2 P-1}{\log_2 e} + \frac{6P-7}{12(P-1)^2} + \frac{\pi}{2e}\right).
\end{array}$$

However, the number of calls to `raise` does not deliver the total number of multiplications, since each call makes a number of multiplications corresponding to the skipped range. Thus, we have to account for the number of calls to `raise (x, m)` for each possible value of m, what we will refer as the "size" of a successful steal. We employ a corollary from Theorem 3 to bound the overhead introduced by each successful steal of a given range size:

Corollary 1 *Let u_m be a random variable whose value is the number of occurrences of the steals of size m in Σ on a work-stealing scheduler with a random selection of the victim. Then, $\mathbb{E}[u_m] \leq (P-1)H_P$.*

Proof Once a worker is a victim of a steal of size m, it cannot be the victim on a steal of the same size again until processor becomes idle (size is strictly decreasing). Thus, for any size m, the maximum M is 1. The remaining follows directly from Theorem 3. ■

Remark 2 The obtained limit can be tightened to $\mathbb{E}[u_m] \leq \min((P-1) \cdot H_{(P-1)}, n/m)$. The n/m min term is introduced to upper-bound large values of m. Large values of m are bounded because there are fewer than P chunks of this size. This is not strictly necessary but increases the tightness of the limit. □

By our simplified model for adaptive algorithms, the range can be only partitioned at its fixed half. Thus, the expected worst-case number of multiplications is the sum of all successful steals of all sizes, each steal "costing" $\log_2$ multiplications that `raise` performs for a successful steal of size 2^i (considering powers of two only for simplicity):

$$\sum_{i=0}^{\log_2 n-1} \log_2(2^i) = \log_2 \frac{n}{2} \cdot \log_2 \sqrt{n}$$

and, thus, we denote the total number of multiplications performed by `raise` as r:

$$\mathbb{E}[r] < \log_2 \frac{n}{2} \cdot \log_2 \sqrt{n} \cdot (P-1) \cdot \left(\frac{\log_2 P}{\log_2 e} + \frac{6P-7}{12(P-1)^2} + \frac{\pi}{2e} \right). \tag{5.1}$$

This algorithm is work-efficient and work-optimal even when memoization is not available — since, for a constant P, Equation 5.1 is dominated by the $\log_2 n/2$ term and the sequential work is $\mathrm{O}(n)$.

For the case when the work-stealing scheduler uses the minimum clock strategy for minimum selection, whose evaluation is simpler. We just ignore the H_{P-1} term, since the multiplier is 1, because the minimum clock increases at each steal, as discussed on Chapter4:

Corollary 2 *Let u_m be a random variable whose value is the number of occurrences of the steals of size m in Σ on a work-stealing scheduler with minimum clock selection of the victim. Then, $\mathbb{E}[u_m] \leq (P-1)$.*

Proof Once a steal of size m is suffered, it cannot be suffered again until processor becomes idle (size is strictly decreasing). Thus, for any size m, the maximum M is 1. The remaining follows directly from Theorem 2. ■

5.5 Simulations

To assert the limits we obtained, a discrete event simulator was written in the Ruby programming language. The simulator is sequential and simulates a P workers machine. The programmer writes its program in terms of a task data structure and the system runs the initial task and performs the work-stealing scheduling with the desired victim selection strategy, specified in the call to the simulator (currently, it supports random and minimal clock selection). It divides the execution in "rounds". At each round, all tasks are executed, in random order. After all executions, the idle workers perform the nano-loop in the work-stealing algorithm also in random order. (If the strategy is minimum clock, and two workers have the same clock value, them one of them is chosen randomly to be executed.)

We run an instance of the `poladapt` algorithm over 32 workers and an input of size/degree of 5.000 elements. The simulator re-runs the simulation 50 times or until the standard deviation is lesser or equal to 0.25 steals. In our experiments, the ranged from 0.01 with 2 workers and 0.25 with 19 workers.

Let us evaluate the tightness of our limits derived from Corollary 1. Recall, for a randomized work-stealing scheduler, that each successful steal of size m obeys, by Remark 2:

$$\mathbb{E}[u_m] < \min\left((P-1)\cdot\left(\frac{\log_2 P}{\log_2 e}+\frac{6P-7}{12(P-1)^2}+\frac{\pi}{2e}\right),\frac{n}{m}\right). \tag{5.2}$$

thus, since the first term is the dominant on the above equation, we have

$$\mathbb{E}[u_m] = \mathrm{O}(P\log_e P).$$

We plotted each steal size as a separate entity. No matter the value of m, Equation 5.2 shall upper-bound the quantity of successful steals of size m in expectation. Figure 5.3 shows the result. Our limits are tight up until 8 workers and remain close until 32 workers — what is the expected behavior for a worst-case expectation. Moreover, we are within a constant value of 1.2 within the asymptotic bound of $P\log_e P$, which does not bound the computation until around 8 workers but is even sharper than the exact predicted value above that. For some size values ($m > 78$) the number of steals "saturates" and flattens in the graph. This is predicted in Remark 2 and notated in Inequation 5.2 through the use of minimum. For steals of larger size, fewer tasks are produced, due to its tree-shaped unfolding.

Finally, for the minimal clock policy the result is no different and is displayed in Figure 5.4. There, the bound is precise and much fewer steals occur since all steals are successful from beginning to end. The execution is deterministic, *mutatis mutandis*, since the random choice of workers with the same clock changes from execution to execution only namely, not in shape.

Adaptive algorithms and SIPS results in mutual benefits especially in the processor-oblivious case, adaptive algorithms are efficient and, at the same time, sufficiently abstract. As this chapter illustrates, the parallel overhead is usually introduced when a successful steal takes place in work-stealing schedulers. This is the direct example where SIPS advances the state of the art.

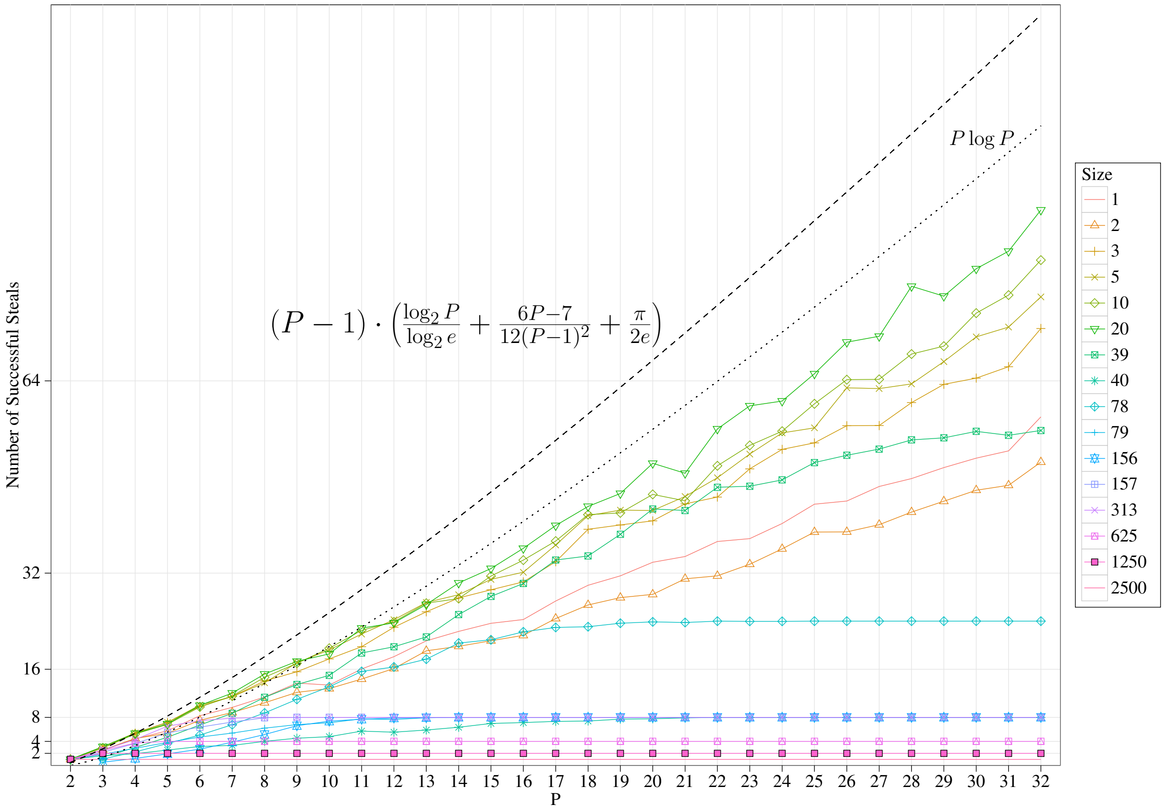


Figure 5.3: Number of successful steals of all sizes for the randomized work-stealing scheduler. The x axis lists the number of workers while the y axis shows the number of steals, each colored line being a different value of m, the size of the steal. The dashed line is the exact bound previously calculated while the dotted line is it in the asymptotic form without the multiplying constant.

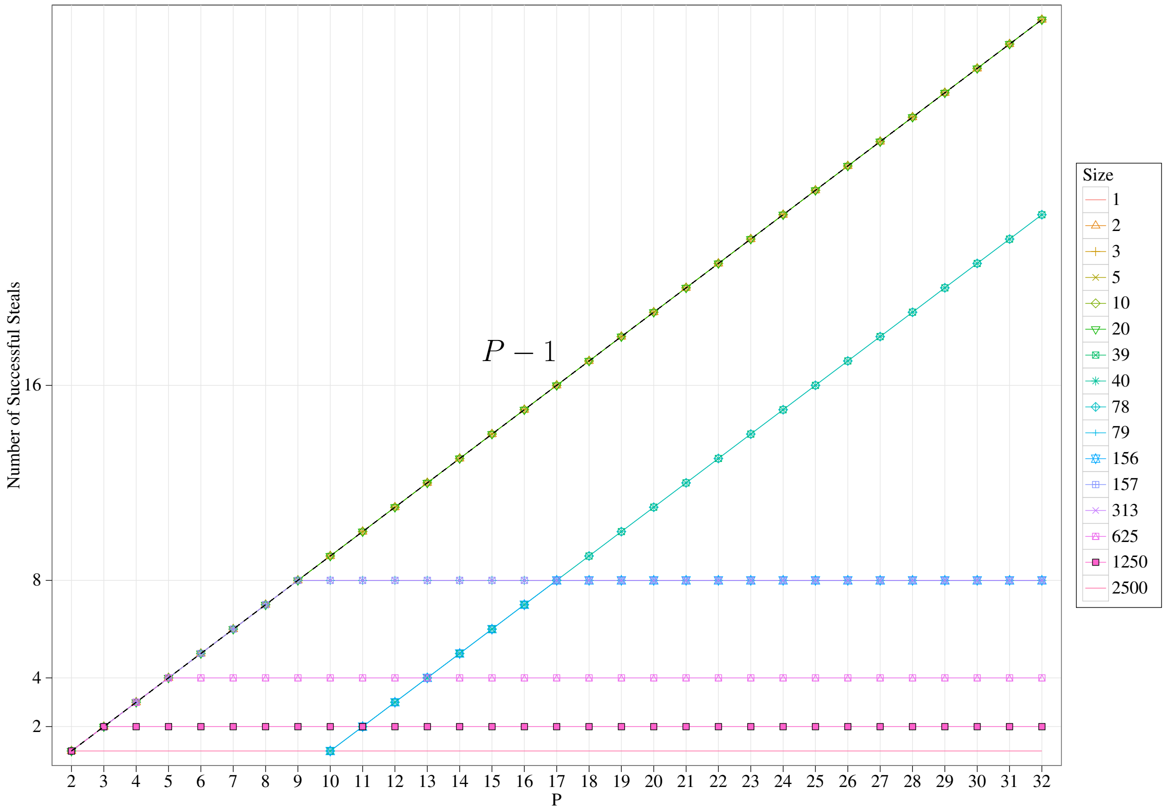


Figure 5.4: Number of successful steals of all sizes for minimum clock work-stealing scheduler. The x axis lists the number of workers while the y axis shows the number of steals, each colored line being a different value of m, the size of the steal. The dashed line is the exact bound previously calculated.

5.6 Closing Remarks

This chapter opens with a definition for adaptive algorithms. The main paper is the one referenced at the beginning by Cung *et al.* (CUNG et al., 2006). It is a paper about the taxonomy of algorithms, but it brings a set of examples — mainly combinatorial optimization and a triangular system solving of linear algebra. The generic scheme it displays is helpful to understand the concepts in a language-independent fashion. We opted, however, to take the inverse path and show it in our quasi-C++ code, more aligned with the proposal of this thesis. Throughout the thesis, we center ourselves in processor-oblivious algorithms and this chapter brings the underlying steps we reproduce in further chapters to build efficient algorithms.

Processor-oblivious algorithms are, in the end, a generalization of the idea of cache-oblivious algorithms by Matteo Frigo *et al.*(FRIGO et al., 1999). Like the considerations we traced at the end of the chapter about the asymptotic limits of `pladapt` and with highlights on the second part of the thesis, the authors consider the asymptotic behavior the algorithms in place of the total number of operations. It centers itself around matrix multiplication and sorting algorithms, providing bounds for cache-misses, showing that even if the algorithms do not have prior knowledge of the cache size, it reaches sub-linear cache-misses (logarithmic in some cases). The processor-oblivious case is addressed by a paper by Bernard, Roch, and Traoré (BERNARD; ROCH; TRAORÉ, 2008), for parallel streams.

In this chapter, we approached the basics of adaptive algorithms, but only briefly. There are theoretical limits, synchronization strategies and other artifacts that are outside the scope of this thesis, but that surpass the discussion about `extract_seq` and `extract_par`. A family of works by Traoré *et al.* discusses these in depth. We highlight its PhD thesis (TRAORÉ, 2008) and the paper about adaptive C++'s Standard Template Library (STL) algorithms (TRAORÉ et al., 2008).

The techniques we have seen here are fundamental to the second part of this thesis, which follows. Together with the SIPS analysis it is used to advance the state of the art of parallel random number generators.

Part II

The Product of Practice: Applications to Parallel Pseudorandom Number Generation

6 A PARALLEL API FOR SEQUENTIAL PSEUDORANDOM NUMBER GENERATORS – PAR-R

In this chapter we define a sequential API for pseudorandom number generators (PRNGs) in order to cope with the algorithms we describe at the next chapters. We model the sequential PRNG concept as a regular type `R` and the API that follows, over which the parallel algorithm runs, `Par-R`.

A generic interface for PRNGs is therefore defined to set a common notation and complexity requirements for our parallel algorithms. It is designed with state-of-the-art generators in mind, in order to provide inexpensive (performance-wise) programming interfaces to usual and standard primitives of current libraries for pseudorandom number generation. Any generator conforming to this API profits from the parallelization techniques we design and analyse later. `Par-R` was designed to make it easy — many times trivial — to adapt a given generator's interface to this minimal set of standard operations it has to support. In what follows, we provide both theoretical complexity requirements on the primitives and implementation constraints that must be followed when implementing it.

We begin by examining preliminary definitions of PRNGs implemented as regular data structures that behave as autonomous objects in memory and the additional concepts they require to function this way (Section 6.1). Next, a mandatory set of primary — *i.e.*, explicit — operations are described: generating the next number on the sequence ("**`next`**"); generating n numbers successively ("`generate`"); and generating directly the $i+n$-th number on the sequence starting at i ("**`jump`**") (Section 6.1). We proceed to discuss a mandatory set of secondary operations that are implicit in programming, but that must also obey the constraints `Par-R` imposes: construction/seeding/reseeding and copy/assignment (Section 6.3). The closing remarks in the chapter consider further requirements that are more abstract yet useful to an actual implementation of `Par-R` (Section 6.4).

6.1 Preliminary Definitions

A PRNG acts as a deterministic stream that provides new random numbers based on its current *state*. A *seed* value gives the initial state. Random streams have a finite orbit (STEPANOV; MCJONES, 2009, *p.* 15), called its *period*, what corresponds to a sequence of numbers that will eventually be repeated over successive generations. It is also assumed that the PRNGs produce generic unsigned integers, for compatibility. There are libraries centered around $[0, 1)$ floating-point types, usually referred to as "Random Bit Generators", a synonym of PRNGs (BARKER; KELSEY, 2012, *p.* 17). The two approaches can be converted interchangeably.

Two PRNG classes are distinguished to generate the stream $\langle r_n \rangle$ from a function r with finite output set and good statistical properties,(SALMON et al., 2011). *Conventional* PRNGs iterate $r_n = r(r_{n-1})$ (*e.g.*, Mersenne Twister (MATSUMOTO; NISHIMURA, 1998), Linear Congruential (KNUTH, 1997b), Tausworth (L'ECUYER, 1996), BBS (BLUM; BLUM; SHUB, 1986)); while *counter-based* PRNGs independently compute $r_n = r(n)$ (*e.g.*, Philox,(SALMON et al., 2011), DotMix). Thus, counter-based are parallel, but conventional PRNGs appear serial: implementations benefit from the previous value r_{n-1} to generate efficiently r_n with less overhead than counter-based ones. Besides, some conventional PRNGs provide efficient jump-ahead over multiple output values in less time than it takes to invoke repeatedly r.

Seeing PRNGs instances as objects provides control to concurrent accesses and freedom to make it visible only to the pertinent parts of our programs. A generic type based interface for PRNGs is defined to set a common notation and complexity requirements for parallel algorithms. Throughout the remain of this thesis we assume that PRNGs are modelled as *regular types*, as defined by Stepanov and McJones (STEPANOV; MCJONES, 2009, *p.* 6-8). A regular type is any type whose basis includes default construction (even if the result is a partially constructed object), copy construction, assignment in terms of copy construction, and equality comparison. Regularity provides uniformity of behavior and interoperability.

Before proceeding, we discuss the referential type function:

`Ref<T>` Returns a reference type for any type `T`.

What `Ref` allows us to do is to overcome the pass-by-value semantics of our dialect, passing the data by reference and assuring ourselves that any parameter of type `Ref<T>` that is modified inside a given function behaves exactly as a variable of type `T`, but all modifications are reflected outside the function's scope. Through `Ref` we can write an increment action that is non-functional: `inc (Ref<int> x) -> void { ++ x ; }`. By using this approach, we introduce collateral effects, of course, but, under control. Collateral effects are a valuable asset in the writing of parallel algorithms that we will approach in the next chapters.

Aside from general use, type functions may also be specific to a given concept. For a given type `R` that models the concept of PRNG we define:

`Val<R>`	Returns the type of the value generated by `R`.

In C++'s Boost library, for instance, `Val<R>` returns **`long long int`** for their implementation of the Mersenne Twister generator.

6.2 Primary Operations

6.2.1 Next

```
concepts <PseudoRandomNumberGenerator R>
    next (Ref<R> gen) -> Val<R>
```

Input:	A reference to a PRNG named `gen`.
Return:	The next random natural number of type `Val<R>` produced by `gen`.
Side-Effect:	Sets `gen`'s internal state.
Time:	$\Theta(1)$

The parameter is passed as a reference to type `R`, what inhibits pass-by-value semantics and thus turn any linear-time cost implied by parameter copying into a constant-time cost.

This is an example of an *adaptor procedure*, which only maps the call to **`next`** to the analogous call of type `R`. Ideally, this function should be compiled inline, telling the compiler to use directly the call to the native generate method defined for `R`, sparing the function call overhead. The compiler's ability to inline a given procedure is a primary trait explored several times henceforward.

This function is referred in NIST's recommendations for Random Bit Generators (BARKER; KELSEY, 2012) as "Generate_function". For non-cryptographic generators, its constant time is usually small. For instance, Lagged Fibonacci generators (KNUTH, 1985, *p.* 26), just perform a sum and a modulus operation. The Xorshift algorithm by George Marsaglia (MARSAG 2003) can perform less than fifteen bitwise operations. There are algorithms whose cost is a little higher, like Mersenne Twister, which is linear on the number of bits.

6.2.2 Generate

The idea to be discussed in the next chapters is to combine the functions in the sequential API to generate random numbers in parallel. Filling a range with random numbers is the building block of several randomized algorithms. We state this problem explicitly to bring along its requirements:

Input. A reference to a PRNG of type `R` and non-negative memory range of size n.

Output. A filled range of n random numbers generated sequentially by `R`.

The reference should be updated by any parallel algorithm as by sequential generation to provide consistency. The same is valid for the generated sequence. Unfortunately, as examined in the next chapter (Remark 3), there is a handful of parallel PRNGs not meeting this requirement.

Now we can write our reference sequential implementation of generate:

```
1       concepts <Iterator I, PseudoRandomNumberGenerator R>
2       generate (I first, I last, Ref<R> gen) -> void
3       {
4           while (first != last) {
5               *first = Val<I> (next (gen)) ;
6               ++ first ;
7           }
8       }
9
```

This function has the same signature and behavior as the Standard C++ Library function `generate`, the only difference being the use of **`next`** in place of `gen()`. As in the C++ implementation, we always assume valid ranges, *i.e.*, the interval is always properly defined by `[first, last)`. On line 5 we make an explicit cast to type `Val<I>`. If it is the same type — or byte-compatible —, the compiler is able to eliminate the conversion. If type `R` provides direct generation through **`operator()`** — *e.g.*, the PRNGs in Boost Library — then one could use `generate` interchangeably with this implementation.

The design takes inspiration from works like Austern *et al.* (AUSTERN; TOWLE; STEPANOV, 1996), where algorithms are orthogonal concepts to data structures.

Parallel implementations of `generate` are proposed in the next chapters. They do not require thread-safeness for the functions provided by `R`.

6.2.3 Jump

Input: A reference to a PRNG and a natural number n.

Return: —

Side-Effect: Performs a *jump-ahead* operation, advancing the generator's state as if **`next`** was called n times.

Time: $O(n)$, $O(\log_2 n)$, or $O(1)$ — see below.

```
concepts <PseudoRandomNumberGenerator R>
 jump (Ref<R> gen, Dist<R> n) -> Ref<R>
```

We introduce a new type function over PRNG `R` named `Dist<R>`, which returns a type whose representation may hold the size of any finite distance between two elements in its period. It is an integer type, and its value is always positive. If no distance is directly associated to `R`, then the type function returns the largest integer type on the machine (in C++ there is the `size_t` macro that represents this type).

Different constraints on the PRNGs usually allow faster jump-ahead implementations. Thus, the cost of jump is modelled as three variations of a function $\delta : \mathbb{N} \to \mathbb{N}$.

Linear $\delta(n) = \mathrm{O}(n)$. Direct implementation. It requires no extra memory to operate, what may be prohibitive for other versions. Trade-offs between memory and space are considered by Haramoto *et al.* (HARAMOTO; MATSUMOTO; L'ECUYER, 2008).

Log $\delta(n) = \mathrm{O}(\log_2(n))$. Could be implemented, *e.g.*, by exponentiation over current state, like the BBS generator (BLUM; BLUM; SHUB, 1986).

Const $\delta(n) = \mathrm{O}(1)$. Could be implemented, *e.g.*, by extending the Log version through pre-computation of the required powers in its finite period. The finitude of period implies that the necessary precomputed powers are also finite. However, this could lead to large memory consumption for longer periods.

Efficient **`jump`** (Log and Const) is frequently not present in random number generation libraries. The downside is that the correspondent binary operation is sometimes expensive. For Const version, there are techniques that mitigate the memory usage, like the ones presented by Haramoto (HARAMOTO; MATSUMOTO; L'ECUYER, 2008; HARAMOTO et al., 2008). It pre-computes only key factors to allow a jump by fast polynomial multiplication within the constant cost of $\mathrm{O}(k^{\log_3 2})$, where k is the size in bits of the space occupied.

6.3 Secondary Operations

These operations do not appear explicitly on the algorithms that we discuss next. However, their detailing is necessary for the analysis of algorithms and considerations over their determinism.

6.3.1 Constructor/Seed/Reseed

As already discussed, the initial state of a PRNG is determined in function of a seed value.

`Par-R` requires at least two types of constructors, the seed constructor, and the default constructor. The seed constructor receives as a parameter only the seed, sets it and assembles the initial state. To simplify things, the type of the seed is admitted to be the same type of the generated value for a generic generator `R`, expressed by type function `Val<R>`. The default constructor, called without arguments, is just a wrapper for the seed constructor being invoked with a default seed. This behavior is coherent with major PRNG implementations. Thus,

```
concepts <PseudoRandomNumberGenerator R>
         // seed constructor
            R (Val<R> seed)

concepts <PseudoRandomNumberGenerator R>
        // default constructor
    R () { this->R (default_seed) ; }
```

The generator can be re-seeded as well. The algorithms we present do not use standalone re-seeding (*i.e.*, outside constructors), but we introduce the procedure for symmetrical completeness with classic PRNGs:

Input: A reference to a PRNG and optionally an unsigned integer serving as the seed for the generator.

Return: Generator's seed after the call.

Side-Effect: Re-sets the generator to the first state with the new seed.

Time: $\Theta(1)$.

```
concepts <PseudoRandomNumberGenerator R>
    reseed (Ref<R> gen, Val<R> seed)
```

Whenever the second parameter, `seed`, is omitted the generator is reseeded to its default seed.

Reseed is referred by NIST (BARKER; KELSEY, 2012, *p*. 18) as "Instantiate_function" when called in constructor form and "Reesed_function" when used as stand-alone.

Reseeding a generator after its initialization has unpredictable behavior in practice. After modifying the seed, two behaviors are admitted:

1. It re-starts the state as if the generator is just created.
2. It positions the current state at the same distance it was from the original seed to the new seed. It depends on the existence of a counter, internal or provided by the programmer.

In practice, few implementations of PRNGs offer dynamic re-seeding, and — mostly for performance — it behaves as in (1).

6.3.2 Copy/Assignment

Admitting `R` to be a regular type, `Par-R` also requires a copy constructor and an assignment operator (**`operator=`**) to be defined in terms of it. For any two PRNGs `x` and `y` of type `R`, invoking the copy constructor of as in `{ R y (x) ; }` must be equivalent to default construct `y` and then assign `x` to it as in `{ R y ; y = x ; }`. Setting copy construction and assignment as an adaptor method allows one to work with implementations that do not implement straightforward copy from **`operator=`**, such as C implementations. This is the case, *e.g.*, of SIMD-oriented Fast Mersenne Twister (SFMT) implementation (HARAMOTO; MATSUMOTO; L'ECUYER, 2008).

To copy a generator is to copy individually each of its parts, individually and recursively. After the copy, both objects will generate the same sequence of numbers. Equality comparison (**`operator==`**), for consistency, is also defined in terms of equality of parts.

Although not stated until now, we implicitly assumed that each generator object occupies constant memory space during its life cycle, independently from its period. This allows us to consider the copy time to be $\Theta(1)$. When copying a generator we expect a constant time, but linear to the size of the generator in memory. For instance, there are linear congruential generators occupying roughly two integers and matrix-based generators whose state is determined by a set of matrices.

Each component of the generator is exclusively "owned" by it, *i.e.*, there are no shared memory among multiple instances of a generator. Optimization techniques to speed up copies like copy-on-write or shared read-only parts are allowed, but they must neither provoke a bottleneck on parallel access nor make explicit its shared components to the programmer.

6.4 Closing Ramarks

In this chapter, we proposed and detailed a parallel API for sequential generators called `Par-R`. In retrospective, it models a state-based pseudorandom number generator, although the complexity requirements on its operands make it easy to use other types of generators — *e.g.*, counter-based — underneath without impacting performance. They allow the programmer to follow either the substream — mainly through jump-ahead operation — or the multistream — primarily through copy and reseeding operations — according to the algorithm at hand. `Par-R` aims, in the end, to increase flexibility without performance penalties.

Other than the explicit references passed as parameters, no side-effect primitives are employed by the presented algorithms. Generators with side-effects — such as the state-based kind — are difficult to parallelize, although common. Moreover, even if generic implementations allow side-effects, this may inhibit the use of such algorithms as a "black-box" software component. An example of a generator with side-effect is DotMix, which requires consistent use of initializations — through reseeding based on scope bounding in order to be consistent.

Mutual exclusion on each function's inputs might be necessary to ensure consistency in a variety of uses of the interface, although our algorithms do not require it. In Chapter 7, parallel implementations of `generate` are detailed that do not presuppose thread-safeness for the functions provided by `R`. Indeed, to benefit from the determinism and efficiency of the **`next`** operation of `R` the parallel overhead is moved at steal operations, as detailed on the next section.

Adaptor procedures are extensively used not only to conciliate different interfaces but also to adapt to different type requirements for a given program. All the primitives presented in this section are provided as adaptor methods in the C++ implementation to Boost Library and DotMix, assuring regular behavior even in the absence of regular data types. This implementation is based on the *template specialization* feature of C++ (STROUSTRUP, 2000).

7 DESIGN AND ANALYSIS OF AN ADAPTIVE GENERATION ALGORITHM

In this chapter we discuss an adaptive parallelization of the `generate` algorithm discussed on Chapter 6. The algorithm relies on `Par-R` to use any sequential PRNG implementing `R` as the underlying generator.

First, we examine a naïve version of parallel `generate` and point its weak points (Section 7.1). Next we present the primary trait on the second part of the thesis: to explore `R`'s the ability to jump ahead on a generated stream of random numbers at least as fast as a sequential generation. Whenever the adaptive algorithm changes from sequential to parallel implementation, which occurs on successful steals performed by the scheduler, a jump is performed for pairing the disjoint sequences between workers. Thanks to bounds provided by SIPS we are able to build two fast algorithms: one *work-efficient* (Section 7.2) when the provided jump complexity is at least linear and one *work-optimal* (Section 7.3), when the jump complexity is at least logarithmic. Also through SIPS we discuss these concepts still in the design phase.

We close the chapter with final remarks about the applicability of the algorithm we study and a parallel implementation of it to an even more general algorithm named iota that is more flexible and will be explored in future works (Section 7.4).

Our considerations apply to a general family of generators, not only PRNGs. A generator is any functor (function object) respecting `Par-R`, the API defined in Chapter 6. If it provides pseudorandom numbers, a mathematical progression, or a fixed constant is not relevant to the algorithm. Thus, we use for `R` the concept `Generator` instead of `PseudorandomNumberGenerator`.

7.1 The Naïve Version

To start, consider a naïve implementation of parallel generate on Figure 7.1.

This code is very alike the `generate` one we used in the previous chapter, but here we divide the interval in two and do the generation recursively until a threshold is reached. On line 4 we compute the size of the range using function `length`, which receives the interval delimiters as parameters. The type of the returned integer is the return of type function `Dist<I>`, which returns an integer type able to represent the number of elements in exactly the same fashion we used `Dist<R>` to determine the jump distance for PRNG.

```
1          concepts <Iterator I, Generator R>
2          generate_par_naive (I first, I last, R gen) -> void
3          {
4              Dist<I> n = length (first, last) ;
5              if (! (n > threshold)) {
6                  generate (first, last, gen) ;
7                  return ;
8              }
9              halve (n) ;
10             I middle = successor (first, n) ;
11             R g = gen ;
12             jump (g, n) ;
13             spawn generate_par_naive (first, middle, gen) ;
14             spawn generate_par_naive (middle, last, g) ;
15         }
16
```

Figure 7.1: Parallel generate algorithm: naïve version.

From line 5 to 8 we test for the threshold. In fact, read-only variable `threshold` is accessible anytime in our implementations, but may change from algorithm to algorithm. If under the threshold, we call the already discussed (inlined) generate on line 6 and return. On lines 10 and 11 we obtain an iterator that divides the semi-open interval in half by first halving `n` and then obtaining the n-th successor of iterator first (*i.e.*, the result of applying **`operator++`** n times). Finally, from line 11 until the end, we copy and jump the generator to compensate for what will be generated in parallel in another thread and perform the recursive call; half of the interval will be filled by the original generator and the other half by the copy, already advanced.

We now proceed to the analysis of this solution. Let n' be the parallel threshold — the minimal grain size for parallel processing. Also, let $\alpha = \Theta(1)$ be the work performed by **`next`** and $\beta = \Theta(1)$ be the same for the assignment of generators. Thus, regarding generator operations, naïve parallel generate has total work $W(n) = \alpha n'$ when $n < n'$. Otherwise,

$$W(n) = \beta + \delta(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + W(\lfloor n/2 \rfloor) \tag{7.1}$$

In closed form (only for powers of two, which maintain asymptotic behavior by the Akra-Bazzi Method (AKRA; BAZZI, 1998)):

$$W(n) = \alpha n + \beta(n-1) + \sum_{i=0}^{\log_2(n)-n'-1} 2^i \delta\left(n/2^{i+1}\right) \tag{7.2}$$

Subtracting from both sides $W_{\text{seq}}(n) = \alpha n + \beta$, delivers the overhead,

$$V(n) = \beta(n-2) + \sum_{i=0}^{\log_2(n)-n'-1} 2^i \delta\left(n/2^{i+1}\right) \tag{7.3}$$

determined by δ and its asymptotic behavior:

$\delta(n)$	Equation 7.3	Work-Efficient?	Work-Optimal?
O(n)	O($n\log_2 n$)	No	No
O($\log_2 n$)	O(n)	Yes	No
O(1)	O(n)	Yes	No

Both Const and Log versions of δ are work-efficient because of their overhead O(n) when applied in Equation 7.3, while Linear version has overhead O($n\log_2 n$), not being work-efficient.

It is possible to reduce the number of jump-ahead operations when the spawned routines run sequentially. Jumps are only performed to guarantee that determinism is preserved when the recursive calls operate in parallel. Since parallelism only unfolds in the presence of steals, execution can jump exclusively when a continuation is stolen; otherwise the original PRNG is used. This tactic effectively moves the determinism overhead to the computation's depth, in a fashion inspired by the *work-first* principle of Cilk's scheduler (FRIGO; LEISERSON; RANDALL, 1998).

The jumps will only occur in successful steals and will introduce overhead only then. Also, the DAG will probably change at each execution since the associated tasks will appear and disappear accordingly to the random behavior of the scheduler. The analysis of this situation is straightforward by employing SIPS, as we verify next.

7.2 The Work-Efficient Version

We now aim to display a work-efficient version of `generate`. The code is written using tail recursion optimization, replacing the final recursive call by a loop to spare us from extra recursive calls and the overhead associated. Also, to use the same generator in the absence of steals, the generators are passed by reference and copied only when needed. This implies an occasional cancelation of the tail recursion optimization, but only when a successful steal takes place, as shown on Figure 7.2.

```
concepts <Iterator I, Generator R>
generate_weff (I first, I last, Ref<R> gen) -> R
{
    Dist<I> n = length (first, last) ;
    while (n > threshold) {
        halve (n) ;
        I middle = successor (first, n) ;
        R g = gen ;
        Auto me = id () ;
        spawn generate_weff (first, middle, gen) ;
        if (me != id ()) { // successful steal
            jump (g, n) ;
            return generate_weff (middle, last, g) ;
        }
        first = middle ;
    }
    generate (first, last, gen) ;
    return gen ;
}
```

Figure 7.2: Parallel generate algorithm: work-efficient version.

Remark 3 Contrary to previous parallel versions, `generate_weff` on Figure 7.2 does return something, a generator. This is done to ensure determinism. Since we cannot guarantee that the original generator will be updated in favor of one of its copies, the referenced generator may not reflect the generated values. For instance, consider two ranges, [`f1`, `l1`) and [`f2`, `l2`). Also, consider two iterators `m1` anywhere within [`f1`, `l1`) and `m2` anywhere within [`f2`, `l2`). In this case, the following assertion would fail:

```
R g1, g2 ;
generate (f1, m1, g1) ;
generate (m1, l1, g1) ;
generate_weff (f2, m2, g2) ;
generate_weff (m2, l2, g2) ;
assert (g1 == g2) ;
```

Also, the contents of ranges [`f1`, `l1`) and [`f2`, `l2`) would differ for any choice of `m1` and `m2`. Moreover, `{ generate_weff (f2, l2, g2) ; }` would produce a different stream from `{ generate_weff (f2, m2, g2) ; generate_weff (m2, l2, g2) ;}`. □

To assure determinism we write an upper-level generate function:

```
concepts <Iterator I, Generator R>
generate_par (I first, I last, Ref<R> gen) -> void
{
    gen = generate_weff (first, last, gen) ;
}
```

By always returning a copy of the generator more ahead on its period we ensure that the final version will be deterministic at the cost of at most (for certain compilers) one extra copy. Since the destiny of the other returns on the recursive call tree is nowhere, the compiler will eliminate the unnecessary copies, remaining the upper-level assignment. Besides, since we return the same variable at line 18 or a function call at line 13, the compiler is allowed to use *return value optimization* and eliminate the one extra copy, operating directly in the reference. Finally, to ensure performance and eliminate this additional copy the programmer may use directly `generate_weff` on Figure 7.2 if determinism among successive calls is not a concern — we find no examples of "consecutive deterministic" generators on the literature. This is how we compare our solution to the solutions on Chapter 8.

We now proceed to the analysis, using the limits found on Corollary 1 (Chapter 5). Jump work δ's overhead is bounded by summing the costs of different u_m. Since half of the range is put at deque's front at each spawn, there are $\log_2 n$ different steal sizes to appear, minus size n. Therefore, the expected overhead by Corollary 1 is:

$$V(n) = (n-1)\beta + H_P(P-1) \sum_{i=0}^{\log_2(n)-1} \delta(2^i). \tag{7.4}$$

what results in

$\delta(n)$	Equation 7.4	Work-Efficient?	Work-Optimal?
$\mathrm{O}(n)$	$\mathrm{O}(n)$	Yes	No
$\mathrm{O}(\log_2 n)$	$\mathrm{O}(n)$	Yes	No
$\mathrm{O}(1)$	$\mathrm{O}(n)$	Yes	No

For a fixed P the expected overhead is $\mathrm{O}(n)$ when using the Linear version of δ. Thus, in expectation, work-efficiency is always assured.

Remark 4 Counter-intuitively, linear version is work-efficient. Each node has to compute sequentially the part that will be calculated recursively, in parallel. Nevertheless, we expect some gain for a reasonable number of workers. Looking to Equation 7.3, we calculate redundant sequential work to be $\sum_{i=1}^{\log_2 n} 2^{i-1} n/2^i = n \log_4 n$. However, parallelism mitigates this cost. As the number of workers grows, the extra-cost becomes closer to $\mathrm{O}(n)$ for two reasons (recall that the overhead is introduced at successful steals). First, if the number of steals is low, the high denominator terms vanish; the result tends to $n/2$.

Second, if the number of steals increases, the number of terms with same denominator executed in parallel tends to increase as well, eliminating the multiplier; the results tends to $\sum_{i=1}^{\log_2 n} n/2^i = n - 1$. In all cases, expected overhead is $\mathrm{O}(n)$. □

7.3 The Work-Optimal Version

Our technique can be refined to obtain work-optimal parallel generation. The problem is to eliminate the fixed overhead introduced by the multiple assignments. We do it by creating a new generator only when a successful steal takes place. The correct placement of the newly created generator is obtained by the use of two extra variables:

`R base`	A constant copy of the generator in its initial state.
`I start`	A constant copy of inital position pointed by `first`.

Both variables are read-only and visible to all scopes by all workers but only on the scope of this algorithm — *i.e.*, they are not global variables. Whenever a successful steal occurs, we copy `base` and jump all the way the length from `start` to the current partition point `middle`. This eliminates unnecessary generator copies, in exchange for paying the price of longer jumps. Nevertheless, the jumps are mitigated by parallelism and "cheap" when the generator provides at most polylog time jump-ahead.

We employ the same return value optimization technique to ensure determinism as with the work-efficient version — returning a generator instead of nothing. The upper-level function would then be:

```
concepts <Iterator I, Generator R>
generate_par (I first, I last, Ref<R> gen) -> void
{
    start = first ;
    basis = gen ;
    gen   = generate_wopt (first, last, gen) ;
}
```

And the work-optimal algorithm is then shown on Figure 7.3.

Now we can cut off the $\beta(n-2)$ term from the asymptotic overhead. Even the more expensive **`jump`** calls are yet upper-bounded by the most expensive possible jump:

$$H_P(P-1) \sum_{i=0}^{\log_2(n)-1} \delta(n - n/2^{i+1}) \tag{7.5}$$

The cost of a call to copy constructor per successful steal is added, but it is constant. Now work-optimality for Const and Log versions can be guaranteed:

```
concepts <Iterator I, Generator R>
generate_wopt (I first, I last, Ref<R> gen) -> R
{
    Dist<I> n = length (first, last) ;
    while (n > threshold) {
        halve (n) ;
        I middle = successor (first, n) ;
        Auto me = id () ;
        spawn generate_wopt (first, middle, gen) ;
        if (me != id ()) { // successful steal
            R g (basis) ;
            jump (g, length (start, middle)) ;
            return generate_wopt (middle, last, g) ;
        }
        first = middle ;
    }
    generate (first, last, gen) ;
    return gen ;
}
```

Figure 7.3: Parallel generate algorithm: work-optimal version.

$\delta(n)$	Equation 7.5	Work-Efficient?	Work-Optimal?
$O(n)$	$O(n\log_2 n)$	No	No
$O(\log_2 n)$	$O({\log_2}^2 n)$	Yes	Yes
$O(1)$	$O(\log_2 n)$	Yes	Yes

The overhead results in $O({\log_2}^{O(1)} n)$, although work-efficiency for Linear version is lost, since it results in an overhead of $O(n\log_2 n)$.

7.4 Closing Remarks

On this chapter, we have discussed the design and analysis of a parallel adaptive generate algorithm. This algorithm is also generic since it works with iterators and concepts.

Sequential incarnations of this type of list-based algorithm is approached in depth in the works of Alexander Stepanov — designer of C++'s STL —, specially on his books *Elements of Programming* (STEPANOV; MCJONES, 2009) and *From Mathematics to Generic Programming* (STEPANOV; ROSE, 2014). These books apply the deductive Euclidean approach to programming in order to allow the writing of the most generic algorithms possible.

The sequential `generate` hold resemblance to a classical algorithm named iota that fills a range $[f, l)$ with consecutive values obtained from an initial value (concept "`Incrementable`" meaning “has operator ++ defined over”):

```
concepts <Iterator I, Incrementable T>
iota (I f, I l, T value) -> T
{
    while (f != l) {
        *f = value ;
        ++ value   ;
        ++ f       ;
    }
    return value ;
}
```

Although not straightforward, if `value` is a PRNG where the `++` operator is mapped to **`next`** and the `=` operator is mapped to copying of the produced value, then what we have is the generate from Chapter 6.

The mapping above may seem only a curiosity, but it allows us to define a more general interface than `Par-R` through the use of a new construct called *virtual iterators*, which enable us to model `T` precisely and produce a *single* version of the algorithm that is work-efficient or work-optimal only by changing the underlying generator, without modifying the algorithm. This is more thoroughly detailed on Chapter 9, on Section 9.3, about future works. There are already implementations of it from the authors that will be published soon.

In the next chapter, we study benchmarks employing the techniques we have seen in this chapter to verify whether these asymptotic limits are valid for processing inputs of reasonable size.

8 ALGORITHMS & BENCHMARKS

This chapter provides experimental evidence that the asymptotic limits shown previously do not excessively penalize the execution with its hidden constants and if they are competitive against Cilk Plus' parallel PRNG, DotMix (LEISERSON; SCHARDL; SUKHA, 2012). DotMix relies on *pedigrees*, thread-unique numerical labels, a feature its authors persuaded Intel to include in its Cilk Plus implementation. This provides us an excellent opportunity to compare our generic solution with a tailored design and reason about the abstraction penalty of using generic PRNGs.

In DotMix, a given reference to a global generator compresses the pedigree and then hashes the result with a low collision probability. To maintain pedigrees on the runtime overcharges it with less than 1% overhead. DotMix shows statistical quality rivaling (with high variance) the one of Mersenne Twister upon the Dieharder random number test suite, although no results are given for other well-recognized tests such as TestU01 (L'ECUYER; SIMARD, 2007). The overheads introduced by DotMix are within a factor of two for adverse cases and are lesser than 20% of the non-deterministic version in suitable cases. Default values were used for DotMix, whose version is the one that comes along with CilkPub — community's Cilk Plus implementations maintained by Intel — contributed code.

We begin presenting environment, runtime (Section 8.1), and evaluation methodology (Section 8.2). Then we approach four benchmark algorithms: Generate (Section 8.3), Introsort (Section 8.4), Maximal Independent Set — by Luby's Method (Section 8.5), and Fibonacci (Section 8.6), designed to evaluate performance in an increasing level of adversity against our methods. Each one approaches one aspect we want to evaluate:

Generate. The basic building block of our algorithms, as discussed in Chapter 7. Experiments show that even so, our performance rivals DotMix for work-efficient algorithms and surpasses it for work-optimal ones.

Introsort. A mix of Quicksort and Heapsort. It requires that we overestimate the generation of random numbers, generating more than what will be effectively needed. Experiments show that even so, our performance rivals DotMix for work-efficient algorithms and surpasses it for work-optimal ones.

Maximal Independent Set. Calculates the maximal independent set of vertices in a graph. We use it to illustrate the first adverse situation, an excessive overestimated algorithm. (It is artificially programmed this way, our methods do not require this level of overestimation.) Experiments show that our technique and DotMix suffer equally from degeneration in performance.

Fibonacci. Calculus of the n-th term on a multiplicative version of Fibonacci's series. We use it to illustrate the second adverse situation, a parallel algorithm with large parallel depth. Experiments show that our technique and DotMix suffer equally from degeneration in performance.

We close the chapter with brief considerations on the size and complexity of the enlisted benchmarks (Section 8.7).

8.1 Environment and Runtime

The discussion is contextualized over Cilk Plus' dynamic multithreading platform (Intel Corporation, 2013), the most recent incarnation of Cilk (FRIGO; LEISERSON; RANDALL, 1998). It provides a spawn-sync abstraction where user threads are spawned as parallel procedures (keyword **`cilk_spawn`**) and joined in a blocking way (keyword **`cilk_sync`**). This implies a processor-oblivious model of computation.

Cilk Plus assigns continuations (ready tasks) to workers through a randomized work-stealing scheduler (FRIGO; LEISERSON; RANDALL, 1998). It is implemented as a collection of worker threads with a deque with two extremes, a front, and back. Parallel continuations produced by the worker are placed in its deque's front. Idle workers with an empty deque keep randomly selecting victim workers until choosing one with a non-empty deque. In this case, it steals the continuation at the deque's back. Idle workers with a non-empty deque remove and execute continuations from its deque's front. The runtime stops when all workers are idle. The principal invariants are the fact that a stolen task is executed without entering the deque (prevents deadlocks) and the spawned task is immediately executed, while the spawner goes to the deque's front (depth-first execution). This model is considered in the implementations that follow.

Three sequential PRNGs from Boost C++ serve as the underlying engine of the generic scheme: Mersenne Twister 19937 (MT19937) (MATSUMOTO; NISHIMURA, 1998), Linear Congruential (Rand48), both over 64-bit integers, and Tausworth Generator (Taus88) (L'ECUYER, 1996), over 32-bit integers. The only Boost generator that implements a jump operation in log time is Rand48, the others executing in linear time. See Table 8.1. We also implemented a Blum Blum Shub (BBS) (BLUM; BLUM; SHUB, 1986) crypto-secure generator over 512-bit integers with a logarithmic time jump. In all tests, work-optimal algorithms are used with Rand48 and BBS and the work-efficient versions with the others.

Name	Description	Period	Speed
mt19937_64	Mersenne Twister	$2^{19937}-1$	38%
rand48	Like Linux's `lrand48`	$2^{48}-1$	64%
taus88	Tausworth Generator	2^{88}	100%

Table 8.1: Boost PRNGs. Data was collected from Boost 1.55's documentation. Column Speed gives the relative speed when compared with the fastest ones, whose value is 100%.

8.2 Evaluation

The benchmark algorithms run for a number of workers $1 \leq P \leq 32$ as well as a sequential version. To provide statistical confidence, the pointed plots are the means of 50 executions for each P and sequential version, lying within a 95.45% confidence interval. The standard deviation is, at the worst case, under 8% of the mean, a reasonable range for randomized algorithms. T_{seq} (resp. T_P) denotes the sequential time (resp. parallel time on P processors) with PRNG `R` (resp. `Par-R`). Yet T_1 is the time of `Par-R` scheduled on one processor.

The comparison criterion is total execution time. Since the algorithms do not have a standard sequential implementation (because of different implementations of the generator components), speedup and efficiency measurements are not meaningful when compared against each other, since a slow sequential implementation may wrongly boost the results. This way, we take out the unfairness of comparing relative speedups, but use it to show anomalies in sequential executions.

In fact, some DotMix benchmarks running in sequential showed unusual measurements for T_{seq} and T_1 but are as expected for T_2 and above. Thus, for clearness of comparison, these execution times are displayed separately; measurements on T_{seq}, T_1, and T_2 are in Table 8.2 and measurements for T_P with $P > 2$, are in Figures 8.1, 8.2, 8.3, and 8.4. The unusual behavior of DotMix is contextualized within each benchmark. Highlights on the implementations and reviews over the results follow.

8.3 Generate

Generates 10^8 64-bit random numbers in parallel.

PRNG	Generate			Introsort			MIS			Fibonacci		
	T_{seq}	T_1	T_2	T_{seq}	T_1	T_2	T_{seq}	T_1	T_2	T_{seq}	T_1	T_2
Rand48	559	529	268	5649	5730	2994	39	67	43	17	194	97
Taus88	703	660	1033	6132	6412	3661	38	67	45	30	193	146
MT19937	877	901	611	6451	6577	3680	38	66	43	30	327	199
DotMix	4201	1713	863	6227	9798	5217	51	67	42	129	389	195
BBS	25954	25602	13006	6316	6424	3503	149	182	102	701	910	455

Table 8.2: Average time (in milliseconds, rounded up) of parallel algorithms' execution.Shown sequential time T_{seq} and parallel times T_1 and T_2.

8.3.1 Implementation

We follow the implementation of `generate` as discussed in Chapter 7. The sequential version for all PRNGs is a for loop calling method **`next`**. The parallel version of DotMix is a call to its own `fill_buffer` function, implemented with the same tail-recursion optimization of our codes, with parallel a threshold of 2,048 elements — the same as DotMix. Target implementation has the same grain size for comparison fairness.

8.3.2 Theoretical Analysis

The theoretical analysis of the work-efficient and work-optimal versions is already exposed in details at Chapter 7.

8.3.3 Experimental Results

Performance is shown in Figure 8.1.

Boost generators and BBS have a minor difference between T_{seq} and T_1, with BBS being much slower because of its extensive use of integer modulus. DotMixhas a T_1 that is 2.45× faster than its T_{seq}. Since DotMix is projected with a parallel-first principle, `fill_buffer` is optimized regarding pedigree initialization (scope bounding), what is mandatory in order to generate deterministic results, introducing significant sequential overhead, what does not affect `Par-R`. A speedup comparison between T_1 and T_2 shows Rand48 (work-optimal), BBS, and DotMix with ≈ 1.97 of speedup while work-efficient MT19937 has ≈ 1.47 of speedup. Taus88, work-efficient and 32 bits, has speed-down of ≈ 0.63. DotMix scales until $P = 11$ processors, being better than MT19937 for $P > 4$ processors (it scales up to 6 processors). DotMix is never better than Rand48. Taus88 does not profit at all from `Par-R`, due to 32 to 64-bit casting. Even BBS is faster for 26 or

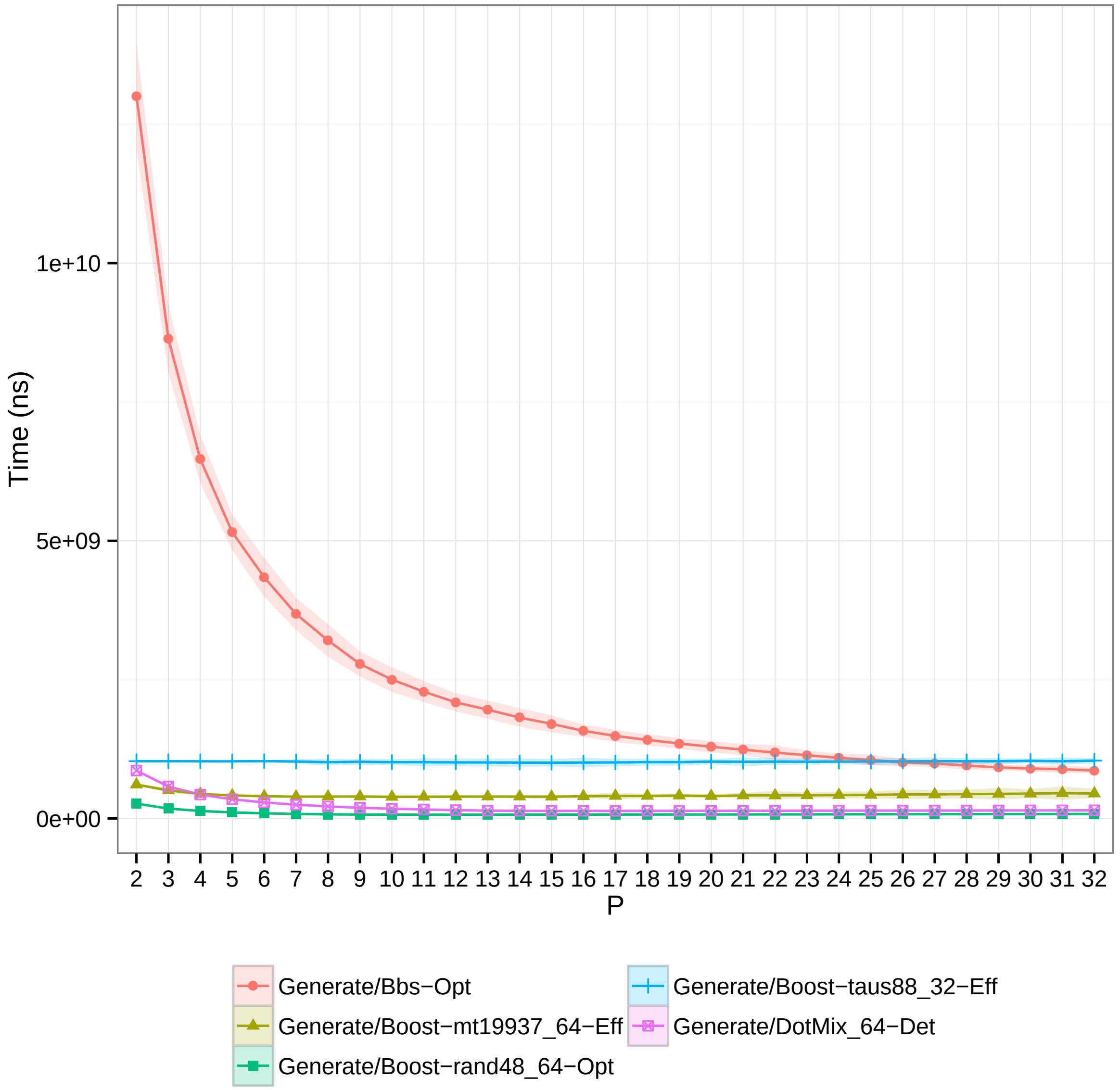


Figure 8.1: Absolute time execution for Generate: P vs. Time (ns) for MT19937 (▲), BBS (●), Rand48 (■), Taus88 (+), and DotMix (☒). The respective colored areas around the points are the confidence interval of 95.45%.

more processors. Overall we are competitive with DotMix for fast underlying generators.

8.4 Introspective Sort

David Musser's Introspective Sort (Introsort) (MUSSER, 1997). It is the algorithm used to code generic sorting in several implementations of C++'s Standard Template Library (like GNU's or SGI's). Introsort is basically a quicksort algorithm that is switched to heapsort whenever its tree depth goes beyond a certain limit. The relevant piece of parallel code to this analysis is the quicksort part, with the random selection of the pivot.

We sort 10^8 integers. The pivots are generated in an "online" fashion, as they are needed — in opposition to producing all at once, prior to the main computation. We use a modified partition procedure always to divide the interval by half, independent of the random pivot, and a sequential threshold of 2,048 elements to provide comparison fairness between PRNGs.

8.4.1 Implementation

The Quicksort part of the algorithm is implemented using the two-way variation, where we divide the range in two based on the pivot value. The Heapsort part of the algorithm is performed on the sequential version that we also call whenever it is under the threshold.

The deterministic version of the algorithm, to be used with DotMix is:

```
    concepts <Iterator I, BinaryPredicate P, PseudoRandomNumberGenerator R>
    introsort_det (I first, I last, P cmp, Dist<I> height, Ref<R> gen) -> void
    // precondition: height >= Dist<I> (0)
    {
        while ((height --) && (length (first, last) > threshold)) {
        Val<I> pivot = *random_pivot (first, last, gen) ;
        I p1 = partition (first, last, bind_2nd (cmp, pivot)) ;
        I p2 = p1 ;
        ++ p2 ;
        Dist<I> left = length (first, p1) ;
        Dist<I> right = length (p2, last) ;
        if (left < right) {
            introsort_det (first, p1, cmp, height, gen) ;
            first = p2 ;
        } else {
            introsort_det (p2, last, cmp, height, gen) ;
            last = p1 ;
        }
    }
    introsort (first, last, cmp) ;
}

```

We introduce a new concept, `BinaryPredicate`, that models a binary function returning a boolean. It is used to define the type of the comparison function the user shall provide to the algorithm. If the user does not provide any, we assume it to be function `less (x, y)` that returns true whenever `x` < `y`, and false otherwise.

On line 6 we use function `random_pivot`, which returns an iterator to the position where the selected pivot lies. Its implementation is:

```
concepts <Iterator I, PseudoRandomNumberGenerator R, BinaryPredicate P>
random_pivot (I first, I last, Ref<R> gen) -> I
{
    Dist<I> n = length (first, last) ;
    Dist<I> r = Dist<I> (next (gen)) ;
    return successor (first, r % n) ;
}

```

On line 7 we use two new functions, `partition`, which re-arranges the elements in range to position all elements satisfying the predicate before all elements not satisfying it — the key function on Quicksort —, and `bind_2nd`. The later receives as a parameter a binary predicate and a value whose type is the same as the second parameter of this function. It returns an unary predicate that behaves exactly as if the previous function was used, but only the first parameter varies. (Receiving a function and returning the same function with fixed parameters is known as currying, in tribute to mathematician Haskell Curry, which developed the technique.) We use it remarkably to give the comparison function and the selected pivot and obtain an unary function capable of comparing the elements in the range directly with the pivot. Since currying is performed in compile-time, this does not imply in performance overhead.

The instruction at line 9 takes out the pivot from the execution because it is already in the right place.

The recursion is implemented again with an unrolled tail recursion optimization, like with `generate`. However, the choice of the pivot is unpredictable, and we want to decrease the number of recursive calls performance-wise. Thus, from line 12 to 19 we measure each sub-chunks size and recurse over the lesser one, producing fewer recursions.

We now can write the work-efficient version of the algorithm:

```
 1     concepts <Iterator I, BinaryPredicate P, PseudoRandomNumberGenerator R>
 2     introsort_weff (I first, I last, P cmp, Dist<I> height, Ref<R> gen) -> R
 3     // precondition: height >= Dist<I> (0)
 4     {
 5         while ((height --) && (length (first, last) > threshold)) {
 6         Val<I> pivot = *random_pivot (first, last, gen) ;
 7         I p1 = partition (first, last, bind_2nd (cmp, pivot)) ;
 8         I p2 = p1 ;
 9         ++ p2 ;
10         Dist<I> left  = length (first, p1) ;
11         Dist<I> right = length (p2, last) ;
12         Auto me = id () ;
13         R g (gen) ;
14         // recurse over smaller side
15         if (left < right) {
16             spawn introsort_weff (first, p1, cmp, height, gen) ;
17             if (me != id ()) { // successful steal
18                 jump (g, left) ;
19                 return introsort_weff (p2, last, cmp, height, g) ;
20             }
21             first = p2 ;
22         } else {
23             spawn introsort_weff (p2, last, cmp, height, gen) ;
24             if (me != id ()) { // successful steal
25                 jump (g, right) ;
26                 return introsort_weff (first, p1, cmp, height, g) ;
27             }
28             last = p1 ;
29         }
30     }
31     introsort (first, last, cmp) ;
32     jump (gen, length (first, last)) ;
33     return gen ;
34 }
```

Here we follow the logic of recursing over the smaller part by spawning only the recursion. A thief, thus, is only able to steal the larger part. With few steals, that represent overhead, the more tasks it takes, the better, so we are settled.

We now explain the extra jump on line 34. This version of the code is parallel deterministic in the sense that the parallel version always choose the same pivots, regardless of the number of workers. Nonetheless, the parallel version — even the one with just one worker — do *not* chooses the same pivots as the deterministic version. This is so because we use overestimation to ensure determinism. For a given sub-chunk of size n we assume the recursive call will choose exactly n pivots. This, however, may not be true depending on how the partition is performed; the threshold may be hit sooner or later on the recursion. Since the problem of knowing the Quicksort tree shape is undecidable, we choose to carry out the jumps following the non-strict upper bound worst case of generating all pivots. This, as we shall see, does not have a large impact on the performance, especially with logarithmic jump generators.

With `generate`, we always recursed on the left sub-chunk, thus giving us implicitly the information of how many numbers were generated. In the work-optimal version, displayed below, we are obliged to abandon the idea of holding the initial value of `first` in static variable `start`, because the point we are on the recursion does not give us precise information about how many numbers were generated by that moment. Since here we choose dynamically to what side recurse, we maintain a counter n of how many pivot choices we made until that moment and pass it forward on the recursion, in order to be a private number to each worker.

```
    concepts <Iterator I, BinaryPredicate P, PseudoRandomNumberGenerator R>
    introsort_wopt (I first, I last, P cmp, Dist<I> height, Dist<R> n,  Ref<R> gen) -> R
    // precondition: height >= Dist<I> (0)
    {
        while ((height --) && (length (first, last) > threshold)) {
        Val<I> pivot = *random_pivot (first, last, gen) ;
        I p1 = partition (first, last, bind_2nd (cmp, pivot)) ;
        I p2 = p1 ;
        ++ p2 ;
        Dist<I> left  = length (first, p1) ;
        Dist<I> right = length (p2, last) ;
        Auto me = id () ;
        if (left < right) {
            spawn introsort_wopt (first, p1, cmp, height, n, gen) ;
            n += left ;
            if (me != id ()) {
                R g (basis) ;
                jump (g, n) ;
                return introsort_wopt (p2, last, cmp, height, n, g) ;
            }
            first = p2 ;
        } else {
            spawn introsort_wopt (p2, last, cmp, height, n, gen) ;
            n += right ;
            if (me != id ()) {
                R g (basis) ;
                jump (g, n) ;
                return introsort_wopt (first, p1, cmp, height, n, g) ;
            }
            last = p1 ;
        }
    }
    introsort (first, last, cmp) ;
    jump (gen, length (first, last)) ;
    return gen ;
}
```

So again we use overestimation and count on jumps having logarithmic complexity to mitigate this cost. Details are given next.

8.4.2 Theoretical Analysis

To determine how many terms are to be jumped, it is supposed that each recursive call will advance the generator as much as the size of the subsequence it takes as input. The analysis of the online algorithm adds some complexity to the analysis of the straightforward parallel generation. While in that case the partitioning was fixed by half of the elements (and thus the number of ranges subject to steal was $\log_2 n$), pivot segmentation may result in degeneration at some points of the execution tree. However, even if we suppose worst-case cost for steals of $\delta(n)$ — regardless of its size — we are able to still assert work-efficiency through the tree's maximum depth, passed as parameter to Introsort.

Instead of applying Corollary 1, Theorem 3 is used directly. In this case, $M = 2\log_2 n$, bounded by Introsort's maximal depth before changing to Heapsort. Introsort runs in time $\mathrm{O}(n\log_2 n)$. The pessimistic approach allows us to implement a work-optimal version and just omit the assignment cost because the different costs of jumps over different chunks are supposed to be the whole distance. This results in an overhead of:

$$\mathbb{E}[V(n)] < H_{P-1}(P-1) \cdot 2\log_2(n) \cdot \delta(n) \tag{8.1}$$

For an work-optimal version, the pessimistic cost for δ is maintained while cutting off the cost for the clone method. For the implementations of δ:

$\delta(n)$	Equation 8.1	Work-Efficient?	Work-Optimal?
$\mathrm{O}(n)$	$\mathrm{O}(n\log_2 n)$	Yes	No
$\mathrm{O}(\log_2 n)$	$\mathrm{O}({\log_2}^2 n)$	Yes	Yes
$\mathrm{O}(1)$	$\mathrm{O}(\log_2 n)$	Yes	Yes

8.4.3 Experiments

Performance is shown in Figure 8.2.

We use a modified partition procedure always to divide the interval by half for comparison fairness between PRNGs. DotMix, because of its use of pedigrees, is not implemented with this overhead. All generators have $T_{\text{seq}} \approx T_1$, except DotMix, which has large overhead $T_1 \approx 1.58T_{\text{seq}}$ without optimized `fill_buffer`. Indeed, until $P = 13$ DotMix has the worst performance, even when comparing to BBS, whose slow performance seems to be mitigated by the work-optimal implementation, placing it at the same level and sometimes better than its work-efficient rivals. For $P > 13$ DotMix is at most statistically equal to the work-efficient implementations. Rand48, being fast and work-optimal, is the incontestable winner. Taus88 has a significant gain since no type casting is necessary.

8.5 Maximal Independent Set: Luby's Method

Implementation of Luby's method to calculate the Maximum Independene Set (MIS) of an acyclic graph. It is divided into three steps, repeated until the input is marked as empty:

1. Select nodes with probability $1/2^i$, where i is the node's degree;

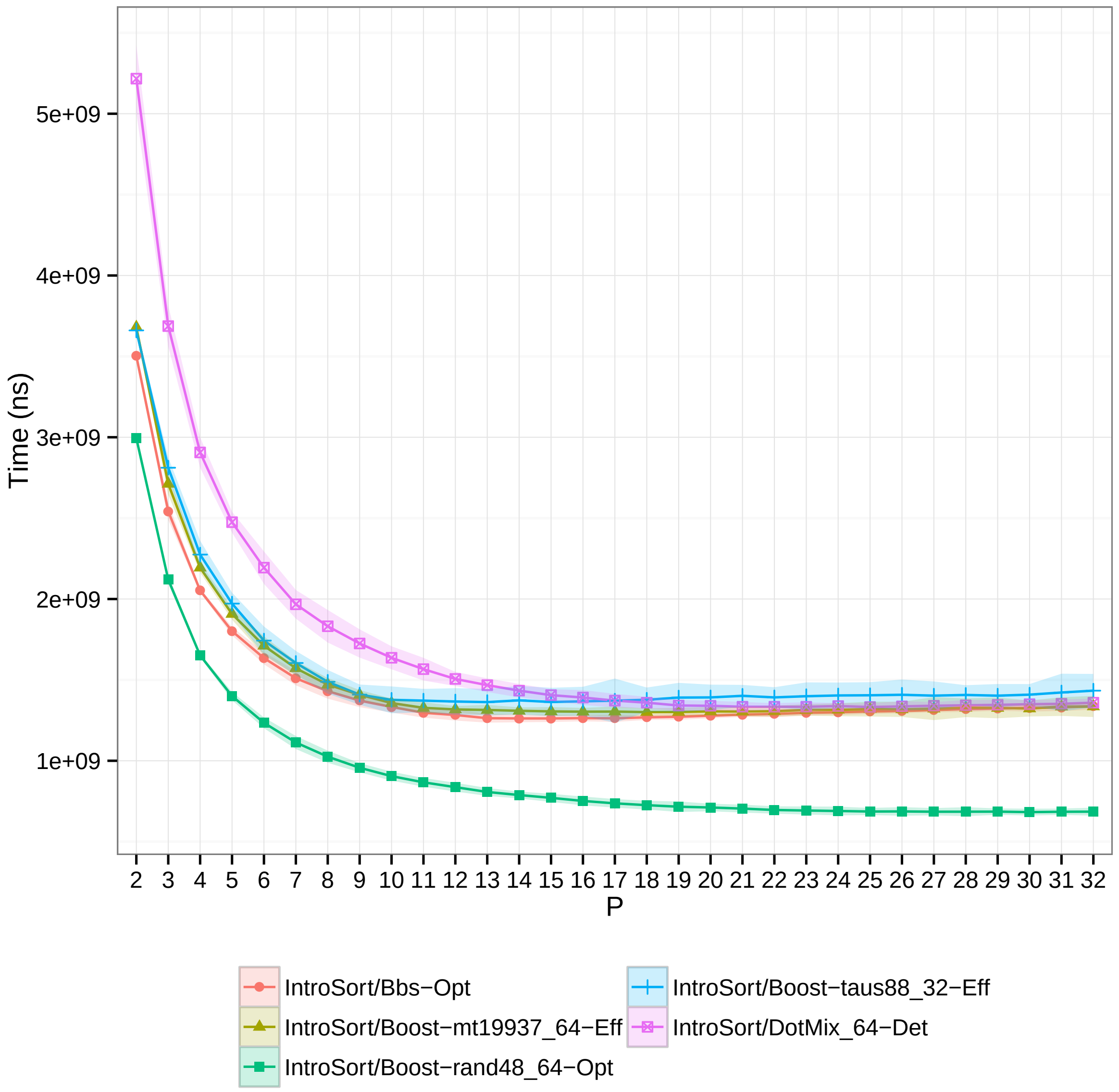


Figure 8.2: Absolute time execution for Introsort: P vs. Time (ns) for MT19937 (▲), BBS (●), Rand48 (■), Taus88 (+), and DotMix (⊠). The respective colored areas around the points are the confidence interval of 95.45%.

2. Deselect lowest degree node of two neighbor selected nodes;
3. Move the remaining selected nodes to the MIS and removes its neighbors from the input.

Steps (1) and (2) are performed in parallel for each node, but the step (2) only executes after (1). We use random numbers for the probabilistic selection in step (1), but the parallel generate function is initialized by a step (0) to generate random numbers in an "off-line" fashion at each round — to the highest level of over-estimation. Luby's method assumes that the PRNG's period is as large as needed to not generate any loops. Robust generators, like Mersenne Twister, provide a period sufficient for most cases. Generators with a small period, however, may result in non-termination.

We intend this algorithm to be adverse to our methods by overestimation, performance-wise. To achieve it, we generate one random number per input vertex even when the number of vertices decreases throughout the algorithm. We show, then, that our methods are equally degraded by this behavior as DotMix.

The input is a grid graph with 10^6 nodes.

8.5.1 Implementation

We use Problem-Based Benchmark Suite (PBBS)'s implementation of a graph data structure (SHUN et al., 2012). In that, vertices are stored in a vector, and we use their positional index as a label. With it we can describe a graph through a pair of indexed iterators (first and last).

Two useful functions for us are defined by each node:

```
concepts <Vertex V, Integer N>
degree (V x) -> N
```

Returns the degree (number of connecting edges) of the node as an integer type.

```
concepts <Vertex V>
neighbors (V x) -> Vector<V>
```

Returns another vector of nodes containing the neighbors of the parameter vertex.

Here we introduce type function `Vector`, which returns a vector type data structure whose elements are of the type passed as a parameter. Vector has an optional constructor that receives as parameter the initial size of the vector. Thus, `Vector<T> v (10) ;` declares a vector of elements of type `T` named `v` whose initial size is 10 elements. A vector provides two functions that return iterators to its elements. For the vector `Vector<T> v` (for any element type `T`) we may invoke:

`begin (v)` Returns an iterator to the initial element in the sequence (`first`).

`end (v)` Returns an iterator to one past the final element in the sequence (`last`).

The MIS is a vector whose values are integers, manipulated through the following enumerated type

```
enum Choice {selected, deselected, removed} ;
```

and thus declared as `Vector<Choice>` and manipulated through its iterators. This is done for performance; throughout the algorithm vertices may be selected and often deselected, only at the end of a step they are allowed to be effectively removed. Thus, in order to avoid the cost of physical removal and at the same time maintain consistency with the relation between a vertex's tag and its position in a vector, we logically mark it as removed once it is not needed anymore.

We can now provide the parallel code, following Figure 1 from Ferreira and Schabannel (FERREIRA; SCHABANEL, 1999):

```
concepts <Iterator II, Iterator IO, PseudoRandomNumberGenerator R>
mis_par (II first, II last, IO mis, Ref<R> gen) -> void
{
    Vector<Val<R>> random (length (first, last)) ;
    while (true) {
        generate_par (begin (random), end (random), gen) ;
        // (I) probabilistic selection
        if (! select_par (first, last, mis, begin (random))) break ;
        // (II) if two selected are neighbors, deselect the one with lowest degree
        deselect_neighbors_par (first, last, mis) ;
        // (III) remove the neighbors of selected vertexes
        remove_neighbors_par (first, last, mis) ;
    }
    return ;
}
```

The basic mainstream on the algorithm is the manipulation of iterators to make every operation in-place and thus improve performance. As with an implicit notation until know all methods ending in the suffix `par` are parallel implementations.

Instead of constructing the random generation built-in — as we have done with Introsort — we generate the parallel numbers we are going to need at each round at once, using `generate_par` (detailed in Chapter 7) as a building block. On each step of the algorithm exactly n random numbers are generated, where n is the initial number of vertices. This occurs even if fewer numbers are needed because we want to stress our method against an algorithm whose speedup is adverse for a larger number of processors.

Since our analysis covers the overhead introduced by parallel generation of random numbers, we detail function `select_par`, which uses the random numbers generated previously:

```
1        concepts <Iterator II, Iterator IO, Iterator IR>
2        select_par (II first, II last, IO mis, IR rand) -> Dist<IO>
3        {
4            Dist<IO> in = 0 ;
5            Dist<IO> out = 0 ;
6            Dist<IO> n = length (first, last) ;
7            parallel_for (Dist<IO> i = Dist<IO> (0) ; i < n ; ++ i, ++ rand) {
8                if (mis[i] == deselected) {
9                    ++ out ;
10                   if (choose_with_probability
11                   ( Dist<IO> (1)
12                   , Dist<IO> (pow2 (degree (*first)))
13                   , Dist<IO> (*rand) ))
14                   {
15                       mis[i] = selected ;
16                       ++ in ;
17                   }
18               }
19           }
20           if (out == Dist<IO> (0)) return Dist<IO> (0) ;
21           return out - in + Dist<IO> (1) ;
22       }
23
```

All this function parameters model the iterator concept, but do not need to be equal types. The iterator type defining the input list is `II`, the iterator type defining the output MIS is `IO`, and the iterator type to the list of random numbers is `IR`. Some may be pointers, some maybe streams, it does not matter once they implement the basic operations we defined earlier for iterators.

On line 7 we use a `parallel_for` construct to selected the nodes in parallel. It splits iterations over active workers and manages the control variables in its header — in this case, `i` and `rand` — by incrementing it accordingly to the current worker. (This is the reason we require indexed iterators.) Since there is no dependency between iterations, the algorithm works fine by choosing probabilistically all nodes that are currently deselected.

Throughout the algorithm — on lines 9 and 16 — we count the number of deselected nodes before and selected nodes after its execution. What remains are removed vertices. In the end, on line 20 and 21, we return `0` if all nodes are already removed — this signals `make_mis_par` to stop — or the number of vertices that remain deselected plus one.

As seen, this implementation of `select_par` may lead active workers to receive non-useful work to perform, splitting the useful work to an increasingly smaller number workers. With this, we enforce a steep ascending curve to the speedup after a certain limit.

Since a number need to be select with probability $1/2^{\sigma}$, a specialization of a more general selection is used. To choose an element of probability x/y, one chooses a random number between 1 and y and verifies if it is lesser than x:

```
concepts <Integer N>
choose_with_probability (N a, N b, N n) -> bool
// precondition : a >= N (0)
// precondition : b >= N (0)
{
    return (n % b) + N (1) <= a ;
}
```

8.5.2 Theoretical Analysis

The analysis of the degenerated algorithm is simple. Since we use parallel `generate_par` as a building block, we may consider k calls to it, where k is determined by the selected numbers at each round. Because this is a degenerated version, we always generate as many random numbers as there are vertices. Standard Luby's Method runs in expected time $\mathrm{O}(\log_2 n)$ (since the expected number of vertices to be selected in step I is 1/2 of the previous iteration (LUBY, 1986)), our degenerated version runs in time $\mathrm{O}(n)$, because of step I. Therefore, the same considerations about work-optimality and work-efficiency of `generate` traced on Chapter 7 hold.

8.5.3 Experiments

Performance is shown in Figure 8.3.

To provide comparison fairness, the same numbers are selected despite a given generators output. The implementation was written to have irregular scalability: at each step a worker may have assigned a node already marked as deselected, performing no useful work. For small P this behavior eliminates node removal operations, but the parallel performance degrades fast for larger values. For the fixed values we generate and for the selected input graph, performance loss begins between 6 and 8 workers. When considering both this irregular scalability and the maximum level of over-estimation the non-secure PRNGs have the same statistical performance — with larger confidence interval due to the other non-PRNG operations the algorithm performs — while BBS penalizes execution because of its integer modulus operations not being mitigated by online generation.

8.6 Randomized Fibonacci

A randomized recursive calculus of 30th Fibonacci term that uses three random numbers (before, after and between the recursive calls), multiplies and adds them to the

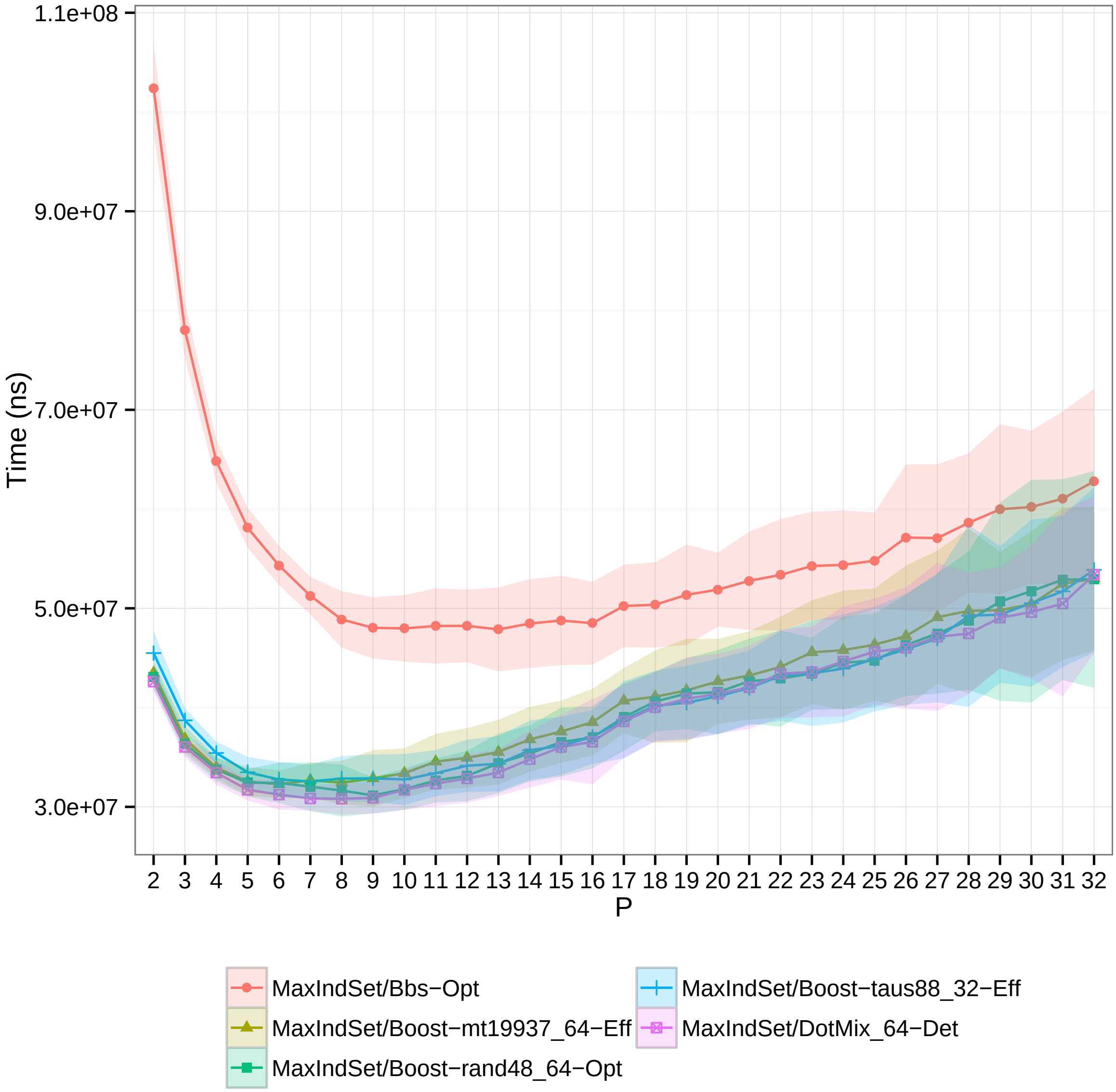


Figure 8.3: Absolute time execution for Maximal Independent Set: P vs. Time (ns) for MT19937 (▲), BBS (●), Rand48 (■), Taus88 (+), and DotMix (☒). The respective colored areas around the points are the confidence interval of 95.45%.

recursive sum. We choose this synthetic benchmark because it is also the algorithm of choice of DotMix in Leiserson *et al.*'s paper (LEISERSON; SCHARDL; SUKHA, 2012) to show a weak point of it, namely when the recursion is too deep. It ends being an weak point for us as well, since the distance of jump is not calculated in constant time, because although we can calculate how much previous calls will advance the main generator, this calculus involves computing how many nodes the tree will spawn. This is as much computational work compared to the computation being performed. We use the fast doubling Fibonacci algorithm to mitigate this cost. This decrease of arithmetical work prevents the randomized algorithm to be work-optimal.

Fibonacci is almost immune to cache issues due to its almost negligent memory consumption. As in Introsort, the random numbers are also generated "online", as needed.

8.6.1 Implementation

We first present a sequential implementation of it:

```
1       concepts <Integer N, PseudoRandomNumberGenerator R>
2       fib_muladd (N n, Ref<R> gen) -> N
3       {
4           if (n < N (2)) return n ;
5           N p = N (next (gen)) * n ;
6           N a = fib_muladd (n - N (1), gen) ;
7           N q = N (next (gen)) * n ;
8           N b = fib_muladd (n - N (2), gen) ;
9           return a + b + p + q ;
10      }
11
```

This is straightforward from the various instances of Fibonacci codes we presented in the thesis until this point. The new multiplicative steps are added on lines 5 and 7.

We now try to implement the work-efficient version:

```
1        concepts <Integer N, PseudoRandomNumberGenerator R>
2        fib_muladd_weff (N n, Ref<R> gen) -> N
3        // precondition: n >= N (0)
4        {
5            if (n < N (2)) return n ;
6            N x, y, sum ;
7            sum = N (next (gen)) * n ;
8            R g (gen) ;
9            Auto me = id () ;
10           x = spawn fib_muladd_weff (n - N (1), gen) ;
11           if (me != id ()) {
12               // each non-leaf node spawned generates two random numbers
13               jump (g, twice (nonleaf (n - N (1)))) ;
14               sum = N (next (g)) * n ;
15               y = spawn fib_muladd_weff (n - N (2), g) ;
16               sync ;
17               gen = g ;
18           } else {
19               sum = N (next (gen)) * n ;
20               y = spawn fib_muladd_weff (n - N (2), gen) ;
21               sync ;
22           }
23           return x + y + sum ;
24       }
25
```

Here, on line 13 we introduce two new functions, `twice` and `nonleaf`. Twice just multiplies an integer by two using fast hardware operations — “shift left”. It requires a binary integer representation of its abstracted type. On the other hand, `nonleaf` is an important function. It calculates the jump distance to update a PRNG according to a sub-branch. This is done by computing the number of non-leaf nodes that this recursive computation will spawn. This is as much computational work compared to the computation being performed.

Function `nonleaf` is short:

```
concepts <Integer N>
nonleaf (N n) -> N
{ return fibonacci_fast (n + N (1)) - N (1) ; }
```

The problem is `fibonacci_fast` is the calculus of Fibonacci series all over again. We try to fake an improvement somehow on our naïve algorithm by using the doubling Fibonacci algorithm, which we show next. (In what follows, `Pair<T, U>` is a data structure containing two variables of type `T` and `U` respectively; function `first` (resp. `second`) returns the first (resp. second) element of the pair, took as parameter.)

```
 1    concepts <Integer N>
 2    fibonacci_fast (N n) -> N
 3    {
 4        if (n <  N (2)) return n ;
 5        return first (fibonacci_pair (n - N (1))) ;
 6    }
 7
 8    concepts <Integer N>
 9    fibonacci_pair (N n) -> Pair<N, N>
10    {
11        if (n == 0) return Pair<N, N> (N (0), N (1)) ;
12        Pair<N, N> ab = fibonacci_pair (half (n)) ;
13        N a = first  (ab) ;
14        N b = second (ab) ;
15        N c = a * (2 * b - a)   ;
16        N d = (a * a) + (b * b) ;
17        if (even (n)) return Pair<N, N> (c, d    ) ;
18                      return Pair<N, N> (d, c + d) ;
19    }
20
```

On line 12, function `half` obtains the integer half of a binary integer using fast "shift right" operations on the hardware. Still, on line 17, function `even` tests if a given number is even through fast bitwise operations rather than the expensive modulus operator.

Let us try to write a work-optimal version of the algorithm:

```
concepts <Integer N, Integer M, PseudoRandomNumberGenerator R>
fib_muladd_wopt (N n, Ref<R> gen, M m) -> N
// precondition: n >= N (0)
// precondition: m >= N (0)
{
    if (n < N (2)) return n ;
    N x, y, sum ;
    sum = N (next (gen)) * n ;
    ++ m ;
    Auto me = id () ;
    x = spawn fib_muladd_wopt (n - N (1), gen, m) ;
    // each non-leaf node spawned generates two random numbers
    m += twice (nonleaf (n - N (1))) ;
    if (me != id ()) {
        R g (basis) ;
        jump (g, m) ;
        sum = N (next (g)) * n ;
        ++ m ;
        y = spawn fib_muladd_wopt (n - N (2), g, m) ;
    } else {
        sum = N (next (gen)) * n ;
        ++ m ;
        y = spawn fib_muladd_wopt (n - N (2), gen, m) ;
    }
    sync ;
    return x + y + sum ;
}
```

Here we use the same counter technique we used in `introsort` in order to keep record of the random numbers already generated, because they are not sequentially generated. The counter is the variable named `m`.

This version, as we will argue next, is unable to achieve work-optimality though.

8.6.2 Theoretical Analysis

The sequential work it performs is given by the recurrence equation

$$R(n) = \begin{cases} 0 & \text{if } n = 0 \\ R(n) = R(n-1) + R(n-2) + \mathrm{O}(1) & \text{if } n > 0 \end{cases} \quad = \quad \mathrm{O}(2^n),$$

Comparing with what is on Section 4.5, we see that this version is equivalent with version `fa` of f_2, and, thus, its local clock upper-bound is $M = \lfloor n/2 + 1 \rfloor$. The fast Fibonacci variation we used has complexity $\mathrm{O}(\log_2 n)$. Thus, each successful steal, on the `fib_muladd_weff` version, has a cost upper-bounded by

$$\log_2 n + \delta(\log_2 n).$$

Applying Theorem 3 and the approximation $H_n \approx \log_e n + \frac{\pi}{2e} + \frac{1}{2n}$ the total number of successful steals is

$$\mathbb{E}[V(n)] < \left(\log_e n + \frac{\pi}{2e} + \frac{1}{2n}\right) \cdot (P-1) \cdot \left\lfloor \frac{n}{2} + 1 \right\rfloor \cdot (\log_2 n + \delta(\log_2 n))$$

that, for any three variants of δ (linear, logarithmic, and constant), is $\mathrm{O}(n \log_2 n)$. The overhead in `fib_muladd_weff` will be then dominated by the clones, one per non-leaf node, which will follow the sequential work we showed to be $\mathrm{O}(2^n)$. Therefore, `fib_muladd_weff` is work-efficient and, as expected, is not work-optimal because of the number of copies it performs.

We now analyse `fib_muladd_wopt`. Each call to `nonleaf` costs $\log_2 n$. However, we have to call `nonleaf` regardless if a steal occurred or not. Thus, we call it $\mathrm{O}(2^n)$ times, and thus its cost is $\mathrm{O}(2^n \log_2 n)$. It is neither work-efficient nor work-optimal, even if the number of successful steals and jump costs is polylogarithmic. Therefore, we are unable to turn it into a work-optimal version.

8.6.3 Experiments

Performance is shown in Figure 8.4.

For this algorithm, DotMix is statistically paired with the work-efficient implementations, although slightly faster for $P > 5$. Taus88 is nearly always better than MT19937,

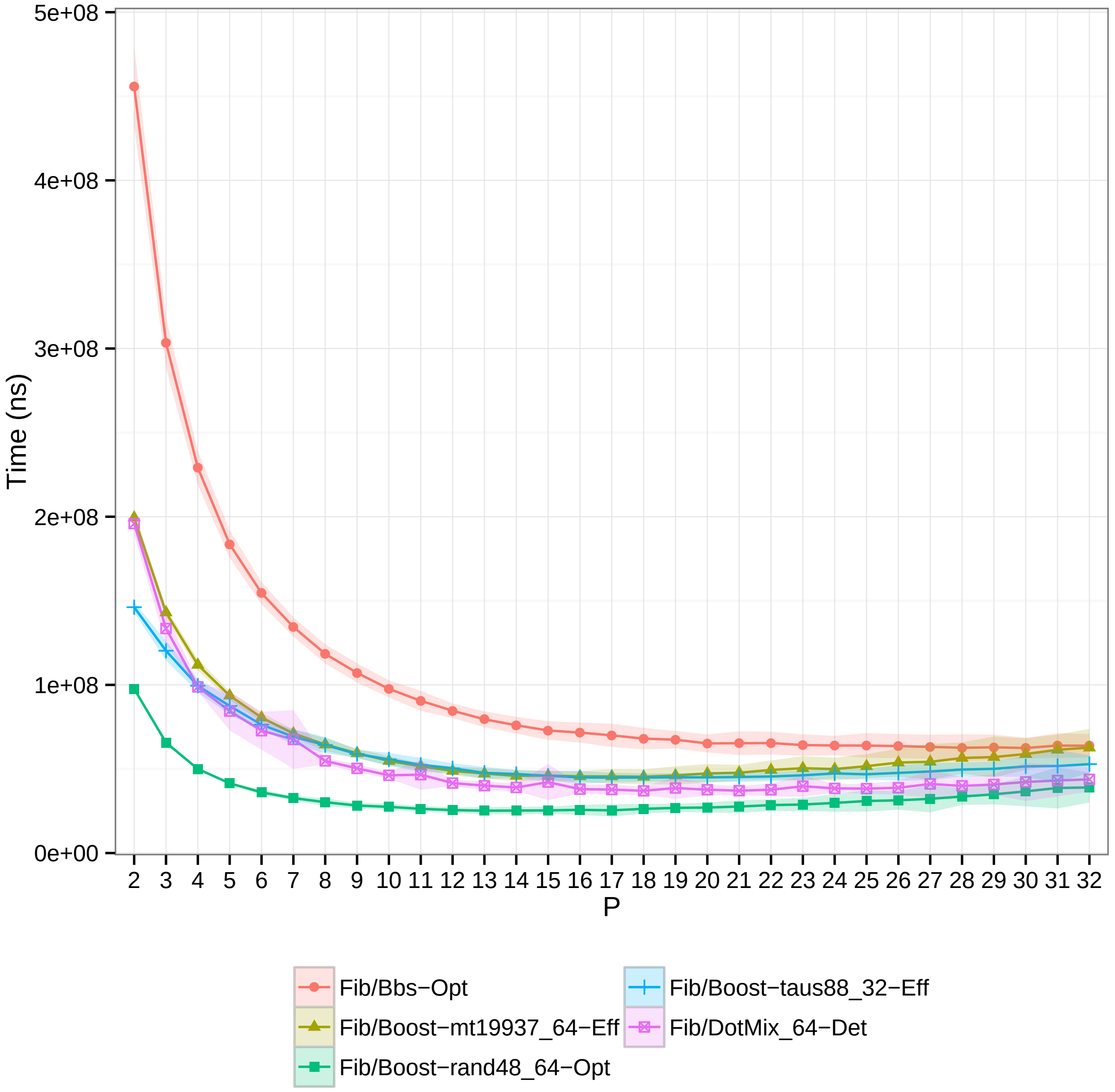


Figure 8.4: Absolute time execution for Fibonacci: P vs. Time (ns) for MT19937 (▲), BBS (●), Rand48 (■), Taus88 (+), and DotMix (☒). The respective colored areas around the points are the confidence interval of 95.45%.

what reinforces its previous improvements for online algorithms. Rand48 is the best until $P = 25$, when it becomes statistically equal to DotMix. For the same range, BBS is the worst, being statistically equal to MT19937 afterwards.

8.7 Closing Remarks

In this chapter, we analysed how our methods behave as underlying blocks to algorithms. We have analysed it in terms of performance (execution time), complexity and implementation. It was shown that our techniques were competitive with state-of-the-art generator DotMix in terms of performance while adding more flexibility to the programmer's options when generating pseudorandom numbers. While we believe to have sound theoretical foundations, the practice shall be confronted in order to measure real gains.

The algorithms we selected are not front-end applications but instead are mostly found in programming libraries. We have chosen to build it this way for two reasons: (1) to follow the state-of-the-art works that evaluate PRNGs, like Leiserson *et al.* (LEISERSON; SCHARDL; SUKHA, 2012), and (2) to show that the footprint of our methods is small enough to fit into tiny primitives, what is somewhat more complex (in performance gain) than to build it into larger applications. The performance of the `Par-R` algorithms was satisfactory in this setting.

9 CONCLUSIONS

Overall we comment over what was presented and summarize it, highlighting its main contributions. We also show the limitations of our work and the frontier on where we expect it to apply. In both cases, we separate the conclusions on each part and trace it separately, since we contemplate both contributions as interlinked, but independent. After, we outline the expected derived contributions to be researched and expanded in future works. Finally, we trace the concluding remarks of the work.

9.1 Summary, Considerations, and Advancements

We restate that both contributions are derived, but they constitute distinct advances. Overall, the `Par-R` scheme is an advance on its own. It uses SIPS underneath, but the techniques applied are a result of advancement in the field of parallel algorithms. We see it not as a mere application of SIPS, but an advance whose one of the main components is the SIPS-based design. There are others, though; *e.g.*, the definition of a standard sequential API that is generic, the construction as regular objects, *etc*.

We highlight that despite the fact that we rely on Cilk Plus to implement our designs, our scheme is not dependent on it. Its coding is simple to be written in another dynamic multithreaded environment, and the theoretical analysis does not rely on a fixed execution's depth. This implies correctness even in the presence of a non-deterministic DAG, such as those on adaptive algorithms. The programming language does not constrain us. Although our dialect is based on C++, it uses standard constructions of modern programming languages. In Chapter 5, for instance, we construct the polynomial evaluation algorithm using the Ruby programming language.

9.1.1 Part I, SIPS

We presented a new technique to analyze the number of synchronizations (in terms of communications) performed during a greedy-scheduled and work-stealing-scheduled parallel computation, SIPS. Using it we obtain an expectation on the worst-case bound of the analyzed algorithms. The classical work on logical clocks by Lamport is the inspiring mechanism for SIPS clocks, whose minimum value ever increases during computation. With SIPS not only the analysis but also the design of parallel algorithms is improved. Using the concepts of work-efficient and work-optimal algorithms, one can reason accurately about the overhead the parallelization introduces and how it affects the performance.

SIPS improves the state of the art by making available an analysis of parallel executions without any reference to the computation's depth. It allows, thus, the analysis of computations with a non-deterministic DAG, what was previously performed in an *ad hoc* fashion at most. Also, SIPS allows an ever-changing number of workers to participate in the computation and is capable of bonding communication on both distributed and shared memory machines. The analysis framework is able to analyze state-of-the-art work-stealing schedulers with a myriad of strategies for victim selection and workload partitioning. On the thesis, we showed detailed analysis for the random choice strategy and minimum clock strategy. We also showed the correspondent bounding values when the workload is partitioned one task at a time and when half of the tasks are taken. Through SIPS, we can reason about the difference in performance from the order of spawning tasks in the source code, what is usually not addressed in the literature.

Our technique pairs well with adaptive algorithms, especially with processor-oblivious ones. It can be used straightforward to obtain a bound on the number of times the primitive `extract_par` is invoked, and, thus, the actual unfolding of parallelism during the computation. In combination with work-efficient and work-optimal concepts, it delivers bounds for the parallel overhead introduced.

9.1.2 Part II, Par-R

We presented a structural design, `Par-R`, that allows us to write generic PRNGs that are still efficient in parallel. `Par-R` has significant performance gains as described in Chapter7 and measured in Chapter 8. We are competitive against DotMix for off-line generation algorithms and usually faster with online generation and fast underlying generators. With our generic scheme, we are able to choose the desired point between quality and speed of several PRNG. Besides, it is possible to be more efficient than DotMix or other parallel PRNGs with fixed implementations by selecting underlying PRNGs whose generated sequence is especially useful for a particular application.

Efficient jump-ahead operations for random number generators are well-known, but implementations of it are still rare. Mascagni *et al.* (MASCAGNI, 1997) presented a review in implementing it with matrix and polynomial methods in 1997, and yet today this fast procedures are missing from most PRNGs we have examined. We already stated that matrix method has substantial associated constant for Mersenne Twister, yet an efficient implementation based on polynomials (HARAMOTO; MATSUMOTO; L'ECUYER, 2008) came only recently. If broadly implemented, this would imply in fast and deterministic random number generators through our methods. The conditions for efficient jump-ahead operations are not impossible to met in most cases. Several linear PRNGs offer a regular structure suitable to it, such as Lagged Fibonacci, MRG32k3a, and Mersenne Twister, although, as mentioned beforehand, large associated constants can provide efficiency just within largely generated ranges. Other types of generators, however, are not suited to this, such as Inversive Congruential generators (EICHENAUER; LEHN, 1986).

One primary family of algorithms that benefit from our techniques are graph algorithms, especially graph generation algorithms. The Stanford's GraphBase (KNUTH, 1993) is full of examples of useful algorithms suitable to `Par-R` — especially random graph generators.

9.2 Limitations

In this section, we discuss the frontiers of the work developed in this thesis. There are, indeed, several open problems and questions. Here, we enlist the limitations. Possible contours are discussed in next section about future work and research.

9.2.1 Part I, SIPS

SIPS provides a tight expectation on the worst-case number of synchronizations. Although this is useful in the analysis and design, it does not model a typical execution. Throughout the work, we used SIPS to validate designs mainly in terms of work-efficiency and optimality. These two definitions are more prone to be used because they are applied over asymptotics, which is derived from the upper-bound. As seen on Section 5.5, the precise number of operations approximates well the asymptotics without being too loose.

Our upper-bound is somewhat conservative, as briefly stated on Remark 1 because we suppose on an idle worker at a time. As enlisted on Subsection 3.1.2, there are more refined methods of handling it to be less pessimistic. We have a few ideas to approach it that are described in next section. This impacts directly on the selected benchmarks designed to show applicability. The use or not of adversaries in the proofs to come to become closer to the upper bounds is still a subject of discussion and evaluation.

Logical clocks are much more general than the focus on work-stealing that the thesis took. Future work already in progress will feature an important generalization to any parallel computation, *including* shared memory computation, which is usually not addressed by this kind of techniques and to which SIPS is applicable.

Our meta-programming solution to act on steals is not free of issues when used outside the generate algorithm. For instance, code grows large when multiple random number generations are performed within the same recursion node. Also, this could introduce constant residual overhead from all the required testing. This is problematic — although not severe —, since it is the standard way to use our techniques upon systems without support for callback invocation on synchronizations. The mere adoption of current parallel middleware of this event-based approach would handle it. In this context, we reinforce that dynamic multithreading environments like Cilk Plus might benefit from the inclusion of steal callbacks, especially within constructs that profit from the lack of sequential overhead. We reinforce that Cilk Plus might benefit from the inclusion of steal callbacks, especially within constructs that profit from the lack of sequential overhead, such as `cilk_for` with reducers.

9.2.2 Part II, Par-R

A hybrid solution of `Par-R` and DotMix is compelling. However, because DotMix does not have an equivalent to the jump-ahead operation, the linear version becomes mandatory. In our tests, this approach was more than 10× slower than SFMT, a 128bit generator. DotMix has internal functions that optimize it further than what is possible with its public interface. However, maybe there is some optimization inside DotMix to allow it. We plan to verify it as future work.

Our approach is generic in the sense that a variety of PRNGs may be used, although it is not suitable for all problems. In `Par-R`, the number of required random numbers must be known *a priori* to the computation. This is a substantial limitation to our method. There is, however, a range of algorithms that are suited to it besides direct parallel generation, such as randomized sorts, randomized graph generation, randomized genetic algorithms (crossing over), *etc.* Additionally, one may overcome this limitation by guessing large non-overlapping ranges between the different workers, thus enabling algorithms to not know exactly how many numbers they will need in runtime, given an upper-bound. Combining over-estimation and polylog jumps mitigate the overheads largely. This is similar to what is done, for instance, by SFMT (HARAMOTO; MATSUMOTO; L'ECUYER, 2008) and RNGStreams (L'ECUYER et al., 2002).

Determinism is a primary concern when implementing parallel PRNGs, as we illustrated on Remark 3. All the algorithms that use `Par-R`-based generators are guaranteed to be deterministic, but only between the parallel versions. The sequential version may generate a different result. This happens because of the over-estimation we use. It does not insert considerable overhead — as discussed in Section 8.4 — but it will probably produce different results from no-over-estimated versions. Other parallel PRNGs (SFMT) do not even offer this kind of guarantee. DotMix only offer a full-deterministic guarantee (sequential and parallel) upon a call to a particular procedure that does insert non-negligent overhead. Moreover, it fails to the generate asserting proposed at Remark 3, since its fast internal algorithm does not preserve the state after invocation.

9.3 Future Works and Research

9.3.1 Part I, SIPS

The subject is not exhausted. Ongoing works by the authors explore the distribution of SIPS clocks. 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 independence, which is stronger and more general than the condition of having different sizes. It varies from algorithm to algorithm.

We have seen two main strategies of victim selection: minimum SIPS clock and randomized. While the computation progresses faster with minimum clock, there is contention. Future research will approach the mixing of these two strategies. Looking once again to Figure 3.1, it is prominent that there are lots of unsuccessful steals in the beginning and end of the computation. Once the workers are full, almost every steal attempt is successful. This suggests that minimum clock is useful "in both ends" while randomized strategy provides optimal performance "in the middle".

About the worst case expectation being "loose", as pointed before, we work on alternative ways to be sharper. Currently, we evaluate an alternative based on the isomorphism of worker configuration along the rounds and a Markov chain evaluated through computer-aided simulation to deliver a tighter bound. Yet, adversary-based proofs are being examined, such as the "Tetris adversary", which tries to maximize the SIPS in all workers, as the classical game.

Derivate designs we currently evaluate include the notion of partially associative operations. During the thesis, we discovered that the monoid structure generally taken as pre-requisite to build parallel reduce algorithms can be replaced by a possibly new algebraic structure we currently develop. A “partially associative” structure is in sight. There two right-associative operations can be combined to produce a fully-associative (monoid-based) computation. The algorithm would process the structure sequentially from left to right using the first operation. Every time another worker steals a part ahead of it to be executed in parallel (see adaptive algorithms on Chapter 5), the second operation is invoked to perform the merge. This “restores the associativity”. The number of times we call the second operation is a straightforward application of SIPS. This approach is promising, especially when analyzing current parallel programming trends such as the Map-Reduce paradigm.

9.3.2 Part II, Par-R

Also, we plan to extend the jump on steals technique to numerical algorithms, such as transform, accumulate, prefix, and iota. This idea brought along the idea of changing the paradigm from a reference to a generator to what we convened to call *virtual iterators*. These iterators point to a virtual sequence, *i.e.*, to a sequence that is not in memory but will be generated at each dereference performed. This gives us several possibilities:

1. It allows us to abstract a larger class of generators, such as counter-based, in addition to the state-based we approached; we see the generator as a virtual infinite sequence of random numbers that are only addressed.
2. To unify work-efficient and work-optimal algorithms under a single interface, using a primitive “detach state” to copy the whole generating mechanism or a reference to it.
3. To apply to different domains, such as numerical algorithms, without modifications to the interface.
4. Allows “to go back” in the sequence, re-generating the numbers of the position or caching it, following the complexity of the sequential underlying generator it generalizes.
5. It allows the programmer to use several libraries of algorithms over iterators that are already implemented and twist it to a new purpose. For instance, one can use the

algorithm accumulate from STL to solve random polynomials if the virtual iterators are used over a random sequence of numbers.

There are already implementations of some algorithms discussed in this thesis in this fashion. These implementations are preliminary, but work as expected and maintain efficiency.

While reviewing counter-based generators, we noticed the use of Feistel networks within to produce a constant-time generation of random numbers in an arbitrary position on the random stream. While it has the same constant cost for all implementations, Feistel networks allow faster constant-time generation of the "next" element in the stream, what was not approached on the original works on the subject. This, combined with the iterator approach above and adaptive algorithms has a vast potential of spawning a new family of algorithms, efficient and scalable.

One variation of our technique would be to maintain a constant number of PRNGs per thread and reuse it (through seed method) at each steal. In our tests this approach resulted in slightly lesser performance gain (probably due to compiler optimizations on read/write variables since it might already reuse the PRNG without reading access concurrency), but it may be useful to transpose the pedigree-hashing approach from DotMix.

9.4 Final Remarks

The overall work is not a *front-end*, but it is rather a structured deductive approach to be used in underlying components of applications and their analysis. It does not apply directly to applications, being useful in the analysis and design of algorithms and as reusable code in source code libraries.

This thesis also represents a joint effort of a long-lasting partnership between research groups MOAIS, in France, and GPPD, in Brazil, through the international laboratory LICIA. The cooperation was fundamental to the development of the work and to surpass its difficulties. We hope this work will make stronger, long-lasting bonds to reinforce this relation.

Finally, once this work advances further, we will be able to review and analyze the algebra of tasks concept once again, as described in Chapter 1.

Appendices

AppendixA EXPANDED BACKGROUND

This appendix expands Chapter 2. The informed reader may skip its parts in conformance to his previous knowledge on the topic. Its contents are aimed at the reader not familiarized with multithreaded parallel programming and scheduling theory.

A.1 Parallel Machine Architectures

We operate over MIMDs, which is a category in Flynn's taxonomy. Classical taxonomy by Michael Flynn (FLYNN, 1972) divides general computer architectures in terms of their parallel operations in affinity with the Von Neumann model. This may be the very reason its model became a widespread taxonomy, even for nowadays. Flynn's classification consists in four classes:

SISD. It means *Single Instruction, Single Data.* This is the uniprocessor machine. Obviously, it is not parallel. These are, roughly, the implementations of the Von Neumann machine.

SIMD. It means *Single Instruction, Multiple Data.* Synchronized processors apply the same instructions to different memory addresses. SIMD computers exploit data-level parallelism. Vector architectures are the largest class of SIMD architectures. SIMD approaches emerged with a new importance for graphics performance, in the implementation of General-Pourpose Graphichs Processing Units (GPGPUs). Its parallel instructions are executed synchronously. The earliest parallel computers, such as the Illiac IV, MPP, and MasPar MP-1 belonged to this class of machines. Variants of this concept are used in co-processing units such as the Intel's MMX and SSE provides this kind of instruction.

MISD. It means *Multiple Instruction, Single Data.* Originally, this category is empty. Currently, there is no commercial processor architecture that follows this model.

MIMD. It means *Multiple Instruction, Multiple Data.* Autonomous processors, each with its instruction flows and data scope. Unlike SIMD systems, MIMD systems are usually asynchronous. The processors can operate at their own pace; there is no global clock, and there may be no relation between the system times on two different processors. Unless the programmer imposes some synchronization, even if the processors are executing exactly the same sequence of instructions, at any given instant they may be executing different statements.

A.2 Parallel Machine Models

We adopted in thesis a general shared-memory model of abstract parallel machine. However, there are more accurate and widespread models. We discuss two of such models here: PRAM and LogP.

PRAM is a shared-memory virtual machine where processors work at a fixed pace, strictly synchronized by a global clock. All processors execute one operation per time unit — at the same time — and memory access time is equal to zero. The programmer has available a set of processors as large as needed. This set is *not* infinite, the processors cannot be created at runtime. The number must be defined prior to the execution and is fixed, although it may change between executions with different inputs and be calculated in function of it.

In PRAM any memory location is accessible to any processor at a given time. Thanks to it, inconsistencies may arise from simultaneous access in memory. There are no built-in synchronization primitives and algorithms must enforce correctness in those cases. One usual way to enforce correctness on memory accesses is to constrain the model regarding read and write operations to the same memory location:

EREW. *Exclusive Read, Exclusive Write.* Every memory cell can be read or written to by only one processor at a time.

CREW. *Concurrent Read, Exclusive Write.* Multiple processors can read a memory cell, but only one can write at a time.

ERCW. *Exclusive Read, Concurrent Write.* Never considered, makes little sense, because reads are usually innocuous to consistency.

CRCW. *Concurrent Read, Concurrent Write.* Multiple processors can read and write.

Whenever a conflict violates one constraint a default criterion is applied (*e.g.*, the processor with the lowest rank performs the operation and the other ones produce a no-operation instruction.)

PRAM is famous for complexity analysis. In the same way that a theoretical random-access machine is used by sequential algorithm designers to model algorithmic performance, the PRAM is used by parallel-algorithm designers to model parallel algorithmic performance. Similar to the way in which the sequential model neglects practical issues, such as access time to cache memory, the PRAM model ignores issues as synchronization and communication. It is useful to provide proofs of correctness or lower bounds (in time or processors × time).

The LogP machine consists of arbitrarily many processing units with distributed memory. While PRAM is usually a constrained version of the shared memory model, LogP fits well with the distributed memory model. The processing units are connected through an abstract communication medium that allows point-to-point communication. This model is pair-wise synchronous and overall asynchronous.

The name LogP is not related to the mathematical logarithmic function. Instead, the virtual machine is described by four parameters:

$$\begin{aligned} L &= \text{The latency of the communication medium.} \\ o &= \text{The overhead of sending and receiving a message.} \\ g &= \text{The gap required between two send/receive operations.} \\ P &= \text{The number of processing units.} \end{aligned}$$

Each local operation on each machine takes the same time — a processor cycle. The units of the parameters L, o, and g are measured in multiples of processor cycles.

A.3 Parallelization

On Chapter 2 we explained in details the task-based parallelization scheme applied to the recursive model.

Sequential programs may be divided and re-written into parallel tasks by distinct (sometimes concurrent) approaches. We cite as non-exhaustive examples (KUMAR, 2002):

Subtask partition. The input is split, each sub-computation is defined as a parallel task, and their outputs are merged. Both split and merge are separate algorithms that may or not also spawn their parallel tasks. The partition is usually defined by the program's coding logic rather than some distribution property on the input. Parallelization of recursive functions falls into this category. Example: sorting algorithms.

Linear decomposition. Data is linearly divided, its segments being a basis plus an offset. Each segment is a parallel task. Parallelization of loops falls into this category. Example: brute-force cryptanalysis.

Domain decomposition. The structural shape of data defines the partition scheme. A parallel task is a coordinate regarding the input data. Parallelization of grid-like data structures falls into this category. Example: mesh refinement algorithms, used largely for weather forecast and dynamic molecular applications in Physics.

Now we detail two other strategies to parallelization besides recursion: bag-of-tasks and communicating.

The *bag-of-tasks* approach decomposes the problem into *independent* parts, *i.e.*, parallel tasks with no sequential dependency. There are several possible entry points: non-nested function calls; loop with independent iterations; delimited code sections; input partitions; *etc.* Bag-of-tasks adds few (if any) constraints to programming. Nonetheless, it is usually applied to solve simple problems. Some examples are brute-force cryptanalysis, map/reduce algorithms, linear search returning *any* occurrence.

The *communicating* approach is similar to bag-of-tasks, except that partial results must be communicated during the computation. This requires synchronization operations that, in their turn, insert sequential dependencies. There are various patterns, such as:

Reduction. Tasks compute in parallel operands, and a final n-ary operation combines them into a single result. Since truly n-ary operations are not implemented in most processors, a binary associative operation is usually used.

Gather. Tasks compute a partition of the final result.

Two parallel algorithms that profit from this approach are Prefix-Sum (JAJA, 1992) (calculates all partial sums of an array) and Odd-Even Mergesort (KNUTH, 1998) (parallelization of Mergesort where different operations occur in alternate turns according to the worker's id being odd or even).

A.4 Middlewares: Libraries and Runtimes

We now survey some modern combined runtime/libraries for high-performance parallelism other than the Cilk family we employed in our examples and benchmarks. The emphasis is on mechanisms to support explicit task parallelism. All are exemplified with implementations to find the n-th Fibonacci term. The discussed middlewares are standard implementations in different levels of the “parallel stack” in Figure 2.1.

A.4.1 PThreads

“POSIX Threads” (MUELLER, 1993), belonging to the POSIX standard, used to refer to systems that implement the UNIX OS interface with an arbitrary degree of fidelity. It is designed for creating and management of OS level threads in the C language. Threads are usually a “lightweight process” encompassing private register content, machine flags, a program counter and a stack pointer, among others. The program code, the heap, and the stack are usually shared between the threads belonging to the same process. One OS process may have several threads affiliated. Implementing workers as threads allows a much faster context change than with processes. The implementation is fast, at the cost of little abstraction that demands management of details. The runtime only translates lower level primitives in C to the OS’s thread Application Binary Interface (ABI).

PThreads offers three synchronization primitives: barriers (join), mutexes and semaphores. The scheduling itself is performed by the OS.

We show an example of the calculus of the n-th Fibonacci term on Figure A.1 (*cf.* the Fibonacci example in Subsection 1.4.2). This code is written in C. In this listing,

1. On line 1 we have the function signature. PThreads demands that a new thread is spawned with a running procedure associated. This procedure, however, is required to receive just one parameter and return one single value of the same type, `void*` (“void pointer”). In C, a pointer is a memory address that holds a variable with type associated and a void pointer is a special type of pointer that points to a variable of any type. Since C does not have facilities for higher-order functions, when passing a procedure as parameter this style of raw pointer manipulation is necessary. The return type must be properly cast by the programmer after the execution. The same is valid for the manipulation of the arguments inside the function. If more than one parameter is needed, the argument reference should point to an array of arguments and the proper cast be made.
2. On line 3, we declare two thread identifiers, `t1` and `t2`. When spawning a new thread, we have to bind it to an identifier that may be employed to make further reference to the spawnies of the parent thread.
3. On line 4 we declare the variables that will serve as input and output on the recursive call.
4. On line 5, since our Fibonacci implementation is over type `int`, we create an `int*`

named `i` and assign the address of `n` to it, telling the compiler that the address of `n` shall be interpreted as an int pointer when using `i`. This is done by type casting the value of `n` before the assignment, **`(int*)`** `n`. From now on we access the parameter and output by verifying the contents of pointer `i`, through operation `*i`.

5. On line 6 we test the recursion limit and return the parameter if nothing has to be done.
6. On lines 7 and 8 we set the parameters to perform the recursive call later.
7. On lines 9 and 10 we spawn child threads as a recursive invocation of `fib`. This is done through the primitive `pthread_create`. We use operator `&` to extract the address (as a pointer type) of a variable. It receives as parameter the address of the thread identifier to be bound (`&t1` and `&t2`), the OS attributes of the thread (we use default ones by passing a `NULL` pointer), a pointer to the spawned function (`fib`) and a pointer to its arguments (`&a` and `&b`). Function `pthread_create` returns an int. If this value is different than zero, then an error occurred, and we return a `NULL` pointer.
8. On lines 11 and 12 we perform a join, where a given thread waits for the completion of another one through primitive `pthread_join`. The first parameter we pass is the identifier of the thread we want to wait (`t1` and `t2`). The second parameter is the address to receive the termination value of the thread, which we are not interested and thus give a `NULL` value to it. Function `pthread_join` returns an int. If this value is different than zero, then an error occurred, and we return a `NULL` pointer.
9. On line 13 we sum the outputs of the recursive calls on the position pointed by `i`.
10. On line 14 we return the input void pointer that now has the final value of the procedure.

The procedure on Figure A.1 shows various implementation details that are better to be hidden from the programmer. Since we are using a low-level middleware, this type of thing is usual. We usually use such structures to implement an underlying layer to higher level libraries or to improve performance. However, we highlight that this code has no control of granularity and is implemented naïvely, performing too many redundant calculations, and spawning one OS thread per recursive call. It is just for syntax demonstration, not meant for performance.

```
void* fib (void* n)
{
    pthread_t t1, t2 ;
    int a, b ;
    int* i = (int*) n ;
    if (*i < 2) return n ;
    a = *i - 1 ;
    b = *i - 2 ;
    if (pthread_create (&t1, NULL, fib, &a)) return NULL ;
    if (pthread_create (&t2, NULL, fib, &b)) return NULL ;
    if (pthread_join (t1, NULL)) return NULL ;
    if (pthread_join (t2, NULL)) return NULL ;
    *i = a + b ;
    return n ;
}

```

Figure A.1: Fibonacci in PThreads (C).

A.4.2 OpenMP

OpenMP (DAGUM; MENON, 1998) is a standard for a collection of higher-level constructions for multithreaded programming in C, C++, and Fortran. It is a higher level abstraction of concepts the uses PThreads underneath. Its workers are mapped to OS level POSIX threads. Its tasks and constructs are user-level threads scheduled on the top of its workers.

Its primary approach is to "parallelize" sequential code by informing the compiler what parts it can parallelize automatically. There is an extensive use of pragma compiler directives, which are ignored if the compiler does not implement the standard, thus implying the execution of an elision code. It allows, among others, to partition the code in parallel, sequential, and critical sections; fork/join style function calls (respectively named `task` and `taskwait`); parallel loop directives, telling the compiler to run each iteration — or set of repetitions — in parallel (named `parallel_for`); resources for higher-level functions, like reduce, implemented on the top of its parallel loops.

OpenMP also provides a library of runtime-querying functions enabling, for instance, get or set the number of participating workers; and get current worker's id. Much of the interaction between the program and the runtime is done through system variables — *e.g.*, `OMP_NUM_PROCS` sets the default number of workers to be used. These values are taken by default by parallel programs running on a given machine and may be overwritten by the function mentioned above.

We show an example of the calculus of the n-th Fibonacci term on Figure A.2 (*cf.* the Fibonacci example in Subsection 1.4.2). This code is written in C. In this listing,

```
1   int fib (int n)
2   {
3       int a, b ;
4       if (n < 2) return n ;
5       #pragma omp task shared (a)
6       a = fib (n - 1) ;
7       #pragma omp task shared (b)
8       b = fib (n - 2) ;
9       #pragma omp taskswait
10      return a + b ;
11  }
12
```

Figure A.2: Fibonacci in OpenMP (C).

1. On line 1 we have the function signature. It receives and returns a C integer.
2. On line 3 we declare the variables that will serve as the output of the recursive call, `a` and `b`.
3. On line 4 we test the recursion limit and return the parameter if nothing has to be done.
4. On lines 5 and 7 we tell the compiler, through a `pragma omp` directive, that the next function call shall be treated as a parallel task. It implies that its execution may be done in another thread, if the scheduler decides for it. The `shared(a)` (line 5) and `shared(b)` (line 7) tells the runtime that the variables `a` and `b` are shared between the threads. In here no mutual exclusion mechanism is necessary to keep consistency since writings on it are only performed by the parent caller.
5. On lines 6 and 8 we spawn child user-level threads as a recursive invocation of `fib`.
6. On line 9 we perform a join through a `taskwait` directive, where a given user-level thread waits for the completion of all spawned threads on the current scope.
7. On line 10 we sum the outputs of the recursive calls and return it.

If the `pragma omp` clauses/lines are removed from Figure A.2 (on lines 5, 7, and 9) the resulting code is valid sequential code. This is a useful guarantee since compilers just ignore unknown pragma clauses.

The scheduling of the user threads/workers on the top of PThreads is left to each implementation.

We highlight that code on Figure A.2 has no control of granularity and is implemented naïvely, performing too many redundant calculations, and spawning one OS thread per recursive call. It is just for syntax demonstration, not meant for performance.

A.4.3 Threading Building Blocks

TBB (REINDERS, 2007) is a C++ generic library from Intel that offers data structures and algorithms ("building blocks") whose mechanics use multicore parallelism. For instance, there are constructs for explicit task parallelism, parallel loops, reducer algorithms, thread-safe containers, *etc.* Beware, though, that the "generic" part of its definition fails to accomplish full generic programming definition (STEPANOV; MCJONES, 2009). It does type instantiation, as other generic constructs, but lacks, *e.g.*,

- orthogonality between algorithms and data structures — consider that its parallel algorithms rely on a range type to work along;
- regularity (assignment, copy construction, and comparison for equivalence) for its manipulated objects — consider its container implementations.

TBB offers tools for the programmer to manipulate parallel tasks explicitly. Parallelism is not unfolded by keywords, but rather by data structures and algorithms used by the programmer. A task data structure is provided as a way to encapsulate procedure calls. These tasks can be manipulated as plain C++ objects within the code. They are scheduled by ABP work-stealing.

Since it is a pure library, some mechanics that are hidden from the programmer in Cilk and OpenMP have to be handled directly. For instance, the process of joining two parallel tasks must be written as a different type of task, called *continuation*, in order to inform the runtime that that particular task can be placed outside the local deques, waiting for their input data to be produced. In Cilk, this process is implicit — it is the code between the last **`cilk_spawn`** and the **`cilk_sync`** — and the compiler generates the correspondent code.

We show an example of the calculus of the n-th Fibonacci term on Figure A.3 (*cf.* the Fibonacci example in Subsection 1.4.2). This code is written in C++. In this listing,

1. On line 1 we have the function signature. It receives and returns a C++ integer.
2. On line 3 we declare the variables that will serve as the output of the recursive call, `a` and `b`.
3. On line 4 we test the recursion limit and return the parameter if nothing has to be done.
4. On line 5 we declare a TBB `task_group` (that belongs to a named scope called `tbb`), which will manage the parallel execution and its interaction with the runtime.

```
1   int fib (int n)
2   {
3       int a, b ;
4       if (n < 2) return n ;
5       tbb::task_group g ;
6       g.run ([&] { a = fib (n - 1) ; }) ;
7       g.run ([&] { b = fib (n - 2) ; }) ;
8       g.wait () ;
9       return a + b ;
10  }
11
```

Figure A.3: Fibonacci in TBB (C++).

5. On lines 5 and 7 we spawn child user-level threads (tasks). Method `run` of task group `g` expects a functor as the argument, to be run according to the scheduling policy. This is done through C++11 lambdas, which encapsulate runnable code in a block delimited by `{}`, which we use to encapsulate the `fib` recursive call. Artifact `[&]` is a scope delimiter and serves to specify that inside our lambda function if we refer to any outside variable this will be done by reference. This is used to write to variables `a` and `b` on the current scope from another — potentially remote — scope.
6. On line 8 we perform a join through method `wait` of task group `g`, where a given user-level thread waits for the completion of all spawned threads on the current scope.
7. On line 9 we sum the outputs of the recursive calls and return it.

Due to its intrusive nature, TBB does not provide elision semantics if its parallel support is removed. Also, some overheads that could be eliminated in compile-time (like type deduction and task dependency analysis) are present in runtime.

We highlight that code on Figure A.3 has no control of granularity and is implemented naïvely, performing too many redundant calculations, and spawning one OS thread per recursive call. It is just for syntax demonstration, not meant for performance.

A.4.4 Kaapi

Kaapi (GAUTIER; BESSERON; PIGEON, 2007) is a C/C++ framework that allows one to execute fine/medium grain multithreaded computation with dynamic data flow synchronizations. It also supports compiler directives in a fashion similar to OpenMP. Its scheduler also implements a work-stealing algorithm in a help-first fashion (*cf.* Cilk and TBB). This scheduler is also implemented in a non-blocking fashion. Kaapi's scheduler can also run in "cooperative" mode, where concurrent thieves may be able to share its load.

A Kaapi task is "annotated" with read/write modifiers. Through this explicit way of indicating dependencies, the runtime calculates the dependency graph as the computation moves forward. This allows general dependencies to be established, departing from and generalizing the strict model of Cilk. Whenever the computation follows a strict schema, the same formal guarantees are provided by the Kaapi runtime.

In Kaapi, a task is completely asynchronous regarding its parent. All reads of shared data follow the last write over it. *Ergo*, a task is not ready until all writes over its data are finished. The parent task has no direct access to its child's data. Another task must be the "continuation" that will examine if the dependencies are satisfied. (Thus, like in TBB, continuations are explicit constructs and must be handled by the programmer.)

The task parallelism in Kaapi relies on two main constructs:

Shared. A generic construct to declare variables and inform the Kaapi runtime its access patterns (`Shared_r` for read, `Shared_w` for write).

Fork. A generic construct that receives a procedure and its parameters and forks it in the Kaapi runtime.

We show an example of the calculus of the n-th Fibonacci term on Figure A.4 (*cf.* the Fibonacci example in Subsection 1.4.2). This code is written in C++. In this listing,

1. From lines 1 to 10 there is the declaration of the main task, `fib`. Like in TBB no compiler support is given, and we are unable to use lambdas as well; the code thus is more verbose. Task `fib` is declared as a C++ **`struct`**, which is equivalent to a **`class`** where all members are defaulted to public access.
2. On line 2, we overload operator parenthesis (**`operator()`**) to enable our task to be run by the framework. This is the standard C++ way (prior to C++11) to produce functor objects, whose instance may be used to pass method invocation as parameters. Thanks to it code blocks like `{ fib f ; f() ; }` are valid. The return value and parameters, however, should be passed by special references provided by the Kaapi runtime. In this case, while the first parameter **`int`** `n` is regular C++, the second parameter, `Shared_w<`**`int`**`> result` (scoped on the `ka` namespace) is a special type that gives write access to other tasks and whose basic type is an **`int`**, specified through C++ template mechanism.
3. On line 4, we declare two other write-access, int-based variables to serve as the

results of the subtasks.

4. On line 5, we test for recursive termination and write the value to the output variable `result` if it is the case. We have to use member method `write` provided by the `Shared_w` interface.
5. On lines 6 and 7 we use a particular functor from the `ka` scope named `Fork`, whose basis type must be a runnable task. Since we are declaring functor `fib`, we use itself as the basis type to emulate a recursive call on the framework. The empty parenthesis that follows declare the fork variables without binding it to a variable name (anonymous instance) and calling its default constructor. On the same line, we invoke overloaded **`operator()`** with the parameters of the recursive parallel call. This corresponds to the second pair of parenthesis on the same line.
6. On line 8 we fork another task, `sum` to serve as a continuation, being executed only when the precedent tasks on the same function scope finish. Although `sum` does not specify whose tasks it shall wait, we will see that it expects read shared variables and that we passed write shared variables. Kaapi's runtime is smart enough to guarantee that all write operations will be performed on the variables before the moment they would be read on a sequential execution.
7. From lines 12 to 20 we implement the continuation task `sum` in the same fashion we implemented `fib`. The differences is that sum expects its input arguments on overloaded **`operator()`** as a `Shared_r` (readable shared type) instead of output `Shared_w` we used for the result.
8. On line 18 we write to the results the sum of what we read on the parameters through methods `read` and `write` of the `Shared` interface.

The code on Figure A.4 was the way to pass elements around prior to C++11, without lambdas and generic facilities. While TBB embraced lambdas and higher level structures like `task_group` to eliminate boilerplate code and simplify implementation, Kaapi followed the OpenMP strategy and adopted pragma-based compiler directives. In Figure A.5 we show the same `fib` algorithm, but on the top of a compiler supporting Kaapi pragmas (*cf.* Figure A.2). This code is written in C. In this listing,

1. On line 1 we have the function signature. It receives both the input, `n`, and an output pointer to an integer, `result`. We return nothing since Kaapi works over shared pointers on its scheduling.
2. On line 3 we declare the variables that will serve as the output of the recursive call,

```
1   struct fib {
2       void operator() (int n, ka::Shared_w<int> result)
3       {
4           ka::Shared_w<int> a, b ;
5           if (n < 2) result.write (n) ;
6           ka::Fork<fib> () (n - 1, a) ;
7           ka::Fork<fib> () (n - 2, b) ;
8           ka::Fork<sum> () (result, a, b) ;
9       }
10  } ;
11
12  struct sum {
13      void operator()
14              ( ka::Shared_w<int> result
15              , ka::Shared_r<int> a
16              , ka::Shared_r<int> b )
17      {
18          result.write (a.read () + b.read ()) ;
19      }
20  } ;
21
```

Figure A.4: Fibonacci in Kaapi, structured version (C++).

`a` and `b`.

3. On line 4 we test the recursion limit and return the parameter if nothing has to be done.
4. On lines 5 and 7 we tell the compiler, through a `pragma kaapi` directive, that the next function call shall be treated as a parallel task. We specify with the `write` on the pragma that the variable between parenthesis has the write access in sharing to maintain the semantics of sequential execution. In this case, since `a` and `b` are input and output parameters, we specify its address `&a` and `&b` to be shared, effectively passing it by reference.
5. On lines 6 and 8 we spawn child user-level threads as a recursive invocation of `fib`.
6. On line 9 we perform a join through a **`sync`** directive, where a given user-level thread waits for the completion of all spawned threads on the current scope.
7. On line 10 we sum the outputs of the recursive calls and return it.

As with the previous Fibonacci examples, we highlight that this naïve recursive form is inefficient.

On its most recent incarnation, XKAAPI, the runtime was embedded with support for accelerators, like Intel Xeon Phi and GPGPU. Formal guarantees and scheduling of hybrid CPU/accelerator tasks is still a trend in its development.

```
void fib (int n, int* result)
{
    int a, b ;
    if (n < 2) *result = n ;
    #pragma kaapi task write(&a)
    fib (n - 1, &a) ;
    #pragma kaapi task write(&b)
    fib (n - 2, &b) ;
    #pragma kaapi sync
    *result = a + b ;
}

```

Figure A.5: Fibonacci in Kaapi, pragma-annotated version (C).

A.4.5 Message-Passing Interface

MPI (FORUM, 2012) is a *standard* for message-passing communication on distributed memory systems using C or Fortran. Given the profusion and depth of its usage, it is considered a *de facto* standard. Contrary to previous examples, the parallelism does not come from a structured way of programming, but rather from concurrent processes exchanging messages. The programmer builds scheduling and distribution of data in its program. The worker is a MPI process that, although usually corresponding to an OS process, may differ between implementations. The scheduling policy of the MPI process, thus, is also a factor, but usually beyond the control of the programmer.

MPI's runtime is usually a coarse-grained process manager. Among its functions, it handles routing messages and allocates MPI processes in physical processes/machines. It must be properly configured to have both permissions to access machines remotely and to access network interface locally.

Groups of processes can be interlinked in a "communicator". A process can query its communicator for the total number of workers and its (guaranteed to be unique) id within it. All processes belong to the `COMM_WORLD` communicator, having a unique integer id. Communicators can be created by the user and may possess non-integer ids — *e.g.*, cartesian coordinates.

The message-passing communication in MPI was initially performed over a LAN, in the top of POSIX sockets, point-to-point or collectively. The six main primitives are (C version):

1. *Initialize*,

   ```
   MPI_Init (int* argc, char** argv[]).
   ```

Open a channel to the manager and initialize the data structures going to be used by the MPI program. Its parameters are references/pointers to the number of arguments passed by the user and an array of strings containing the arguments themselves respectively.

2. *Finalize,*

```
MPI_Finalize ().
```

Closes all open channels between executables and managers, de-allocating resources and freeing the memory.

3. *Obtain ID.*

```
MPI_Comm_rank (MPI_Comm comm, int *rank)
```

Variable `rank` will contain the unique identifier of the invoking process on the context of communicator `comm`.

4. *Obtain number of workers.*

```
MPI_Comm_size (MPI_Comm comm, int *size)
```

Variable `size` will contain the number of workers in communicator `comm`.

5. *Send.*

```
MPI_Send (void* buf, int count, MPI_Datatype T,
    int dest, int tag, MPI_Comm comm).
```

Sends `count` values of type `T` inside buffer `buf` to the process that is identified by rank `dest` at communicator `comm`, waiting a message tagged with label `tag`. This operation is blocking until the middleware copies the buffer's content. In practice, it is *non-blocking* in the vast majority of cases. Since C does not allow us to pass a type as a parameter we have a special type variable whose type itself is a `MPI_Datatype` struct with the size of the corresponding type in that compiler within. The tag is used to differentiate messages with same content between workers. We highlight that the rank of a process has only meaning inside a communicator, possibly changing between different communicators.

6. *Receive.*

```
MPI_Recv (void* buf, int count, MPI_Datatype T,
int source, int tag, MPI_Comm comm, MPI_Status status).
```

Receives up to `count` values of type `T` inside buffer `buf` from the process that is identified by rank source at communicator comm tagged with label tag. This operation is by default *blocking* until the message arrives. All equally named attributes have the same meaning and constraints as in MPI_Send. In order to be more flexible, the programmer may specify MPI_ANY_SOURCE as source and/or MPI_ANY_TAG as a tag, to receive messages in a first-in, first-served fashion. An extra parameter, status of type MPI_Status is added in order to allow the receiver to know some meta-data on the receive operation. It is especially useful to discover the source and tag when accepting any message.

Anytime a match between sender and receiver parameters occurs a message is effectively delivered. Beware, however, that not all parameters must match; only source/destiny, communicator, and tag suffice. If the other arguments are not matched, then data will be re-interpreted by the programming language, without any extra guarantees.

With this six primitives, one can model any message-passing parallel program to be written in MPI. However, there are dozens of others, aiming at programmer's convenience, like collective communication (broadcast, gather, *etc*) and non-blocking communication (blocking receive, buffered send, *etc*), primitives for standard parallel operations (reduction, diffusion, *etc*), and fine-tuning of the runtime (type support, buffer management, *etc*).

On its first incarnation, MPI-1, all MPI processes were created before execution by the runtime. If there are more processes than machines, then they were distributed through round-robin. It follows the Single Program, Multiple Data (SPMD) model where an identical copy of the executable runs on each worker. Disjoint execution control is handled through testing of the worker's unique id — named *rank*. There is no threading support specified in MPI-1 whatsoever. On its second version, MPI-2, the standard was made more flexible, adding, among others, support for remote memory operations; support for parallel Input/Output (IO); support for shared memory systems (*e.g.*, multithreaded processors) — workers may be threads, located on the local machine or in others; compliance levels for multithreaded implementations employing other multithreaded runtimes (*e.g.* OpenMP) at the same time (different levels are required because of performance issues). The main new feature, however, is that MPI-2 supports the dynamic creation of workers during execution. The programmer may spawn an external MPI binary inside a particular type of communicator called an "intercommunicator", used to conciliate dynamically and statically created workers. The operation is slower than static creation but helps to organize execution in heterogeneous processing. Also, because any valid binary can be spawned, the paradigm is no longer within the SPMD class.

REFERENCES

AKRA, M.; BAZZI, L. On the solution of linear recurrence equations. *Comput. Optim. Appl.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 10, n. 2, p. 195–210, may 1998. ISSN 0926-6003.

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. New York, NY, USA: ACM, 1967. (AFIPS '67 (Spring)), p. 483–485. Available from Internet: <http://doi.acm.org/10.1145/1465482.1465560>.

ARORA, N. S.; BLUMOFE, R. D.; PLAXTON, C. G. Thread scheduling for multiprogrammed multiprocessors. In: *Proc. of SPAA'98*. New York, NY, USA: ACM, 1998. p. 119–129. ISBN 0-89791-989-0.

ASANOVIC, K.; BODIK, R.; DEMMEL, J.; KEAVENY, T.; KEUTZER, K.; KUBIATOWICZ, J.; MORGAN, N.; PATTERSON, D.; SEN, K.; WAWRZYNEK, J.; WESSEL, D.; YELICK, K. A view of the parallel computing landscape. *Commun. ACM*, ACM, New York, NY, USA, v. 52, n. 10, p. 56–67, oct. 2009. ISSN 0001-0782. Available from Internet: <http://doi.acm.org/10.1145/1562764.1562783>.

AUSTERN, M. H.; TOWLE, R. A.; STEPANOV, A. A. Range partition adaptors: a mechanism for parallelizing stl. *SIGAPP Appl. Comput. Rev.*, ACM, New York, NY, USA, v. 4, n. 1, p. 5–6, abr. 1996. ISSN 1559-6915.

BARKER, E.; KELSEY, J. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. [S.l.], 2012.

BENDER, M. A.; RABIN, M. O. Scheduling cilk multithreaded parallel programs on processors of different speeds. In: *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*. New York, NY, USA: ACM, 2000. (SPAA '00), p. 13–21. ISBN 1-58113-185-2. Available from Internet: <http://doi.acm.org/10.1145/341800.341803>.

BERENBRINK, P.; FRIEDETZKY, T.; GOLDBERG, L. A. The natural work-stealing algorithm is stable. *SIAM J. Comput.*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 32, n. 5, p. 1260–1279, may 2003. ISSN 0097-5397. Available from Internet: <http://dx.doi.org/10.1137/S0097539701399551>.

BERNARD, J.; ROCH, J.-L.; TRAORÉ, D. Processor-oblivious parallel stream computations. In: *PDP'08, 16th Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing*. Toulouse, France: IEEE Computer Society Press, 2008. p. 72–76.

BLUM, L.; BLUM, M.; SHUB, M. A simple unpredictable pseudo random number generator. *SIAM J. Comput.*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 15, n. 2, p. 364–383, may 1986. ISSN 0097-5397.

BLUMOFE, R. D. Scheduling multithreaded computations by work stealing. In: *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. [S.l.: s.n.], 1994. p. 356–368.

BLUMOFE, R. D.; LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *J. ACM*, v. 46, n. 5, p. 720–748, 1999.

BLUMOFE, R. D.; PAPADOPOULOS, D. *Hood: A User-Level Threads Library for Multiprogrammed Multiprocessors.* [S.l.], 1998.

BRENT, R. P. The parallel evaluation of general arithmetic expressions. *J. ACM*, ACM, New York, NY, USA, v. 21, n. 2, p. 201–206, abr. 1974. ISSN 0004-5411. Available from Internet: <http://doi.acm.org/10.1145/321812.321815>.

CAPPELLO, F.; CARON, E.; DAYDE, M.; DESPREZ, F.; JEGOU, Y.; PRIMET, P.; JEANNOT, E.; LANTERI, S.; LEDUC, J.; MELAB, N.; MORNET, G.; NAMYST, R.; QUETIER, B.; RICHARD, O. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In: *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing.* Washington, DC, USA: IEEE Computer Society, 2005. (GRID '05), p. 99–106. ISBN 0-7803-9492-5. Available from Internet: <http://dx.doi.org/10.1109/GRID.2005.1542730>.

CASANOVA, H.; LEGRAND, A.; ROBERT, Y. *Parallel Algorithms.* 1st. ed. [S.l.]: Chapman & Hall/CRC, 2008. ISBN 9781584889458.

CHATTERJEE, S.; GROSSMAN, M.; SBîRLEA, A.; SARKAR, V. Dynamic task parallelism with a gpu work-stealing runtime system. In: RAJOPADHYE, S.; STROUT, M. M. (Ed.). *Languages and Compilers for Parallel Computing.* Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 7146). p. 203–217. ISBN 978-3-642-36035-0. Available from Internet: <http://dx.doi.org/10.1007/978-3-642-36036-7_14>.

CODDINGTON, P. Random number generators for parallel computers. *The NHSE Review*, v. 2, 1997.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to Algorithms, Third Edition.* 3rd. ed. [S.l.]: The MIT Press, 2009. ISBN 0262033844, 9780262033848.

CULLER, D.; KARP, R.; PATTERSON, D.; SAHAY, A.; SCHAUSER, K. E.; SANTOS, E.; SUBRAMONIAN, R.; EICKEN, T. von. Logp: Towards a realistic model of parallel computation. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 28, n. 7, p. 1–12, jul. 1993. ISSN 0362-1340. Available from Internet: <http://doi.acm.org/10.1145/173284.155333>.

CUNG, V. D. C.; DANJEAN, V.; DUMAS, J.-G.; GAUTIER, T.; HUARD, G.; RAFFIN, B.; RAPINE, C.; ROCH, J.-L.; TRYSTRAM, D. Adaptive and hybrid algorithms: classification and illustration on triangular system solving. In: DUMAS, J. (Ed.). *Transgressive Computing TC'2006.* Granada, Spain: [s.n.], 2006. p. 131–148. ISBN 84-689-8381-0.

DAGUM, L.; MENON, R. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 5, n. 1, p. 46–55, jan. 1998. ISSN 1070-9924. Available from Internet: <http://dx.doi.org/10.1109/99.660313>.

DIJKSTRA, E. W. Solution of a problem in concurrent programming control. *Commun. ACM*, ACM, New York, NY, USA, v. 8, n. 9, p. 569–, sep. 1965. ISSN 0001-0782. Available from Internet: <http://doi.acm.org/10.1145/365559.365617>.

DIJKSTRA, E. W. *A Discipline of Programming*. 1st. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997. ISBN 013215871X.

EICHENAUER, J.; LEHN, J. A non-linear congruential pseudo random number generator. *Statistische Hefte*, Springer-Verlag, Berlin, v. 27, p. 315–326, 1986. ISSN 0932-5026.

ESTRIN, G. Organization of computer systems: The fixed plus variable structure computer. In: *Papers Presented at the May 3-5, 1960, Western Joint IRE-AIEE-ACM Computer Conference*. New York, NY, USA: ACM, 1960. (IRE-AIEE-ACM '60 (Western)), p. 33–40.

FERREIRA, A.; SCHABANEL, N. A randomized BSP/CGM algorithm for the maximal independent set problem. *Parallel Processing Letters*, v. 9, n. 3, p. 411–422, 1999. Available from Internet: <http://dx.doi.org/10.1142/S0129626499000384>.

FISCHER, G. W.; CARMON, Z.; ARIELY, D.; ZAUBERMAN, G.; L'ECUYER, P. Good parameters and implementations for combined multiple recursive random number generators. *Oper. Res.*, INFORMS, Institute for Operations Research and the Management Sciences (INFORMS), Linthicum, Maryland, USA, v. 47, n. 1, p. 159–164, jan. 1999. ISSN 0030-364X.

FLYNN, M. J. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, IEEE Computer Society, Washington, DC, USA, v. 21, n. 9, p. 948–960, sep. 1972. ISSN 0018-9340. Available from Internet: <http://dx.doi.org/10.1109/TC.1972.5009071>.

FORUM, M. P. I. *MPI: A Message-Passing Interface Standard Version 3.0*. 2012.

FOSTER, I.; KESSELMAN, C. *The Grid: Blueprint for a New Computing Infrastructure*. [S.l.]: Morgan Kaufmann, 1999.

FRIGO, M.; HALPERN, P.; LEISERSON, C. E.; LEWIN-BERLIN, S. Reducers and other cilk++ hyperobjects. In: *Proc. of SPAA'09*. New York, NY, USA: ACM, 2009. p. 79–90. ISBN 978-1-60558-606-9.

FRIGO, M.; LEISERSON, C. E.; PROKOP, H.; RAMACHANDRAN, S. Cache-oblivious algorithms. In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1999. (FOCS '99), p. 285–. ISBN 0-7695-0409-4. Available from Internet: <http://dl.acm.org/citation.cfm?id=795665.796479>.

FRIGO, M.; LEISERSON, C. E.; RANDALL, K. H. The implementation of the cilk-5 multithreaded language. In: *Proc. of PLDI'98*. New York, NY, USA: ACM, 1998. p. 212–223. ISBN 0-89791-987-4.

GAUTIER, T.; BESSERON, X.; PIGEON, L. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: *Proceedings of the*

2007 International Workshop on Parallel Symbolic Computation. New York, NY, USA: ACM, 2007. (PASCO '07), p. 15–23. ISBN 978-1-59593-741-4. Available from Internet: <http://doi.acm.org/10.1145/1278177.1278182>.

GRAHAM, R. L. Bounds on multiprocessing timing anomalies. *SIAM JOURNAL ON APPLIED MATHEMATICS*, v. 17, n. 2, p. 416–429, 1969.

GUO, Y.; BARIK, R.; RAMAN, R.; SARKAR, V. Work-first and help-first scheduling policies for async-finish task parallelism. In: *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009. (IPDPS '09), p. 1–12. ISBN 978-1-4244-3751-1. Available from Internet: <http://dx.doi.org/10.1109/IPDPS.2009.5161079>.

HARAMOTO, H.; MATSUMOTO, M.; L'ECUYER, P. A fast jump ahead algorithm for linear recurrences in a polynomial space. In: *Proc. of SETA'08*. Berlin, Heidelberg: Springer-Verlag, 2008. p. 290–298. ISBN 978-3-540-85911-6.

HARAMOTO, H.; MATSUMOTO, M.; NISHIMURA, T.; PANNETON, F.; L'ECUYER, P. Efficient jump ahead for f2-linear random number generators. *INFORMS J. on Computing*, INFORMS, Institute for Operations Research and the Management Sciences (INFORMS), Linthicum, Maryland, USA, v. 20, n. 3, p. 385–390, jul. 2008. ISSN 1526-5528.

HERLIHY, M.; SHAVIT, N. *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN 0123705916, 9780123705914.

HOARE, C. A. R. Communicating sequential processes. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 8, p. 666–677, aug. 1978. ISSN 0001-0782. Available from Internet: <http://doi.acm.org/10.1145/359576.359585>.

Intel Corporation. *Intel Cilk Plus Language Specification*. 2013.

JAJA, J. *An Introduction to Parallel Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1992. ISBN 0-201-54856-9.

KERNIGHAN, B. W. *The C Programming Language*. 2nd. ed. [S.l.]: Prentice Hall Professional Technical Reference, 1988. ISBN 0131103709.

KNUTH, D. E. *The art of computer programming, volume 2 (2nd ed.): seminumerical algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1985. ISBN 0-201-89684-2.

KNUTH, D. E. *The Stanford GraphBase: A Platform for Combinatorial Computing*. New York, NY, USA: ACM, 1993. ISBN 0-201-54275-7.

KNUTH, D. E. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-89683-4.

KNUTH, D. E. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0201896842. Available from Internet: <http://portal.acm.org/citation.cfm?id=270146>.

KNUTH, D. E. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN 0-201-89685-0.

KRUSKAL, C. P.; RUDOLPH, L.; SNIR, M. A complexity theory of efficient parallel algorithms. *Theor. Comput. Sci.*, Elsevier Science Publishers Ltd., Essex, UK, v. 71, n. 1, p. 95–132, mar. 1990. ISSN 0304-3975. Available from Internet: <http://dx.doi.org/10.1016/0304-3975(90)90192-K>.

KUMAR, V. *Introduction to Parallel Computing*. 2nd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0201648652.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 7, p. 558–565, jul. 1978. ISSN 0001-0782. Available from Internet: <http://doi.acm.org/10.1145/359545.359563>.

L'ECUYER, P. Maximally equidistributed combined tausworthe generators. *Mathematics of Computation*, v. 65, n. 213, p. 203–213, 1996.

L'ECUYER, P.; SIMARD, R. Testu01: A c library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, ACM, New York, NY, USA, v. 33, n. 4, aug. 2007. ISSN 0098-3500. Available from Internet: <http://doi.acm.org/10.1145/1268776.1268777>.

L'ECUYER, P.; SIMARD, R. J.; CHEN, E. J.; KELTON, W. D. An object-oriented random-number package with many long streams and substreams. *Operations Research*, v. 50, n. 6, p. 1073–1075, 2002.

LEE, I.-T. A.; BOYD-WICKIZER, S.; HUANG, Z.; LEISERSON, C. E. Using memory mapping to support cactus stacks in work-stealing runtime systems. In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. New York, NY, USA: ACM, 2010. (PACT '10), p. 411–420. ISBN 978-1-4503-0178-7. Available from Internet: <http://doi.acm.org/10.1145/1854273.1854324>.

LEISERSON, C. E. The cilk++ concurrency platform. In: *Proc. of DAC'09*. New York, NY, USA: ACM, 2009. p. 522–527. ISBN 978-1-60558-497-3.

LEISERSON, C. E.; SCHARDL, T. B.; SUKHA, J. Deterministic parallel random-number generation for dynamic-multithreading platforms. In: *Proc. of PPoPP'12*. New York, NY, USA: ACM, 2012. p. 193–204. ISBN 978-1-4503-1160-1.

LIMA, J. V. F.; GAUTIER, T.; MAILLARD, N.; RAFFIN, B. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2013. (IPDPS '13), p. 1299–1308. ISBN 978-0-7695-4971-2. Available from Internet: <http://dx.doi.org/10.1109/IPDPS.2013.66>.

LUBY, M. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 15, n. 4, p. 1036–1055, nov. 1986. ISSN 0097-5397. Available from Internet: <http://dx.doi.org/10.1137/0215074>.

MARSAGLIA, G. Xorshift rngs. *Journal of Statistical Software*, v. 8, n. 14, p. 1–6, 7 2003. ISSN 1548-7660.

MASCAGNI, M. *Polynomial Versus Matrix Methods for Leap-ahead in Shift-register Type Pseudorandom Number Generators*. [S.l.]: Institute for Mathematics and its Applications, University of Minnesota, 1997.

MASCAGNI, M.; SRINIVASAN, A. Algorithm 806: Sprng: a scalable library for pseudorandom number generation. *ACM Trans. Math. Softw.*, ACM, New York, NY, USA, v. 26, n. 3, p. 436–461, sep. 2000. ISSN 0098-3500.

MATSUMOTO, M.; NISHIMURA, T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, ACM, New York, NY, USA, v. 8, n. 1, p. 3–30, jan. 1998. ISSN 1049-3301.

MICHAEL, M. M.; VECHEV, M. T.; SARASWAT, V. A. Idempotent work stealing. In: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2009. (PPoPP '09), p. 45–54. ISBN 978-1-60558-397-6. Available from Internet: <http://doi.acm.org/10.1145/1504176.1504186>.

MOR, S.; MAILLARD, N. Dynamic workload balancing deques for branch and bound algorithms in the message passing interface. *International Journal on High Performance Systems Architecture*, Inderscience Publishers, Inderscience Publishers, Geneva, SWITZERLAND, v. 3, n. 2/3, p. 77–86, may 2011. ISSN 1751-6528. Available from Internet: <http://dx.doi.org/10.1504/IJHPSA.2011.040461>.

MOR, S.; ROCH, J.; MAILLARD, N. Generic deterministic random number generation in dynamic-multithreaded platforms. In: *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*. [s.n.], 2014. p. 427–438. Available from Internet: <http://dx.doi.org/10.1007/978-3-319-09873-9_36>.

MUELLER, F. A library implementation of posix threads under unix. In: *In Proceedings of the USENIX Conference*. [S.l.: s.n.], 1993. p. 29–41.

MUSSER, D. R. Introspective sorting and selection algorithms. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 27, n. 8, p. 983–993, aug. 1997. ISSN 0038-0644.

PACHECO, P. *An Introduction to Parallel Programming*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 9780123742605.

PACHECO, P. S. *Parallel Programming with MPI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996. ISBN 1-55860-339-5.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. 4th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN 0123744938, 9780123744937.

PFISTER, G. F. *In Search of Clusters (2Nd Ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998. ISBN 0-13-899709-8.

PLAUGER, P.; LEE, M.; MUSSER, D.; STEPANOV, A. A. *C++ Standard Template Library*. 1st. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000. ISBN 0134376331.

REINDERS, J. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. First. Sebastopol, CA, USA: O'Reilly, 2007. I-XXV, 1-303 p. ISBN 978-0-596-51480-8.

ROCH, J.-L. *Work complexity analysis of parallel algorithm in a concurrent system using clocks*. 2012. Presentation - Seminar ID. Personal Communication.

SALMON, J. K.; MORAES, M. A.; DROR, R. O.; SHAW, D. E. Parallel random numbers: as easy as 1, 2, 3. In: *Proc. of SC'11*. New York, NY, USA: ACM, 2011. p. 16:1–16:12. ISBN 978-1-4503-0771-0.

SCHALLER, R. R. Moore's law: Past, present, and future. *IEEE Spectr.*, IEEE Press, Piscataway, NJ, USA, v. 34, n. 6, p. 52–59, jun. 1997. ISSN 0018-9235. Available from Internet: <http://dx.doi.org/10.1109/6.591665>.

SHUN, J.; BLELLOCH, G. E.; FINEMAN, J. T.; GIBBONS, P. B.; KYROLA, A.; SIMHADRI, H. V.; TANGWONGSAN, K. Brief announcement: The problem based benchmark suite. In: *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA: ACM, 2012. (SPAA '12), p. 68–70. ISBN 978-1-4503-1213-4. Available from Internet: <http://doi.acm.org/10.1145/2312005.2312018>.

SHUN, J.; BLELLOCH, G. E.; FINEMAN, J. T.; GIBBONS, P. B. Reducing contention through priority updates. In: *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA: ACM, 2013. (SPAA '13), p. 152–163. ISBN 978-1-4503-1572-2. Available from Internet: <http://doi.acm.org/10.1145/2486159.2486189>.

SIMHADRI, H. V.; BLELLOCH, G. E.; FINEMAN, J. T.; GIBBONS, P. B.; KYROLA, A. Experimental analysis of space-bounded schedulers. In: *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA: ACM, 2014. (SPAA '14), p. 30–41. ISBN 978-1-4503-2821-0. Available from Internet: <http://doi.acm.org/10.1145/2612669.2612678>.

STEELE JR., G. L.; LEA, D.; FLOOD, C. H. Fast splittable pseudorandom number generators. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &#38; Applications*. New York, NY, USA: ACM, 2014. (OOPSLA '14), p. 453–472. ISBN 978-1-4503-2585-1. Available from Internet: <http://doi.acm.org/10.1145/2660193.2660195>.

STEELE JR., G. L.; LEA, D.; FLOOD, C. H. Fast splittable pseudorandom number generators. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 49, n. 10, p. 453–472, oct. 2014. ISSN 0362-1340. Available from Internet: <http://doi.acm.org/10.1145/2714064.2660195>.

STEPANOV, A.; MCJONES, P. *Elements of Programming*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2009. ISBN 032163537X, 9780321635372.

STEPANOV, A. A.; ROSE, D. E. *From Mathematics to Generic Programming*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2014. ISBN 0321942043, 9780321942043.

STROUSTRUP, B. *The C++ Programming Language*. 3rd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 0201700735.

SUKSOMPONG, W. *Bounds on Multithreaded Computations by Work Stealing*. Dissertation (Master) — Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, jun. 2014.

TCHIBOUKDJIAN, M.; GAST, N.; TRYSTRAM, D. Decentralized list scheduling. *Annals OR*, v. 207, n. 1, p. 237–259, 2013. Available from Internet: <http://dx.doi.org/10.1007/s10479-012-1149-7>.

TCHIBOUKDJIAN, M.; GAST, N.; TRYSTRAM, D.; ROCH, J.-L.; BERNARD, J. A tighter analysis of work stealing. In: CHEONG, O.; CHWA, K.-Y.; PARK, K. (Ed.). *ISAAC (2)*. [S.l.]: Springer, 2010. (Lecture Notes in Computer Science, v. 6507), p. 291–302. ISBN 978-3-642-17513-8.

TRAORÉ, D. *Self-adaptive parallel algorithms and applications*. Thesis (Theses) — Institut National Polytechnique de Grenoble - INPG, dec. 2008. Available from Internet: <https://tel.archives-ouvertes.fr/tel-00353274>.

TRAORÉ, D.; ROCH, J.-L.; MAILLARD, N.; GAUTIER, T.; BERNARD, J. Deque-free work-optimal parallel stl algorithms. In: *Proc. of Euro-Par'08*. Berlin, Heidelberg: Springer-Verlag, 2008. p. 887–897. ISBN 978-3-540-85450-0. Available from Internet: <http://dx.doi.org/10.1007/978-3-540-85451-7_95>.