MSc. Informatica Thesis

Parameter-Bounded Parallelism for Fixed Parameter Tractable Problems

Cyrus K. Vattes
c.k.vattes@uu.nl
4800516

Supervisor
Prof. Dr. H.L. Bodlaender

Examiner
Dr. J.M.M. van Rooij

Utrecht University

March 31, 2025

Abstract

Fixed-parameter tractability (FPT) addresses computational hardness by isolating complexity to a problem-specific parameter. As input sizes grow, so does the need for parallelism, particularly when resources are constrained. Current parameterized parallel complexity classes often assume unbounded process counts, leading to inefficient or unpredictable workloads when the process count is limited. This paper introduces Bounded-Process Parallel FPT (BPP-FPT) as a class of parallel FPT algorithms with polynomial runtime in the input size and process count bounded by a function of the parameter. We demonstrate the feasibility of this class with BPP-FPT algorithms for Vertex Cover, Feedback Vertex Set, and Longest path, and show that problems solvable by a bounded search tree of parameter-dependent depth can be parallelized into a BPP-FPT algorithm. Analysis of the work and process efficiency, including computational overhead and process overprovisioning, establishes lower bounds for BPP-FPT work. Finally, we propose solutions to the efficiency trade-offs of this new class of algorithms and discuss ways to optimize resource allocation and preprocessing time. This work addresses a gap in the current models of parameterized parallel complexity theory and provides a practical framework for developing parallel FPT algorithms under realistic resource constraints.

Contents

1 Introduction **2**
1.1 Organization of this Paper . . . 2
1.2 Related Work . . . 2

2 Terminology **4**

3 Bounded-Process Parallel FPT (BPP-FPT) **5**
3.1 Time and Process Scaling . . . 5
3.2 Vertex Cover (BST) . . . 6
3.3 Frontier Set Generation . . . 7
3.4 Process Count Identifiers . . . 9
3.5 Parallel Construction . . . 9
3.6 Subprocess General Form . . . 10

4 Structural Boundaries for BPP-FPT **10**
4.1 Vertex Cover (IC) . . . 10
4.2 Feedback Vertex Set (BST) . . . 12
4.3 Feedback Vertex Set (IC) . . . 14
4.4 Longest Path (CC) . . . 16

5 Work and Process Bounds **18**
5.1 Overhead . . . 18
5.2 Overprovisioning . . . 18
5.3 Lower Bounds . . . 19

6 Work-Efficiency **19**
6.1 Assumed Hardness . . . 19
6.2 Optimal Provisioning . . . 20
6.3 Kernelization . . . 20

7 Conclusion **21**
7.1 Further Work . . . 21

1 Introduction

Fixed-parameter tractability (FPT) provides a framework for solving otherwise intractable problems by isolating the source of computational hardness to an advice parameter. As input sizes continue to grow, and parallel architectures become more common, understanding how parameterized problems can be parallelized — particularly under resource constraints — has become increasingly important.

While class hierarchies for parameterized parallel complexity exist, they often assume large, input-dependent process counts. In practice, hardware limitations often impose strict upper limits on parallelism, raising a fundamental question: is there a class of FPT problems that can be efficiently parallelized using a tightly constrained number of processes? This paper examines the structure and practicality of such a class, asking how and when the parameter function can be shifted from runtime to process count.

1.1 Organization of this Paper

This paper examines the existence, boundaries, and applications for a class of problems solvable by parallel fixed-parameter algorithms when the number of processes is bounded by some function of k. Section 3 describes the theoretical foundations for this new class of problems, proves its existence, and gives a detailed algorithmic implementation for the vertex cover problem. Section 4 identifies requirements and FPT techniques for constructing a bounded-process parallel FPT algorithm. Section 5 discusses lower bounds for bounded-process parallel FPT algorithms, and Section 6 explores tools for improving the efficiency of such algorithms.

1.2 Related Work

Multi-Resource complexity: In designing and analyzing computational problems, we are chiefly concerned with algorithmic runtime. However, the structures of many problems, defined by size, shape, topology, etc., may contribute to the complexity of algorithmic solutions [18]. In the study of algorithmic complexity, we refer to these types of problems as *multi-resource*, *multivariate*, or *parameterized* problems.

Hard problems may become tractable with the introduction of a parameter. Generally, parameterization aims to isolate the source of *exponential complexity* in the problem [20] from the input size. Parameterized problems may be solved more efficiently than with a brute-force approach [16, 20].

We must then ask which parts of a problem may be parameterized and what trade-offs must be made to effectively use the parameter. Time and space are traditionally the most important resources in this regard. Whether a parameterized algorithm is feasible depends on whether a problem can be reduced to some maximum size (kernelization) or split into a parameter-dependent number of subproblems (branching) [16].

In the scope of this work, we consider the time and space requirements of parameterized problems in the context of parallelism: whether a parameter-dependent number of processes can efficiently solve a given problem.

Parallelism & Process Scaling: As processing speeds approach their theoretical limit, alternative strategies are needed to compute large or difficult problems [21]. Parallelism and distributed computing exploit natural structures (such as branching) to overcome the physical limitations of sequential algorithms. Because parameterized problems often use such structures, parallelism can solve problems that are otherwise infeasible [15].

Typically, parallel algorithms must use a large number of processes (dependent on the input size) to efficiently solve a problem [14]. The process count for a parallel parameterized algorithm is known to be scalable to some maximal number depending on the algorithmic structure [2, 3] or circuit size and depth [10, 11].

Though using input-dependent-many processes may lead to significant runtime improvements (in some cases, *constant time* [8]), there are practical limits to the number of processes available for any given problem. Even in cases where the number of available processes is close to the optimal limit, poor implementation or scheduling may drastically reduce the overall speedup over a sequential algorithm [3, 23].

In an effort to clearly discuss *speed* and *efficiency* as they relate to parallelism and algorithmic complexity, this work will focus on the total *work* performed by an algorithm. In this context, algorithmic work is defined as the cumulative runtime of all processes of a parallel algorithm. While a parallel algorithm may have a smaller overall runtime, its work can never be less than that of the fastest sequential algorithm for the same problem [21]. We call an algorithm *work-optimal* if it matches this lower bound [7]. Work-efficiency, work-optimality, and work/process bounding are discussed further in Sections 5 and 6.

Parameterized Parallel Subclasses of FPT: Classes of parameterized parallel complexity were introduced and formally defined by Cesati and Ianni [14], then further refined by Abu-Zham et al. [4]. These class definitions and hierarchies build a foundation for the time and process scale described in Section 3.1.

The NC-hierarchy for parallel complexity is given by $\mathrm{NC}^i \subseteq \mathrm{NC}^{i+1} \subseteq \cdots \mathrm{NC}$ for $i \geq 1$, where NC^i is the set of problems that can be solved in $\mathcal{O}(\log^i n)$ time using a polynomial number of processes [25]. This hierarchy describes the complexity of classical problems, and is further refined to accommodate parameterized problems with the classes PNC, FPP, and FPPT.

PNC is a parameterized analog of NC, where PNC $\subset$ NC and PNC $\subset$ FPT. A problem is in PNC if it can be solved in $\mathcal{O}(f(k) \cdot \log^{h(k)} n)$ time using $\mathcal{O}(g(k) \cdot n^{\beta})$ processes [14].

FPP is a restricted subclass of PNC which replaces the arbitrary function h with a constant α. A problem is in FPP if it can be solved in $\mathcal{O}(f(k) \cdot \log^{\alpha} n)$ time using $\mathcal{O}(g(k) \cdot n^{\beta})$ processes. FPP $\subseteq$ PNC [14].

FPPT is an equivalent refinement of FPP, obtained by simulating the work of $\mathcal{O}(g(k) \cdot n^{\beta})$ processes on $\mathcal{O}(n^{\beta})$ processes [2, 4].

$$\mathrm{FPPT} = \mathrm{FPP} \subseteq \mathrm{PNC} \subset \mathrm{NC}$$
$$\mathrm{FPPT} = \mathrm{FPP} \subseteq \mathrm{PNC} \subset \mathrm{FPT}$$

Parameterized AC Complexity Classes: The AC-hierarchy ($\mathrm{AC}^0 \subseteq \mathrm{AC}^i \subseteq \cdots \mathrm{AC}$ for $i \geq 1$) describes the set of problems recognized by Boolean circuits of depth $\mathcal{O}(\log^i n)$ and a polynomial number of unbounded fan-in AND and OR gates [2]. The Boolean circuit of AC^0 is of constant depth. There exists a relation between the AC-hierarchy and NC hierarchy, as given by $\mathrm{AC}^{i-1} \subseteq \mathrm{NC}^i \subseteq \mathrm{AC}^i \subseteq \mathrm{NC}^{i+1}$ for $i \geq 1$ [6]. In this case, the Boolean circuit can be represented as a tree of depth $\mathcal{O}(\log^i n)$ and a polynomial number of nodes.

Using a equivalent definition for FPT = para-P gives the following parameterized class inclusion hierarchy: para-$\mathrm{AC}^0 \subsetneq$ para-$\mathrm{NC}^1 \subseteq$ para-$\mathrm{AC}^1 \subseteq \cdots$ para-P [9]. For $i \geq 0$, the depth of a para-AC function is defined as para-$\mathrm{AC}_{\mathcal{O}(f(k) \cdot \log^i n)}$ = para-$\mathrm{AC}^{i\uparrow}$.

The class of problems solvable in parameter time and FPT work is para-$AC^{0\uparrow}$. For $i > 0$, para-$AC^{i\uparrow}$ = FPPT.

$$\text{para-AC}^0 \subsetneq \text{para-AC}^{0\uparrow} \subseteq \text{para-AC}^1 \subseteq \text{para-P} = \text{FPT}$$
$$\text{para-AC}^0 \subsetneq \text{para-AC}^{0\uparrow} \subseteq \text{FPPT} \subset \text{FPT}$$

Parallel Kernelization: Kernelization techniques are notoriously sequential, and parallel and distributed kernelization are areas of frequent study within parameterized complexity theory. Building on the connection between kernelization and parameterized complexity [20], Bannach and Tantau (2018) define a *parallel analog* of FPT as "kernels computed in polynomial time" [11]. Multiple methods now exist within distributed computation [26], parallel computation [3, 7, 11], kernel reduction [1, 5, 11], and kernel application [3, 4, 7] to overcome this bottleneck.

Of particular interest to this work, a kernel can be resized by $f(k)$ rounds of distributed communication [26] or by reshaping the circuit family to remove the parameter dependence from its depth [11]. Although here is some interesting overlap between the parameterized, parallel, and distributed complexity hierarchies [26], this work will only consider distributed kernelization for its relevance to parallelism and kernel interleaving [3]. Kernelization is examined in Section 6.3.

2 Terminology

Parameterized Problem — A problem where the complexity is measured in terms of the input and an additional parameter. Formally, a language $(x, k) \in \Sigma^* \times \mathbb{N}$, where Σ is a fixed, finite alphabet and k is a numeric parameter [16, 20].

Work — The total number of computational steps performed by an algorithm, summed over all processing units [7]. For a sequential algorithm, $W(n)$ equals the runtime. A parallel algorithm with $W(n)$ work can never have a runtime less than $\frac{W(n)}{p}$. In this case, work can be understood as (*runtime* · *process count*).

***n*, *k*, *p*, p_i** — The *size* of an algorithmic input is given by n. The numeric *parameter* is given by k. The total *processes count* is given by p. The *process identifier* is given by p_i, where $0 \leq p_i < p$.

$N(v)$ — The *open neighborhood* of a vertex $v \in V$ of graph $G = (V, E)$.

BST — *Bounded Search Tree*: a recursive branching algorithm with a fixed depth and branching factor. Branches that exceed a parameter bound are pruned.

IC — *Iterative Compression*: a method for solving problems by building a solution incrementally. Given a solution for a smaller instance, a new solution set is found (compression) as new elements are added to the problem instance (iteration).

CC — *Color Coding*: a randomized algorithmic technique for detecting subgraph components. The algorithm randomly colors vertices and checks for a solution. Because the probability of finding a solution with a random coloring is low, this process is repeated many times.

VC — Vertex Cover Problem

Given: Graph $G = (V, E)$, integer $k \geq 0$.
Question: Does G have a subset $S \subseteq V$ of at most k vertices such that every edge in E has at least one endpoint in S?

FVS — Feedback Vertex Set Problem

> **Given:** Graph $G = (V, E)$, integer $k \geq 0$.
> **Question:** Does G have a subset $S \subseteq V$ of at most k vertices such that $G - S$ is acyclic?

LONGPATH — Longest Path Problem

> **Given:** Graph $G = (V, E)$, integer $k \geq 0$.
> **Question:** Does G have a simple path of length at least k?

3 Bounded-Process Parallel FPT (BPP-FPT)

Abu-Kzam et al. (2006) describe one problem when implementing parallel algorithms on a limited number of processes:

> Suppose both n and k are large, and 32 processors are available. Because the search tree has a branching factor of two, decomposition will have used the first $5 \ll k$ levels of its tree to split the input into 32 subgraphs, one for each processor. In turn, each processor will, in parallel, examine its subgraph using the search tree technique. [3]

The theoretical runtime in this example is drastically different than the practical runtime. This discrepancy grows as $f(k)$ and the process count diverge. This section seeks to develop a theory around such limitations. Reframing parallel FPT algorithms around a parameter-dependent number of processes will improve runtime estimations and allow for more flexibility in algorithm choice for different computing environments.

The goal of this section is to determine whether the parameter function can be removed from the runtime of some FPT problems when parallelized to $f(k)$ processes. Algorithms of this class should run in polynomial time on the input size. We refer to this proposed class of algorithms as "Bounded-Process Parallel, Fixed-Parameter Tractable", or **BPP-FPT**.

3.1 Time and Process Scaling

The question of whether certain problems in FPT can be parallelized to $f(k)$ processes and solved in n^c time begins with reviewing the class inclusion structures presented in Section 1.2, *Parameterized Parallel Subclasses of FPT* and *Parameterized AC Complexity Classes*:

$$\text{FPPT} = \text{FPP} \subseteq \text{PNC} \subset \text{FPT} \tag{1}$$

$$\text{para-AC}^0 \subsetneq \text{para-AC}^{0\uparrow} \subseteq \text{FPPT} \subset \text{FPT} \tag{2}$$

Hierarchy 1 shows the parameterized parallel subclasses of FPT under classical computational complexity. Hierarchy 2 adds the parameterized alternating circuit complexity classes requiring FPT work. Taken together, these inclusion structures reveal a scale for parallelizing FPT problems, with runtimes ranging from constant time (para-AC^0) to polylogarithmic time (PNC).

Considering that all problems contained in hierarchies 1 and 2 must perform FPT work, the time and process count requirements for these classes can be plotted together. Figure 1 shows the relationship between process count and runtime for FPT-work algorithms and highlights the gap for algorithms parallelized to $f(k)$ processes.

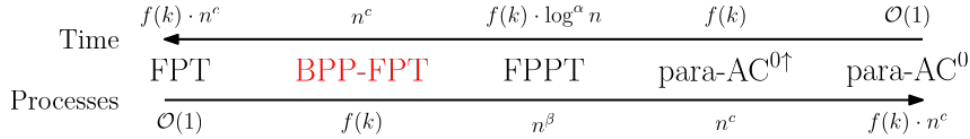


Figure 1: Time and process scale for parallel FPT-work.

3.2 Vertex Cover (BST)

To construct a BPP-FPT algorithm, we must idendify problems for which the parameter function can be moved from the runtime to the process count. Here we will examine the vertex cover problem, which can be solved with FPT work using a bounded search tree (BST) of depth k. The sample graph given in Figure 2 will be used as the basis for algorithms explored here and in Section 4.

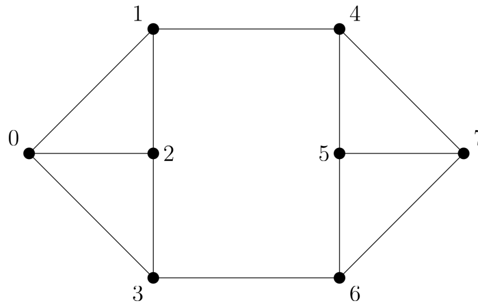


Figure 2: Sample graph G

The parallel runtime of a BST algorithm is proportional to the depth of the search tree, even in cases where aggressive branching can reduce it [7]. Our goal is to show that a k-depth BST can be parallelized to $f(k)$ independent processes, thus removing the parameter function from our runtime.

The *vertex cover problem* (VC) asks whether all edges of a graph $G = (V, E)$ have at least one endpoint in a set $S \subseteq V$ of size at most k. Equivalently, the set S is a vertex cover of G if the subgraph $G - S$ contains no edges [16].

For each vertex $v \in V$ to be covered, either v or $N(v)$ must be in S. A simple BST algorithm exploits this fact by branching on the selection of v or $N(v)$. The resulting tree has depth k and contains at most $2^{k+1} - 1$ nodes (see Figure 3).

This algorithm runs sequentially in $\mathcal{O}(2^k \cdot n)$ time. If multiple processors are available, the search tree can be decomposed into p distinct subgraphs with a parameter value of $k' \leq k - \lfloor \log_2 p \rfloor$. However, this sort of static parallelization can lead to performance penalties and resource underutilization [3].

For the runtime of a BPP-FPT algorithm to be fully independent of the parameter function, we must find a way to parallelize the $f(k)$ leaves of the search tree onto $f(k)$ processes without sequentially branching at each step. To accomplish this, we first look at another algorithmic approach to solving vertex cover.

Figure 4 shows a partial implementation (up to $k = 3$) of an $\mathcal{O}^*(2^k)$ edge removal algorithm. A new edge is selected at each level of the search tree, and the tree branches on which of its two endpoints are removed. The correctness of this algorithm follows from a simplification of the rules for the BST algorithm: for each edge selected, each of its endpoints must be in v or $N(v)$ [18]. As with Figure 3, the selection set S contains the progress of the algorithm.

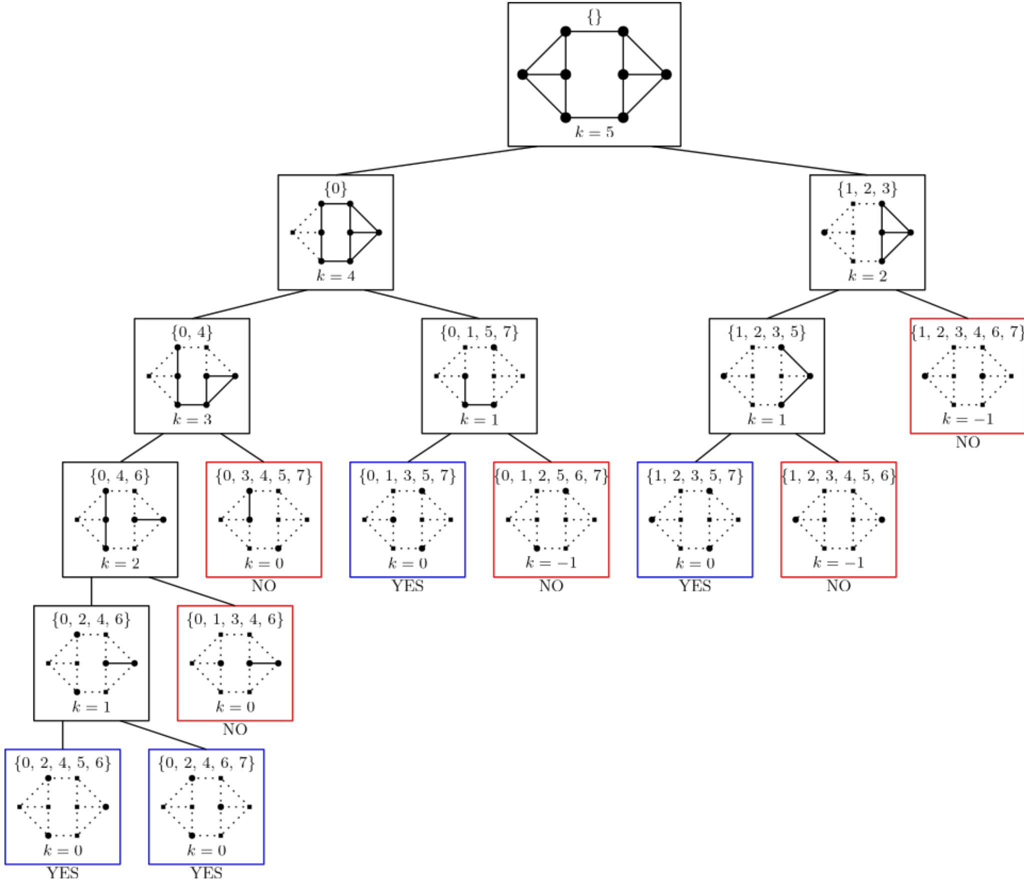


Figure 3: Branching algorithm for Vertex Cover

The leaves of this tree can be encoded as binary addresses and reconstructed into the desired subgraphs in parallel using an ordered list of edge selections. This approach avoids the sequential processing of the k-depth tree: exactly what we wish to achieve with our BPP-FPT algorithm.

The three steps to constructing a BPP-FPT algorithm are:

1. Encode the shape of the BST into an ordered list.
2. Encode the location of each leaf into a binary address.
3. Construct subproblems in parallel from the shape and location data.

3.3 Frontier Set Generation

The first step of building a BPP-FPT algorithm is to build an ordered list of operations which will represent the overall shape of the search tree. We do this by constructing a frontier set F. The intuition for such a set comes from observing the differences in tree structure from Figures 3 and 4. In Figure 3, the branching choice at each node is dependent on the unique graph induced by its parent nodes, whereas in figure 4, the choices at each level are the same. The purpose of the frontier set is to find a decision ordering in the vertices or edges of G such that the nodes at depth k are unique.

For vertex cover, we want to minimize duplicated work by selecting vertices which are not covered by previous selections. One such vertex ordering is given by FRONTIER-SET-VC (see Algorithm 1). We step upward through the vertices in V, skipping the first neighbor of each, as shown in Figure 5. The resulting set contains the first vertex of each edge in Figure 4.

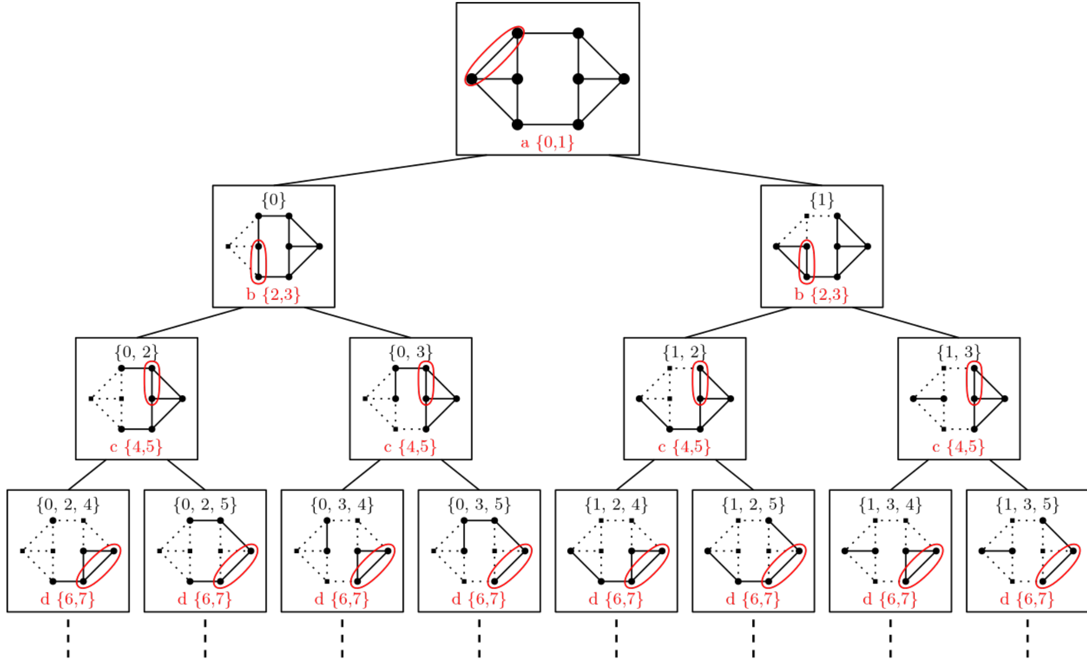
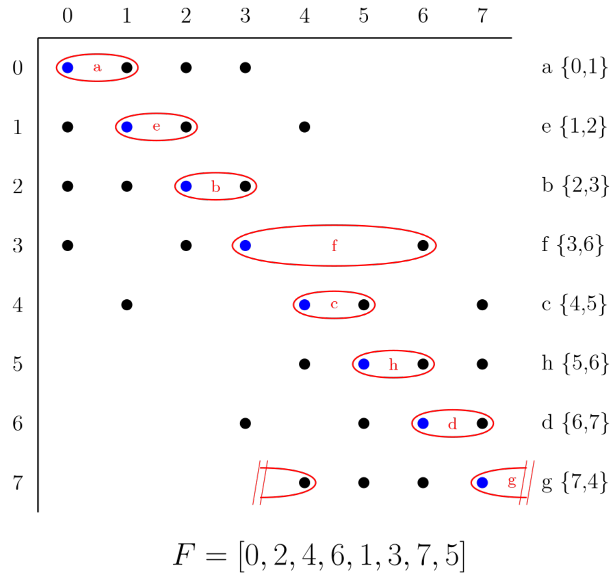


Figure 4: Edge endpoint removal for Vertex Cover



$$F = [0, 2, 4, 6, 1, 3, 7, 5]$$

Figure 5: Frontier set selection for Vertex Cover

Algorithm 1 Generate VC Frontier Set $\mathcal{O}(n)$ | sequential

Input: graph $G = (V, E)$
Output: vertex list F of size $|V|$

1: **function** FRONTIER-SET-VC(G) $\longrightarrow F$
2: $F \leftarrow [\,]$ ▷ *empty ordered list*
3: **while** $|F| < |V|$ **do**
4: $v_1 \leftarrow$ first $v \in V \setminus F$ ▷ *cycle through V from last v_2*
5: $v_2 \leftarrow$ first $v \in N(v_1)$ ▷ *cycle through $N(v_1)$ from v_1*
6: $F \leftarrow F + v_1$ ▷ *append v_1 to F*
7: **return** F

3.4 Process Count Identifiers

As defined in Section 2, *Terminology*, we refer to the total number of processes as p, with each process having its own unique identifier p_i. In the simplest case, the values of p_i are simply integer values ranging from 0 to $p-1$.

Because each identifier must represent a path traversed through the BST, we will instead read each value as a bitstring $p_i \in \{0,1\}^{\log_2 p}$. Thus, in this first example, each value of p_i is a bitstring of length $\log_2 p = \log_2(2^k) = k$. Later algorithms will use different encoding schema and lengths for values of p_i, but will follow this method of encoding the process identifier as a tree traversal path.

3.5 Parallel Construction

Now that we have a decision tree defined by F and a set of leaf node addresses denoted p_i for $i \in p$ we can examine each leaf at depth k without sequentially searching the tree. This final step, performed in parallel, allows each process to construct an endpoint of the decision tree and determine whether it is a YES or NO instance. In this paper we will assume that each process knows its own unique identifier, as this assumption simplifies the theoretical runtimes. Practical implementations of these techniques are discussed in Section 6.

Our BPP-FPT algorithm is given by VERTEX-COVER-A (see Algorithm 2). After constructing the frontier set F (line 2), we pass the decision procedure to each of the 2^k processes (lines 3–7). Following the logic of the BST algorithm for VC, the algorithm adds to the selection set S all vertices from F where the corresponding bit in p_i is 1 (line 4). When a bit in p_i is instead 0, the neighbors of that vertex in F are added. The resulting set S now contains k vertices where $p_i = p-1$ and at most n vertices where $p_i = 0$.

Removing all vertices in S from G (line 5) and checking whether any edges remain (line 6) will determine a vertex cover exists on k vertices. If any process contains a YES instance, the algorithm will return YES; otherwise, it returns NO.

Algorithm 2 Vertex Cover — BPP-FPT (A) $\mathcal{O}(n+m)$ | 2^k processes

Input: graph $G = (V, E)$, parameter k
Output: YES or NO

```
1: function VERTEX-COVER-A(G, k) ⟶ YES / NO
2:     F ← FRONTIER-SET-VC(G)                     ▷ generate frontier list
3:     for p_i ∈ {0,1}^k do                       ▷ pass (G, k, p_i, F) to 2^k processes
4:         S ← { F[x]      where the x^th bit of p_i is 1
                { N(F[x])   where the x^th bit of p_i is 0
5:         G' ← G \ S_{≤k}                        ▷ remove first k vertices in S from G
6:         if E(G') = ∅ then                      ▷ vertex cover exists if no edges remain
7:             return YES
8:     return NO
```

VERTEX-COVER-A runs in $\mathcal{O}(n+m)$ time on 2^k processes:

- Generating the frontier set is done in $\mathcal{O}(n)$ time.
- Each assignment to S touches at most $m = (k \cdot n)$ vertices.
- Removing edges incident to S from G is done in $\mathcal{O}(m)$ time.

Theorem 1. BST $\subseteq$ BPP-FPT

Proof. Following the steps in Section 3.2, a BPP-FPT algorithm can be constructed for any problem in FPT for which a BST algorithm exists:

- The traversal order of a bounded search tree is encoded as a frontier set F.
- The location of each leaf node is given by a bitstring p_i.
- Each of the $f(k)$ leaf nodes constructs a unique subproblem using F and p_i.

Any regular tree structure with a depth of k can be restructured to perform $f(k) \cdot n^{\mathcal{O}(1)}$ work on $f(k)$ parallel processes in $n^{\mathcal{O}(1)}$ time. $\square$

The process of encoding a traversal order into the search tree (generating the frontier set F) imposes an $\mathcal{O}(n)$ overhead; this appears to be unavoidable for problems without some inherently ordered structure (see Section 5.1). Similarly, a BPP-FPT algorithm will always examine all p leaf nodes, even when a BST algorithm would explore fewer nodes (as in Figure 3). This *overprovisioning* problem is discussed in Section 5.2.

3.6 Subprocess General Form

Having shown the existence of the class BPP-FPT and demonstrated how a BPP-FPT algorithm can be constructed from an instance of a BST algorithm, we can finally define a general form for subprocess calls used by all BPP-FPT algorithms presented in this paper:

$$\forall i \in \{0, \ldots, p-1\} : (G, k, p_i, F) \rightarrow \{\text{YES,NO}\}$$

where $G = (V, E)$, k is the parameter, p_i is a process identifier, and F is the frontier set.

4 Structural Boundaries for BPP-FPT

We now turn our attention to other FPT problems and algorithmic techniques to determine what structures permit or preclude the creation of a BPP-FPT algorithm. This section presents an alternative BPP-FPT formulation for Vertex Cover using iterative compression (IC), algorithms for Feedback Vertex Set based on the BST and IC methods, and a color coding (CC) algorithm for Longest Path. Because producing a kernel is generally done sequentially, kernelization procedures and their runtimes will be highlighted where relevant.

4.1 Vertex Cover (IC)

Given an arbitrary ordering $(v_1, v_2, \ldots, v_n)$ for V, we denote G_j as the subgraph induced by the first j vertices. At each step, we add vertex v_j to the solution set S_j. For $|S_j| \leq k$, the solution set contains all vertices in G_j. When $|S_j| > k$, we must compress our solution to size at most k by calling the compression function VC-Compression(G_j, k, S_j).

VC-Compression checks all 2^{k+1} partitions of $S_j = (S', \overline{S'})$. For each partition, $S^* = S' \cup N(\overline{S'})$. If no edges remain in $\overline{S'}$ when S^* is removed and $|S^*| \leq k$, then S^* is returned. Otherwise, S^* is not a vertex cover of size at most k in G_j.

This algorithm runs sequentially in $\mathcal{O}^*(2^k)$ time, with the runtime dominated by the $\mathcal{O}(2^{k+1} \cdot n)$ compression function. Figure 6 shows one partitioning the 6th iteration of the IC algorithm.

As with the BST example, this IC algorithm provides a clear opportunity for parallelization to $f(k)$ processes. For each compression step, there are 2^{k+1} possible partitions of the $k+1$ vertices in S. This is equivalent to a binary search tree of depth $k+1$. Using the encoding

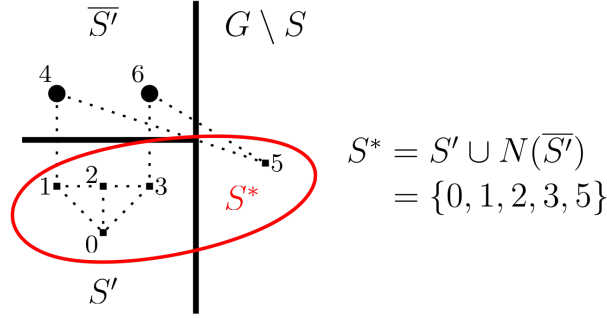


Figure 6: Compression step for Vertex Cover

for process identifiers described in Section 3.4, we can parallelize each compression step to 2^{k+1} processes, each with process identifier $p_i \in \{0,1\}^{k+1}$, and uniquely partition S using the set bits of p_i.

VERTEX-COVER-B (Algorithm 3) gives a BPP-FPT implementation for vertex cover using iterative compression. Generating a frontier set is unnecessary in this case: the solution set S_j performs the same function. As before, vertices from S are added to S^* where the corresponding bit of p_i is 1 (line 15); the neighbors of that vertex are added if the bit is 0.

In this algorithm, the subprocesses no longer perform a simple decision procedure, but must pass a new solution set S^* for use in the next iteration. For simplicity, we do this with an accumulator (lines 6, 8, 10); however, the precise implementation for this step will depend on the parallel memory model available. Viable solutions from each subprocess are appended to the accumulator A, and S is assigned an arbitrary solution if one exists. If S contains at most k vertices after the last iteration step, the algorithm returns YES; otherwise, NO.

Algorithm 3 Vertex Cover — BPP-FPT (B) $\quad \mathcal{O}(n+m) \mid 2^{k+1}$ processes

Input: graph $G = (V, E)$, parameter k
Output: YES or NO

```
1:  function VERTEX-COVER-B(G, k) ⟶ YES / NO
2:      S ← {}                                        ▷ empty unordered set
3:      for j ∈ {1, ..., n} do
4:          G′ ← G[V_{≤j}]                            ▷ graph induced by first j vertices of G
5:          S ← S ∪ {v_j}
6:          A ← ∅                                     ▷ empty accumulator set
7:          for p_i ∈ {0,1}^{k+1} do                  ▷ pass (G′, k, p_i, S) to 2^{k+1} processes
8:              A ← A ∪ VC-COMPRESSION(G′, k, p_i, S)  ▷ collate results
9:          if A ≠ ∅ then                             ▷ check if results were returned
10:             S ← A[0]                              ▷ select any valid solution S*
11:     if |S| ≤ k then
12:         return YES
13:     return NO

14: function VC-COMPRESSION(G′, k, p_i, S) ⟶ S*
15:     S* ← { S[x]       where the x^th bit of p_i is 1
               N(S[x])    where the x^th bit of p_i is 0
16:     G* ← G′ \ S*                                  ▷ remove S* from G_j
17:     if E(G*) = ∅ AND |S*| ≤ k then                ▷ vertex cover exists if no edges remain
18:         return S*
```

Vertex-Cover-B runs in $\mathcal{O}(n+m)$ time on 2^{k+1} processes:

- There are n iteration steps.
- Each assignment to S^* touches $\mathcal{O}^*(m)$ vertices.
- Removing edges incident to S^* from G' is done in $\mathcal{O}(m)$ time.

Theorem 2. $(*)$-Compression = BST

Proof. The equivalence of $(*)$-Compression and BST algorithms can be shown by simulation:

$(*)$-Compression $\subseteq$ BST — The 2^k possible partitions of k vertices can be decomposed into a binary tree of depth k where each layer represents the inclusion or exclusion of a vertex in partition S' of S. The tree contains 2^k leaf nodes, accessible via depth-first search.

BST $\subseteq$ $(*)$-Compression — A bounded search tree of depth k and branching factor δ can be folded into a discrete sequence where each iteration represents a step between nodes at a constant depth. There are δ^k such iterations at depth $|S'| = k$. □

Corollary 1. $(*)$-Compression $\subseteq$ BPP-FPT

Corollary 2. IC $\subseteq$ BPP-FPT

4.2 Feedback Vertex Set (BST)

With the *feedback vertex set problem* (FVS), we are given an undirected graph $G = (V, E)$ and a parameter k. We attempt to find a set $S \subseteq V$ of size at most k such that $G \setminus S$ is acyclic. There exists a BST algorithm for FVS which builds a tree of depth k and a branching factor of $3k$, resulting in a runtime of $(3k)^k \cdot n^{\mathcal{O}(1)}$. This algorithm orders the vertices in descending order by degree ($d(v_1) \leq d(v_2) \leq \cdots \leq d(v_n)$) and tests all k-sized subsets of the first $3k$ vertices (V_{3k}). This follows from an assertion that every FVS must contain at least one vertex from V_{3k} [16].

This algorithm again lends itself well to a BPP-FPT construction. If we take V_{3k} as our frontier set F, it is clear that we can construct each of the $(3k)^k$ subproblems using processes $p_i \in \{0, \ldots, (3k)^k\}$. However, Figure 7 shows how this approach produces many redundant (in red) or undersized (red and crossed) solution sets, even for small values of k.

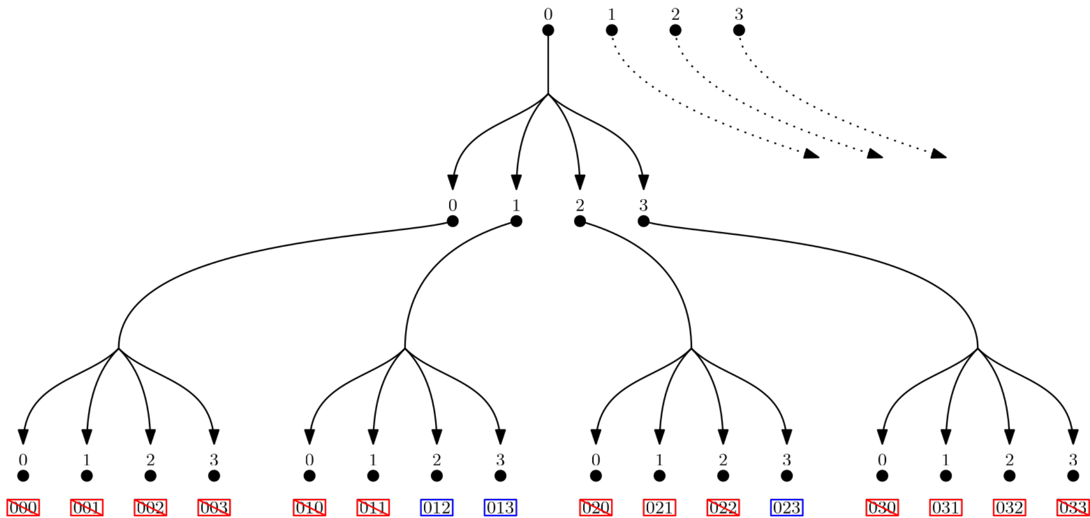


Figure 7: Overprovisioning in parallel FVS

To get an intuition for a better parallel encoding strategy, consider that we only need to test solutions where k of the $3k$ vertices are chosen in F. If each bit in p_i represents a vertex in V_{3k}, we need only test solutions where the Hamming Weight (the number of 1's) of p_i is equal to k. This reduces our search space from $(3k)^k$ to $\binom{3k}{k}$, massively reducing the overhead of our function, as shown by Table 1.

k	$(3k)^k$	$\binom{3k}{k}$
3	729	84
4	20736	495
5	759375	3003

Table 1: Growth of $(3k)^k$ and $\binom{3k}{k}$

The $p_i \in \{0,1\}^{3k}$ encoding strategy results in $8^k - \binom{3k}{k}$ processes where $\text{HW}(p_i) \neq k$. This overprovisioning can be removed by instead using $\binom{3k}{k}$ processes and letting each process p_i calculate the ith bitstring with Hamming Weight k. However, this adds $\mathcal{O}(2^k)$ to the subprocess runtime: such an algorithm would no longer be in BPP-FPT. Thus, we will use our original encoding, using a total of $2^{3k} = 8^k$ processes.

We start Algorithm 4 by sorting the vertices of G by their degree (in descending order) and assigning the first $3k$ sorted vertices to F. Each of the subprocesses calculates the Hamming Weight (population count) of p_i: if $\text{HW}(p_i) \neq k$, then process p_i returns nothing. As with previous algorithms, we select vertices from F for the solution set S where the corresponding bit of p_i is 1. Finally, all vertices in S are removed from G. If no cycles remain in the resulting graph, then a feedback vertex set exists on k vertices, and we return YES.

Algorithm 4 Feedback Vertex Set — BPP-FPT (A) $\mathcal{O}(n^2)$ | 8^k processes

Input: graph $G = (V, E)$, parameter k
Output: YES or NO

```
1:  function FEEDBACK-VERTEX-SET-A(G, k) ⟶ YES / NO
2:      F ← V_{≤3k} ordered by decreasing degree
3:      for p_i ∈ {0,1}^{3k} do                  ▷ pass (G, k, p_i, F) to 2^{3k} processes
4:          if HW(p_i) ≠ k then
5:              skip                             ▷ only consider cases where HW(p_i) = k
6:          S ← F[x] where x^{th} bit of p_i is 1
7:          G' ← G \ S                           ▷ remove S from G
8:          if G' is acyclic then                ▷ feedback vertex set exists no cycles remain
9:              return YES
10:     return NO
```

FEEDBACK-VERTEX-SET-A runs in $\mathcal{O}(n^2)$ time on 8^k processes:

- Assigning V_{3k} to F takes $\mathcal{O}(n^2)$.
- Checking $\text{HW}(p_i)$ takes $\mathcal{O}(1)$ on most hardware.
- Each assignment to S touches $\mathcal{O}(m)$ vertices.
- Removing S from G takes $\mathcal{O}(m)$.
- Detecting cycles in G' takes $\mathcal{O}(n)$.
- The total runtime is bounded by $\mathcal{O}(n^2 + m + n) = \mathcal{O}(n^2)$.

4.3 Feedback Vertex Set (IC)

An iterative compression algorithm for FVS begins with two assertions, given by Cygan et al. [16]:

1. If there exists an algorithm solving (∗)-Compression in time $f(k) \cdot n^c$, then there exists an algorithm solving problem (∗) in time $\mathcal{O}(f(k) \cdot n^{c+1})$.
2. If there exists an algorithm solving Disjoint-(∗) in time $\alpha^k \cdot n^{\mathcal{O}(1)}$, then there exists an algorithm solving (∗)-Compression in time $(1+\alpha)^k \cdot n^{\mathcal{O}(1)}$.

We have seen the effects of Statement 1 in Section 4.1, where the iteration step calls VC-Compression a total of n times. There exists an algorithm which solves Disjoint-FVS in $4^k \cdot n^{\mathcal{O}(1)}$ time. From Statement 2, we can use this algorithm for Disjoint-FVS to solve FVS in $5^k \cdot n^{\mathcal{O}(1)}$ time. Faster algorithms exist for Disjoint-FVS [16,18], but the branching algorithm described here is most convenient for the purposes of this paper. Theorem 3 follows from these assertions.

Theorem 3. Disjoint-(∗) $\subseteq$ (∗)-Compression

Corollary 3. Disjoint-(∗) $\subseteq$ BPP-FPT

The branching algorithm for Disjoint-FVS takes a graph G, parameter k, and feedback vertex set S of size $k+1$ and returns a set S^* of size $\leq k$ which is disjoint from S. The algorithm exhaustively applies the three reduction rules (given below), then branches on whether a vertex $V \in G \setminus S$ is added to the solution set S^* or disregarded (added to S).

The reduction rules each apply to any vertex $v \in G \setminus S$:

R1 — if $d(v) \leq 1$, remove v from G.

R2 — if $S \cup \{v\}$ is not acyclic, add v to S^* and remove it from G.

R3 — if $d(v) = 2$ and v has least one neighbor in $G \setminus S$, connect its neighbors and remove it from G.

Parallelizing Disjoint-FVS into a BPP-FPT algorithm once again depends on encoding used for p_i and S. As with Vertex-Cover-B, S receives a new vertex for each iterative step, and S_j is equivalent to a frontier set for iteration j. The reduction steps ensure that each operation on p_i is performed sequentially, but the encoding of a decision tree onto p_i is otherwise unrestricted. This branching algorithm bounds the number of operations by 2^{2k+1}, so we can take this as our process count p. The values for p_i are then bitstrings of length $2k+1$.

The iterative step for Feedback-Vertex-Set-B (Algorithm 5) functions identically to that of Vertex-Cover-B. The Disjoint-FVS subprocess is passed to 2^{2k+1} processes, which sequentially iterate through the $2k+1$ bits in p_i. In each iteration, we exhaustively apply the reduction steps in order. When the graph cannot be reduced further, we add the next vertex $v \in G' \setminus S$ to either the disjoint solution set S^* or to S. If $G' \setminus S^*$ is acyclic and $|S^*| \leq k$, then we return S^* to the accumulator and continue as with Vertex-Cover-B. If, after the final iteration, the solution set S contains at most k vertices, then we return that it is a YES instance. Otherwise, no such feedback vertex set exists.

Algorithm 5 Feedback Vertex Set — BPP-FPT (B) $\mathcal{O}(n^2m)$ | $4k$ processes

Input: graph $G = (V, E)$, parameter k
Output: YES or NO

```
1:  function Feedback-Vertex-Set-B(G, k) ⟶ YES / NO
2:      S ← {}                                          ▷ empty unordered set
3:      for j ∈ {1, ..., n} do
4:          G′ ← G[V_{≤j}]                              ▷ graph induced by first j vertices of G
5:          S ← S ∪ {v_j}
6:          A ← ∅                                       ▷ empty accumulator set
7:          for p_i ∈ {0,1}^{2k+1} do                   ▷ pass (G, k, p_i, S) to 2^{2k+1} processes
8:              A ← A ∪ Disjoint-FVS(G′, k, p_i, S)     ▷ collate results
9:          if A ≠ ∅ then                               ▷ check if results were returned
10:             S ← A[0]                                ▷ select any valid solution S*
11:     if |S| ≤ k then
12:         return YES
13:     return NO

14: function Disjoint-FVS(G′, k, p_i, S) ⟶ S*
15:     for bit x ∈ p_i do
16:         if S is not acyclic OR |S*| > k then
17:             skip                                    ▷ branch p_i is not viable
18:         repeat
19:             v ← first v ∈ G′ \ S
20:             if d(v) ≤ 1 then                        ▷ reduction R1
21:                 G′ ← G′ \ {v}
22:             else if S ∪ {v} is not acyclic then     ▷ reduction R2
23:                 S* ← S* ∪ {v}
24:                 G′ ← G′ \ {v}
25:             else if d(v) = 2 AND for {u, w} = N(v), u ∉ S OR w ∉ S then   ▷ reduction R3
26:                 E ← edge {u, w}
27:                 G′ ← G′ \ {v}
28:         until no reductions can be applied
29:         v ← first v ∈ G′ \ S
30:         if x = 1 then                               ▷ x^{th} bit of p_i is 1
31:             S* ← S* ∪ {v}
32:             G′ ← G′ \ {v}
33:         else                                        ▷ x^{th} bit of p_i is 0
34:             S ← S ∪ {v}
35:         if G′ \ S* is acyclic AND |S*| ≤ k then     ▷ fvs exists if no cycles remain
36:             return S*
```

Feedback-Vertex-Set-A runs in $\mathcal{O}(n^2m)$ time on 2^{2k+1} processes:

- There are n iteration steps.
- Performing the reduction steps takes $\mathcal{O}(n^2)$.
- Removing $\{v\}$ from G' takes $\mathcal{O}(n)$.
- Detecting cycles in G' takes $\mathcal{O}(n)$.
- Disjoint-FVS performs k iterations, bounding the runtime by $k \cdot \mathcal{O}(n^2) = \mathcal{O}(nm)$.

4.4 Longest Path (CC)

In the *longest path problem* (LONGPATH), we are given a graph $G = (V, E)$ and a parameter k. Our goal is to find a simple path of length k or determine that one does not exist. There exists a randomized algorithm which uses color coding and dynamic programming to solve the problem.

We first define a graph coloring, denoted by χ, where the vertices are colored uniformly and at random from a set of k colors. Of the possible k^n colorings, the probability that a subset $S \in V$ of size k contains only pairwise distinct colors is e^{-k} [16].

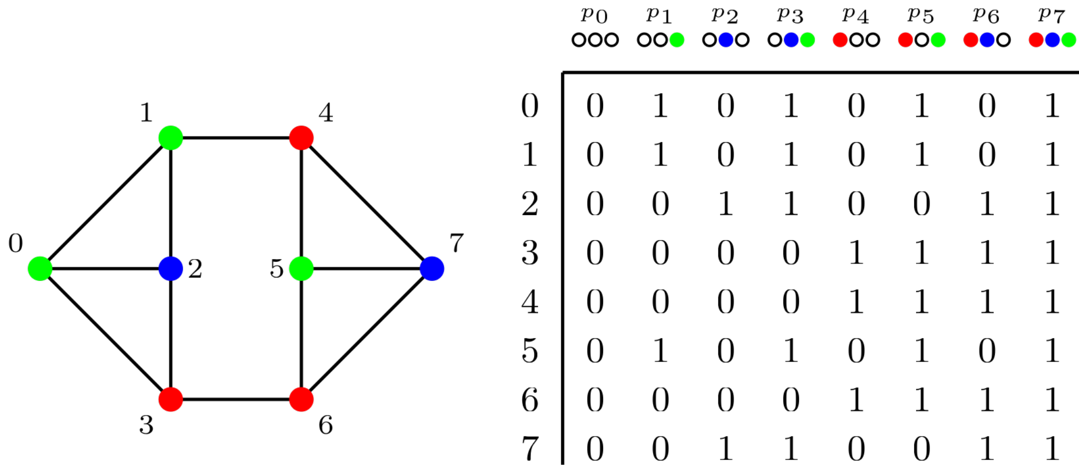
To find whether a k-path exists within χ, we build a dynamic programming table for a subset $S \in \{1, \ldots, k\}$ and each vertex $v \in V$. If $|S| = 1$ and $S = \{\chi(v)\}$, then $\text{PATH}(S, v) = \text{TRUE}$. Otherwise, we fill the table with the following recurrence:

$$\text{PATH}(S, v) = \begin{cases} \vee\{\text{PATH}(S \setminus \{\chi(v)\}, u) : \{v, u\} \in E\} & \text{if } \chi(v) \in S \\ \text{FALSE} & \text{otherwise} \end{cases}$$

A colorful k-path exists if $\text{PATH}(S = [k], v) = \text{TRUE}$ for some vertex $v \in V$. This DP method takes time $2^k \cdot n^{\mathcal{O}(1)}$. If we repeat this method e^k times, each with a new random coloring χ, then we will find a k-path in G with constant probability. This algorithm has an overall runtime of $(2e)^k \cdot n^{\mathcal{O}(1)}$.

The BPP-FPT algorithm Longest-Path-A (Algorithm 6) will use two layers of parallelization: one for the e^k random colorings of V, and one for the 2^k possible subsets of the k colors. The first $\lceil \ln k \rceil$ bits of the process identifier (denoting the coloring count) are ignored, thus we use only k bits for the values of p_i. This algorithm also relies on a parallel-safe implementation of the dynamic programming PATH table. This can be accomplished using a lock-free concurrent hash table [27], DAG optimization [17], or entry sequencing [17, 23, 27]. For simplicity, we will assume that PATH supports concurrent read and write operations.

Longest-Path-A begins by generating a k-coloring F of the vertices V. PATH is updated in parallel for each subset S of the colors in F. We can visualize the possible subsets of S as a BST of depth k where each level represents the inclusion or exclusion of one of the colors from S. Figure 8 shows one such coloring and the resulting table generated by $p = 2^k$ processes iterating through the vertices of V. If the value for $\text{PATH}(S, v)$ is 1 for any vertex $v \in V$ where $|S| = k$, we have found a YES instance. If no k-path has been found for any of the e^k random colorings of F, then we conclude that no such path exists.



	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7
0	0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0	1
2	0	0	1	1	0	0	1	1
3	0	0	0	0	1	1	1	1
4	0	0	0	0	1	1	1	1
5	0	1	0	1	0	1	0	1
6	0	0	0	0	1	1	1	1
7	0	0	1	1	0	0	1	1

Figure 8: PATH table for coloring $F = \{2, 2, 1, 0, 0, 2, 0, 1\}$

Algorithm 6 Longest Path — BPP-FPT (A) $\mathcal{O}(nm) \mid (2e)^k$ processes

Input: graph $G = (V, E)$, parameter k
Output: YES or NO

```
1: function LONGEST-PATH-A(G, k) ⟶ YES / NO
2:     for j ∈ {0, ..., e^k} do                         ▷ pass (G, k) to e^k processes
3:         F ← coloring F : V → [k]                      ▷ k-coloring of V
4:         PATH ← [ ][ ]                                 ▷ empty Boolean DP table
5:         for p_i ∈ {0,1}^k do                          ▷ pass (G, PATH, p_i, F) to 2^k processes
6:             COLORFUL-PATH(G, PATH, p_i, F)
7:         if PATH([k], v) = 1 for any v ∈ V then        ▷ k-path exists, ending at v
8:             return YES
9:     return NO

10: function COLORFUL-PATH(G, PATH, p_i, F)
11:     S ← {color x} where the x^th bit of p_i is 1
12:     for v ∈ V do
13:         if |S| = 1 AND S = {F[v]} then               ▷ F[v] is only color in S
14:             PATH(S, v) = 1
15:         else if F[v] ∈ S then                         ▷ color F[v] is in S
16:             PATH(S, v) = ∨{PATH(S \ {F[v]}, u) : ∀u ∈ N(v)}
17:         else                                          ▷ color F[v] is not in S
18:             PATH(S, v) = 0
```

Longest-Path-A runs in $\mathcal{O}(nm)$ time on $(2e)^k$ processes:

- Generating a random coloring F takes $\mathcal{O}(n)$.
- Colorful-Path performs n iterations.
- Building PATH touches at most $m = k \cdot n$ entries.
- Checking PATH$([k], V)$ for a colorful path takes $\mathcal{O}(n)$.

Colorful-Path tests each of the 2^k possible color subsets of F. We can equivalently say that it tests all *partitions* of k colors. Theorem 2 tells us that a partitioning of k vertices can be restructured as a BST of depth k. It follows that Colorful-Path is an instance of a BST algorithm.

Corollary 4. Colorful-Path = BST

From this, we can construct a proof for inclusion of some randomized algorithms in BPP-FPT:

Theorem 4. *For a randomized FPT algorithm A which performs $f(k)$ repetitions of sub-problem B, there exists a BPP-FPT algorithm for A if $B \subseteq$ BPP-FPT.*

Proof. A BPP-FPT algorithm for B will take $n^{\mathcal{O}(1)}$ time on $g(k)$ processes. All repetitions of B can be run in parallel on $f(k) \cdot g(k)$ processes with minimal overhead. Thus, if $B \subseteq$ BPP-FPT, $A \subseteq$ BPP-FPT. □

Corollary 5. CC $\subseteq$ BPP-FPT

5 Work and Process Bounds

The BPP-FPT algorithms presented in Sections 3 and 4 all suffer from unavoidable runtime and processing limitations. These limitations fall into two general categories: overhead (additional runtime components) and overprovisioning (additional processing load). This section explores the causes of these limitations and the effects they have on work and process bounds.

5.1 Overhead

Overhead is an additional work component which is imposed on an algorithm during restructuring or parallelization. Overhead is applied any time a graph is restructured (as in kernelization, tree decomposition, or vertex/edge ordering), and is often an unavoidable cost associated with simplifying a problem. Overhead reduction techniques are a common area of study, and are discussed in Section 7.1.

In general, there are three sources of overhead for a BPP-FPT algorithm:

- W_F — *Frontier Set Generation* (Section 3.3) imposes an $\mathcal{O}(n)$ work overhead for any BPP-FPT algorithm constructed using the steps from this paper. These operations are generally sequential, and do not benefit from additional parallelization. The overhead for IC algorithms include a constant-time component, which is included in the $\mathcal{O}(n)$ iteration overhead.
- W_C — *Parallel Construction* (Section 3.5) adds an additional $\mathcal{O}(k)$ time to any operation performed using p_i and F. In effect, the runtimes of subproblem reconstruction operations increase from $\mathcal{O}(n)$ to $\mathcal{O}(m)$.
- W_P — *Process Communication* adds a polylogarithmic runtime factor to most parallel algorithms, depending on the implementation methods and hardware. In this paper, process communication time is assumed to be negligible. The question of whether practical BPP-FPT algorithms exist for different parallel models is discussed in Section 7.1.

The total overhead for a BPP-FPT algorithm is given by:

$$W_{\uparrow} = W_F + (p \cdot W_C) + W_P$$

5.2 Overprovisioning

Overprovisioning is when a parallel algorithm allocates more than the optimal number of processes for a given problem. This problem extends beyond BPP-FPT algorithms, and can be seen in cases where a greedy algorithm overzealously generates subprocesses or where care is not taken when parallelizing dynamic programming algorithms [3]. This is distinct from under-utilization, where too few processes are provisioned, causing bottlenecks and performance penalties.

The overprovisioned process count is the difference between the BPP-FPT process count and the optimal number of subprocess steps performed by a sequential FPT algorithm ($\Delta p = p - p_{\text{opt}}$).

For example, the sequential BST algorithm for vertex cover (Figure 3) performs 17 steps. If we assume this to be the optimal number of processes, then Vertex-Cover-A (Algorithm 2) overprovisions $\Delta p = 15$ processes.

To better understand the impact of overprovisioning in a BPP-FPT algorithm, we can use a scalar value to show the percentage of work an algorithm performs beyond that of an optimal sequential algorithm. Continuing the previous example, Vertex-Cover-A performs 46.875% more work than the "optimal" algorithm (not including overhead).

The overprovisioned work is given by the ratio of Δp to p:

$$p_{\uparrow} = \frac{\Delta p}{p} \in [0, 1)$$

5.3 Lower Bounds

A parallel algorithm can never perform less work than the best sequential algorithm [21], and the parallel runtime reduction is generally offset by the overhead of splitting and provisioning a problem (see Section 5.1). Given these assertions, we must then determine the lower work bounds for BPP-FPT: what is the smallest difference from the work of a sequential algorithm that a BPP-FPT algorithm can achieve?

Sections 5.1 and 5.2 show the additional work accrued from runtime and process components. If we define W_{FPT} as the basis work done by a sequential FPT algorithm, we can then calculate the total work done by a corresponding BPP-FPT algorithm. The runtime overhead is an unavoidable workload made up of individual (W_F) and per-process (W_R, W_P) parts, which we can combine and denote as $W_{\text{overhead}} = W_{\uparrow}$. Overprovisioning is the amount of superoptimal work done by Δp additional processes, and is found by scaling the basis work by the overprovisioned amount: $W_{\text{overprovisioning}} = (p_{\uparrow} \cdot W_{\text{FPT}})$.

This gives us the following general formula for the work done by a BPP-FPT algorithm:

$$\begin{aligned} W_{\text{BPP-FPT}} &= W_{\text{FPT}} + W_{\text{overhead}} + W_{\text{overprovisioning}} \\ &= W_{\text{FPT}} + W_{\uparrow} \qquad\quad + (p_{\uparrow} \cdot W_{\text{FPT}}) \end{aligned}$$

For a BPP-FPT algorithm to be work-optimal (i.e., $W_{\text{BPP-FPT}}$ matches the lower bound of W_{FPT}), $W_{\uparrow}$ and $p_{\uparrow}$ must be zero. While there are cases where the overprovisioned work might be negligible (as with Feedback-Vertex-Set-B and Longest-Path-A under ideal conditions), the overhead for all algorithms presented in this paper is unavoidable. At minimum, an $f(k)$ time component is hidden in each subproblem (which increases the overall runtime from $\mathcal{O}(n)$ to $\mathcal{O}(m)$) or the selection of a frontier set adds an additional $\mathcal{O}(n)$ runtime to the sequential portion of the algorithm. Vertex-Cover-A highlights both of these work increases: the sequential VC algorithm performs $\mathcal{O}(2^k \cdot n)$ work, while the BPP-FPT algorithm performs $\mathcal{O}(n + (2^k \cdot m))$ work.

Unless there is a method to reduce $W_{\uparrow}$ or $p_{\uparrow}$ to zero, the minimum additional work for a BPP-FPT algorithm is $\mathcal{O}(n + (p \cdot k))$. Whether such methods exist is explored in Sections 6 and 7.1.

6 Work-Efficiency

An algorithm is *work-optimal* if its work matches that of the lower bound for the fastest sequential algorithm [7]. Section 5 shows that the work of a BPP-FPT algorithm cannot match that of a comparable sequential algorithm unless certain constraints are met; overhead and overprovisioning must be reduced to zero. This section explores which techniques may allow for work-efficient or work-optimal implementations of BPP-FPT algorithms.

6.1 Assumed Hardness

The exponential time hypothesis (ETH) states that 3CNFSat cannot be solved in less than $2^{o(n)}$ [18]. This implies the existence of the W hierarchy, where FPT $\subsetneq W[1]$. More specific to this paper, no FPT algorithm exists which can solve *vertex cover* in $\mathcal{O}^*((1+\epsilon)^k)$ unless ETH is false [18, 24]. Thus, there is a value ϵ which bounds the runtime of FPT algorithms.

Because BPP-FPT $\subseteq$ FPT, this lower bound applies to any BPP-FPT algorithm constructed using *bounded search tree* (Theorem 1), *iterative compression* (Corollary 2), or *color coding* (Corollary 5) techniques. It is unlikely that any BPP-FPT algorithm can perform less than $2^{o(k)}$ work, even if constructed using techniques not discussed in this paper.

6.2 Optimal Provisioning

As discussed in Section 5.2, the BPP-FPT algorithms presented in this paper assign a superoptimal number of processes. However, there is potential for optimizing provisioning in BPP-FPT algorithms. Section 4.2 (and Figure 7 in particular) highlight areas where a well-constructed BPP-FPT algorithm may require less branching than a sequential FPT algorithm.

The BST algorithm for *feedback vertex set* branches into $(3k)^k$ subproblems, whereas an optimal BPP-FPT algorithm requires only $\binom{3k}{k}$ processes. For such an optimal provisioning to be in BPP-FPT (i.e., the runtime is independent of k), the algorithm would need to send the ith bitstring with Hamming Weight k to each process p_i. This can be done by:

1. Building a lookup table for the $\binom{3k}{k}$ bitstrings with Hamming Weight k in $\mathcal{O}(n)$ time.
2. Allowing each process p_i to read entry i from the lookup table.

The concurrent read (CREW/CRCW) version of Feedback-Vertex-Set-A requires $\mathcal{O}(n^2)$ time on $\binom{3k}{k}$ processes: significantly less work than the sequential algorithm.

Parallel decomposition strategies [3] may allow for more efficient parallelization of some problems. Proactive decomposition could prune subproblem branches with a low likelihood of finding a solution, such as where Vertex-Cover-A would add many neighborhoods to the selection set on processes where p_i approaches p. Dynamic decomposition may allow for better process count utilization for IC algorithms where some partitionings of G' consistently give better solutions than others. Both of these scenarios require some amount of heuristic approximation or additional processing and may only work in a narrow set of cases.

6.3 Kernelization

This paper has previously ignored kernelization steps for FPT algorithms. However, many of the fastest FPT algorithms rely on graph reductions or decompositions to exploit certain structures or attributes of the input graph. For example, the $\mathcal{O}(2^k \cdot n)$ sequential VC algorithm given in Section 3.2 accepts a kernel with $\mathcal{O}(k^2)$ vertices and edges, drastically reducing the search space. There are algorithms that can solve vertex cover in less than $\mathcal{O}(n\sqrt{m} + 1.4656^k k^{\mathcal{O}1})$ time by combining kernelization and more complex branching rules [16].

Kernelization presents an opportunity to hide the W_F overhead costs of a BPP-FPT algorithm: an $\mathcal{O}(n)$-time frontier set generation will not meaningfully increase the runtime of a kernelization which runs sequentially in $\mathcal{O}(n\sqrt{m})$. From this, we can generalize that any BPP-FPT algorithm with a kernelization step with a sequential runtime of at least $\mathcal{O}(n)$ may trivially include the sequential overhead component (W_F) during kernelization.

Additional speedups may be gained from kernel parallelization. Though most kernelizations are inherently sequential, several methods exist to parallelize certain aspects of kernelization. Of particular note are the parallel interleaving techniques presented in [7] and parallel crown decompositions presented in [5]. Such parallel kernelizations are currently achieved using a number of processes that is polynomial in n or m, and therefore may be difficult to add into a fully BPP-FPT algorithm.

However, some interleaving techniques produce a partially-parallel kernel of depth $\mathcal{O}(\log k)$. There may exist BPP-FPT algorithms for maximum matching and crown decomposition using

fast subset convolution, given that perfect matchings can be counted in $\mathcal{O}(k^2 \cdot 2^k)$ on graphs with treewidth k [28]. Either of these techniques may result in a BPP-FPT kernelization.

7 Conclusion

This paper introduces Bounded-Process Parallel FPT (BPP-FPT) as a new class of parallel fixed-parameter algorithms, characterized by a runtime polynomial in the input size and a process count bounded by a function of the parameter. BPP-FPT fills a gap in existing models of parameterized parallel complexity and offers a practical framework for solving parallel FPT problems under realistic resource constraints.

This class is demonstrated through BPP-FPT algorithms for the Vertex Cover, Feedback Vertex Set, and Longest Path problems. We show that any FPT problem solvable by a bounded search tree (BST) of depth $f(k)$ can be parallelized into a BPP-FPT algorithm. This result generalizes to other FPT algorithmic methods, including iterative compression (IC) and color coding (CC). An analysis of work and process bounds reveals fundamental challenges to work-optimality in BPP-FPT algorithms, particularly the costs associated with subproblem reconstruction and process overprovisioning. We discuss strategies to improve algorithm performance, such as efficient encoding, memory model optimization, and kernelization techniques.

This work provides both a theoretical foundation and a practical toolkit for developing parallel FPT algorithms when the number of available processes is small relative to the input size. It establishes a basis for future research on refining the parameterized parallel complexity hierarchies and developing work-efficient BPP-FPT algorithms, aiming to bridge the gap between theoretical parallelism and practical, resource-constrained computation.

7.1 Further Work

Compression Dependence: The parallel construction overhead (W_C) appears to depend on the compression of a BST search path into $\log_2(p)$ bits. This adds an additional $p = \mathcal{O}(k)$ runtime component to each subprocess step, in effect increasing subprocess runtimes from $\mathcal{O}(n)$ to $\mathcal{O}(m)$ (see Section 5.1). Several methods for removing this overhead are presented by Elberfeld et al. (2012) [19], including preprocessing (kernelization), advice parameterization, para-L reductions, and graph contractions. Optimal BPP-FPT algorithms may require that W_C be reduced to a constant runtime factor or removed entirely.

Overhead Reduction: Frontier set generation adds a sequential overhead component (W_F) to the runtime of BPP-FPT algorithms as presented in this paper. Section 6.3 discusses how W_F may be hidden by combining it with kernelization, but other research shows that this runtime factor can be further reduced or removed. Parallel kernelization [11] may reduce W_F to a logarithmic factor, while k-perfect hashing [9] may reduce it to a constant factor. How these methods apply to BPP-FPT algorithms, and under what restrictions, is an area of interest.

Process Communication: The added overhead of provisioning subprocesses, parallel memory access, and interprocess communication typically imposes a logarithmic runtime factor to parallel algorithms [14, 21], as shown in Section 1.2 (*Parameterized Parallel Subclasses of FPT*). This paper refers to this *process communication overhead* as W_P and has previously disregarded its impact.

Graphics processing units (GPUs) allow for identifier-based parallel algorithms as presented in this paper ($W_P \approx \mathcal{O}(1)$). Further, optimized parallel data structures may reduce the parallel construction cost W_C to a constant factor for sparse graphs. Hwu et

al. (2023) [23] explain how both of these methods can be employed. Specific hardware implementations for BPP-FPT algorithms will show whether FPT problems can be feasibly solved without a parameter-dependent runtime.

Non-Branching Membership: This paper has only explored BPP-FPT algorithms which exploit branching to parallelize a problem. Elberfield et al. (2012) [19] state that consideration of parallel and circuit complexity classes might inform solutions to parameterized problems which are not slicewise P-complete. As BPP-FPT shares structural similarities with both FPPT and para-$AC^{0\uparrow}$, it is likely that new algorithmic techniques for either of these classes could improve or refine BPP-FPT.

Evaluating FPT problems by space complexity [19] and layered iteration [8] have already informed the construction of BPP-FPT algorithms based on BST and IC methods. Further investigations into the relationship between algorithmic structure and parameterized parallel complexity are needed. Of particular interest is whether BPP-FPT algorithms exist for problems without an inherent branching structure.

Advice Parameterization: This paper only explores problems with simple inputs. However, input restructuring and advice parameterization (treewidth, branchwidth, modular-width, etc.) can lead to drastic runtime improvements for many problems [2,12,13,22]. Whether BPP-FPT algorithms exist for problems parameterized by a structural parameter is unknown.

References

[1] Faisal N. Abu-Khzam, Michael R. Fellows, Michael A. Langston, and W. Henry Suters. Crown structures for vertex cover kernelization. *Theory of Computing Systems*, 41(3):411–430, 2007.

[2] Faisal N. Abu-Khzam and Karam Al Kontar. A brief survey of fixed-parameter parallelism. *Algorithms*, 13(8):197, 2020.

[3] Faisal N. Abu-Khzam, Michael A. Langston, Pushkar Shanbhag, and Christopher T. Symons. Scalable parallel algorithms for FPT problems. *Algorithmica*, 45(3):269–284, 2006.

[4] Faisal N. Abu-Khzam, Shouwei Li, Christine Markarian, Friedhelm Meyer auf der Heide, and Pavel Podlipyan. On the parameterized parallel complexity and the vertex cover problem. In *Proceedings of the 10th International Conference on Combinatorial Optimization and Applications (COCOA 2016)*, volume 10043 of *Lecture Notes in Computer Science*, pages 477–488, Hong Kong, China, December 2016. Springer.

[5] Faisal N. Abu-Khzam, Shouwei Li, Christine Markarian, Friedhelm Meyer auf der Heide, and Pavel Podlipyan. Efficient parallel algorithms for parameterized problems. *Theoretical Computer Science*, 786:2–12, 2019.

[6] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[7] Max Bannach, Malte Skambath, and Till Tantau. Towards work-efficient parallel parameterized algorithms. In *Proceedings of the 13th International Conference and Workshop on Algorithms and Computation (WALCOM 2019)*, volume 11355 of *Lecture Notes in Computer Science*, pages 341–353, Guwahati, India, February 2019. Springer.

[8] Max Bannach, Malte Skambath, and Till Tantau. On the parallel parameterized complexity of MaxSAT variants. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*, volume 236 of *LIPIcs*, pages 19:1–19:19, Haifa, Israel, August 2022. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[9] Max Bannach, Christoph Stockhusen, and Till Tantau. Fast parallel fixed-parameter algorithms via color coding. In *Proceedings of the 10th International Symposium on Parameterized and Exact Computation (IPEC 2015)*, volume 43 of *LIPIcs*, pages 224–235, Patras, Greece, September 2015. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[10] Max Bannach and Till Tantau. Parallel multivariate meta-theorems. In *Proceedings of the 11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*, volume 63 of *LIPIcs*, pages 4:1–4:17, Aarhus, Denmark, August 2016. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[11] Max Bannach and Till Tantau. Computing kernels in parallel: Lower and upper bounds. In *Proceedings of the 13th International Symposium on Parameterized and Exact Computation (IPEC 2018)*, volume 115 of *LIPIcs*, pages 13:1–13:14, Helsinki, Finland, August 2018. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[12] Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1-2):1–45, 1998.

[13] Hans L. Bodlaender and Torben Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. In *Proceedings of the 22nd International Colloquium on Automata,*

Languages and Programming (ICALP95), volume 944 of *Lecture Notes in Computer Science*, pages 268–279, Szeged, Hungary, July 1995. Springer.

[14] Marco Cesati and Miriam Di Ianni. Parameterized parallel complexity. In *Proceedings of the 4th International Euro-Par Conference on Parallel Processing (Euro-Par '98)*, volume 1470 of *Lecture Notes in Computer Science*, pages 892–896, Southampton, UK, September 1998. Springer.

[15] James Cheetham, Frank K. H. A. Dehne, Andrew Rau-Chaplin, Ulrike Stege, and Peter J. Taillon. Solving large FPT problems on coarse-grained parallel machines. *Journal of Computer and System Sciences*, 67(4):691–706, 2003.

[16] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

[17] Xiangyun Ding, Yan Gu, and Yihan Sun. Parallel and (nearly) work-efficient dynamic programming. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24)*, SPAA '24, page 219–232, Nantes, France, June 2024. ACM.

[18] Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013.

[19] Michael Elberfeld, Christoph Stockhusen, and Till Tantau. On the space complexity of parameterized problems. In *Proceedings of the 7th International Symposium on Parameterized and Exact Computation (IPEC 2012)*, volume 7535 of *Lecture Notes in Computer Science*, pages 206–217, Ljubljana, Slevenia, September 2012. Springer.

[20] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[21] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, STOC '78, pages 114–118, San Diego, USA, May 1978. ACM.

[22] Jakub Gajarský, Michael Lampis, and Sebastian Ordyniak. Parameterized algorithms for modular-width. In *Proceedings of the 8th International Symposium on Parameterized and Exact Computation (IPEC 2013)*, volume 8246 of *Lecture Notes in Computer Science*, pages 163–176, Sophia Antipolis, France, September 2013. Springer.

[23] Wen-mei W. Hwu, David Blair Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors - A Hands-on Approach*. Morgan Kaufmann, 2023.

[24] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.

[25] Christos H. Papadimitriou. *Encyclopedia of Computer Science*, chapter Computational Complexity, page 260–265. John Wiley and Sons Ltd., 2003.

[26] Sebastian Siebertz. Parameterized distributed complexity theory: A logical approach. *ArXiv*, abs/1903.00505, 2021.

[27] Alex Stivala, Peter J. Stuckey, Maria Garcia de la Banda, Manuel Hermenegildo, and Anthony Wirth. Lock-free parallel dynamic programming. *Journal of Parallel and Distributed Computing*, 70(8):839–848, 2010.

[28] Johan M. M. van Rooij, Hans L. Bodlaender, and Peter Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In *Proceedings of*

the 17th Annual European Symposium on Algorithms (ESA 2009), volume 5757 of *Lecture Notes in Computer Science*, pages 566–577, Copenhagen, Denmark, September 2009. Springer.