Progress in Parallel Algorithms

by

Damian Tontici

SB in Electrical Engineering and Computer Science

Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2024

Authored by:	Damian Tontici Department of Electrical Engineering and Computer Science January 19, 2024
Certified by:	Jayson Lynch Research Scientist, MIT CSAIL, Thesis Supervisor
Certified by:	Neil Thompson Research Scientist, MIT CSAIL, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Progress in Parallel Algorithms

by

Damian Tontici

Submitted to the Department of Electrical Engineering and Computer Science on January 19, 2024, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Parallel computing offers the promise of increased performance over sequential computing, and parallel algorithms are one of its key components. There has been no aggregated or generalized comparative analysis of parallel algorithms. In this thesis, we investigate this field as a whole. We aim to understand the trends in algorithmic progress, improvement patterns, and the importance and interactions of various commonly used metrics. We collect parallel algorithms solving problems in our set and analyze them. We look at four major themes: how parallel algorithms have progressed, including in relationship to sequential algorithms and parallel hardware; how the work and span of algorithms influence performance; how problem size and available parallelism affect performance; and what researchers' observable priorities look like. We find that more problems have had parallel improvements than sequential ones since the '80s, that most parallel algorithms don't improve algorithmic complexities, and much more. This research is important for us to understand how the field of parallel algorithms has changed throughout time, and what it looks like now.

Thesis Supervisor: Jayson Lynch
Title: Research Scientist, MIT CSAIL

Thesis Supervisor: Neil Thompson
Title: Research Scientist, MIT CSAIL

Acknowledgments

I'd like to first and foremost thank my advisors, Jayson Lynch and Neil Thompson, for their guidance, support, feedback, and direction. An especially big thanks to Jayson for being available on short notice on a frequent basis whenever I had questions. And a huge thanks to Neil for his expertise regarding data analysis methodology which has proven invaluable.

I'd also like to give many thanks to Jeffery Li and Liva Olina, who've been gathering parallel algorithms alongside me. This project would not have been possible without them. On the same note, thank you to any other students and researchers who helped with data gathering.

Last but not least, I'd like to thank Rezaul Chowdhury and Charles Leiserson, who with their expertise helped me better understand the background and helped flesh out the methodology for this project. And along with TB Schardl and Julian Shun, for providing useful feedback, helping with tricky algorithms in data gathering, and catching my mistakes. I'm very thankful to all of them for this and for their general support.

A joint-authored paper based on the results of this thesis is in the works.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Algorithms are ubiquitous in the modern world, as they form the basis of the field of computer science. Scientists have always looked for ways to increase the speed of the algorithms we use — the idea is simple: the more efficient algorithms, the faster they take to complete. One way of doing that is **parallelization**, or making different parts of an algorithm run at the same time (in parallel) instead of one after another (sequentially).

There have been proposals for parallel algorithms even before the invention of the modern computer. In 1922, mathematician Lewis Fry Richardson suggested an algorithm for predicting the weather. [3] That could be done by solving a system of many differential equations, however, it would have taken one person too much time to compute a solution. Richardson proposed having multiple people work on smaller parts of those equations at the same time. While it was only a thought experiment at the time, it's now possible to implement it (with processors instead of people), because parallel computers have actually been built.

The idea of a parallel computer had been floated around for a while, but it was both very novel and very complex, which made people reluctant to invest in it. After a long development period, the first parallel computer, ILLIAC IV, was completed in 1972. [4] It had 64 processors, which was only a quarter of what the initial design was meant to have. ILLIAC IV was followed by other parallel machines in the 1980s, such as the connection machine series. [5, 6, 7]

Today, parallelism is ubiquitous, with commercially available personal computers having up to 64 cores. [2] And the biggest supercomputer in the world has a core count of almost 20 million! [8] At the same time, there have been complementary advances in algorithms on the theoretical side — for this project, we've collected hundreds of such parallel algorithms.

But just how useful are parallel algorithms? What's their impact? And what are their limits? In this thesis, we examine parallel algorithms in an attempt to answer these questions.

1.1 Prior Work

When we talk about analyzing trends in technology, the first thing that comes to mind is Moore's Law. [9] Gordon Moore predicted that the number of transistors in an integrated circuit would double (computer hardware will become twice as good) every year. That has been surprisingly accurate until the mid-2000s.

But computer performance depends on both the hardware and software, as well as the algorithms that run on them. To this point, Sherry and Thompson [10] have recently inquired into the improvements of sequential algorithms and their impact on computer technology progress. Their research uncovered interesting findings, such as "for moderate-sized problems, 30%–43% of algorithmic families had improvements comparable or greater than those that users experienced from Moore's Law and other hardware advances". Liu [11, 12] and Rome [13, 14] have further developed this line of research by investigating sequential algorithm lower bounds and space usage respectively.

In a paper reviewing predicting computer performance improvement after the end of Moore's law, Leiserson et al. mention parallelism as one of the crucial aspects needed for computer performance to increase. [15] Additionally, they argue that most of the future improvement will come from hardware architecture, software, and algorithms. Therefore, we're interested in examining one of these three areas — algorithms — as applied to parallelism.

1.2 Objectives and Thesis Organization

In this thesis, we extend this analysis to the realm of parallel algorithms. We're interested to see how quickly they've improved, how that improvement compares to sequential algorithms' progress, how different parts of the parallel computation impact the performance, and what the extent of such progress is.

Work done for this thesis will be incorporated in the Algorithm Wiki project[1].

This thesis is organized as follows. We start with some theoretical background in Chapter 2, which includes an in-depth look at parallel models of computation, as well as an organizational framework for them. In Chapter 3, we address our methodology, detailing our process and pointing out some limitations of our data. We show our first set of results in Chapter 4, where we compare sequential and parallel improvement and performance. We then take a look at the existing trade-offs between the span and work of algorithms and analyze their frequency and magnitude in Chapter 5. Chapter 6 examines how improvement depends on both problem size and available parallelism. Finally, Chapter 7 highlights some unexpected results regarding parallel algorithms.

[1] `algorithm-wiki.csail.mit.edu`

Chapter 2

Background

Research in parallel computing has been varied. Ultimately, some research directions are of more interest to us than others, and we shall focus on providing a broad background related to the concepts that are most relevant to this thesis.

We start with a note on the wide variety of what parallelism can refer to. We then give an overview of parallel models, including a description of the cost model and other metrics used to describe parallel performance in section 2.1. We then classify parallel models in section 2.2 and present a few models in more depth in section 2.3. A discussion on simulating models on each other follows in section 2.4. We conclude this chapter by briefly defining a few problems in computer science that are going to be used as examples throughout the rest of the thesis.

Types of Parallelism

Parallel computing can refer to a lot of things. We can parallelize various parts of a computation, and we can define parallelism based on those.

Bit-level parallelism involves performing operations on more than one bit at a time. This is hardware-dependent, and it was largely the first type of parallelism to be implemented — starting with 4-bit word lengths in the '70s, and reaching 32-bit words in the mid-'80s and 64-bit in 1996, which are the most widely used today. [16]

Instruction-level parallelism happens at the compiler and hardware level — a

processor can perform multiple instructions that use different units (after potentially reordering them).

Task parallelism is all about making different tasks run on different processes/ threads at the same time.

Data parallelism, a.k.a. **word parallelism**, implies manipulating more than one word or piece of data at the same time.

Algorithms have the biggest impact on task and data parallelism — since what tasks can be performed in parallel depends on the algorithm as well as the hardware, and what data you can operate in parallel as much on the algorithm you're using as the hardware you're running it on. Therefore, this thesis will focus mainly on these types of parallelism.

2.1 Parallel Models of Computation

We've seen that parallel computers can be very valuable in speeding up computation. However, parallel algorithms come with several additional complications. Sequential algorithms are being conceived without a specific computer in mind, and still, they can be adapted to any computer. To make it easier to reason about algorithms, computer scientists have abstracted the general computer to models. Although others (such as Turing machines) exist, by far the most popular one is the **Word Random Access Model**, or Word RAM for short. Models like this are very useful, as there are a certain set of basic operations that every algorithm can be broken down to and each of those has a predetermined cost (e.g. read a memory address, which takes constant time).

This becomes more complicated with parallel computing, as there is no one ubiquitous model, but rather a multitude are often used, and new ones are devised frequently. Parallel models need to balance simplicity with how well they would translate to real parallel computers. Certain things that are abstracted in Word RAM can't be as easily eliminated when working with bigger data sets and more complicated real machines, as we have to do with parallel computing. Some of the most notable issues

include communication delays and reading from memory. In addition, with multiple processors comes the issue of interactions between them, and there are various aspects to consider, e.g. what happens when multiple processors want to write to the same memory address at the same time? It turns out that there's a fine line between all these considerations, and no model is ideal on all axes.

These circumstances generated a lot of models, among the most popular ones being PRAM (similar to word RAM, but parallel), SIMD (Single-Instruction, Multiple-Data), MIMD (Multiple-Instruction, Multiple-Data), BSP (Bulk Synchronous Parallel). Most of these models have subdivisions of their own, to accommodate different answers to the considerations outlined above. In the sections that follow, we look closer into what makes each of these special and attempt to classify the parallel models.

Note that there are multiple levels of abstraction within any computation, and there is an important distinction to be made between models whose purpose is algorithm design (models of computation) and models that abstract execution (models of execution a.k.a. programming models). Here, we concern ourselves with theoretical models of computation only.

Work-Span Cost Model

There's the notion of a **work-span** model (also known as a work-depth model) [17], which defines the following two metrics: **work** — the total number of operations an algorithm performs, and **span** (also known as **depth**) — the longest chain of operations performed sequentially. These two measures can be defined for any model of computation. In that sense, all models of computation are work-span models and work-span is a **cost model**.

The span and work are the two possible ends of the algorithm's **runtime**.

For models that assume computation is done on a number of processors, another way of looking at work and span collectively is to define T_P — the time an algorithm takes to run using P processors (its runtime). In that case, T_1 is equivalent to work, and T_∞ is the span. For any algorithm, the work law $T_P \geq T_1/P$ and the span law

$T_P \geq T_\infty$ hold, because parallel algorithms can never reduce the total amount of work that needs to be done. [18]

Parallelism Metrics and other Definitions

Additional useful metrics can be defined based on work and span.

Parallelism is the ratio between work and span (T_1/T_∞ for processor-based models) and is a metric that tells us how much faster can an algorithm be made by parallelizing it.

Absolute (parallel) Speedup is the ratio of work to runtime (T_1/T_P).

This is not to be confused with **relative speedup** with respect to another (not necessarily parallel) algorithm, which is the ratio between the running time of the reference algorithm and the runtime.

Strong scaling analysis of an algorithm is done by measuring the performance (as speedup) when the problem size n is kept fixed, and the number of processors p is varied. **Weak scaling** is varying both n and p.

Work efficiency $T_{seq}/work$ is a metric comparing the work of a parallel algorithm to the time of the best-known sequential algorithm T_{seq}. If the ratio is $O(1)$, it means that they're asymptotically equal, in which case we call the parallel algorithm **work-efficient** (also known as **work-optimal** or **work-preserving**).

The inverse of work efficiency is **work overhead** — how much more work (multiplicatively) one has to do on top of the best sequential runtime. Algorithms with a constant overhead are work-efficient.

In the following sections, we'll show how models are classified, then we'll discuss some well-known models, and end on how we can simulate different types of models.

2.2 Models of Computation Classification

Throughout the years, multiple approaches to conceptualizing models have arisen. There are two main ones: in the first, **processor-based models**, we consider the

perspective of parallel machines and their processors. For the second one, we take a more theoretical perspective and look at **abstract models**.

A schematic summary of the above is presented in Table 2.1, and Table 2.2 exemplifies it by categorizing selected models.

Processor-based Models			**Abstract Models**
Instruction/Data			Language-based models
MISD	MIMD		
SISD	SIMD		
Communication			Vector models
Shared Memory	Distributed Shared Memory	Distributed Memory	
Synchrony			Circuit models
Lock-step	Bulk	Asynchrony	

Table 2.1: Classifying models of computation

Abstract Models

There are three main types of abstract models: **circuit**, **vector machine**, and **language-based**.

Circuit

In **circuit models**, the input is processed through **gates**, which are connected by **wires**. The wires carry information, which isn't stored anywhere else, so they could be viewed as the equivalent of "memory". Gates have one or more input wires and one or more output wires. Each gate performs an operation on the data carried through the input wires and outputs it through the output wires. In some sense, each gate is like a specialized processor, as it only computes one function. A given circuit solves the problem for inputs of a given size; in order to solve a problem, one needs to give a **family of circuits**, by defining a circuit for each input size.

The wires are one-directional, and there can't be any cycles, so the circuit is a DAG. This means we can group the gates into **levels** so that a gate's inputs only depend on computations performed on previous levels. Then a circuit's **span** is the number of levels, while its **work** is the total number of gates.

By restricting some of the above characteristics, one can get different types of circuits. For example, **boolean circuits** only use AND, OR, and NOT gates. [19] Two other circuit model variations are the bounded fan-in circuit model (the fan-in is bounded to 2 or a different constant) and the unbounded fan-in circuit model. Note that universal gates (such as NAND and NOR) can simulate another constant size universal gate set without asymptotically increasing the number of gates used.

Vector Machine Models

Vector machines [20] are a good example of data parallelism. They are machines that operate on vectors, where a (bit) vector is a sequence of bits. Vectors can be seen as infinitely long, and made of two parts: the significant bits, called the **non-constant** part, and the leading 0s or 1s, called the **constant** part of the vector. The vector machine model consists of bit processors and registers that can hold bit vectors (a register is essentially a memory location). Vector machines define the set of instructions they can perform, such as bitwise boolean operations, shift operations, and loading vectors into registers. [21] In a parallel vector machine, each of the processors is responsible for the same bit location in all the vectors in memory. In that sense, it's performing instructions on multiple data and is thus a type of MIMD (discussed below).

Language-based Models

The last types of abstract models are **language-based** models. [22, 23] They directly model programming languages, without considering the underlying machine. They are defined by specifying the language constructs. For example, the PAL model (parallel applicative lambda calculus) is based on call-by-value lambda calculus. It's easily described because it uses the same semantics, the only additional thing to be

described being their complexity (i.e. how long each of the basic operations takes).

Processor Models

Now we turn our attention to processor models. As implied by the name, these model the parallel computer as a set of processors, each of which performs computations to solve the problem. We classify models on a few different axes. The most widely used are the following:

- distinguishing models based on **instruction/data** parallelism. Any program/ computer/model (not necessarily parallel) can perform one or multiple instructions, and operate on either one or multiple data at the same time (in a single parallel operation). This gives rise to the following four categories. This classification is called **Flynn's taxonomy**. [24]
  - **Single Instruction, Single Data (SISD)**: no parallelism; most sequential computers are of this type
  - **Multiple Instruction, Single Data (MISD)**: multiple instructions are applied at the same time to the same piece of data; this is usually used for fault-tolerance
  - **Single Instruction, Multiple Data (SIMD)**: multiple processors do the same job at the same time on their assigned piece of data; in a way, this is similar to word-level parallelism
  - **Multiple Instruction, Multiple Data (MIMD)**: like SIMD, but one can have multiple types of instructions, not just one. Notice that if we set all instructions to be the same, we can effectively get a SIMD machine. In that sense, SIMD is a type of MIMD.
- based on **communication**. When multiple processors work together to accomplish one common goal, we need to make sure they can communicate with each other (e.g. about data that they both need to modify). They can either pass

messages directly from processor to processor, or a processor can write its message down somewhere for the other processor (or processors) to read. We get the following three types of models.

- **shared memory**: the processors are usually close enough that they can just write to a shared memory, where other processors can then read from. Shared memory models can be further subdivided according to their access policies: multiple processors could be allowed to read the same memory address at the same time (called **concurrent** read), or that could be restricted to only one processor at a time (called **exclusive** read). Applying a similar rationale to writing to a memory space, we have the following 4 submodels:
  - **EREW:** Exclusive Read, Exclusive Write
  - **ERCW:** Exclusive Read, Concurrent Write
  - **CREW:** Concurrent Read, Exclusive Write
  - **CRCW:** Concurrent Read, Concurrent Write

  All versions but the ERCW one are widely used.
- Distributed Memory with **Message Passing**: under the assumption that processors don't have access to a common memory, they need to communicate by sending messages to each other. Often such message passing costs are non-trivial, and models need to define how they measure the communication performance of algorithms in addition to computation. There are generally two ways to model message passing:
  - The communication network is a complete graph — in other words, each processor can directly reach every other processor. An example of such a model is given in [25]
  - Each processor can only reach a limited number of other processors directly. Examples of such networks include MIMD hypercubes and meshes [26]

– **Distributed Shared Memory**: this is a mix of the above — all underlying communication is done through message passing, but algorithm designers can assume access to a shared memory, which is abstracted away [27]

Models can also use a combination of the above strategies, such as hierarchical memory models. [28]

	Abstract model type	Flynn Taxonomy	Comm[a]	Synch	R/W Policy	Parameters	Performance Measures	Work	Span	
Word RAM	-	SISD	-	-	-	-	time t	t	t	
PRAM	-	MIMD	SM	lock-step	4 types	processors p	time t	$p \cdot t$	t	
BSP	-	MIMD	MP	bulk	-	barrier synch l, gap g, processors p	supersteps S, total local computation W, total comm H	$W + Hg + Sl$	$S \cdot L$	[29]
Binary Forking	language	-	-	-	-	-	work w, span s	w	s	[30]
Queueing Shared Memory	-	MIMD	SM	bulk	CRxorCW[b] (arbitrary)	processors p, gap g, maximum contention K	r_i reads, w_i writes, c_i local computations per processor	$p \cdot \text{span}$	$\sum \max\{\max_i\{c_i\}, g \cdot \max_i\{r_i, w_i\}, K\}$	[31]
LogP	-	MIMD	MP	asynch	-	processors p, comm delay L, overhead o, bandwidth g	r_i reads, w_i writes, c_i local computations per processor	$\sum_i^p r_i(L+o)+ w_i o+ \max\{g(r_i+w_i), c_i\}$	$\max_i r_i(L+o)+ w_i o+ \max\{g(r_i+w_i), c_i\}$	[32]
VRAM	vector	-	-	-	-	-	step complexity, element complexity	element complexity	step complexity	
BDM	-	MIMD	DSM	-	-	processors p, latency t, comm rate s, packet size m	computation time T_{comp}, communication time T_{comm}	$p \cdot \text{span}$	$T_{comp} + T_{comm}$	[33]
DMM	-	MIMD	DSM	-	CRCW	processors p	time t	$p \cdot t$	t	[34]

Table 2.2: Some models of computation and their categorization

[a] Communication: SM (shared memory), MP (distributed memory), or DSM (distributed shared memory)

[b] Concurrent reads or writes, but not both.

- based on **synchrony**. Some algorithms rely on processors being in synch with each other, while others don't. There are three main ways to categorize that.
  - **lock-step**: every step happens based on a clock, with all processors starting steps at the same time and finishing before the next step starts
  - **bulk-synchronous**: at certain points throughout the program, all processors wait for everyone else to be finished before they start the next step. This process is also called **barrier synchronization**.
  - **asynchronous**: there are no synchrony guarantees
- based on **branching factor**, which is the number of children at every node of the computation tree.
  - branching factor of 2, for example, the binary forking model
  - everything else has an arbitrary branching factor

2.3 Some Notable models

PRAM

PRAM, which stands for Parallel Random Access Model, is the most commonly used model of parallel computation. It's the parallel equivalent of the Word-RAM model. It assumes a computer with p Word-RAM processors. It is a synchronous, shared-memory MIMD model. There are 4 varieties, as differentiated by the read/write access, of which the following 3 are used: EREW, CREW, and CRCW.

BSP

The **BSP (Bulk Synchronous Parallel)** model [29] was introduced by L. Valiant in 1990 to serve as a "bridging" model of computation, which are models that inform both algorithm and hardware design. The main difference from other models popular at the time was how it modeled communication. BSP does not have shared memory,

so it keeps track of the communication complexity. A BSP system is composed of a few processing components, a router that facilitates communication between them, and ways to synchronize computations.

The model is defined by the following parameters:

- the number of processes (virtual processors) p
- the cost of barrier synchronization l
- g — the average time it takes a processor to deliver a message of size 1

It is a bulk-synchronous model, which means that throughout a program, at certain predetermined breakpoints, all processors need to be synchronized. Each set of steps between breakpoints is called a **superstep**. The breakpoints occur every L time units, which is called the **periodicity** of the system.

A BSP algorithm's performance is then defined by the following measures:

- S — number of supersteps used
- $W = \sum_{s=1}^{S} w_s$, where w_i is processor i's local computation cost
- $H = \sum_{s=1}^{S} h_s$, where h_i is the number of messages sent by processor i

In terms of those quantities, the work of a BSP algorithm is $W + Hg + Sl$, and its span is $S \cdot L$.

Comparison Models

Sorting is one of the most fundamental, common, and well-studied problems in parallel computing and computer science in general. Therefore numerous models have been proposed that focus on comparisons. In addition to sorting, these models are also useful for various other problems that depend on comparisons, such as finding the minimum element, finding medians, or searching.

- One of the first such models was proposed by Valiant [35]: the **Parallel comparison model** performs p comparisons at each step. It notably allows performing multiple comparisons involving the same element at the same time.
- A more restrictive version of that is the **comparator network**, whose basic operation *i-j comparison-exchange* can also be performed more than once at a time, but with the restriction that each element is being compared to at most one other element. [19] A **sorting network** is a comparator network that guarantees to sort the items in a non-decreasing order.
- **Comparator circuits** are circuit models, where the only allowed gate is the comparison with 2 inputs and 2 outputs — the first wire outputs the maximum of the input bits, and the second one outputs the minimum. [36]
- The **comparison PRAM** is a PRAM with the additional operations of comparison and moving data around in memory.

2.4 Simulations

If we have a good algorithm designed for a certain model, we would like to be able to "translate" it to run on a different model. We can do that by **simulating** a **guest model** on a **host model**. When simulating models, we're concerned with a few things. First of all, is possible to achieve the same results on a model as we could on a different one? How can we do it? Different models have their own parameters through which they measure performance. What's the performance of an algorithm on a simulated model?

In the following subsections, we discuss some tactics that are helpful for simulating models that differ in certain aspects.

Synchrony

Simulating an asynchronous model on a synchronous one comes with no hindrance in performance, as synchrony isn't an impediment to an asynchronous program. The

other direction is slightly harder to do. One general way to do it is to stop after each time step and synchronize all the processors. This comes at a cost since synchronizing p processors takes at least $\Omega(\log p)$ time on most models (to send a message to every processor). [37]

We can perform the simulation with the help of a **synchronizer**. A synchronizer is an asynchronous algorithm that simulates a synchronous system on top of an asynchronous one. [38] A synchronizer essentially coordinates with all the processors. It keeps a clock and sends pulses to all the processors at each simulated timestep. Awerbuch proposed several synchronizers in [39].

For example, a CRCW PRAM synchronous model using p processors can be simulated on an asynchronous model with $O(p)$ expected work per step and $O(\frac{p}{\log p \log^* p})$ asynchronous processors. [40]

In certain studied cases, however, the simulation overhead cost could be constant. It's an open problem exactly what algorithms could be modified in such a way, but some types have been proven to achieve this performance. [41]

Concurrent vs Exclusive Reads and Writes

Simulating exclusive reads/writes on a concurrent read/write machine comes with no worsening in performance. The performance might become better, although that would depend on the program. The general guarantee is the same span and work bounds.

The reverse is not true: simulating concurrent reads on an exclusive-read machine is more difficult. One way to do it is using binary trees. [42] We "expand" every time step by $O(\log p)$; in the first sub-timestep, one processor gets to read the value and copy it as a child in a binary tree, in the second sub-timestep, two processors can do the same, and so on; at the i$^{\text{th}}$ sub-timestep, 2^i processors can read the value (since there are 2^i copies) and create another new copy, which again doubles the number of available copies of the same value. This essentially creates a binary tree, which means that it takes $O(\log p)$ substeps to get enough copies for every processor to read. So simulating concurrent reads on an exclusive-read machine can be done with a $O(\log p)$

overhead in span, work, and space.

A similar reasoning works for writes — the processors write potential values in the binary tree in $O(\log p)$ time, and then each of the non-leaf nodes gets evaluated according to the concurrent write policy (e.g. if it's min-CRCW, it writes the minimum of the values in the parent node) in a process called a **tournament**, which will take an additional $O(\log p)$ time. The overall overhead is the same as for the reads — $O(\log p)$ in span, work, and space.

Simulating Between Shared and Distributed Memory

Simulating message-passing on shared-memory systems can be done with no span and work overhead with $O(p^2)$ additional space, where p is the number of processors. For each ordered pair of processors p_i and p_j, designate some part of the memory as their "inbox". Then p_i can pass a message by writing into that space, and p_j can read it from there.

To simulate shared on distributed memory with message-passing, we need to consider the communication overhead, which is going to be the time it takes to send a message in the host model. Let that be t_m. Since writing to and reading from shared memory can happen at each time step, the total span of the simulated algorithm will be $O(t_m \cdot \text{guest span})$ and the work will be $O(t_m \cdot \text{guest work})$. See [43] for a more rigorous discussion.

Branching Factor

Models with branching factor c can be simulated on a machine with branching factor $c' \geq c$ with no change in performance. When $c' < c$, we incur a $\lceil \log_{c'} c \rceil$ factor slowdown: instead of a node branching into c nodes, they can branch into c' nodes, each of which will have to branch into c/c' nodes each if $c' \geq c/c'$, or if $c' < c/c'$, we will need to recurse the same way $\lceil \log_{c'} c \rceil$ times.

MIMD to SIMD

SIMD is a special case of MIMD, so we can simulate SIMD on MIMD with no changes in performance (just set all instructions to be the same). The other direction incurs a $O(p)$ overhead in span and work, where p is the number of processors used. We can do that in the following way: at any time step, the algorithm is running at most p different instructions; let that number be k. For each MIMD time step, we have k sub-timesteps, where processors execute the same i^{th} instruction during sub-timestep i if that's the instruction they were meant to execute, and otherwise stay idle. Note that we would need to perform all communication at the end of each set of k sub-timesteps in a similar way. [44]

In the worst case, the span complexity of the algorithm becomes equal to its work, and it's essentially just running the algorithm sequentially.

2.5 Example Problems

Throughout this thesis, we will give examples based on specific problems. We define these problems here.

The **All-Pairs Shortest Paths** problem, also known as **APSP**, asks, given a graph with $|V|$ vertices and $|E|$ edges, what is the shortest distance between any pair of vertices.

The **Minimum Spanning Tree** problem, also known as **MST**, is also a graph problem, looking for the minimum spanning tree. For a given connected graph G, a spanning tree is a tree that includes all the vertices of G and a subset of its edges. A minimum spanning tree (MST) is a spanning tree with the minimum weight.

The **Longest Common Subsequence** problem, also known as **LCS**, has two sequences a and b as input, and the goal is to find a sequence that is a subsequence of both a and b, where to obtain a subsequence of s we remove characters from s. The resulting subsequence doesn't need to be contiguous.

Chapter 3

Methods

In this chapter, we cover the approach we took to obtain the results in the following chapters, as well as some important limitations resulting from generalizing data.

We focus on the process of generating the data on which we base our results. It started by looking for parallel algorithms that fit our scope. Then we had to process this newly acquired data. Finally, we could use that to compute characteristics we were interested in and analyze them to reach meaningful conclusions. We take a deeper look into certain aspects of this data flow and its steps.

3.1 Scope

We start by setting the scope of our analysis. There are all kinds of parallel algorithms, utilizing different forms of parallelism. Since our purpose was to be able to analyze them together, all of our algorithms had to be comparable — and their differences quantifiable.

First, we had to determine what problems we wanted to investigate. As a starting point, we used the set of 140 problem families that Sherry and Thompson created [10] and Rome [13] further refined. In this context, a **problem family** is a collection of various formulations of the same problem. These formulations have the same premise, but each answers a slightly different question. We call each of these **variations**.

There's a wide variety in how these variations are defined, which is specific to each

family. For an example on one end of the spectrum, consider the *Longest Common Subsequence (LCS)* family. Given two sequences a and b, LCS asks what is the longest subsequence common to both a and b, where a subsequence is obtained by removing some characters from the sequence. For this problem family, we only considered two variations: *LCS* (as stated) and *Multiple Longest Common Subsequence*, which is looking for the longest common subsequence of d sequences, where d is any number more than 1. On the other end, we have a problem whose variations are further apart — the *Maximum Flow* family: for a given graph with n nodes and m edges, each of those with a given weight $w_{i,j}$ representing its capacity, find the maximum amount of flow between a source node s and a sink node t. This family's variations include *Integer Maximum Flow* (the graph weights have to be integers), *Unweighted Maximum Flow* (all capacities are set to 1), *All-Pairs Maximum Flow* (find the maximum flow between any pair of two nodes), and even *Minimum-Cost Flow* (find the minimum flow above a given threshold). Clearly, an algorithm for *Minimum-Cost Flow* does not obviously solve any other *Maximum Flow* algorithms, while an *Multiple Longest Common Subsequence* algorithm would also solve the standard *LCS* problem by setting the number of sequences to $d = 2$. As a consequence, some algorithms can solve multiple variations.

For the purposes of our analysis, we split problem families into *problems* by grouping certain variations together in such a way that algorithms for any variation in a *problem* can solve all of the other variations. For example, this meant splitting all graph problem families into their directed and undirected versions; and we combined the *st-Maximum Flow* (using the basic definition) and the *Integer Maximum Flow* variations. In case an algorithm for a problem *family* solved the different *problems*, we chose the more restrictive problem.

Out of the 140 problem families, 17 didn't have an exact problem statement; out of the remaining 123, a further 35 (28.5%) did not have any relevant parallel algorithms, either because they aren't easily parallelizable, or just because there hasn't been enough interest for anyone to research parallel algorithms for them. A list of these problem families can be found in Appendix B. We've excluded them as well as

families with an inexact problem statement from our analysis. Out of the remaining 88 families, 25 were discarded because none of the algorithms we found fit our analysis criteria (see the Data Collection section below). By splitting and regrouping some of the variations as above, we ended up with **70 problems** with at least one analyzable parallel algorithm that make up the data we're using for this thesis.

3.2 Data Collection

Most of the data collection took place in the period of June 2023 to January 2024. Data collection worked as follows: after choosing a problem family, we searched for algorithms that satisfied our conditions (see below), then we read the papers to extract the information we were looking for. Algorithms that didn't satisfy our requirements were noted down for future projects instead of being discarded. In the end, we've looked at 1373 different papers and collected 486 relevant algorithms that make up our analyzable dataset.

We looked for papers mostly through Google Scholar. In addition, we used textbooks such as [45] and [46] to double-check that we haven't missed any important parallel algorithms.

Let's now dive deeper into the type of data we collected for each algorithm. The fields we considered are as follows:

- the problem family it pertains to
- the variation(s) it solves
- the authors of the algorithm
- the year it was published
- the span of the algorithm, as a function of problem size — the running time assuming infinitely many processors are available (see section 2.1 for more details)

- the work of the algorithm, as a function of problem size — the running time using a single processor
- the model for which this algorithm was designed for and analyzed in
- whether it's randomized (i.e. using a probabilistic model, or utilizing randomness as part of the algorithm); it should be noted that while we've collected this data field for every algorithm, randomized algorithms that return the correct result with high probability ($1 - n^{\epsilon}$ for any small $\epsilon > 0$) are treated the same way as deterministic algorithms
- whether it's approximate — we did not analyze approximate parallel algorithms, instead we put them aside for the future
- whether it's heuristic-based; such algorithms don't have a proof of correctness, and as such not comparable to algorithms with theoretical guarantees
- whether it's parallel
- whether it's GPU-based; GPUs use a different type of parallelism — one processing unit performs multiple computations at the same time, as opposed to data and task parallelism, where computations are performed simultaneously by *different* processors, each processor doing at most one at a time. For this reason, we did not include them in our analysis.

Processor Data

While most of the data collection has been focused on the parallel algorithms themselves, we've used other types of data for the analysis. More specifically, we used data on the maximum number of available processors in both supercomputers and commercially available (personal) computers. For supercomputers, this processor data was scraped from the TOP500 dataset [1], which ranks the top supercomputers in the world. Twice-yearly rankings are available, starting with June 1993. The other successful parallel computers not covered by this dataset are the **ILLIAC IV** [4], the

first parallel computer to be built, and the **Connection Machines** series. [5, 6, 7] For commercially available computers, we've used a dataset mainly composed of the **CHIPS** dataset. [47, 2]

3.3 Data Processing

After the algorithm data was collected, we then had to process it. To be able to do a general analysis, every aspect of every algorithm had to be categorized. We looked at the main aspects that we needed to standardize this way.

Models

Since there are many models of computation (as seen in section 2.1), we'd like to be able to group together the ones that share the same main characteristics, in such a way that all models in the same group are comparable to each other. Here is the categorization we chose:

- **PRAM**, split up by read/write policies into EREW, CREW, and CRCW. When the PRAM subtype is not specified, the most powerful model, CRCW, is assumed
- **SIMD-SM**, similarly separated by its read/write policy
- **MIMD-TC** — tightly-coupled (a.k.a. shared memory) MIMD. This is also separated by its read/write policy
- **BSP** — Bulk Synchronous Parallel Model
- **comparator circuits**, including sorting networks and hardware sorters
- **Distributed memory** models — a general category grouping together all distributed memory models, such as MIMD hypercubes, or unnamed models using message passing. These models aren't always directly comparable, but all of them imply (sometimes weaker) bounds on simulated machines

- **Other** — any model that doesn't fit into any of the above categories is collected, but not analyzed

It's worth noting that sometimes it can be useful to compare different algorithms, even though they're designed for different models. To do this, we simulate all our model categories on one "main" model. We use a shared memory MIMD with concurrent reads and writes (**CRCW MIMD-TC**) as the main model. We've already discussed details on how models can be simulated on other models, as well as what the new simulated bounds are, in the background subsection 2.1.

We've examined some trends related to the model data we've collected. This analysis is available in Appendix A.

Running Time

Running times don't always depend purely on the problem size n. For example, running times for graph problems often depend on both $|V|$, the number of vertices, and $|E|$, the number of edges. The number of parameters that runtimes are based on vary widely with the problem families and their variations. We needed to standardize these so we'd be able to compare runtimes the same way irrespective of the problem.

We settled on defining n as the problem size, and for every problem family/ variation, choosing parameters in terms of this size n to be something "universal", which would be a special case of the problem that is widely used. For example, for graph problems, we take the special case of dense graphs. We define **dense** as $|E| = \Theta(|V|^2)$. Then $n = |V| + |E| = \Theta(|V|^2)$, and we get $|E| = \Theta(n)$ and $|V| = \Theta(\sqrt{n})$. You can find some common generalizations in Appendix C.

Note that we define n as the problem input size. This sometimes goes against convention (e.g. for graphs usually $n = |V|$, while here we take it to be $n = |V|+|E|$). We think that this would make the analysis more meaningful, as n is defined the same across all problems.

All span, work, parallelism, and sequential time measures have been redefined this way in terms of n.

3.4 Data Analysis

In this section, we briefly describe implementation details of our analysis.

Datasets Used

Based on the above data, we've created two parallel datasets that we used for different parts of the analysis:

- the **original** dataset: we include all of the analyzable parallel algorithms from our parallel data. This does not include algorithms that have been designed for a distributed memory model or for a model we labeled as "Other".
- the **simulated** dataset: we simulate all algorithms (except ones with an "Other" model) on our main model (CRCW MIMD-TC). Since this is one of the most powerful models, all running time bounds for the other models hold here as well. This dataset allows us to compare more algorithms and get a more meaningful analysis.

The sequential dataset used was created using data collected for the initial algorithms improvement project. [10] That dataset was updated by Rome et al. [13, 14] and further by us to a limited extent.

Parallel Running Time Calculation

Lastly, we describe how certain metrics are calculated. These are the basic tools that we used in our analysis in the following chapters.

One of the metrics we've used the most is that of **running time**. Often it's useful to be able to compute runtime numerically. Sequential time, span, and work are all asymptotic functions of the problem size n.

For numerical sequential times, we can just evaluate the asymptotic function for a given n; the work and span for a given n can be found in a similar way. That allows us to compute parallel times, which should be a function of both problem size n and the number of processors used p. Here we have two options:

1. runtime $= \max\{\text{work}/p, \text{span}\}$

2. runtime $= \text{work}/p + \text{span}$

These are equivalent asymptotically, but numerically they differ slightly. In fact, runtime is bounded by both options when scheduled by a Greedy scheduler:

$$\max\{\text{work}/p, \text{span}\} \leq \text{runtime} \leq \text{work}/p + \text{span}$$

and either option is within a factor of 2 of the optimal runtime. [18]

We will be mostly using the second *upper bound* definition throughout this thesis. The only exception is section 7.2, where we'd like to be able to compare pairwise performances that are equivalent asymptotically, and the max function ensures that two algorithms with the same asymptotic runtimes get treated as equal.

It's worth it to note that for most of our results, switching to the first *lower bound* definition would lead to insignificant differences. The only part where this distinction matters is section 6.1, where we plot the running times and speedups separately. We discuss this in more detail in that section.

Note that we're not taking into account leading constant factors, so this only leads to rough estimates, but Sherry and Thompson show that for most problems, at least serially, this is a good approximation. [10]

Calculation of Other Metrics

Another commonly used metric is that of **relative speedup**. The speedup of an algorithm A with respect to another *base* algorithm B is calculated as the ratio between the runtime of the base algorithm B and the runtime of A. This way, the relative speedup is > 1 if our algorithm is faster (i.e. has a lower running time). The base algorithm is often the first known sequential algorithm, but can be any other one, such as the first known parallel algorithm or the best sequential algorithm at the same time.

$$\text{relative speedup}_B(A) = \frac{\text{runtime}(B)}{\text{runtime}(A)}$$

We also use the notion of **improvement**. We define **sequential improvement** by the algorithm's running time — an algorithm makes a sequential improvement if its running time is asymptotically faster than the best-known algorithm at the time (i.e. the year before). The **span** and **work** improvements are defined similarly. The **performance improvement** for a given problem size n and number of processors p is defined as a faster *runtime* (as calculated above). A general **parallel improvement** is defined as an algorithm having pushed out the Pareto frontier, i.e. there's no other existing algorithm with the same or better span and the same or better work.

Chapter 4

Achievements of Parallelism

4.1 Parallel Algorithms Progress

When studying parallel algorithms, it's only natural to compare them to their sequential counterparts. Our first question then is, how much progress has there been in parallel algorithms as compared to sequential ones? We analyze how many improvements there have been for each category over time for cases where there are both. It turns out that almost since the start of this field, there have been more parallel algorithms with improvements than sequential ones.

Figure 4.1 shows what proportion of the problems had improvements in each decade, for both sequential algorithms and parallel algorithms. Note that we only consider the 70 problems for which we have at least one parallel algorithm. Here, we define sequential improvement as having an algorithm with a smaller asymptotic runtime at the end of the decade compared to the beginning of the decade. For parallel algorithms, that metric needs to change slightly, since the running time of an algorithm depends both on its work and its span. We settled on considering an algorithm as being better if its span (respectively work) was better than that of the algorithm with the best span (respectively work), or if at the end of the decade, there was an algorithm for which none of the algorithms at the beginning of the decade had both better span and better work. In other words, if it pushed the **Pareto frontier** of the work-span trade-off (see a deeper discussion of this concept in Chapter 5).

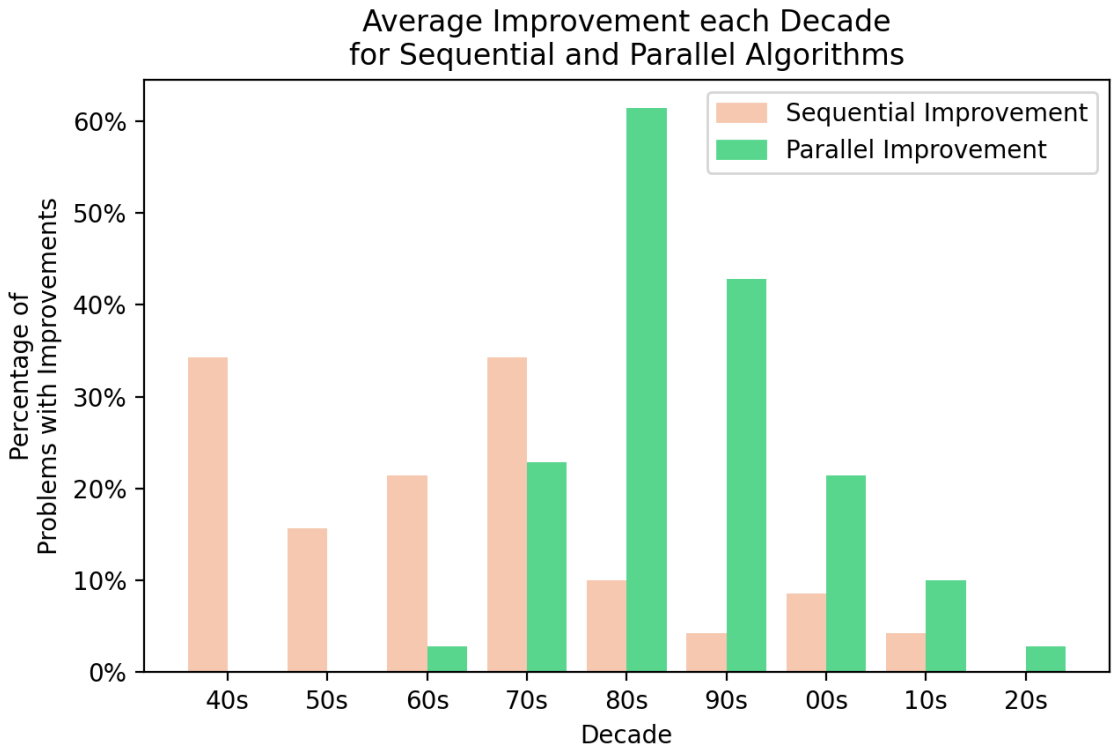

Figure 4.1: Progress over the decades — a comparison between sequential and parallel algorithms

One of the first details one notices about this plot is that there was a lot a lot more sequential improvement until the '70s — 56% of problems had sequential improvements by the end of the '70s, and more parallel improvement after, skewing the parallel bars towards the present. Both of these match our expectations — for the sequential side, there weren't that many problems with significant improvements in runtime after the '70s because many had already hit their theoretical lower bound. [12] On the other hand, there was lots of progress to be made on the parallel algorithms side since the start of the field was only in the late '60s. [1]

Parallel algorithms start in the late '60s, the first algorithm in our database is from 1968. Parallel improvements happen at a much higher rate in the '80s and '90s — 61.4% and 42.9% of problems had parallel improvements respectively.

[1]Note that Liu's thesis [11] has a similar analysis for the sequential upper bounds in Figure 1-1b, which looks slightly different from the sequential part in our Figure 4.1. This difference mainly comes from a slight restructuring of how we define our problems, as well as the fact that we're not considering problems without any parallel algorithms.

The '80s constitute an impressive peak of parallel progress, after which we've had a consistent decline. In the last decade, there has been progress for almost 10% .

4.2 Parallel Hardware Improvements

None of the algorithms designed for parallel computers would be of any use if there were no parallel computers. Figure 4.2 shows the number of available processing cores over time for two different situations: the best supercomputers in the world, and commercially available personal computers.

The supercomputer data is mostly based on the TOP500 lists, and the data on commercially available machines is from WikiChip. [1, 2] A historical note is that today, the standard word size for supercomputers is 64 bits; that has not always been the case. This affects the early supercomputers up to roughly 1990, with the Connection Machine series being especially relevant to our analysis. We accounted for this by treating 4 32-bit processors as 1 64-bit processor for CM-2 — with 2048 32-bit floating point units, it's the second green step in Figure 4.2 at 512 processors. For the same reason, we've also excluded the CM-1 machine because its 65536 processors were all 1-bit.

The first step in the picture is the ILLIAC IV machine, the first ever parallel computer to be built (in 1972). ILLIAC IV had a 64-bit mode of operation for its 64 processors.

The Connection Machines (1985-1993) sparked a more or less consistent exponential increase in the number of supercomputer processors, reaching almost 20 million cores in 2017 thanks to the ExaScaler **Gyoukou** computer. [8]

On the front of commercially available computers, the first ones with more than 1 core were Intel Core Duo T2050 and AMD Athlon 64 X2 3800+, both released in 2005. AMD released EPYC models 7702, 7702P, and 7742 in 2019, and with 64 cores, they currently have the highest available number of processors.

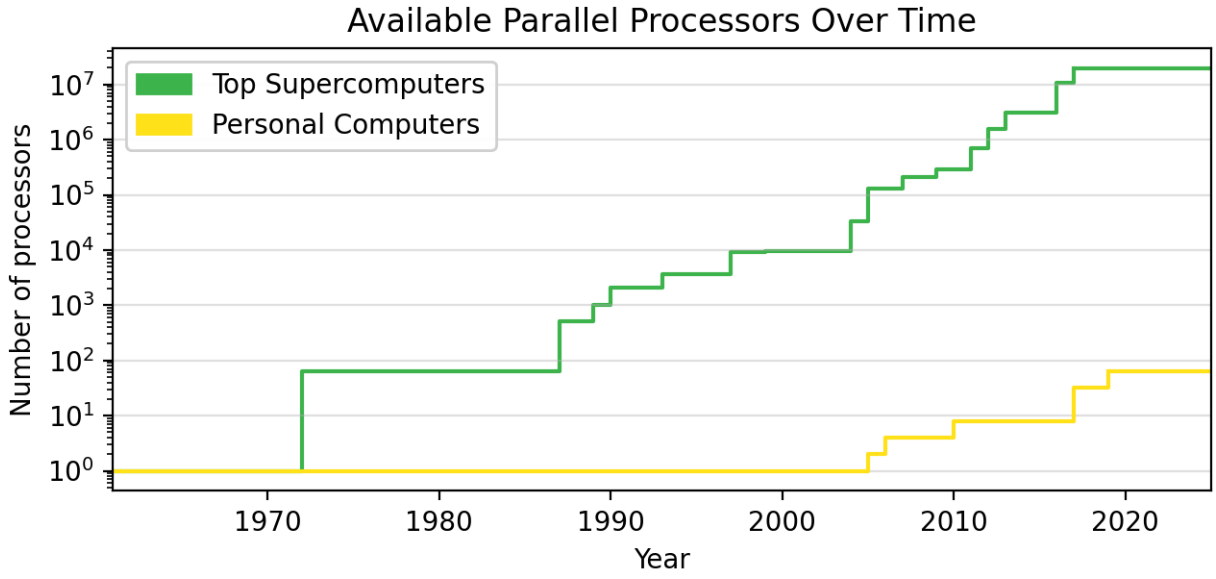

Figure 4.2: The maximum number of cores available every year for the most powerful supercomputers in the world [1] and for commercially available personal computers [2]

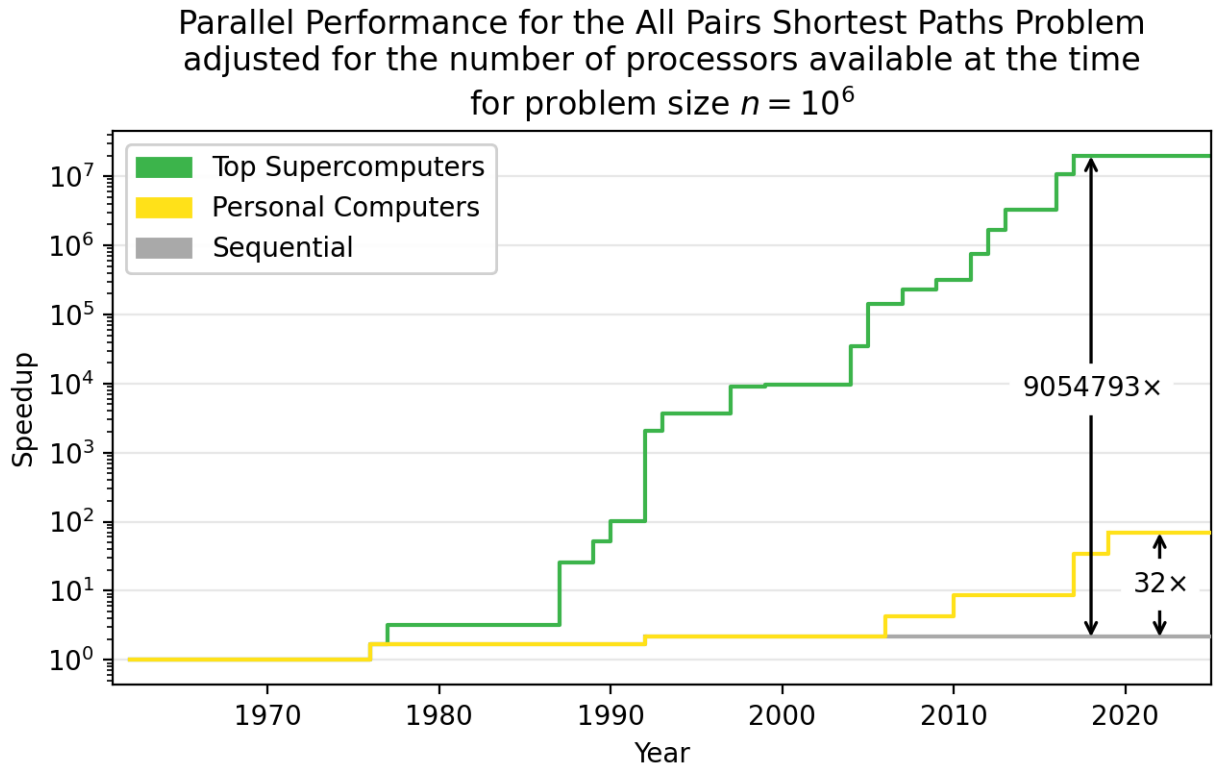

Figure 4.3: Relative speedup for the All Pairs Shortest Paths problem

4.3 One Problem's Progress

We'd like to observe how both hardware and algorithms have impacted parallelism progress. We study an individual problem and track its performance, which we define here as the speedup relative to the first known algorithm. As a reminder, the relative speedup of an algorithm A with respect to B is the ratio between the running time of B and runtime of A, and the runtime of an algorithm is computed just from the asymptotic complexities of its span and work (see section 3.3 for more details).

We've used the highest number of available processors in computing parallel running times — both the top supercomputer and commercially available ones. So in some sense, this represents the maximum practically achievable speedup at the time, assuming each of the two scenarios.[2]

Figure 4.3 shows the performance for the All Pairs Shortest Paths problem for a problem size of $n = 10^6$. The three lines represent sequential progress, or speedup with only one available processor, parallel progress with commercially available numbers of processors, and parallel progress with the highest existing numbers of processors.

The sequential improvements are small. We start out with speedup 1 with the first known APSP algorithm corresponding to the Floyd–Warshall sequential algorithm in 1962, [48] with $O(|V|^3)$ running time. This is improved to the subcubic time of $O(n^3(\log\log n/\log n)^{1/3})$ by Fredman in 1976. [49] In the dense graphs case (the case we're considering, see Appendix C), $|V| = O(\sqrt{n})$, which leaves us with a speedup of 1.67.

The first parallel improvement we see is Savage's 1977 algorithm with $O(|V|^3 \log |V|$ work and $O(\log |V|^2)$ span. [50] On an ILLIAC IV, it would outperform the best sequential algorithm at the time with a speedup of 3.2.

As we go towards the present, we can see that the shapes of the yellow and green lines look similar to the lines in Figure 4.2 — those improvements correspond to hardware improvements using the same algorithm. This is because the relative speedup

[2] A caveat is that the architectures of existing supercomputing machines oftentimes use a mix of shared and distributed memory, and throughout this thesis, we're assuming that algorithms are either designed for or simulated on shared memory models. See section 2.4 for more on model simulations.

of an algorithm is linear in the number of processors roughly until we reach the algorithm's available parallelism. In this case, most of the APSP parallel algorithms have almost cubic available parallelism, which is roughly 18 orders of magnitude. We discuss available parallelism more in Chapter 6.

Other times, the line raises even more, which signifies a new better algorithm, such as the significant 1992 improvement with a randomized algorithm by Han, Pan, and Reif. [51] They improved on both span and work, the new measures being $O(\log |V|)$ and $O(|V|^3)$ respectively. 1992 had a separate improvement with an unrelated sequential algorithm by Takaoka. [52]

Today, APSP parallel algorithms running on supercomputers are able to provide more than a 9 million-fold theoretical increase in performance over the best available sequential algorithm.

4.4 Relative Speedup Improvements for all Problems

Some problems achieve bigger speedups than others, and we'd like to see what this speedup looks like in the general case. To do this, we observe the 25^{th}, 50^{th}, and 75^{th} percentiles for speedup over all problems, for each year. Just as before, we consider these in three situations: sequential improvement, parallel improvement assuming commercially available levels of parallelism, and parallel improvement assuming supercomputing parallelism. As before, we take the problem size to be $n = 10^6$.

The shapes of the parallel lines are once again reminiscent of the ones for the available processors in Figure 4.2. This means that the above reasoning holds in general, and the current speedup values are bounded by hardware more than algorithms.

The lines start at speedup 1, and stay constant for a very long time (progressively shorter as we consider a higher percentile) — this makes sense because a lot of problems don't have that much improvement, especially in the beginning. On top of that, algorithms with tight bounds have been found before that for many problems, see

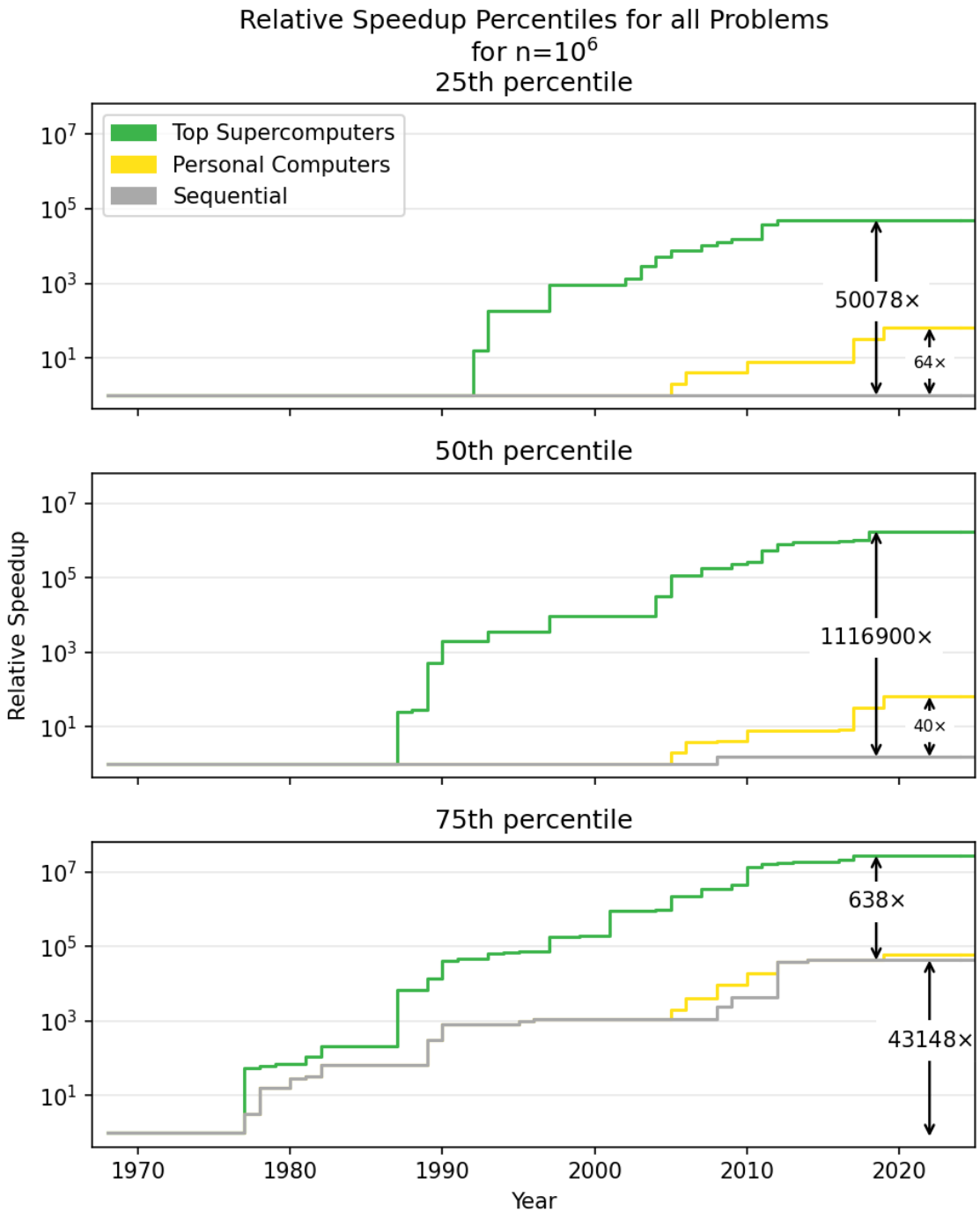

Figure 4.4: Relative speedup at the level of problem families over time as the 25$^{\text{th}}$, 50$^{\text{th}}$, and 75$^{\text{th}}$ percentiles for sequential algorithms, parallel algorithm using the maximum commercially available parallelism, and parallelism available from the most powerful supercomputers, calculated for input size 10^6

[11]). We're using 1968 as the speedup reference year since it's the year corresponding to the first parallel algorithm in our database. In addition, some problems were not defined or didn't have parallel algorithms by that point.

For those reasons, we see that the sequential lines for the 25^{th} and 50^{th} percentiles stay at or slightly 1 throughout the present, and for a significant portion of time, so do the parallel lines. This shows that at least 50% of the problems didn't have any sequential improvements in the decade between 1968 and the late 2000s. On the other hand, 50% of problems have seen a parallel speedup greater than 1 starting in 1987, with 25% more problems getting improvements in the short span between 1987 and 1992.

The 75^{th} percentile lines start increasing early, 25% of problems having had both a sequential and parallel improvement by 1977. All 75^{th} percentile lines rise a lot. An interesting phenomenon to notice happens in 2012: the 75^{th} percentile sequential algorithm goes above the then-best parallel algorithm with personal computer parallelism. This happens occasionally because PC-level parallelism is small enough that new good sequential algorithms are able to overtake it.

In 2024, the speedup of the 75^{th} percentile is 43148, 941, and 550 times greater compared to the 25^{th} percentile for sequential, PC parallelism, and supercomputer parallelism respectively. For the top 25% problems, the speedup gain of supercomputer-level algorithms is over 10^7. For the median and top 75%, the gain is at roughly 10^6 and 10^5 respectively.

4.5 Overall Parallel Progress

Finally, we'd like to look at the overall progress. How much impact do parallel algorithms actually have? We look at the overall improvement of each problem, as measured by the relative speedup of the best current-day algorithm taken with respect to the first existing algorithm. Once again, we're interested in those measures in three situations: sequential, personal computer parallelism, and supercomputer parallelism. We want to see how much improvement in each of these accounts for the

total improvement, as measured against the highest speedup. We use the geometric proportion for this. Figure 4.5 shows their arithmetic averages for three different problem sizes.

Sequential algorithms account for roughly 27% of algorithmic progress across all problem sizes. We also notice that the amount of parallelism is very important — the more parallelism we have, the more improvement we can get. This seems to be especially important with increasing problem size — for n=10^3 it's only 36%, but for 10^9 it increases by roughly half to 56%. On the other hand, personal computers seem to make a bigger difference for smaller problem sizes.

Overall, we see that parallel algorithms can have a lot of impact, especially for large problem sizes and especially if lots of processors are available.

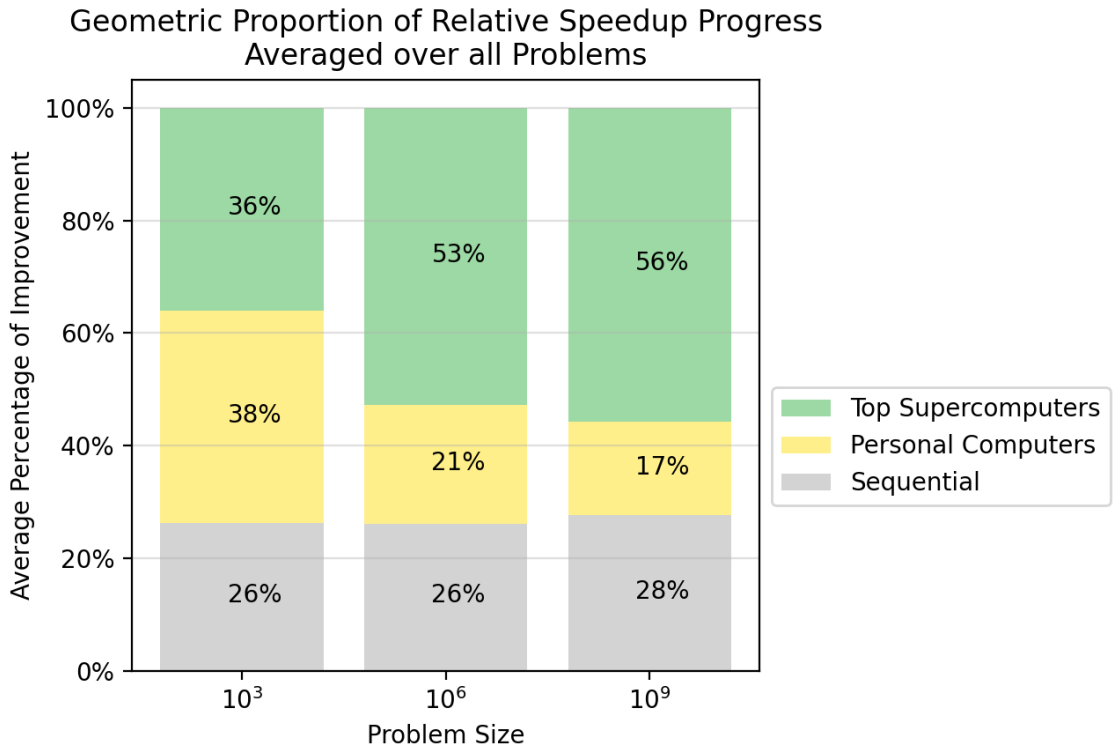

Figure 4.5: Maximum sequential and parallel (in the cases of commercially available personal computers as well as supercomputers) relative speedups from the first algorithm for all problems

Chapter 5

Cost of Improvement and Nontrivial Trade-offs

5.1 Work and Span

The relative speedup that we've studied in the previous chapter is directly related to the running time of a parallel algorithm. That running time depends, in turn, on the work and span of the algorithm. When trying to improve the runtime of an algorithm, one can either improve the span, or the work, or both. Naturally, improving both at the same time is the hardest. We'd like to investigate how these two measures interact. We start by plotting them for all algorithms of an individual problem. Figure 5.1 shows this for the Longest Common Subsequence (LCS) problem, which, given two sequences a and b, asks to find the longest subsequence common to both of them, where a subsequence of a string s can be obtained by removing (not-necessarily contiguous) characters from s.

Figure 5.1 plots all algorithms for the LCS problem according to their spans and works — each dot depicts an algorithm; its color is representative of the model for which the algorithm was devised, and the year next to it is the year in which the algorithm was published. In addition to parallel algorithms, the plot also shows one sequential algorithm — the one with the best running time, shown in lavender on the lower right side.

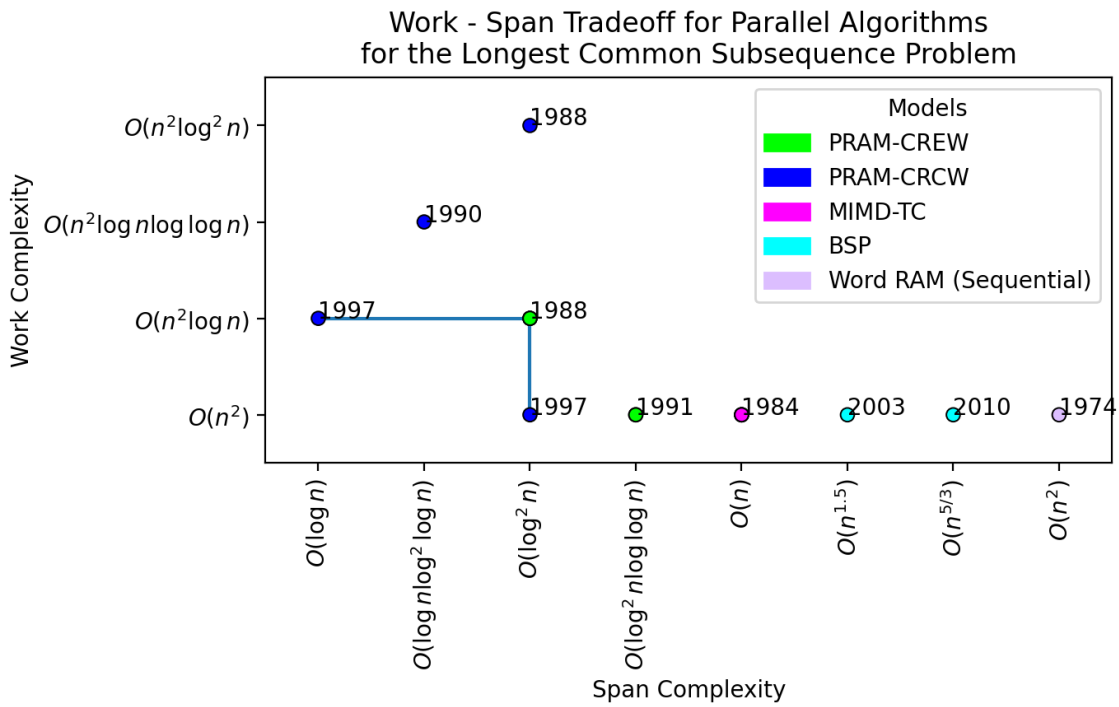

Figure 5.1: Work vs span for the Longest Common Subsequence problem. Each colored dot represents a parallel algorithm for the respective model, the best-known sequential algorithm is plotted for reference. The line shows the Pareto frontier if all the algorithms were simulated on a shared memory model.

For this problem, the best sequential running time is $O(n^2)$. [53] We can interpret it as a parallel algorithm that always runs on exactly 1 processor. In that case, both its span and work would be $O(n^2)$. Notice that there are no algorithms with work lower than that — this is to be expected because a parallel algorithm with better work would imply the existence of a sequential algorithm with better time (since we can just run it on a single processor). The algorithms on the same line all have the same work as the best sequential algorithm, therefore they are all **work-efficient**.

On the other hand, one can notice that none of the spans are *greater* than $O(n^2)$. While that can sometimes be the case, it's rare to see parallel algorithms with worse spans than the best sequential time — on a parallel computer, the sequential one would still be faster. Therefore this mostly happens when the parallel algorithm is designed before the sequential one is published. For the Longest Common Subsequence problem, however, all parallel algorithms appeared after 1974.

In terms of reading performance on this graph, an algorithm is "better" the lower it is and the more to the left it is. However, there is no LCS algorithm that's the best in both respects — that would have to be in the lower left corner, e.g. an $O(n^2)$ work, $O(\log n)$ span algorithm.

Sometimes, a new algorithm is better in both span and work — for example, for CRCW PRAM algorithms, the 1997 $O(n^2 \log n)$-work $O(\log n)$-span algorithm is better in both dimensions than the previous one from 1990. Other times, improvements occur only in one direction — such as the span improvements for the work-efficient algorithms. Algorithms improving one metric can also sometimes make the other metric worse.

The blue line represents the **Pareto frontier** of span and work for the algorithms simulated on an asynchronous CRCW shared-memory MIMD. Algorithms on this line represent the best trade-offs between work and span.

This plot also shows us how the Pareto frontier was pushed through the years. For example, The 1988 CREW PRAM algorithm was improved upon in 1990 by two algorithms in two different directions — improving work but having a slightly worse span for the MIMD-TC algorithm, and the opposite for the PRAM-CRCW one (a CREW algorithm implies both a PRAM CRCW algorithm and a MIMD-TC one with the same bounds, see section 2.4).

5.2 Nontrivial Work Span Trade-offs

LCS is an example of a problem with non-trivial trade-offs — there is no algorithm with both the best span and the best work. But how often does this happen? We analyzed how many of our problems have such nontrivial trade-offs between work and span. Currently, the figure stands at **35.7%**, a significant proportion.

This number has changed throughout the years, as Figure 5.2 shows. We track the portion of such algorithms for each decade starting from the 1980s. Specifically, we check how many problems had work-span trade-offs at the end of each decade out of how many problems had parallel algorithms at all. A problem is considered to have

a work-span trade-off in a decade if, at the end of it, no algorithm has both the best span and the best work.

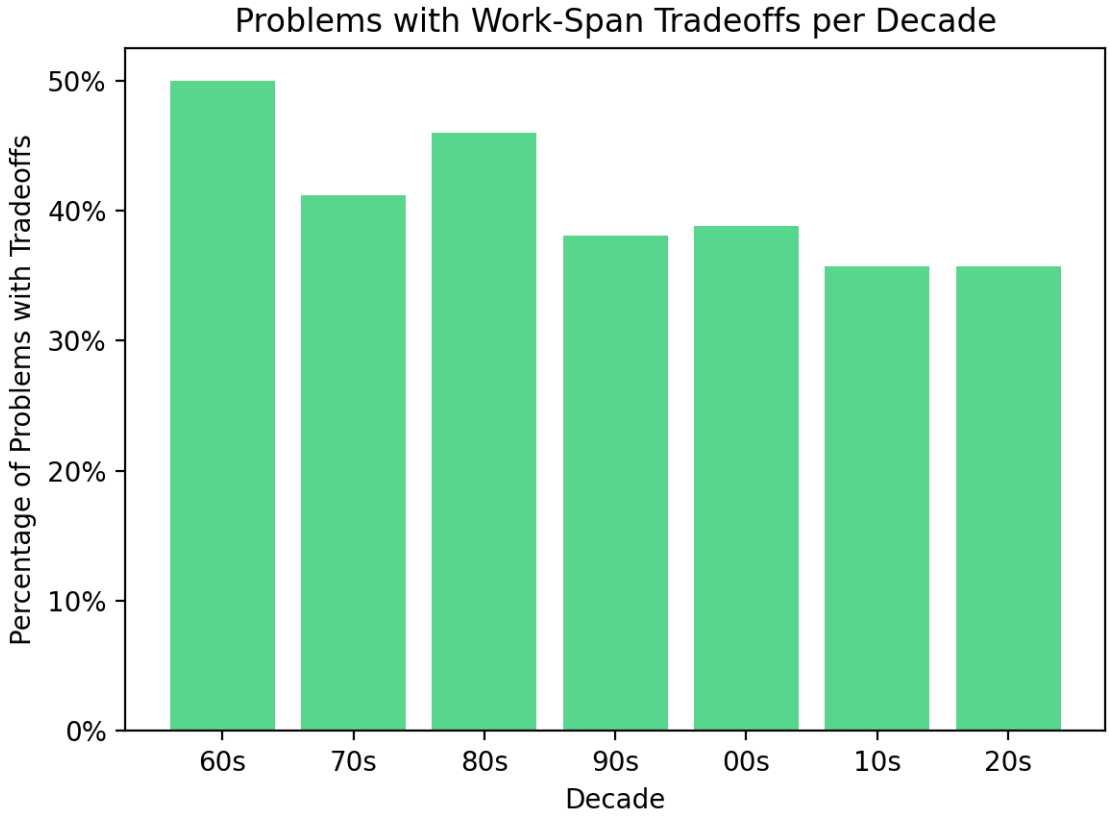

Figure 5.2: Work-span trade-offs throughout time

The number of problems considered for each bar also varies — every decade we only consider the problems that had at least one parallel algorithm by the end. For example, the first decade only had two problems with parallel algorithms by the end of 1970 (Comparison Sorting and Discrete Fourier Transform). Discrete Fourier Transform had an $O(n \log n)$ work algorithm, the same as the best sequential at the time. Since all Comparison Sorting algorithms had $O(\log^2 n)$ span and $O(n \log^2 n)$ work, and the best sequential algorithm was $O(n \log n)$, it turns out that 50% problems with parallel algorithms had work-span trade-offs. 50 out of our 70 problems have had their first parallel algorithm by the end of the '80s however, with a further 13 by the end of the '90s.

We can see that the proportion has stayed pretty consistent since then, mildly

decreasing, probably because as more algorithms get designed, we're able to find ones that are better in both span and work.

5.3 The Span of Two Algorithmic Extremes

A different way to look at this existing trade-off would be to look at both extremes and compare them. In the work-span graph of a problem (such as Figure 5.1 for LCS), this would correspond to the leftmost vertical line — representing the lowest achievable span, and the lowest horizontal line — representing work-efficient algorithms. We will be comparing these two types of algorithms in different contexts.

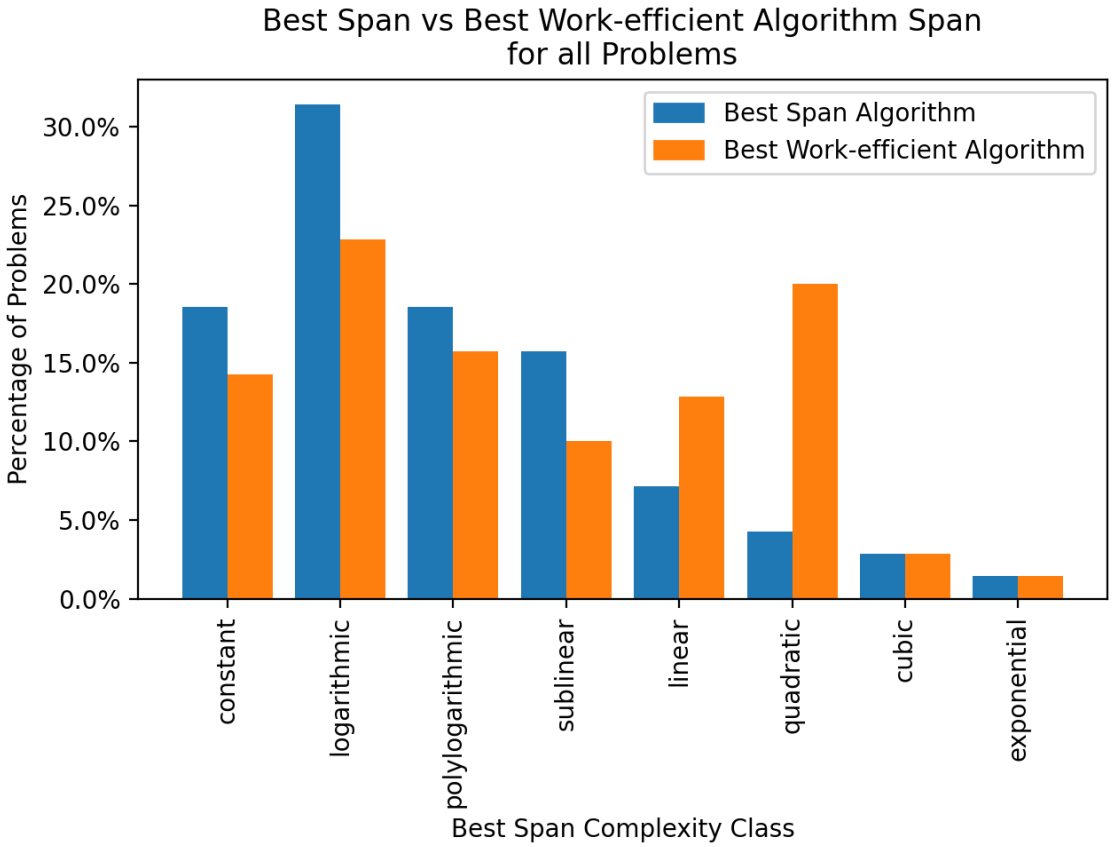

Figure 5.3: Comparing the best span and work-efficient algorithms
The blue bars show the distribution of best spans — the percentage of problems with an algorithm of the corresponding complexity class. The orange bars show the distribution of spans of the work-efficient algorithms, taking the lowest ones in the case of ties.

Looking at the theoretical limit on how fast algorithms can run, we wish to start by examining the best spans for these two categories across all problems. Note that a best-span algorithm can be defined for all problems (as long as they have at least one algorithm in general), and the same applies to work-efficient algorithms as well. In the case that there are no work-efficient *parallel* algorithms, the best sequential algorithm is the work-efficient one, since it can be viewed as a parallel algorithm with a maximum parallelism of 1.

Therefore for each problem, we can take the span of the best-span algorithm and the span of the work-efficient algorithm (we take the lowest such span in the case of ties). We plot the distribution obtained this way in Figure 5.3.

We can see that the lowest span distribution is skewed to the left compared to the work-efficient distribution. This is to be expected because, for a given problem, a work-efficient algorithm's span can only be as low as the best span one. If they're equal, that would mean that there's no nontrivial trade-off for that problem, and both of its algorithms fall into the same complexity class bucket. The fact that the distributions aren't the same is therefore because nontrivial trade-offs exist.

5.4 Overhead of Best Span Algorithms

Ideally, we'd like the algorithms we use to be work-efficient. Besides using fewer resources in general, this also minimizes the running time — especially for small values of p (the number of processors), the magnitude of the work has a bigger impact than the span (see Chapter 6). We can quantify how "far" an algorithm is from being work-efficient by defining **work overhead** as the ratio between work and the best sequential time, asymptotically. By this definition, a **work-efficient** algorithm has $O(1)$ work overhead.

Then we can see what the "cost" of using the lowest span algorithm would be, as represented by the work overhead of the algorithm. Figure 5.4 shows that in the form of a grid comparing the span and the work overhead of the lowest span algorithm for each problem. Each square shows the percentage of problems that fall into the

corresponding span and overhead buckets. The Total column and row show the sums of each row and column, and therefore also represent the distributions of span and overhead.

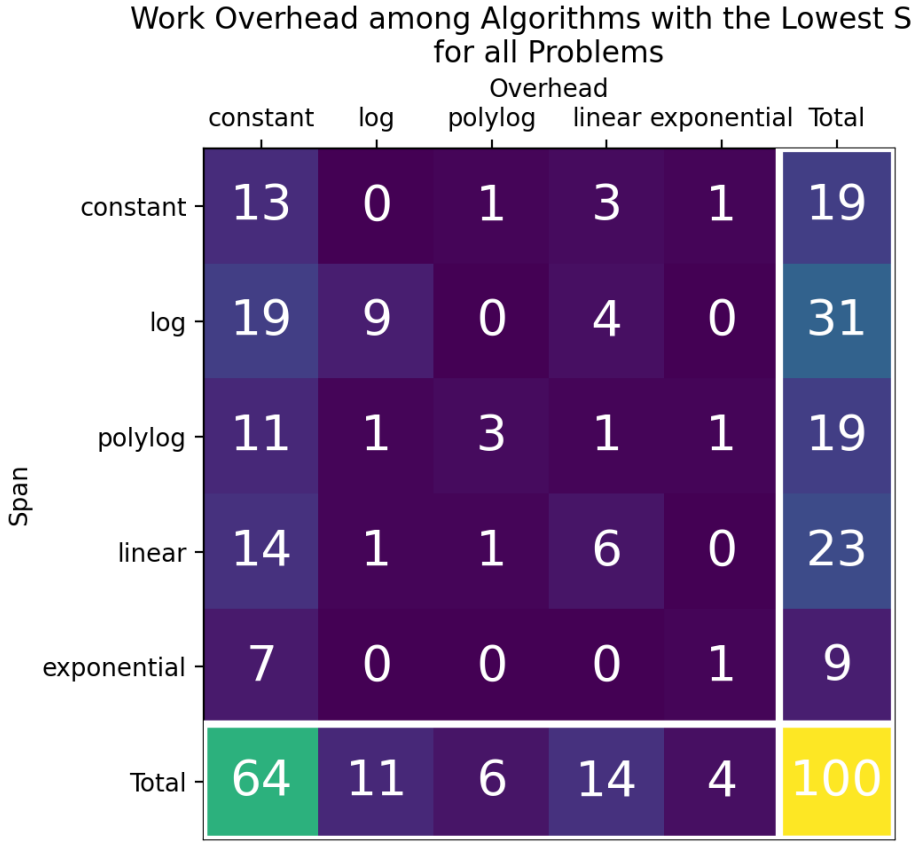

Figure 5.4: Percentage of problems whose best-span algorithm falls into the respective span and overhead buckets.

We can see that most of the problems are concentrated in the upper-left corner, which is where the fastest algorithms fall. Looking at the bottom Total row, we can see the distribution of work overhead among the lowest span algorithms. We can see that most of our problems (64%) have constant overhead, which means that they're work-efficient. This is consistent with the fact that 35.7% of the problems have a nontrivial work-span trade-off. Most of the rest are split between (poly)logarithmic and linear complexity (17% and 14%), leaving only 4% of problems with exponential overhead.

The rightmost Total column shows the lowest span histogram. It's consistent with

the lowest span (blue) bars from Figure 5.3, with slightly different categories. Most lowest spans are logarithmic, and only 9% are superlinear.

Chapter 6

Problem Size and Parallelism

We've talked about how work is the leading factor in running time when the number of processors is small; that's one of the main reasons why we prefer work-efficient algorithms to non-work-efficient ones, especially in practice. However, work efficiency is not necessarily strictly better. In particular, in this section, we look at the **available parallelism** of an algorithm, the largest number of non-idle processors it can use. Asymptotically, this is achieved when the runtime becomes equal to the span, which is why we calculate the available parallelism complexity as work/span.

6.1 Two Minimum Spanning Tree Algorithms

To illustrate why available parallelism can be as important as work efficiency, we show two different algorithms for the same problem, one which is work-efficient, and one which has a better span. We consider the Minimum Spanning Tree problem (see section 2.5). The work-efficient algorithm is due to Deo and Yoo (1981), [54] and has $O(n^{0.75})$ span and $O(n)$ work, and $O(n^{0.25})$ available parallelism. The other algorithm is due to Johnson and Metaxas (1992), [55] with $O(\log n^{1.5})$ span, $O(n \log n^{1.5})$ work, and $O(n)$ available parallelism.

First, we look at what happens to the algorithms' performance when we vary the number of processors for three values of problem size n — 10^3, 10^6, 10^9 (see Figure 6.1). In this instance, we take the performance to be the speedup of the algorithm

relative to the best sequential algorithm. This is sometimes referred to as **strong scaling**.

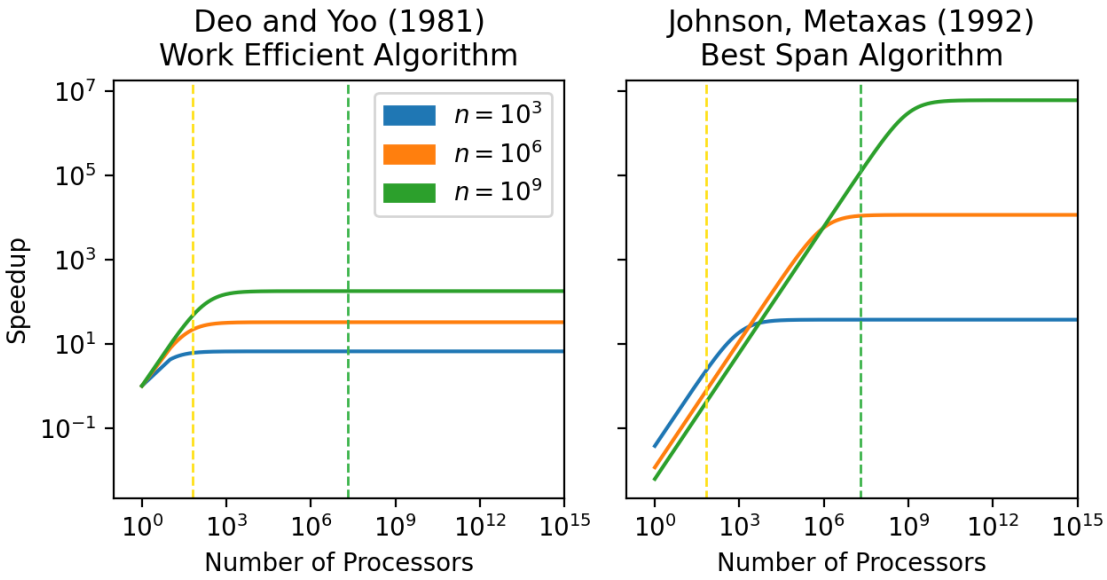

Figure 6.1: Strong scaling for two different algorithms: a work-efficient one, and one with a better span but not work-efficient. Speedup is relative to the work-efficient algorithm at $p = 1$.

Since the work-efficient algorithm has the same work as the best sequential runtime, all three lines for the DY81 algorithm start at speedup 1 for all $p = 1$. On the other hand, JM92 is not work-efficient, so its speedup with respect to the best sequential algorithm is less than 1 — in this case their speedups at $p = 1$ start at 0.037, 0.0116, and 0.006. The larger n, the larger the difference between their runtimes and the best sequential time. Ultimately, at $p = 1$, JM92 has a worse performance.

However, as we increase the number of processors, DY81 reaches a plateau much more quickly — this is due to its available parallelism being much smaller — it will reach this plateau at $O(n^{0.25})$ processors. For our values of n, that's only 6, 32, and 176 processors — a personal computer with 64 processors would theoretically reach this limit for problem sizes smaller than 1.6×10^7. On the other hand, JM92's performance keeps growing. For $n = 10^9$, the speedup only reaches a plateau at almost 10^7, which is more than 4 orders of magnitude difference compared to DY81.

The vertical lines show the currently available numbers of processors for easy ref-

erence — 64 for commercially available multicore computers and almost 20 million for supercomputers. While DY92 is better at $p = 64$, JM92 is better if using supercomputer parallelism. Therefore, available parallelism needs to be kept in mind when choosing an algorithm — work efficiency is a more important metric for smaller values of p, while available parallelism is more valuable with higher p.

Note that we use the *upper bound* of runtime for computing speedup (see section 3.3) in Figure 6.1. If we were to use the *lower bound* instead, the main difference would be that the transition between the linear growing line and the constant one would be sudden instead of gradual. Each curve would be made out of two distinct straight line segments — one with a positive slope (the work) and a constant one — which intersect at the point where span = work/p.

Running Time with Varying Problem Size

We can also make the reverse comparison: how does problem size affect performance for certain values of p? In Figure 6.2, we plot the running time against the problem size. This is an equally good measure of performance, but in this case, running time

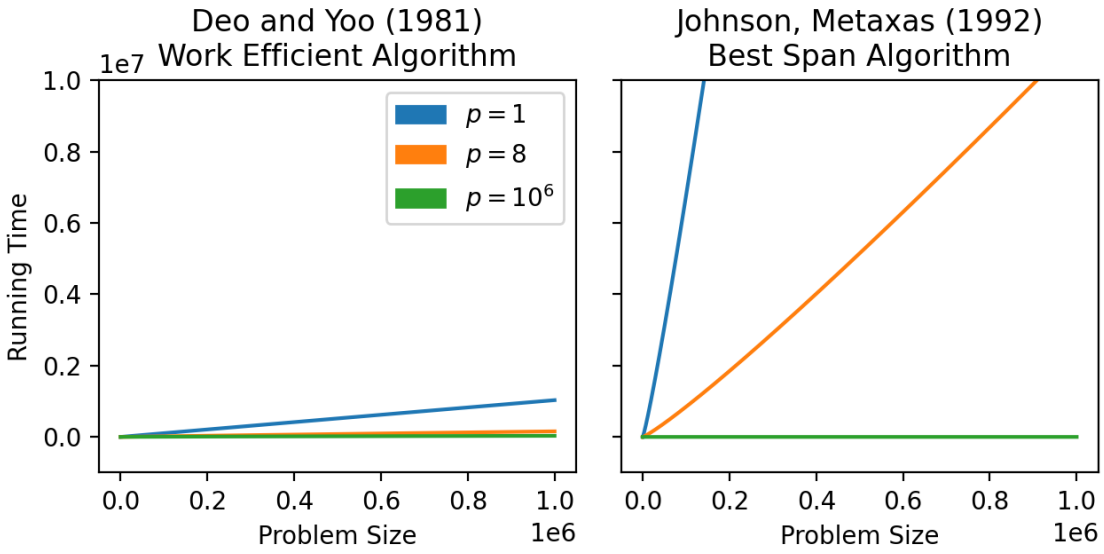

Figure 6.2: Running time as a function of problem size for two different algorithms: a work-efficient one, and one with a better span but not work-efficient.

is more meaningful. Speedups increase when running times decrease, which happens as we increase p; however, increasing n increases runtimes, which leads to negative speedups (when compared to a constant performance, such as for $n = 1$).

DY81, with the runtime $O(n/p + n^{0.75})$, generates lines for $p = 1$ and $p = 8$ that mostly grow as $n^{0.75}$ — since that's the significant part of the runtime for $n > 1$ and $n > 4096$ respectively. Even though the curvature is slight, it can be noticed between the $p = 8$ and $p = 10^6$ lines — since for $p = 10^6$ the runtime is mainly dominated by the n term until $n = 10^8$, it looks more linear.

JM92's runtime, $O(n \log n^{1.5}/p + \log n^{1.5})$, is dominated by the work for the range of values shown in the plot. The slopes of these quasilinear lines decrease with p. This makes JM92 runtime lines more spread out than the DY81, and we can see that for small values of p, they're less efficient.

Running Time with p as a Function of n

Now let's examine what happens when p and n interact, keeping neither constant. In Figure 6.3, we look at running time as we vary the problem size; however, here we investigate p values that depend on n: $p = n^{0.25}$, $p = \sqrt{n}$, and $p = n$.

For the DY81 subplot, all lines are asymptotically $O(n^{0.75})$, the dominating term in the runtime. However, their slopes are different — for $p = n^{0.25}$ the slope is 2, and it approaches 1 as p's complexity function increases.

For JM92, the line shapes are all different. For the $p = n$ line, it looks like it's a constant zero function. It's actually $O(\log n^{1.5})$, which is much smaller than $O(\sqrt{n} \log n^{1.5})$ at large values of n. The $p = n$ line corresponds to runtime equalling span and therefore it's the minimum possible runtime (within a factor of 2).

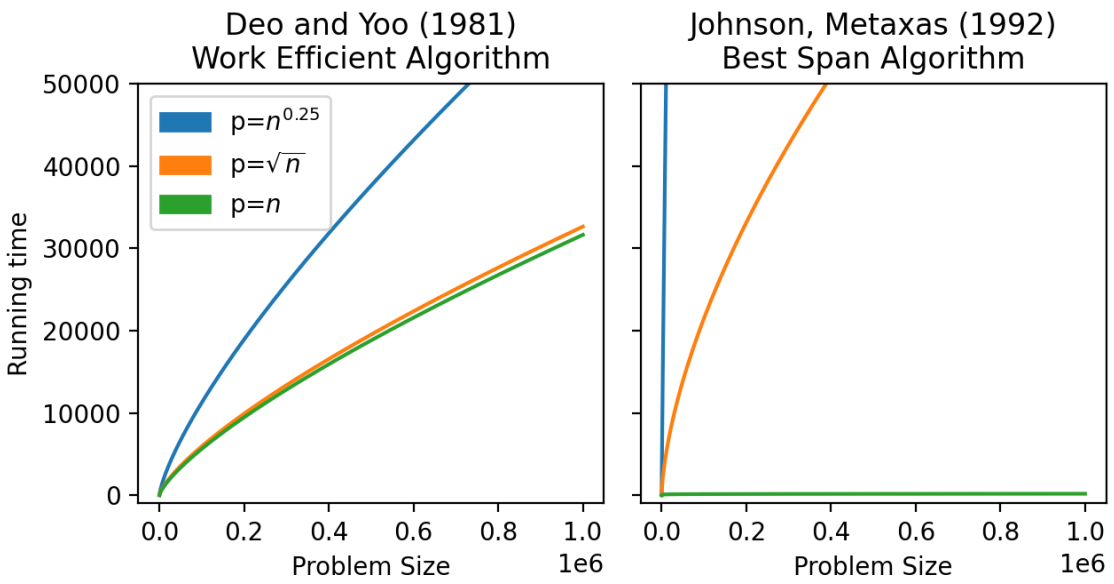

Figure 6.3: Varying the problem size n with the number of processors as a function of n for two different algorithms: a work-efficient one, and one with a better span but not work-efficient.

6.2 Available Parallelism for Best Span Algorithms

As we've seen in the previous section, available parallelism for an algorithm is quite important. So how does available parallelism in general compare for the best span and work-efficient algorithms? Take a look at Figure 6.4, which plots the histogram of available parallelisms for the best work-efficient and lowest-span algorithms for every problem. The y-axis shows the percentage of problem families that whose respective algorithms fall into each complexity class. As opposed to Figure 5.3, where lower complexity classes were better, here we would like to have more parallelism.

As expected, the lowest span histogram is skewed to the right, which tells us that the lowest span algorithms have higher available parallelism.

A curious case is the constant complexity class — a problem being included in this bar implies that the span is asymptotically the same as the work, and few parallel algorithms have that. Consequently, all problems that don't have a work-efficient *parallel* algorithm fall into this category, and a big part of it is sequential algorithms

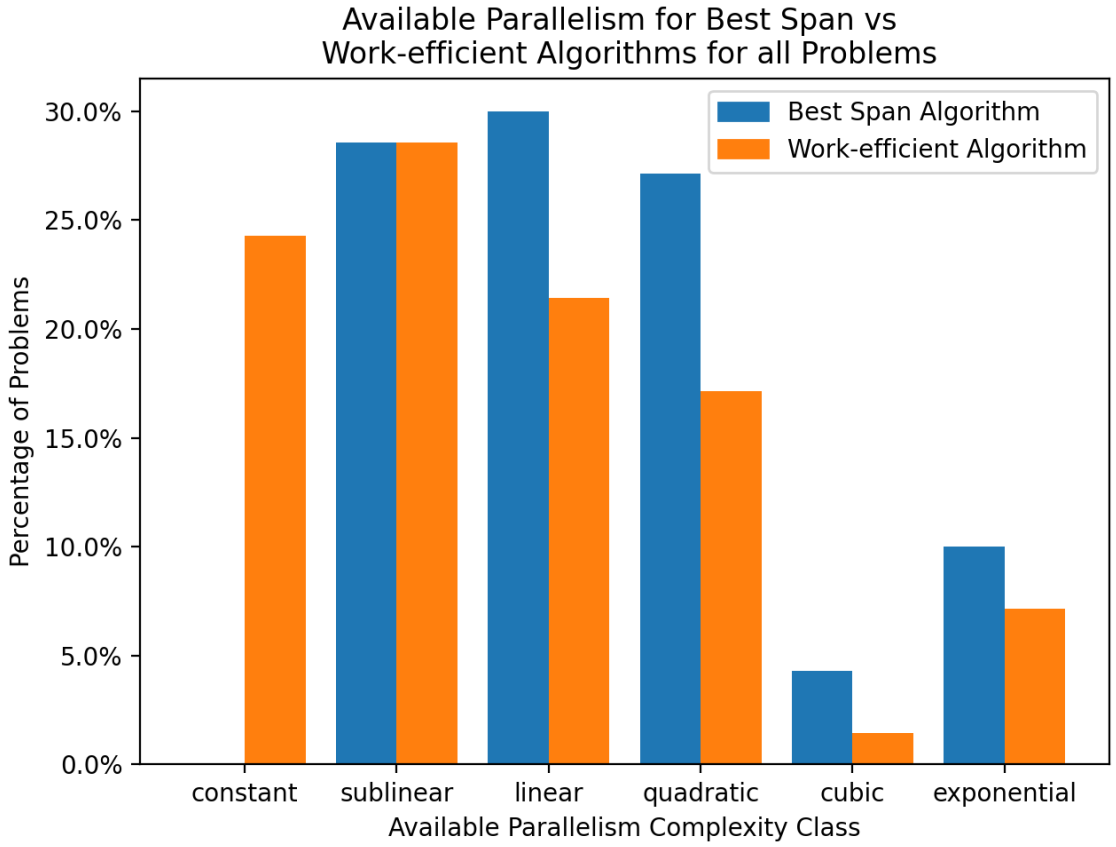

Figure 6.4: Distribution of available parallelism for best span and for work-efficient algorithms

for problems where no parallel algorithm has achieved the same work.

In practice, available parallelism matters only in some contexts — an algorithm with high available parallelism isn't very useful on machines with few processors trying to solve large problems.

Chapter 7

Unexpected Results

We've left some results that we found surprising for this last chapter. We start with a more "social" perspective in which we want to see what researchers are most working on — span and work are both measures of complexity for parallel algorithms, but which one gets improved on more often?

7.1 Work vs Span Improvement

Figure 7.1 shows the distribution of all algorithms, categorizing them based on whether they improve or not upon the best span and best work at the time. In practice, that means that we compare it to the previous year. Works and spans are *better* if they have an asymptotically strictly smaller complexity; the same and worse cases are defined similarly (equal and strictly greater complexity respectively). Note that the spans and works are compared *independently* to each other. Therefore, for example, an algorithm that pushes the Pareto frontier but does not improve upon either the best span or the best work would be categorized as worse-worse and would fall into the lower left square.

We see that the categories of worse span with same and worse work are by far the most common with roughly 28.6% and 23.9% respectively. On the other hand, improvements in complexity seem to be more rare — with roughly 24.2% improving at least one of span and work, and only 3.5% improving both. This might strike the

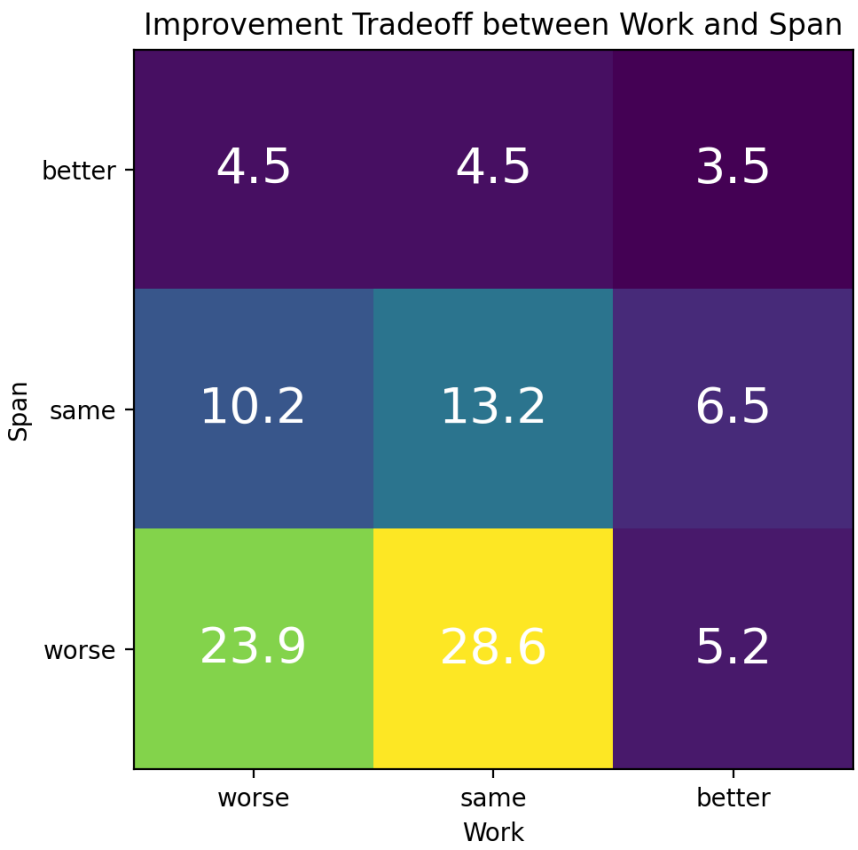

Figure 7.1: Improvement distribution between span and work

reader as surprising. We give a few reasons for why that's the case, besides it being hard to design better algorithms.

There are other worthwhile reasons to design and publish new algorithms besides improving complexity. We've collected all kinds of algorithms, not just the ones that make a complexity improvement over the best existing metric. Some common qualities that are being improved on in such algorithms include smaller constants, less communication (for distributed-memory models), simpler algorithms, etc. Occasionally, it can also be the case that researchers aren't aware of existing better results, although that's been less common in recent decades, thanks to easier access to research and digital tools.

Additionally, as explained above, algorithms that push the Pareto frontier can fall into any of the squares. **198** (**40.74%**) of the algorithms in our database have

pushed the Pareto frontier at their time of publication. One may wonder how much of that 76% of seemingly non-improving algorithms from Figure 7.1 still make a Pareto frontier improvement. To answer that question, we look at Figure 7.2, for which we ran a similar analysis, but we restricted it to only include algorithms that had pushed the Pareto frontier at the time of their publishing.

First, we notice that this figure looks almost completely reversed. In fact, the raw *numbers* (not proportions) of the 5 squares with at least one better measure have stayed the same. This is because an algorithm that has improved over the best span and/or the best work has most certainly extended the Pareto frontier to that new measure. We can get each square's Figure 7.2 proportions by multiplying its Figure 7.1 proportion by 402/114, the ratio between the number of algorithms considered for the first and the second figures. Notice that for both scenarios, 84 of the algorithms

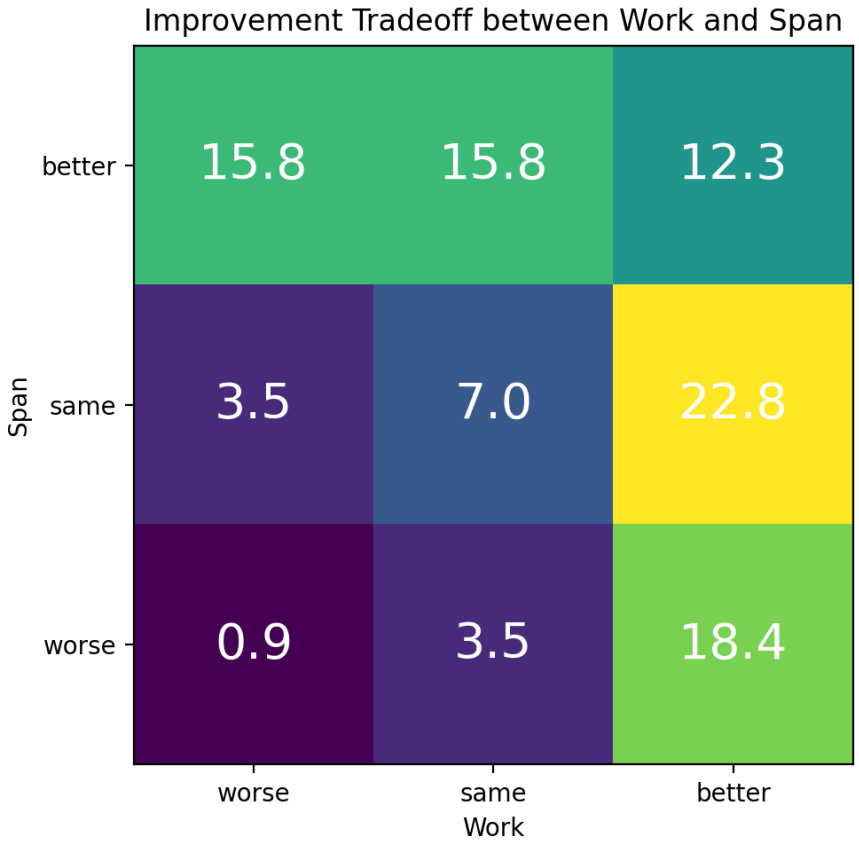

Figure 7.2: Improvement distribution between span and work, only for algorithms that pushed the Pareto frontier at the time of their publishing

aren't included because, at the time of their publishing, there was no other parallel algorithm to compare them to.

Only 17 Pareto frontier algorithms (14.9%) do not improve on either the best span or the best work. That means that 4.2% of the algorithms have a Pareto frontier improvement, but are in the lower left 4 squares in Figure 7.1. That leaves 288 (or 71.6%) of the algorithms that don't improve complexity in any of the ways we've discussed.

As for the work versus span question, it seems like there are a few more algorithms improving work — some possible causes for that might be that researchers recognize work as being slightly more important to improve on, or just that span improvements happen with bigger strides at a time.

7.2 Improvement in Performance and Span

Work is important, but ultimately the most practically useful measure of how an algorithm performs is its runtime. How often does it get improvements? We present Figure 7.3, where for 9 different combinations of problem size n and number of processors p, we examine the performance of all algorithms in our database and compare it to their spans. An algorithm's runtime can be classified as worse, same, or better than the best runtime existing at the time just as well as span and work; however, in this case, we must resort to using numerical.

As a reminder, for this section, we're using the *lower bound* definition of runtime:

$$\text{runtime} = \max\{\text{work}/p, \text{span}\}$$

Based on this definition, we can see two cases: when the work/p term dominates (is greater than the span), which we call **work-bound**, and when the span dominates the runtime, which we call **span-bound**.

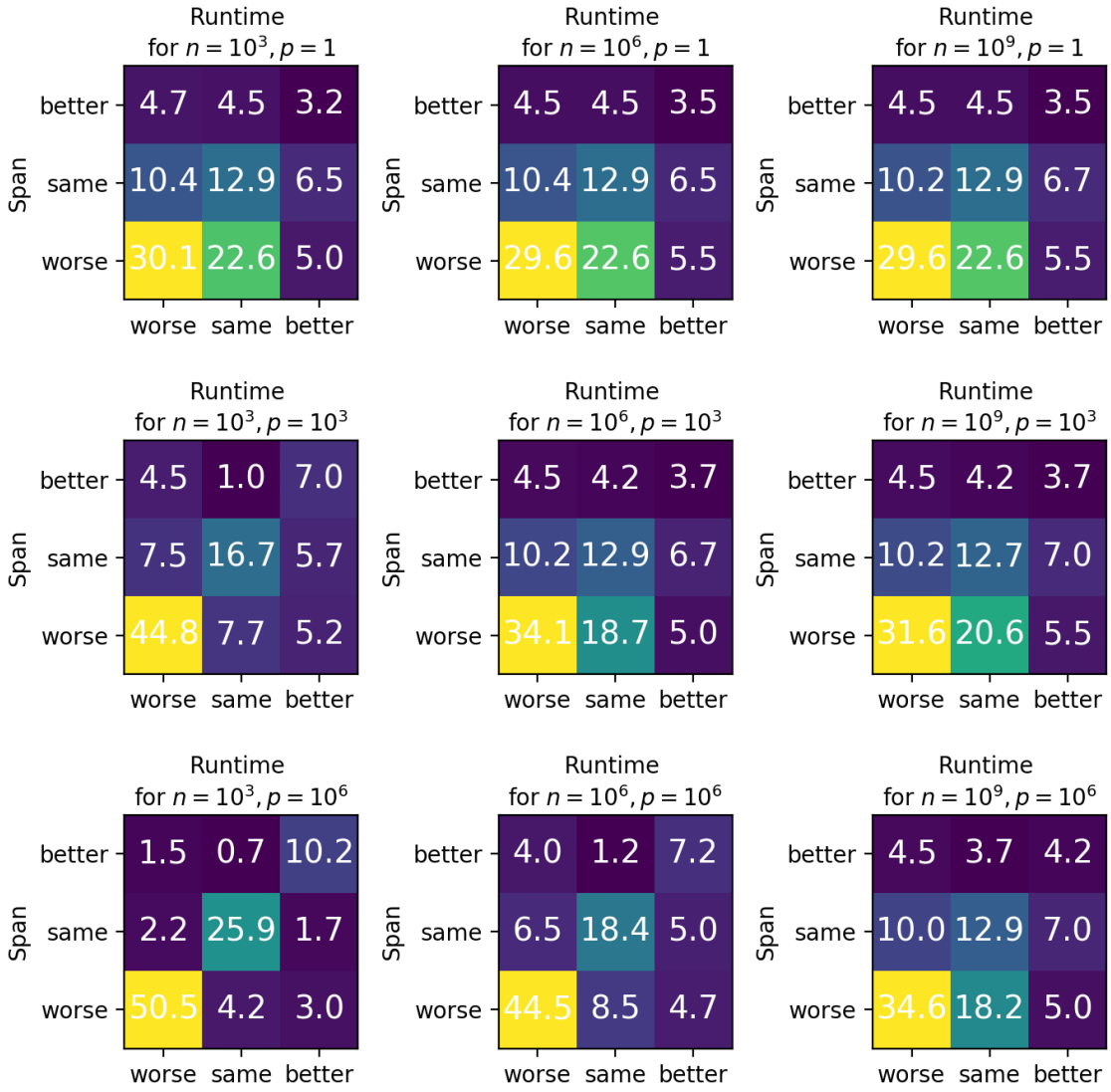

Figure 7.3: Improvement distribution based on span and runtime, computed for 9 different combinations of problem size n and number of processors p

First of all, the totals for each span category row stay consistent, because the span isn't calculated numerically in terms of n and p. This means that the only changes we see across each of the subplots consist of algorithms being moved from one bucket to another in the same row. Notice that as we *increase* the number of processors p and *decrease* the problem size n, we see it has different effects on the three different span category rows.

For the top span category row, corresponding to better span, we see algorithms

get moved right, towards better runtime; for the middle row, "same span", both worse and better runtime algorithms tend to combine into the "same runtime" category; and for the bottom "worse span" row, algorithms get classified more frequently as having worse runtimes. This looks very interesting, but there's a good reason for it to be this way.

Because we compute an algorithm's runtime by taking the dominating factor out of work/p and span, we need to look at how varying p and n affect both of these terms in relationship to each other. Increasing p means that work/p gets smaller while span doesn't change; therefore, span is likelier to become the dominating factor. We see the same happening when we *lower* the problem size n — in that case, both work and span decrease, which makes p have a bigger influence on the work/p term, and therefore it has a similar effect to increasing p by becoming work-bound.

At this point, we'd like to draw the reader's attention to how this figure compares to Figure 7.1. Since work is the asymptotic runtime at $p = 1$ as $n \to \infty$, we conclude that Figure 7.1 is the natural continuation of the **top** row (for $p = 1$) to the **right**. In fact, all subplots in the top row are the same: for $p = 1$, the runtime becomes $\max\{\text{work}, \text{span}\}$, and no matter the value of n, and since work $\geq$ span, the calculated runtime is the same as the work.[1]

On the opposite end, letting p grow to infinity to the **bottom left** reduces the work/p term to the point that span becomes the dominating factor. Therefore, as suggested by the $n = 10^3$, $p = 10^6$ subplot, runtime and *span* will be equal, and all but the diagonal squares will be equal to 0, and the diagonal squares will show the totals for the span improvement categories.

This ties in back to the parallelism discussion from the last chapter: the increasing n has the opposite effect of increasing p, which leads to the two opposite corners being the two extremes.

[1]The subplots in the top row of Figure 7.3 have slightly different values from the ones in Figure 7.1, contrary to the theoretical argument presented here. This difference however stems from the fact that for Figure 7.3, runtime is calculated *numerically*, while in Figure 7.1, we compare work asymptotic complexities. For example, $O(\log^c n) \in O(n^\epsilon)$ for an arbitrarily small ϵ and arbitrarily large constant c, however when computing them with the parameters that we've chosen ($\epsilon = 0.01$, $c = 6$), we get the opposite $O(\log^c n) > O(n^\epsilon)$ for the relatively small values of n we're considering.

The version of this figure using only Pareto algorithms undergoes similar changes, which is why we don't include it here. The curious reader can find it in Appendix D.

7.3 Annual Progress

Lastly, we'd like to see how much yearly progress has been made. We measure yearly progress with the **compound growth rate** (CGR).

The **compound growth rate** is computed as

$$\text{CGR} = (f/b)^{1/t} - 1$$

where f is the present-day best parallel speedup, b is the best sequential speedup, both with respect to the first algorithm for the problem and t is the number of years since the first parallel algorithm.

We've computed the compound growth rate for each problem for 9 different combinations of n and p, and then we plotted their distributions in Figure 7.4.

We see that the number of processors has the biggest impact on the CGR — the higher p, the more the distribution skews right towards bigger CGRs. We notice that this effect is bigger with bigger n. When increasing p, the only part of the CGR equation that changes is the parallel speedup. Since speedup increases linearly with p, all CGRs have to increase as p increases.

Similarly, but to a smaller extent, increasing n skews the distribution towards higher CGR. The reasoning behind this is slightly more complicated, as now both f and b increase with n. However, we know that $f > b$, which means that f increases slightly faster than b, which leads to the ratio f/b also increasing as n increases.

In general, the compound growth rates for parallel algorithms stay under 30%, even for $n = 10^9$ and $p = 10^6$. This is below the hardware improvement rate as calculated using the SPECInt benchmark. [56, 10] We can also compare the growth rates with the sequential ones given in [10]. In general, parallel improvement rates are worse than their sequential counterparts for low numbers of processors. This is

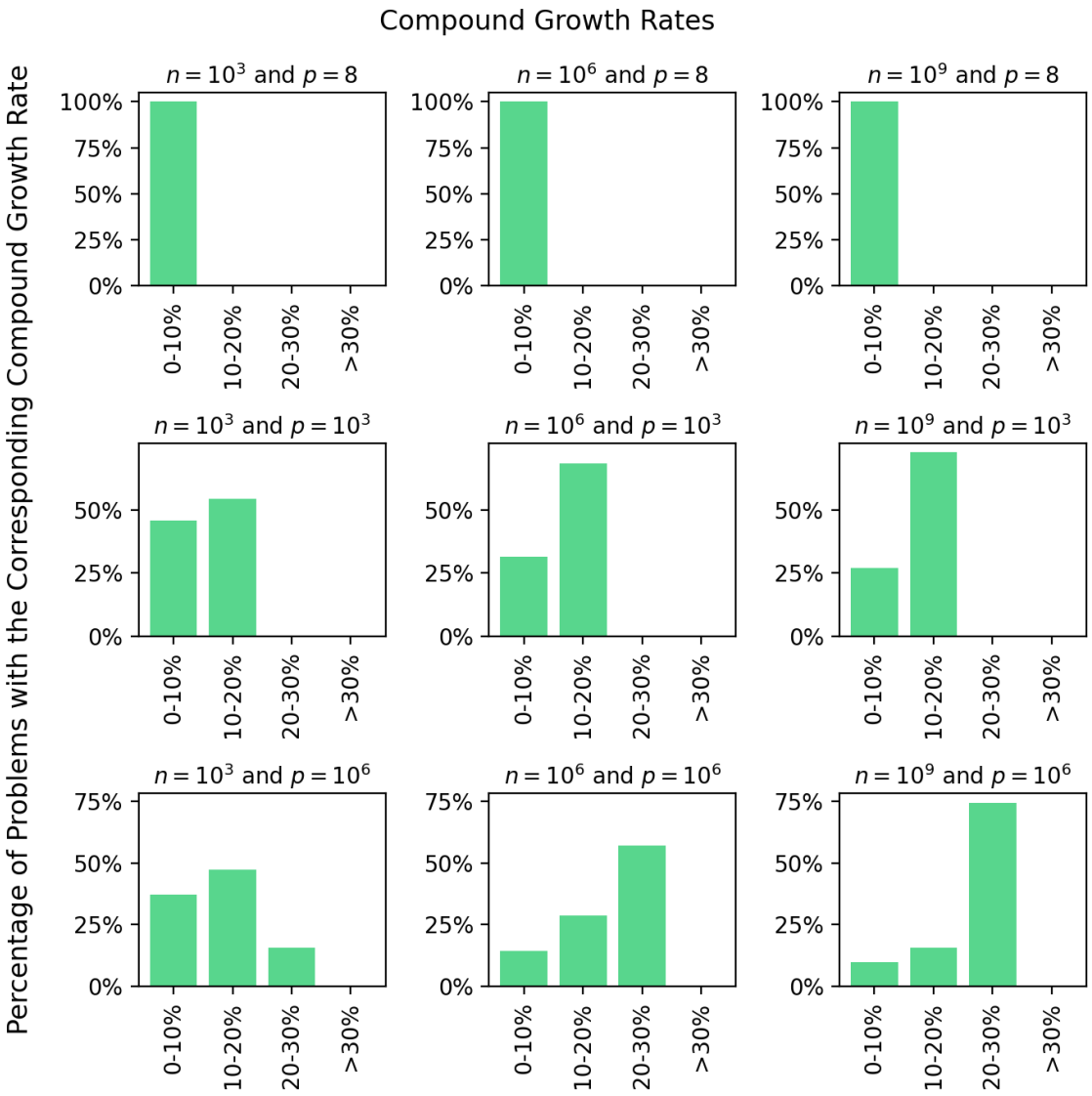

Figure 7.4: Compound Growth Rate distribution

because, at such low — almost constant — values of parallelism, the parallel running times are very close to the sequential running times, so f/b is very close to 1. This explains why for $p = 8$ all the problems have CGRs of under 10%.

For both $p = 10^3$ and $p = 10^6$, we can see bigger differences with respect to sequential improvement rates — the majority category of the parallel problems is always either 10-20% or 20-30%, while for sequential problems, it's always $< 10\%$ for all values of n. On the other hand, a significant number of sequential problems have improvements that go above the hardware average improvement rate — 16%, 27%,

and 37% for each of $n = 10^3$, $n = 10^6$, and $n = 10^9$ respectively. [10]

Overall, one can see that the compound growth rates for parallel problems tend to not vary as much as the sequential improvement rates; however, on average they tend to get higher growth for large enough values of p than their sequential counterparts.

Chapter 8

Conclusion

It's important to know what parallel algorithms have to offer. In this thesis, we've provided a quantitative assessment of this field and how it evolved since its inception. We hope this is useful to anyone who wants to understand the state of the art of parallel algorithms. We also hope that this thesis provides more rigorous quantitative evidence to supplement any of the reader's empirical observations, whether it proves or disproves any such acquired preconceptions. We aimed to provide a landscape of what parallel algorithms can do at their best, worst, and across the whole range. On its most basic level, the goal of this work is to provide more knowledge and a deeper understanding of the state of parallel algorithms.

Parallel algorithms have had significantly more progress than sequential ones since the '80s. Hardware has also improved: commercially available computers have up to 64 cores, and the best supercomputers in the world have up to almost 20 million. Assuming these numbers of processors, most parallel algorithms have had improved speedups when compared to sequential algorithm speedups. For the All Pair Shortest Paths problem, the best speedup achievable by a supercomputer is 9 million higher than the best sequential speedup (for a problem size of 1 million), both relative to the first known algorithm. Similarly, all problems with parallel algorithms have parallel speedups greater than sequential ones. Overall, progress in sequential algorithms accounts for only a quarter of the progress, compared to the maximum available speedup due to parallel algorithms.

Since parallel algorithm performance is measured by both span and work, this gives rise to potential trade-offs. Sometimes there's no single algorithm that has both the best span and the best work for a given problem. In fact, more than a third of our problems have such nontrivial trade-offs, a proportion that has slightly decreased since the 60s. This gives rise to the two extremes: algorithms with the best span, and algorithms that are work-efficient (having the same work as the best sequential algorithm). In general, the spans are better for the best span algorithms than their work-efficient counterparts, which makes them faster with enough processors. The downside however is that such best span algorithms have non-constant work overheads. This makes them not as advantageous with few available processors.

Another way to look at the same trade-off is through available parallelism (more with better span) and work overhead (lower with better work). Best span algorithms generally have more available parallelism than work-efficient ones. Performance depends on both work overhead and available parallelism: for small values of the number of processors p, the work-efficient algorithms have better speedups; but that quickly changes as we increase p to values above those on the scale of commercially available personal computers — that's when the best span algorithms get significantly better speedup for the same problem size. In general, with small numbers of processors, work efficiency is more important; however, with increased processor numbers, smaller spans give us bigger gains. It's important to know what system one is designing algorithms for, to know what to prioritize. This knowledge can help with choosing an algorithm that better suits the system's purpose.

We've also had some unexpected results. Annual progress for parallel algorithms, as expressed by the compound growth rate, doesn't exceed 30% for common values of p and n, which is smaller than the annual progress of hardware and sequential algorithms. In addition, only about 40% of the algorithms have improved at least one of span or work. More research could be done into the other issues researchers try to improve when designing parallel algorithms, such as locality, constants, communication, and so on.

Exploring other issues is just one potential direction of future work. Other di-

rections to be explored are other types of algorithms that we've excluded from our analysis. In this thesis we've limited the scope to consider only specific algorithms: we haven't included approximate, heuristic, GPU-based algorithms, etc. We also plan to expand our algorithms database and will do so before the updated paper is published.

While the presented theoretical gains of parallel algorithms are impressive, in practice they're not implemented to their fullest extent. This is for many reasons, the main one being the difficulty of writing correct and efficient parallel code. This is yet another part of the field that would benefit from more research. We have many good parallel algorithms, thanks to the amazing progress throughout the last few decades. Coming up with ways to make parallel implementation easier and more widely available would help bring more of these theoretical benefits into reality.

Parallel computing has progressed significantly throughout the years, and we can't wait to see how the field will continue to advance.

Appendix A

Model Statistics

Keeping in mind the limitations outlined in Chapter 3, we can look at what the distribution of models looks like for the algorithms we've collected. While this doesn't tell us much about progress in parallel algorithms per se, it's very interesting to analyze the use and interest of various types of models and their historical trends.

Figures A.1 and A.2 show the distribution of the models for the algorithms we've collected over the decades. It's worth noting that data for the "Other" model category is not very reliable, since we haven't tried collecting all the algorithms with such models.

One of the first interesting facts one notices while looking at both of these figures is that the PRAM models seem to be the most popular. They've been widely used throughout the '80s and the '90s. Their distribution is also fascinating — CREW and CRCW seem to have been more popular a little earlier on, and EREW only got more of a following at the end of the '80s — based on Figure A.2 it looks like EREW replaced a significant part of CREW algorithms. Around this time, lots of papers started mentioning the practicality of EREW when compared to CRCW. It seems to not have been a very long-lasting trend though, as EREW algorithms went extinct after 2010, unlike their CREW and CRCW counterparts.

Looking at other models, shared-memory SIMD models were the first to make a big wave; however, that was short-lived until the PRAM models took over to replace

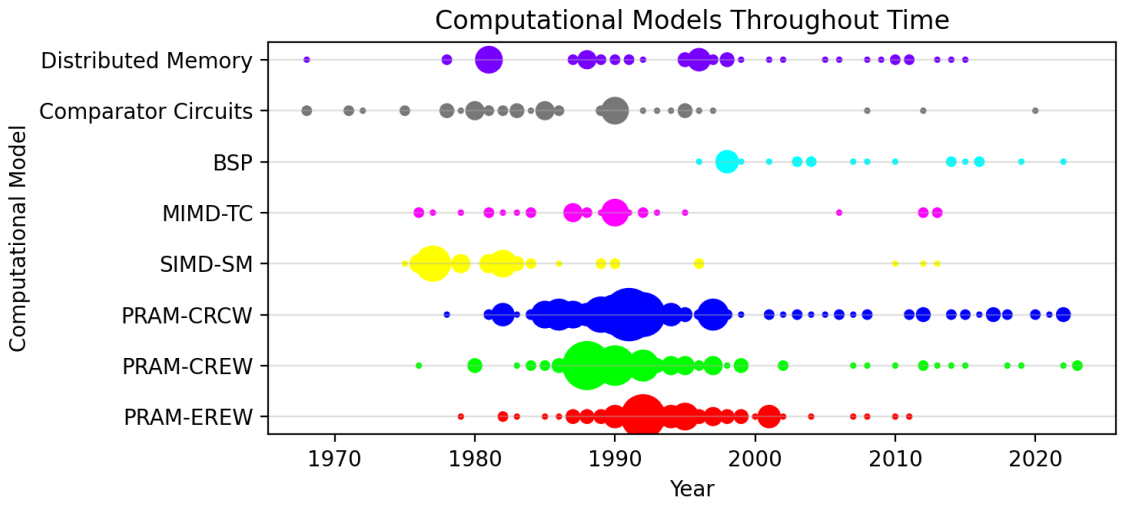

Figure A.1: Utilization of each model over time.
Each dot represents one or more algorithms using that model which was published that year; the size of the dot indicates how many algorithms of that model were published that year

SIMD. Tightly-coupled MIMD has had a relatively more consistent presence in that period, albeit slightly smaller. Consistency can also be seen with BSP models after 1995; it was the last big model to appear in Valiant's 1990 paper. [29]

The earliest were comparator circuits, models that are especially useful for the problem of sorting a given list of elements. As one of the most ubiquitous problems in computer science, it's understandable why it was one of the first to be explored in the context of parallelism.

Figure A.2 shows the number of algorithms using the models per decade. We can see that the '80s and '90s were the most active period for new parallel algorithms, with 185 and 156 algorithms respectively. It coincides with the period of increased advances in building parallel computers. Since then it's been steadily slowing down, partly because progress has been made in most problems where progress was more easily achievable, and partly because the focus has shifted to working on more recent technologies.

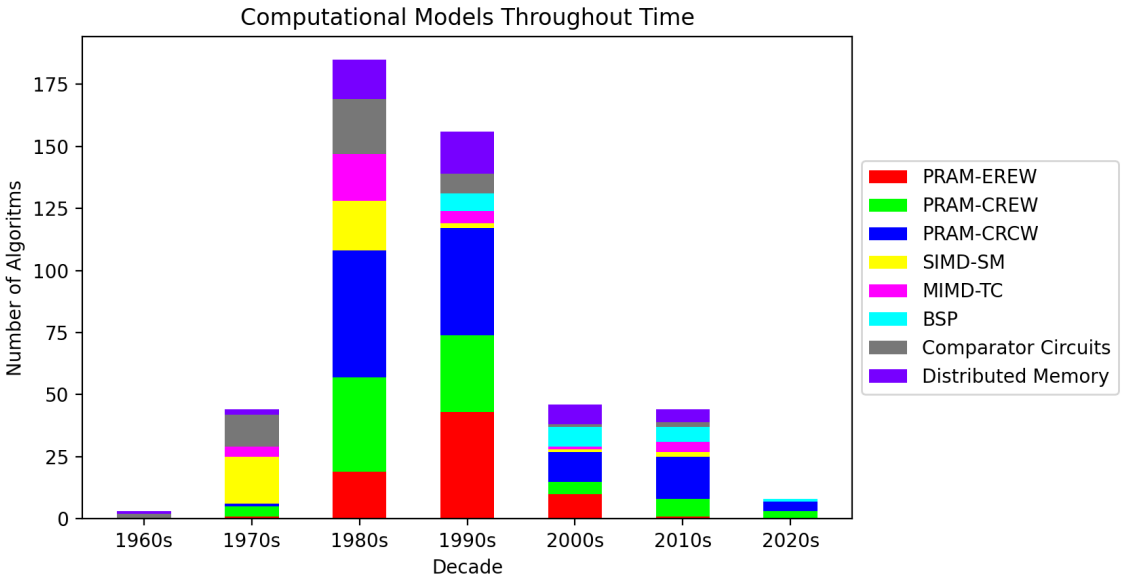

Figure A.2: Distribution of models used over time

It should be noted that while the bar for the 1960s is small, that's because parallel computers had only been brought to attention recently — the first algorithm in our database is from 1968. A similar argument applies to the last bar, since at the time of writing there have only been 3 years in this decade.

Appendix B

List of Problem Families with No Parallel Algorithms Found

Informed Search

Joins

NFA to DFA conversion

Key Exchange

Mutual Exclusion

Minimum value in each row of an implicitly-defined totally monotone matrix

Register Allocation

Dependency Inference Problem

BCNF Decomposition

4NF Decomposition

Cryptanalysis of Linear Feedback Shift Registers

Longest Palindromic Substring

AST to Code Translation

Graph Realization Problems

Duplicate Elimination

Hyperbolic Spline Interpolation

Maximum Likelihood Methods in Unknown Latent Variables

Filtering Problem (Stochastic Processes)

All Maximal Non-Branching Paths in a Graph

Distributed Locking Algorithms

Cyclic Peptide Sequencing Problem

Clock Synchronization in Distributed Systems

Wiener Index

Rod-Cutting Problem

Turnpike Problem

Median String Problem

Frequent Words with Mismatches Problem

Tower of Hanoi

The Frequent Words Problem

d-Neighborhood of a String

Change-Making Problem

Secret Sharing

Page Replacements

Recovery

Gröbner Bases

Appendix C

Parameters

Here are the non-problem size parameters we encountered most, along with the value we set them to, as a function of the problem size n.

	Parameters	**Value**	**Applicable Problem(s)**
ϵ	arbitrarily small number	0.01	all problems
c	highest log exponent for polylogarithmic expressions	6	all problems
$\|E\|$	number of edges	$\|V\|^2 = n$	all graph problems
d	depth of the graph	$\|V\|$	all graph problems
U	maximum edge capacity	$2^{\|V\|} = 2^{\sqrt{n}}$	Maximum Flow
d	number of dimensions	n	Linear Programming, Convex Hull, Closest Pair Problem
k	number of points of intersection	n^2	Line segment intersection
h	number of points on the convex hull	n	Convex Hull
m	length of shorter string	n	Sequence Alignment, Longest Common Subsequence
	length of the LCS	n	Longest Common Subsequence
m	length of pattern	n	String Search
m, k	m x k and k x n matrices	n	Matrix Product

	Parameters	**Value**	**Applicable Problem(s)**
d	maximum degree	n	Single Source Shortest Paths
d_c	the maximum weight of a shortest path	2^n	Single Source Shortest Paths
L	maximum magnitude of lengths	2^n	Single Source Shortest Paths
N	integer to be factorized	$\log n$	Integer Factoring
k	size of alphabet	$O(1)$	DFA Minimization
n	number of vertices	n	Clique Problems
M	number of cliques	n	Clique Problems
w_{min}	minimum weight	2^n	The Subset-Sum Problem
w_{max}	maximum weight	2^n	The Subset-Sum Problem
t	target sum	2^n	The Subset-Sum Problem
Δ	maximum number of acceptable partners for one participant	n	Stable Matching Problem
$\|G\|$	grammar size	$O(1)$	CFG Problems

Table C.1: Values for commonly-used parameters

Appendix D

Improvement in Runtime and Span for Pareto Frontier Algorithms

We present the plot analyzing improvement quality in span and work in Figure D.1 on the next page. The discussion in section 7.2 applies to this graph as well.

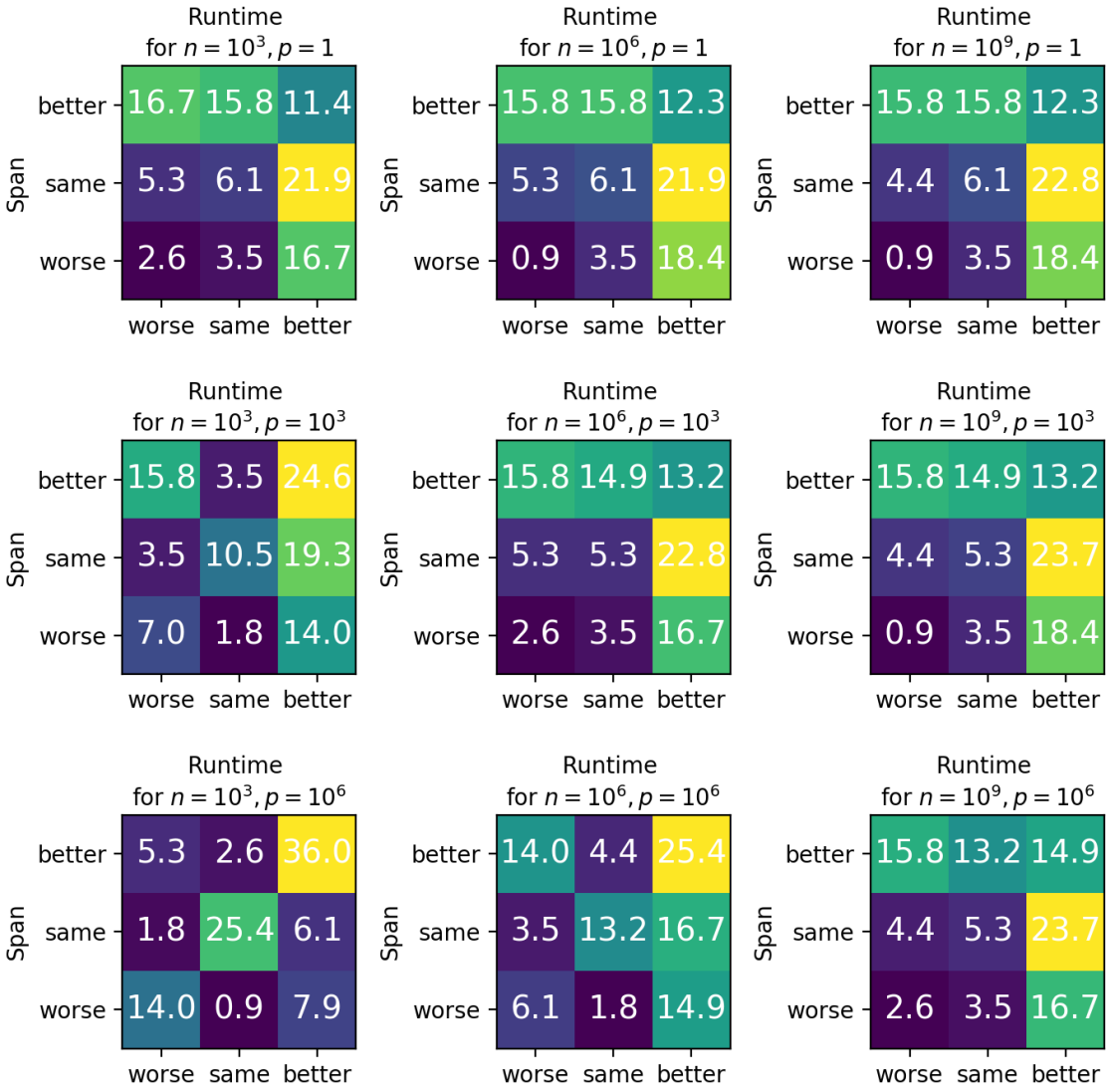

Figure D.1: Improvement in running time and span for Pareto frontier algorithms only

Bibliography

[1] "TOP500 Dataset." `www.top500.org`, 2023. Accessed: 2023-12-07.

[2] "Wikichip." `en.wikichip.org/wiki/WikiChip`, 2024. Accessed: 2024-01-07.

[3] L. F. Richardson, *Weather prediction by numerical process*. University Press, 1922.

[4] W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, J. Randall, A. H. Sameh, and D. L. Slotnick, "The Illiac IV system," *Proceedings of the IEEE*, vol. 60, no. 4, pp. 369–388, 1972.

[5] B. A. Kahle and W. D. Hillis, "The Connection Machine model CM-1 architecture," *IEEE transactions on systems, man, and cybernetics*, vol. 19, no. 4, pp. 707–713, 1989.

[6] T. Machines, "Connection Machine model CM-2 technical summary," *Thinking Machines Technical Report HA87-4*, 1987.

[7] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, *et al.*, "The network architecture of the Connection Machine CM-5," in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pp. 272–285, 1992.

[8] "Gyoukou - ZettaScaler-2.2 HPC system, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 700Mhz — TOP500 — top500.org." `https://www.top500.org/system/179102/`. [Accessed 16-08-2023].

[9] G. E. Moore, "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp. 114 ff.," *IEEE solid-state circuits society newsletter*, vol. 11, no. 3, pp. 33–35, 2006.

[10] Y. Sherry and N. C. Thompson, "How fast do algorithms improve?[point of view]," *Proceedings of the IEEE*, vol. 109, no. 11, pp. 1768–1777, 2021.

[11] E. Liu, "A metastudy of algorithm lower bounds," Master's thesis, Massachusetts Institute of Technology, 2021.

[12] E. Liu, Y. Sherry, W. Kuszmaul, J. Lynch, and N. C. Thompson, "How close are algorithms to being optimal?," *Working Paper*, 2023.

[13] H. Rome, "The space race: Progress in algorithm space complexity," Master's thesis, Massachusetts Institute of Technology, 2023.

[14] H. Rome, J. Lynch, J. Li, C. Falor, and N. C. Thompson, "How fast are algorithms reducing the demands on memory? A survey of progress in space complexity," *Working Paper*, 2024.

[15] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's plenty of room at the Top: What will drive computer performance after Moore's law?," *Science*, vol. 368, no. 6495, p. eaam9744, 2020.

[16] D. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: A hardware/software approach*. Gulf Professional Publishing, 1999.

[17] Y. Shiloach and U. Vishkin, "An $O(n^2 \log n)$ parallel max-flow algorithm," *Journal of Algorithms*, vol. 3, no. 2, pp. 128–146, 1982.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[19] R. M. Karp and V. Ramachandran, "Chapter 17 - Parallel algorithms for shared-memory machines," in *Algorithms and Complexity* (J. VAN LEEUWEN, ed.),

Handbook of Theoretical Computer Science, pp. 869–941, Amsterdam: Elsevier, 1990.

[20] G. E. Blelloch, *Vector models for data-parallel computing*, vol. 2. Citeseer, 1990.

[21] V. R. Pratt, M. O. Rabin, and L. J. Stockmeyer, "A characterization of the power of vector machines," in *Proceedings of the sixth annual ACM symposium on Theory of computing*, pp. 122–134, 1974.

[22] G. Blelloch and J. Greiner, "Parallelism in sequential functional languages," in *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pp. 226–237, 1995.

[23] G. E. Blelloch, "Programming parallel algorithms," *Communications of the ACM*, vol. 39, no. 3, pp. 85–97, 1996.

[24] M. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[25] A. Bar-Noy and S. Kipnis, "Designing broadcasting algorithms in the postal model for message-passing systems," in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pp. 13–22, 1992.

[26] F. T. Leighton, *Introduction to parallel algorithms and architectures: Arrays· trees· hypercubes*. Elsevier, 2014.

[27] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," *Computer*, vol. 24, no. 8, pp. 52–60, 1991.

[28] B. H. Juurlink and H. A. Wijshoff, "The parallel hierarchical memory model," in *Algorithm Theory—SWAT'94: 4th Scandinavian Workshop on Algorithm Theory Aarhus, Denmark, July 6–8, 1994 Proceedings 4*, pp. 240–251, Springer, 1994.

[29] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[30] Z. Ahmad, R. Chowdhury, R. Das, P. Ganapathi, A. Gregory, and M. M. Javanmard, "Low-span parallel algorithms for the binary-forking model," in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 22–34, 2021.

[31] P. B. Gibbons, Y. Matias, and V. Ramachandran, "Can shared-memory model serve as a bridging model for parallel computation?," in *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pp. 72–83, 1997.

[32] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken, "LogP: Towards a realistic model of parallel computation," in *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 1–12, 1993.

[33] J. JaJa and K. W. Ryu, "The block distributed memory model for shared memory multiprocessors," in *Proceedings of 8th International Parallel Processing Symposium*, pp. 752–756, IEEE, 1994.

[34] R. M. Karp, M. Luby, and F. Meyer auf der Heide, "Efficient PRAM simulation on a distributed memory machine," in *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pp. 318–326, 1992.

[35] L. G. Valiant, "Parallelism in comparison problems," *SIAM Journal on Computing*, vol. 4, no. 3, pp. 348–355, 1975.

[36] S. Aaronson, "The power of the Digi-Comp II: My first conscious paperlet," Dec 2016.

[37] N. Nishimura, "Efficient asynchronous simulation of a class of synchronous parallel algorithms," *Journal of Computer and System Sciences*, vol. 50, no. 1, pp. 98–113, 1995.

[38] M. Raynal and M. Raynal, "Simulating synchrony on top of asynchronous systems," *Distributed Algorithms for Message-Passing Systems*, pp. 219–244, 2013.

[39] B. Awerbuch, "Complexity of network synchronization," *J. ACM*, vol. 32, p. 804–823, oct 1985.

[40] C. Martel, R. Subramonian, and A. Part, "Asynchronous PRAMs are (almost) as good as synchronous PRAMs," in *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pp. 590–599, IEEE, 1990.

[41] R. Subramonian, "Designing synchronous algorithms for asynchronous processors," in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pp. 189–198, 1992.

[42] T. Hagerup and T. Radzik, "Every robust CRCW PRAM can efficiently simulate a Priority PRAM," in *Proceedings of the second annual ACM symposium on Parallel algorithms and architectures*, pp. 117–124, 1990.

[43] O. Aguilar, A. Datta, and S. Ghosh, "Simulating shared memory in message passing model," in *[1991 Proceedings] Tenth Annual International Phoenix Conference on Computers and Communications*, pp. 232–238, 1991.

[44] J. E. Savage, *Models of Computation: Exploring the Power of Computing*. USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1997.

[45] J. JáJá, *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., 1992.

[46] S. G. Akl, *The design and analysis of parallel algorithms*. Prentice-Hall, Inc., 1989.

[47] Y. Sun, N. B. Agostini, S. Dong, and D. Kaeli, "Summarizing CPU and GPU design trends with product data," *arXiv preprint arXiv:1911.11313*, 2019.

[48] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, p. 345, June 1962.

[49] M. L. Fredman, "New bounds on the complexity of the shortest path problem," *SIAM Journal on Computing*, vol. 5, no. 1, pp. 83–89, 1976.

[50] C. Savage, "Parallel algorithms for graph theoretic problems," *University of Illinois Coordinated Science Laboratory Report UILU-ENG-77-2231*, 1977.

[51] Y. Han, V. Pan, and J. Reif, "Efficient parallel algorithms for computing all pair shortest paths in directed graphs," in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pp. 353–362, 1992.

[52] T. Takaoka, "A new upper bound on the complexity of the all pairs shortest path problem," *Information Processing Letters*, vol. 43, no. 4, pp. 195–199, 1992.

[53] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *J. ACM*, vol. 21, p. 168–173, jan 1974.

[54] M. J. Quinn and N. Deo, "Parallel graph algorithms," *ACM Comput. Surv.*, vol. 16, p. 319–348, sep 1984.

[55] D. B. Johnson and P. Metaxas, "A parallel algorithm for computing minimum spanning trees," in *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '92, (New York, NY, USA), p. 363–372, Association for Computing Machinery, 1992.

[56] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.