Department of Computer Science

Submitted in part fulfilment for the degree of BSc.

Turing Machine Networks

Jack Romo

30th April 2019

Supervisor: Dr. Detlef Plump

For my beloved cat Pooh, whose unparalleled research skills would surely have benefited this project, had he not been asleep throughout.

Acknowledgements

I now take this time to extend my gratitude to my supervisor, Detlef Plump, for his invaluable experience, insight and knowledge in theoretical computer science - without such guidance, I sincerely doubt this project would have been as satisfying as it has turned out to be. I also thank my friends for their everlasting support, and my parents for both their love and for giving me the opportunities I now have the privilege to take.

Contents

List of Figures

Executive Summary

Introduction

The importance of parallelism is difficult to overstate. As sequential computers have failed to meet modern computational challenges, engineers have begun to distribute computations across many processors, making prior algorithms immensely faster. Such a prevalent concept demands modelling so its power can be best understood. Many diverse models have been developed, ranging from pragmatic models like BSP [1] and LogP [2], to theoretically involved and abstract ones like Parallel Turing Machines [3].

It is unfortunate then that a direct connection of results for classical Turing Machines to parallel complexity theory has seen little effort [4], with the only major model doing this being Parallel Turing Machines [3], proposed by Wiedermann. This model, while well-developed in its own right, only emulates shared memory parallelism, not accounting for message-passing distributed models like MapReduce [5]. This makes it difficult to bring the hefty theory surrounding Turing Machines to bear on message-passing parallelism, with models like BSP and LogP giving little such theory and the $\mathcal{LOCAL}$, $\mathcal{ASYNC}$ and $\mathcal{CONGEST}$ [6] models focusing on communication challenges instead. Thus, it is harder to analyze problems such as how much faster an problem can be solved in a parallel message-passing environment than sequentially by a Turing Machine, a question that debatably defines the efficacy of such an approach to parallelism.

This project aims to produce a model that resolves this conundrum, linking Turing Machines and message-passing parallelism more strongly and providing an environment in which substantial complexity results can be produced. It should be noted that this project will restrict itself to classical computation, not straying into the domain of nonstandard models like quantum computing; these are beyond the scope of this project.

Project Outline

In defining our model for parallel complexity theory, we wish for it to fulfill the following four requirements:

1. The model should intuitively emulate a network of communicating processors without shared memory.
2. The model should allow for meaningful complexity analysis.
3. The model should establish a way to compare sequential complexity classes as defined with Turing Machines and parallel ones.
4. The model should be able to simulate other parallel models efficiently.

To achieve this, the project proceeds in three major steps. First, the model for parallelism is defined in full, after reflection upon the design decisions made by other prevalent models of parallelism. A metric for computational time is defined, and an example is investigated to show the model is sound. The model has intuitively been defined as a network of Turing Machines with extra send/receive transitions, clearly satisfying our first requirement.

Next, complexity theoretic results are pursued to verify that this model can provide meaningful connections between sequential and parallel complexity. Lower and upper bounds on complexity are produced, and their intuitive meanings expanded upon; in particular, it is shown that problems beyond EXPTIME can only be made tractable if the network topology we use is itself intractable to compute. This shows our model can derive powerful results for parallel complexity theory, satisfying our second requirement. In the process, these relations of our model's complexity classes to standard Turing Machine ones verifies our third requirement; this is aided by the fact that our model is a network of Turing Machines itself.

Finally, other models are simulated within this one to establish the model's relevance to others and that its complexity results do indeed translate to other areas of parallelism research, cementing the importance of the model overall. We in particular choose to simulate BSP, a practical model of parallelism, and Boolean Circuits, a model of great theoretical significance. This shows our model can encapsulate both practical and theoretically inclined approaches to parallel modelling, demonstrating its widespread applicability as a model of parallelism.

Overall, the project fulfilled its requirements, proving its versatility as a model for message-passing parallel complexity. Future work remains in investigating dynamic networks and further classifying network topologies.

Ethical Considerations

This project has involved no physical experiments and concerns itself only with abstract reasoning. As such, confidential data has not been collected, no human participants have been involved, and neither its results nor their immediate applications find ethical consequence.

1 Literature Review

We begin by exploring current research into related lines of work, outlining well-understood models of parallel and distributed computation along with their respective approaches to complexity analysis.

1.1 Multicore Processor Models

One of the most well-known models for parallel computation is PRAM. An elaboration upon the earlier RAM (Random Access Machine) model for sequential algorithms by Fortune and Wyllie [7], a PRAM, or Parallel Random Access Machine, is defined as an unbounded set of indexed RAM processors P_0, P_1, P_2... along with a similarly unbounded set of shared memory cells C_0, C_1, C_2... such that each processor holds its own local memory, knows its own index and is able to execute read/write instructions to any shared memory unit [8, p. 22]. This is a natural expansion of RAM, which was merely a single such processor with memory where accessing any address would take constant time. The PRAM model will normally include an instruction set [8, pp. 22-23], each instruction taking one constant unit of time to execute. Measurements of algorithmic complexity are thus done by finding upper bounds on the number of such instructions executed.

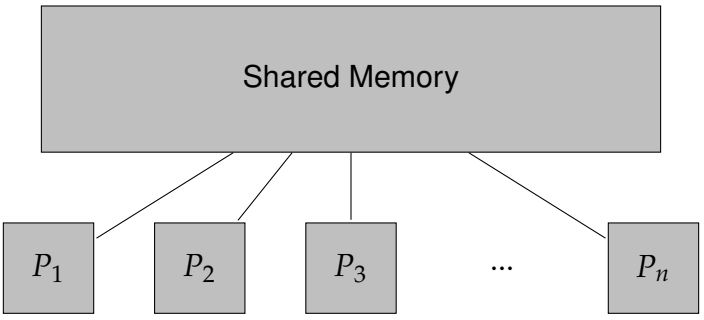

Figure 1.1: The shared memory model of PRAM.

While this model provides a simple and versatile framework to model parallel computation, it suffers a number of issues. From the perspective of practicality, Eijkhout [9, p. 71] argues that this model is "...unrealistic in practice...", stating that assumptions about multiple accesses to the same

location being possible and a lack of memory hierarchy inherently disqualify PRAM from being an honest emulation of modern processors. Another issue to be noted is that of PRAM's network topology; PRAM inherently assumes every memory unit is equally accessible with no overhead by every processor, meaning the network topology is always the same, as seen in [10, p. 4]. This means communication overhead and topology issues cannot be explored, both being pertinent enough to motivate the creation of the $\mathcal{LOCAL}$ model for distributed computation [11].

To tackle a few of these issues, further models were proposed, such as LogP [2]. This framework attempts to rectify the overly simplistic assumptions PRAM makes, adding four constant factors to incur extra overhead. These consist of L, an upper bound on the latency in sending data, o, the overhead time taken by a processor to engage with message transmission, g, the minimum time gap between sending messages, and P, the number of processors/memory modules. Further assumptions of asynchronous execution and a finite network bandwidth are made as well. This model notably makes no assumptions of the network topology, allowing for a diverse range of possible implementations. Furthermore, LogP assumes local memory, making communication a far more central issue than in PRAM, where shared memory makes communication complexity far more trivial.

Another such model is BSP, or Bulk Synchronous Parallelism, which extends sequential computation in a similar way to LogP. In a similar manner, it allows the topology to be of any form and fails to restrict independent processor layouts, citing itself to be "independent of target architecture" [1, p. 2]. However, it differs greatly in its addition of global supersteps of computation [1, p. 3], within each of which a number of local asynchronous computations occur across processors, followed by a final 'barrier synchronization' where the entire network waits to synchronize before a number of communications occur *en masse*. By doing this, deadlock becomes impossible and programming is made easier [9, p. 116]. This model has seen a great deal of focus, even possessing a full library implementation [12] and an algebraic semantics by He [13].

A proposition was made by Valiant in 2010 to extend the BSP model further to Multi-BSP [14]. Valiant defined an instance of Multi-BSP to be "...a tree structure of nested components where the lowest level of leaf components are processors and every other level contains some storage capacity." [14, p. 6] A parent vertex represents a storage level that contains its children; all such vertices at each level of depth in the tree share a predefined degree and communication bandwidth with their children. Computation proceeds similarly to in BSP, consisting of a series of supersteps in which communication between parents and children may occur. This model makes a clear attempt to simulate a memory hierarchy in a parallel environment, aimed at "...capturing the most basic resource parameters of multi-core architectures." [14, p. 1] To this end, it makes

explicit the substantial issues of network topology, being the first such model we have explored to do this.

While all of the frameworks discussed up to this point are effective practical models of parallel computation, they all diverge substantially from work in sequential complexity on Turing Machines, being more akin to modern processors by design. This makes it nontrivial to translate results in sequential complexity theory to parallel versions when using these models. On top of this, many of these models are heavily parameterized, adding great difficulty in producing comprehensive complexity theoretic results. Multi-BSP makes an effort to overcome this with a parameter-free notion of optimality [14, p. 4] by allowing discrepancy in some parameters up to constant factors; however, Valiant cites an aim in future research of this model to be finding optimal such parameters for a given algorithm [14, p. 4], making clear the importance of the many parameters in this approach to parallelism.

1.2 Boolean Circuits

One model that better bridges the gap between classical sequential and parallel paradigms is that of Boolean circuits [15, ch. 6], which play a vital role in classical complexity theory in their own right. A Boolean Circuit with n inputs and a single output is defined as a directed acyclic graph with n sources (vertices with no incoming edges) and one sink (vertex with no outgoing edges), with all non-input vertices being 'gates' and ascribed with some logical operation (OR, AND and NOT) [15, p. 107]. An example circuit is shown below:

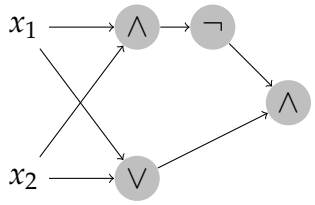

Figure 1.2: A Boolean Circuit that decides $x_1 \neq x_2$ for $x_1x_2 \in \{0,1\}^2$.

A circuit's size is considered to be the number of vertices it contains. An input is a string $x \in \{0,1\}^n$; to compute the output, the source vertices are each given a value equal to each of the input string's n characters, and every gate is recursively given a value equal to its logical operation performed upon the value of its incoming edges, until a value is finally attained at the sink. This value is taken to be the output. For instance, the input 01 above would incur an output of 1, by evaluating the circuit in the

natural way described.

It is perhaps surprising that Boolean circuits play any role in parallel complexity theory, but the complexity class NC of all circuits of polynomial size and polylogarithmic depth [15, p. 117] is in fact a reasonable complexity class for algorithms that have some fast parallel implementation [15, p. 118]; indeed, one can also define NC as the class of all languages decidable in polylogarithmic time with a polynomial number of processors [8, p. 44]. It is well-known that $NC \subseteq P$ [8, p. 44]; however, it is unknown whether $NC = P$, a statement that if true would imply all tractable algorithms can be drastically improved by parallelism.

Boolean circuits are an effective candidate to consider theoretical questions about parallel complexity theory, as their connections with classical complexity theory are well-understood to the point of their embedding in Turing Machines being fully explored [8, p. 71]. One must note, however, the lack of immediate intuition as to how one would implement a parallel process in a Boolean circuit; models like PRAM, LogP and BSP as explored before stand nearer to how parallel algorithms are written by design, meaning analysis of specific algorithms is difficult at best.

1.3 Parallel Turing Machines

Another important model for parallelism to investigate is that of Parallel Turing Machines [3], an approach to parallelism involving many heads reading to one tape. A Parallel Turing Machine is similar to a standard Turing Machine; however, at any given stage, a multitude of read-write heads may exist rather than just one. Computation starts with only one head as a standard sequential Turing Machine, and proceeds identically, however a head may choose to multiply into several heads, which then each execute on their own. If two heads try and write a different character to the same position on the tape, the computation is illegal, leaving undefined behavior [3, p. 5]. This means parallelism is achieved by choosing to 'spawn' new processes with the same shared memory.

It should be noted that Parallel Turing Machines can only give polynomial speedup on sequential ones [3, p. 16]. Wiedermann, the creator of the PTM model, believes this to be "in good agreement with basic physical laws" [3, p. 3], as shared memory models of parallelism which can spawn an exponential number of processors in polynomial time fail to take into consideration implicit communication and synchronization costs between processors, making them inaccurate. [3, p. 2]

1.4 Machine Classes

An important insight made in the study of parallel complexity theory was, as with sequential complexities, many different machines to analyze (eg. PRAM, Parallel Turing Machines) produced varying complexity classes, which were not all identical. This is to be expected; a 2-processor parallel machine where one processor is an oracle that solves anything in one step will clearly give different complexities to PRAM!

In sequential complexity theory, discrepancies between machines were resolved with the **invariance thesis**:

Invariance Thesis. *Any two "reasonable" sequential machines can simulate one another in polynomial-bounded time and constant-bounded space. [16, p. 5]*

We note that 'reasonable' is an inherently qualitative concept; machines like Turing Machines fit into the complexity theory community's idea of reasonable. An equivalent formulation was later developed for parallel models of complexity, as a connection was observed between time taken to execute on a parallel machine and space required on a sequential one:

Parallel Computation Thesis. *The time taken to solve a problem in a "reasonable" parallel machine is polynomially related to the space taken on a "reasonable" sequential machine. [16, p. 5]*

The concepts of reasonable sequential machines are of course equivalent in both theses. Models for parallelism can now be distinguished by whether they satisfy the Invariance Thesis, Parallel Computation Thesis or neither. Machines that satisfy the first case are said to be in the **first machine class**; in the second case they are in the **second machine class**. Models which fit into the first include standard Turing Machines, while those fitting into the second include a variant of RAM called k-PRAM, developed by Savitch and Stimson [17], where processors can create up to k immediate copies of themselves, which can then do the same recursively.

However, many models of parallelism do not fall into the second machine class; notably, Parallel Turing Machines fall short unless $PSPACE = P$, as they can only provide polynomial speedup to a sequential Turing Machine. It is thus a topic of debate whether the Parallel Computation Thesis is a reasonable estimate of parallelism, with papers arguing for [18] and against [19]. It is, nevertheless, always of interest to show whether a model is in these classes or not, something we will investigate.

1.5 Distributed Computing Models

We finally touch upon approaches to distributed complexity analysis, a paradigm distinct from parallel complexity theory but still relevant to consider. This distinction lies in what 'success', 'failure' and 'complexity' mean to a distributed system. Three main disciplines exist in distributed computing theory - analysis of *timing*, *congestion* and *locality* issues [6, p. 27]. These approaches respectively investigate issues arising from asynchronicity of messages and of limited communication.

With respect to locality, the $\mathcal{LOCAL}$ model is standard [6]. It should be noted that the aim of this model is to consider complexities arising from communication rather than from computation itself; to this end, a graph network of processors $G = (V, E)$ of distinct processors V and channels of communication $E \subseteq V \times V$ is assumed such that processors awaken all at once, messages of arbitrary size and complete correctness are transmitted in synchronized rounds of communication, and arbitrary computations are performed by each processor with its local data [11, p. 6]. These decisions all constitute an effort to isolate communication complexities; indeed, a central class of complexity here is $LD(t)$, the class of decision problems that all participant processors can agree on a solution for in t rounds of communication, $LD(O(1))$ being of particular interest [20].

It is noteworthy that, while the $\mathcal{LOCAL}$ model does not directly aim to produce classical complexity theoretic results, it is possible to bound the complexity of computation performed between each round of communication r for a vertex $v \in V$ by a function $f_A(H(r, v))$ [20], where $H(r, v)$ denotes the total amount of data received by v in all rounds before r. This approach makes sense as $H(r, v)$ thus denotes the size of our input for the computation v is to perform this round. Fraiginaud, Korman and Peleg note that investigating this could lead to "...interesting connections between the theory of locality and classical computational complexity theory." [20, p. 6] However, it seems that there has been little to no response to this in the literature, as research continues to focus on communication complexity.

We should also give consideration to the $\mathcal{CONGEST}$ model [6], which aims to add to the $\mathcal{LOCAL}$ model the complexity of message size, and thus of congestion. More specifically, it requires that message size is bounded by $O(\log n)$ bits for a graph with n nodes [11, p. 5], as each node has a unique identifier of this length. We may generalize this to the $\mathcal{CONGEST}(B)$ model, where we may send messages starting with an identifier of size $O(\log n)$, then with B bits of content. This approach more accurately reflects real systems, as message transmission is in reality not instantaneous. This should not discredit the $\mathcal{LOCAL}$ model's importance in the reader's mind however - adding message time to a model can cloud the study of locality on its own.

2 Definitions

Armed with a comprehension of current parallel complexity models, we now begin to consider a discrepancy in the models we have seen. Notably, we find no model for message-passing parallelism with connections to the classical theory of Turing Machines, like Parallel Turing Machines give for shared memory. One could argue the proof of Boolean circuits being a model for parallelism in [15, ch. 6] generalizes to parallelism without shared memory; however, the nature of this connection makes it difficult to analyze specific algorithms or communication networks, as is possible in more practical models like BSP or LogP. Unfortunately, these models in turn lack the theoretical connections to Turing Machines to themselves suffice.

We now attempt to resolve this issue with a new model for message-passing parallelism. We will aim to satisfy the following four requirements:

1. The model should intuitively emulate a network of communicating processors without shared memory.
2. The model should allow for meaningful complexity analysis.
3. The model should establish a way to compare sequential complexity classes as defined with Turing Machines and parallel ones.
4. The model should be able to simulate other parallel models efficiently.

2.1 Turing Networks

To satisfy our first requirement, our model will explicitly consist of a network of processors. To this end we note that, as per the approach of $\mathcal{LOCAL}$ and $\mathcal{CONGEST}$ [6], networks can be effectively modelled with simple undirected graphs, with vertices representing processors and edges representing communication channels. Undirectedness implies communication is two-way, a reasonable choice as such a system can emulate one-way communication by simply only communicating in one direction.

To satisfy the third requirement, we will assume each processor in the network is a Turing Machine, a model that is known to emulate sequential complexity well and will give us the connection to Turing Machines we seek. The second and fourth are to be verified after our model is defined.

With regards to communication, we will follow the example of $\mathcal{CONGEST}$

and demand communication takes time proportional to message size. This forces us to consider communication as a major issue in parallel complexity, a decision that more directly reflects the very real overhead of sending long messages. Communication will be synchronous for each character sent; asynchronicity may be emulated via intermediate buffer processors.

For simplicity, we make a number of assumptions. The first of these is that our network is static and known *a priori*; mobile and unknown networks present a range of challenges we lack the time to investigate here. Furthermore, we assume input is given to a single processor and output is required from it alone. This is reasonable as we are not emulating distributed computing where consensus is meaningful - instead, we are emulating algorithms that reap parallelism to compute functions efficiently, so one input and one output emulate this. Requiring output at the same processor is not a substantial demand either, as we simply wish for the answer to exist somewhere, and sending it back to the input processor can no more than double the time taken by returning where you came from. This is a constant factor, so does not concern complexity analysis.

Finally, we assume all processors can be given the same algorithm to execute, knowing only whether they are the input/output processor or not. This is because, even if there exist a finite number of different algorithms used by the different processors, we can identify each processor by the path of communications taken to reach it, and choose from a finite set of algorithms to execute. Note we are implicitly assuming processors do not begin computation before being communicated with - this is justified, as any computations done before this happens must be agnostic to input, giving results that can simply be encoded in the processor beforehand. This will allow us to represent unbounded networks with an infinite graph, ensuring that an infinite amount of computations cannot occur in finite time as only a finite number of processors could then be interacted with.

Armed with this, we now begin to develop definitions. We start by defining a single processor in our model, essentially a Turing Machine augmented to support communication:

Definition 1. *A **Communicative Turing Machine**, or CTM, is a tuple*

$$T = \langle Q, \Sigma, \Gamma, q_m, q_s, h_a, h_r, \delta_t, \delta_s, \delta_r \rangle$$

where Q, Σ are nonempty and finite, $\Sigma \cup \{\Lambda\} \subset \Gamma$, $q_m, q_s, h_a, h_r \in Q$,

$$\delta_t : Q \times \Gamma \nrightarrow Q \times \Gamma \times \{L, R, S\}$$
$$\delta_s : Q \times \Gamma \nrightarrow \mathbb{N} \times \Gamma \times Q^2$$
$$\delta_r : Q \nrightarrow \mathbb{N} \times Q$$

are partial functions, and $\delta_t(q_s, x), \delta_s(q_s, x)$ undefined $\forall\, x \in \Gamma$.

This definition expands upon the standard Turing Machine (see Appendix A.3) in a number of ways. There are now two start states, q_m and q_s; these are respectively the *master* and *slave* start states. Henceforth, the processor that is given input and produces output shall be called the master node and all others slave nodes. Naturally, slave nodes will start in the state q_s and the master in q_m; this allows nodes to identify their status.

The three transition functions are another notable difference. δ_t is the classical transition function as per standard Turing Machines, albeit partial now. δ_s is the *send function*, identifying in what states we wish to send a character, which integer-indexed neighbor we wish to send it to, and the states we will go to if this operation succeeds or if the neighbor doesn't exist. δ_r is the *receive function*, identifying in which states we are able to receive a character from a given neighbor, and what state we will transition to if this happens. Intuitively, all processors in our network will be of this form, and any one may transition, send or receive if this is currently possible at a given time. Note the end of the definition forces a CTM in the slave starting state to wait for a message before beginning computation.

Definition 2. *An **oriented graph** $\overline{G} = \langle G, \phi \rangle$ is a pair consisting of a simple graph $G = \langle V, E \rangle, E \subseteq V \times V$ and a partial injective function $\phi : V \times \mathbb{N} \nrightarrow V$ such that for any $v \in V$, there exists an $n \in \mathbb{N}$ such that $\{\phi(v,1), ..., \phi(v,n)\}$ is v's neighborhood in G and $\phi(v,m)$ is undefined for all $m > n$. We call ϕ the **orientation** of $\overline{G}$.*

*A **network topology** $\overline{G} = \langle G = \langle V, E \rangle, \phi \rangle$ is an oriented graph where G is connected, E is a decidable symmetric relation, G has bounded degree, $1 \in V \subseteq \mathbb{N}$, and membership in V is decidable.*

Definition 3. *A **Turing Network** is a tuple $\mathcal{T} = \langle \overline{G}, T \rangle$ where T is a CTM and $\overline{G}$ is a network topology.*

Intuitively, the orientation gives an indexing of every neighbor for each vertex. Vertex 1 will represent our master node. Note that, as all vertices of our network are identified by natural numbers, the network topology will be countably infinite at its largest; this is acceptable, as only a finite amount of vertices will be interacted with in a finitely long computation anyways. Having an infinite graph lets us emulate unbounded numbers of processors, similar to a botnet or other distributed computing system.

We also enforce that every topology has bounded degree, as a CTM can only reference a fixed number of distinct neighbors. We will not consider networks of unbounded degree here, for the sake of simplicity.

To define a computation, we must first define a configuration of our network, similarly to a standard Turing Machine (see Appendix A.3). This begins with CTMs, and is then expanded to general networks:

Definition 4. *A **CTM configuration** of a CTM T is a 4-tuple of the form $C \in \Gamma^* \times \Gamma \times Q \times \Gamma^*$. We name the set of all CTM configurations for the CTM T $\mathcal{C}(T)$.*

We say, for CTM configurations $C_n = \langle r_n, s_n, q_n, t_n \rangle, n \in \mathbb{N}$, a network topology $\overline{G} = \langle G, \phi \rangle$, $G = \langle V, E \rangle$ and $v_1, v_2 \in V$,

$$
\begin{aligned}
C_1 \vdash C_2 \Leftrightarrow\ & C_1 \text{ transitions to } C_2 \text{ as configurations of } T' \\
& \text{and } \langle q_1, s_1 \rangle \in dom(\delta_t) \\
\langle C_1, C_2 \rangle \vdash_{v_1}^{v_2} \langle C_3, C_4 \rangle \Leftrightarrow\ & \phi(v_1, n_1) = v_2 \wedge \phi(v_2, n_2) = v_1 \\
& \wedge \delta_s(q_1, s_1) = \langle n_1, s_4, q_3, q \rangle \\
& \wedge \delta_r(q_2) = \langle n_2, q_4 \rangle \\
& \wedge r_1 = r_3, r_2 = r_4, t_1 = t_3, t_2 = t_4, s_1 = s_3 \\
C_1 \nrightarrow_{v_1} C_2 \Leftrightarrow\ & \delta_s(q_1, s_1) = \langle n, s_1, q, q_2 \rangle \\
& \wedge \langle v, n \rangle \notin dom(\phi)
\end{aligned}
$$

where $T' = \langle Q, \Sigma, \Gamma, q_m, h_a, h_r, \delta \rangle$ is a Turing Machine such that $\delta \mid_{dom(\delta_t)} = \delta_t$ and $\delta(q, s) = \langle q, s, S \rangle$ for all $\langle q, s \rangle \notin dom(\delta_t)$.

This definition introduces the three basic transitions CTMs are capable of: **standard transitions** ($\vdash$), **communication transitions** ($\vdash_*^*$), and **failed-to-send transitions** ($\nrightarrow_*$). The first represents a standard change of state as though the CTM were a Turing Machine. The second represents a single character being successfully sent from one CTM in configuration C_1 to one in configuration C_2, each transitioning to configurations C_3, C_4 in the process respectively. The third represents a CTM in configuration C_1 trying to send a message to its n^{th} neighbor, but that neighbor not existing and the CTM entering a chosen recovery state in configuration C_2.

From these three forms of transition, we may now define a configuration of a Turing Network and the appropriate types of transition it can make:

Definition 5. *A **TN configuration** of a TN $\mathcal{T}$ is a function of the form $\Omega : V \to \mathcal{C}(T)$. We say, for CTM configurations $\Omega_n, n \in \mathbb{N}$ of a Turing Network $\mathcal{T}$ and $v_1, v_2 \in V$,*

$$
\begin{aligned}
\Omega_1 \vdash_{v_1} \Omega_2 \Leftrightarrow\ & \Omega_1 \mid_{V \setminus \{v_1\}} = \Omega_2 \mid_{V \setminus \{v_1\}} \\
& \wedge (\Omega_1(v_1) \vdash \Omega_2(v_1) \vee \Omega_1(v_1) \nrightarrow_{v_1} \Omega_2(v_1)) \\
\Omega_1 \vdash_{v_1}^{v_2} \Omega_2 \Leftrightarrow\ & \Omega_1 \mid_{V \setminus \{v_1\}} = \Omega_2 \mid_{V \setminus \{v_1\}} \\
& \wedge \langle \Omega_1(v_1), \Omega_1(v_2) \rangle \vdash_{v_1}^{v_2} \langle \Omega_2(v_1), \Omega_2(v_2) \rangle \\
\Omega_1 \vdash \Omega_2 \Leftrightarrow\ & (\exists\, v \in V \bullet \Omega_1 \vdash_v \Omega_2) \\
& \vee (\exists\, v_1, v_2 \in V \bullet \Omega_1 \vdash_{v_1}^{v_2} \Omega_2)
\end{aligned}
$$

This definition is characterized by a map from vertices to configurations, providing finally a notion of the state of the entire network. The two first forms of transition simply capture a transition involving one or two vertices, respectively. The last is simply a generalized transition, where one of the two above happen. Now, we may define a computation of a Turing Network:

Definition 6. *An **initial state** of a TN $\mathcal{T}$ is a configuration Ω_S for some $S \in \Sigma^*$ where*

$$\Omega_S(1) = \langle \lambda, \Lambda, q_m, S \rangle$$
$$\Omega_S(v) = \langle \lambda, \Lambda, q_s, \lambda \rangle \quad \forall\, v \in V \setminus \{1\}$$

*A **final state** of $\mathcal{T}$ is a configuration Ω_h where $\Omega_h(1) = \langle A, b, q, C \rangle$ where $q \in \{h_a, h_r\}$. The output string is AbC with all characters not in Σ deleted. We say Ω_h is **accepting** if $q = h_a$ and **rejecting** otherwise.*

*A **derivation sequence** $\Psi = \{\Omega_n\}_{n \in X}$ is a sequence of indexed configurations of $\mathcal{T}$ where $X \subseteq \mathbb{N}$ and for any $n, m \in X$, $\Omega_n \vdash \Omega_m$ if m is the least element of X greater than n. We say Ψ is a **computation** if it starts with an initial state and ends with a final state.*

Say that, for two derivation sequences of $\mathcal{T}$, Ψ_1, Ψ_2, $\Psi_1 \leq \Psi_2$ if the former is a prefix of the latter as a sequence.

*We say $\mathcal{T}$ **accepts** a string $S \in \Sigma^*$ if every derivation sequence starting with Ω_S is less than an accepting computation and all rejecting computations are greater than an accepting computation. We say it **rejects** if there exists a rejecting computation not greater than some accepting computation.*

*We say $\mathcal{T}$ **computes** a function $f : \Sigma^* \nrightarrow \Sigma^*$ if, for every input string $s \in dom(f)$, $\mathcal{T}$ accepts s and every computation of $\mathcal{T}$ with input string s has a final state with output string $f(s)$.*

This now gives us the terminology to discuss computation as a series of configurations. We now restrict ourselves to considering computations only.

2.1.1 An Example Network

To help illustrate our model, we now produce an example Turing Network. The network topology will simply be an infinite path starting at 1; the network will, given a number, communicate with that many vertices in the network before terminating. This solves no specific problem beyond showcasing what a Turing Network looks like and how to produce a time function. Formally, we have a network $\mathcal{T} = \langle G, T, \phi \rangle$ where

$$
\begin{aligned}
&G = \langle V, E \rangle, V = \mathbb{N} \\
&E = \{\langle n, n+1 \rangle : n \in \mathbb{N}\} \cup \{\langle n+1, n \rangle : n \in \mathbb{N}\} \\
&\phi(1,1) = 2 \\
&\phi(n,1) = n-1, \qquad n > 1 \\
&\phi(n,2) = n+1, \qquad n > 1
\end{aligned}
$$

A diagram of the CTM T is given in Figure 2.1 below. Note we use similar conventions to classical Turing Machine diagrams, albeit including new types of arrows for sending characters and receiving them. An arrow splitting in two at a black circle is a send transition, with the one-line arrow identifying the state reached on success and the dashed arrow the state reached on failure. A double-lined arrow identifies the state reached when a character is received. We define the alphabets as $\Sigma = \{a\}, \Gamma = \Sigma \cup \{\Lambda\}$.

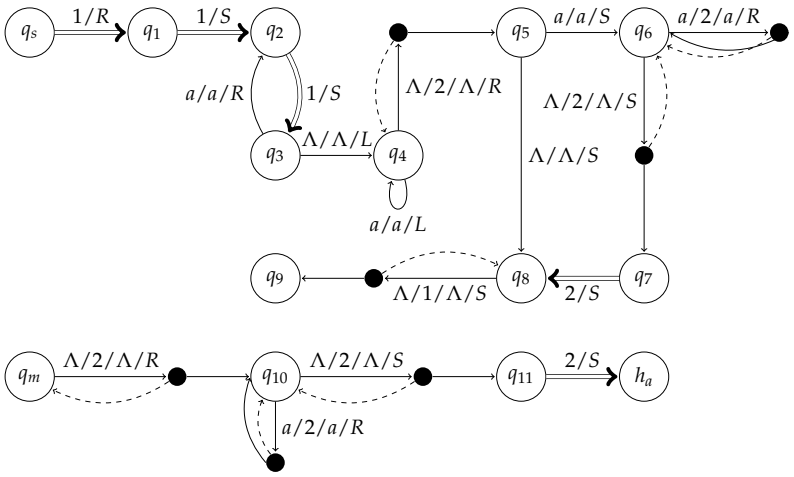

Figure 2.1: Diagram of the CTM T.

It is evident that our machine will, given input a^n, send the message a^{n-1} from vertex 1 to 2, and continue sending strings decreasing linearly in size until nothing is sent to vertex n+1. At this point, a confirmation message will be sent to vertex n, which will send one to vertex n-1, and so forth until vertex 1 receives a confirmation and knows the message has been received. The output string is identical to the input when this halts.

2.2 Measuring Complexity

We now have the facilities necessary to start defining the cost of a given computation. In doing this, we establish exactly how much can be gained by introducing maximal parallelism:

Definition 7. *A **parallel derivation sequence** of a Turing Network $\mathcal{T}$ is a derivation sequence $\Psi = \{\Omega_n\}_{n \in X}$ where for all $i, j \in X, i \neq j$ and $v_n \in V$,*

$$\begin{aligned}
\Omega_i \vdash_{v_1} \Omega_x \ \wedge \ \Omega_j \vdash_{v_2} \Omega_y \ &\Rightarrow \ v_1 \neq v_2 \\
\Omega_i \vdash^{v_2}_{v_1} \Omega_x \ \wedge \ \Omega_j \vdash_{v_3} \Omega_y \ &\Rightarrow \ v_1 \neq v_2, v_3 \\
\Omega_i \vdash^{v_2}_{v_1} \Omega_x \ \wedge \ \Omega_j \vdash^{v_4}_{v_3} \Omega_y \ &\Rightarrow \ v_1, v_2 \neq v_3, v_4
\end{aligned}$$

where $x, y \in X$ are the successors of i, j in X, if they exist.

Definition 8. *The **parallel time** of a derivation sequence is the smallest number of parallel derivation sequences it itself is a concatenated sequence of.*

*The **time function** $\tau : \Sigma^* \to \mathbb{N}_0$ of a Turing Network $\mathcal{T}$ maps input strings to the longest parallel time of any accepting computation starting with said string as input the Turing Network is capable of.*

2.2.1 Example Time Function

Armed with this knowledge, we can now begin to classify the time function of our previous example TN $\mathcal{T}$. At any given time, only one vertex may transition or one pair of vertices may communicate, hence any parallel subsequence is trivial and we may simply count the number of transitions in this trivial case. We note that, for a given processor that is neither the vertex n nor the master node, it takes $2n + 2$ transitions to receive a message of length n and a further $n + 1$ to return the message's start, 1 to determine that a further message of length $n - 1$ must be sent, and 1 to receive an acknowledgement. The master will make one unaccounted for transition to receive the final acknowledgement, and the n^{th} node will take 6 unaccounted for transitions to determine it has nothing to send. Hence,

$$\tau_{\mathcal{T}}(n) = \sum_{r=1}^{n-1} (2r + 2 + r + 1 + 1 + 1) + 6 + 1 = \frac{3}{2}n^2 + \frac{7}{2}n + 2$$

Thus, $\tau_{\mathcal{T}}(n) = O(n^2)$ and we can see our network takes quadratic time to terminate given an input of length n.

It should be noted that there are a multitude of other potential ways we could gauge complexity, including communication costs, space complexity and the number of processors needed to complete a computation. However, we do not investigate these for the sake of brevity.

3 Complexity Theory

Armed with a measure for the time taken by a Turing Network to complete a parallel computation, we may begin to build a complexity theoretic framework around our model for parallelism. We begin by introducing our standard complexity class:

Definition 9. *For any given function* $f : \mathbb{N}_0 \to \mathbb{N}_0$ *and network topology* $\overline{G}$*,*

$$PARTIME_{\overline{G}}(f(n)) = \{g \mid \Sigma \text{ set, } g : \Sigma^* \nrightarrow \Sigma^* \text{ computable by some TN } \mathcal{T} \text{ with topology } \overline{G} \text{ and } \tau_{\mathcal{T}}(n) = O(f(n))\}$$

The reader may rightly question this specific choice of complexity class, namely why we are only willing to consider problems solvable given a specific network topology. This is because, given free choice of network topology, we are in fact able to compute literally any decision problem in quadratic time, including undecidable ones; intuitively, the topology can, if left unchecked, encode arbitrary information we may retrieve by traversing it. (See Appendix A.2 for details.) The reader may thus believe this model is far too general, encompassing behaviors too broad to be considered a model of computation. However, this problem plagues Boolean Circuits as well [15, p. 110], since without considering how hard it is to construct a circuit, these too can compute undecidable problems. Later, we will consider the difficulty of constructing a given network topology, the method by which Boolean Circuits resolve this conundrum in classes like NC.

We note a few simple laws pertaining to our complexity class, the first relating to the sequential complexity class $DTIME(f(n))$ in [15, p. 25]:

Theorem 1. *For any given function* $f : \mathbb{N}_0 \to \mathbb{N}_0$ *and network topologies* $\overline{G}, \overline{H}$*,*

$$DTIME(f(n)) \subseteq PARTIME_{\overline{G}}(f(n)) \tag{3.1}$$

$$\overline{H} \simeq \overline{G} \;\Rightarrow\; PARTIME_{\overline{H}}(f(n)) = PARTIME_{\overline{G}}(f(n)) \tag{3.2}$$

(The operator $\simeq$ depicts isomorphism. See Appendix A.1 for details.)

Seeing (3.1) is trivial, as the master processor can compute anything a classical Turing Machine can by acting as a Turing Machine without communing with any neighbors. (3.2) is true as, since orientation is preserved and 1 remains the same, identical derivations are possible in both network topologies up to renaming of nodes greater than 1. Intuitively, the graphs will look the same to computations as vertex names are not explicitly used, only indexing by orientation.

3.1 The Parallel Computation Thesis

Ideally, we would like some verification of the well-known Parallel Computation Thesis [21, p. 3], mentioned in Section 1.4. Hence, we shall develop an understanding of a wide class of network topologies that can compute PSPACE-complete problems in polynomial time in an effort to identify topologies that verify this thesis for our model.

We begin by defining the class of polynomial time problems for a TN:

Definition 10. *For a given network topology $\overline{G}$,*

$$P_{par}\langle\overline{G}\rangle = \bigcup_{d\in\mathbb{N}} PARTIME_{\overline{G}}(n^d)$$

Initially we consider only the binary tree, namely $\overline{B} = \langle G = \langle V, E\rangle, \phi\rangle$ where $V = \mathbb{N}$ and

$$\begin{aligned}
&E = \{(n, \left\lfloor \frac{n}{2} \right\rfloor), (\left\lfloor \frac{n}{2} \right\rfloor, n) : n \in \mathbb{N}, n > 1\}\\
&\phi(n,1) = 2n, n \in \mathbb{N}\\
&\phi(n,2) = 2n+1, n \in \mathbb{N}\\
&\phi(n,3) = \left\lfloor \frac{n}{2} \right\rfloor, n \in \mathbb{N} \setminus \{1\}
\end{aligned}$$

We note that with this graph, we can reach an exponential number of vertices if we were to broadcast a message across the network starting at the master vertex for linear time. Hence, this network has potential for enormous distribution of computations.

We choose the QBF (Quantified Boolean Formulae) problem [15] as the PSPACE-complete problem we will attempt to solve on this network; if we can compute it in polynomial time, we will have a strong positive result for the Parallel Computation Thesis, as then any problem computable in polynomial space can be computed in parallel in polynomial time.

QBF is stated here as follows:

Definition 11. *For the alphabet* $\Sigma = \{\forall, \exists, \bullet, \wedge, \vee, \neg, (,), \top, \bot, a\}$,

$$QBF = \{\alpha \in \Sigma^* : \alpha \textit{ is a valid satisfiable Boolean formula}\}$$

Our algorithm is given below, where Σ is as in the definition of QBF:

Data: a string $\alpha \in \Sigma^*$
Result: $\top$ if α is a satisfiable Boolean formula, $\bot$ otherwise
if *master node* **then**
  if α *valid Boolean formula* **then**
    convert to Prenex normal form;
    send to neighbor 1;
    return *message from neighbor 1*;
  else
    return $\bot$;
  end
else
  $\sigma \leftarrow$ message from neighbor 3;
  if σ *starts with '*$\forall\, x\bullet$*' for some* $x \in \{a\}^*$ **then**
    delete '$\forall\, x\bullet$' from start of σ;
    send σ to neighbor 1 with all occurrences of x set to 'true';
    send σ to neighbor 2 with all occurrences of x set to 'false';
    wait for both neighbors 1 and 2 to respond;
    if *both neighbors 1 and 2 respond with* $\top$ **then**
      send $\top$ to neighbor 3;
    else
      send $\bot$ to neighbor 3;
    end
  else
    if σ *starts with '*$\exists\, x\bullet$*' for some* $x \in \{a\}^*$ **then**
      same as for $\forall$, but send $\top$ if either 1 or 2 respond with $\top$;
    else
      if σ *contains a free variable* **then**
        same as $\exists$, but setting this variable and no deletion;
      else
        evaluate formula;
        send $\top$ to neighbor 3 if 'true', $\bot$ otherwise;
      end
    end
  end
end

Algorithm 1: Binary tree algorithm to decide QBF.

In devising this algorithm, we have assumed a single CTM is capable of

computing the following in polynomial time:

- Checking whether a string is a valid quantified Boolean formula.
- Converting a formula to Prenex Normal Form. [22]
- Replacing all occurrences of a variable with the value 'true' or 'false'.
- Checking whether any variables are left in a formula.
- Evaluating a formula with no variables.

These are all reasonable assumptions, as a CTM can clearly do anything a standard Turing Machine can do in the same time. With this, we may now begin to consider our algorithm's complexity.

For any given node, no iteration occurs and all steps are polynomial time as per our assumptions. Clearly, if conversion to Prenex normal form is polynomial time, the result formula must be a polynomial of the original one in size. On top of this, each successive slave node communicated with removes a quantifier or free variable, reducing the formula to a variable-free Boolean expression in a linear number of such changes relative to the Prenex normal form of the formula, hence a polynomial number of such steps. Finally, the last slave nodes communicated with simply evaluate variable-free formulae, and results are propagated back up a polynomial depth of the binary tree back to the master node. Overall, this algorithm must therefore take polynomial time, giving us the following theorem:

Theorem 2. $PSPACE \subseteq P_{par}\langle \overline{B} \rangle$.

We are able to extend this result to topologies other than binary trees if we are willing to consider a factor we define below:

Definition 12. *For a given network topology $\overline{G}$, the **neighborhood function** $N_{\overline{G}} : \mathbb{N} \to \mathcal{P}(V)$ is defined as*

$$N_{\overline{G}}(n) = \{v \in V : \exists \text{ path in G from 1 to v of length} \leq n\}$$

*and the **growth function** $\gamma_{\overline{G}} : \mathbb{N} \to \mathbb{N}$ is defined as*

$$\gamma_{\overline{G}}(n) = |N_{\overline{G}}(n)|$$

Theorem 3. *For any network topology $\overline{G}$ where $\gamma_{\overline{G}}(n) = \Omega(2^n)$,*

$$PSPACE \subseteq P_{par}\langle \overline{G} \rangle$$

We omit the full proof of the above theorem here; in order to prove this, we must show that we can find a spanning tree of exponential growth function

in our topology, and then prune all dead end branches. We will assume this is possible, as it is merely an issue of traversing the network and having each node record its assigned parent and children in the subtree.

After this, we have a tree with some vertices having only one child and some having more. Our algorithm can now, if a vertex has only one child, simply pass on the formula to the child and pass the result back up to the parent. We also, instead of sending one formula to each child, send both to each child; the children each first try to evaluate the first formula they receive then the second. The parent simply waits for answers about both formulae from any source, rather than one answer from each child. This stops problems where one child simply has one further child and so forth, which would give an insufficent rate of growth - the other child can compensate. The overall rate of growth, though, will be sufficient to ensure all exponential possible assignments of values to variables are checked when all vertices in a polynomial depth of the tree are used. □

This gives a positive result towards the Parallel Computation Thesis for our model, showing for a large class of topologies, any problem sequentially solvable with polynomial space is solvable in parallel in polynomial time.

3.2 Simulation by Turing Machines

In order to identify upper bounds on complexity classes for Turing Networks, we will follow in the footsteps of Boolean Circuits [8] and attempt to simulate a TN on a classical Turing Machine. Clearly, if a computation taking time t_1 on a Turing Network can be simulated in time t_2 on a Turing Machine, and a problem cannot be solved in time t_2 sequentially, it cannot be solved in parallel in time t_1 as we could then simulate its parallel execution sequentially in time t_2. Hence, we are interested in deriving t_2 from t_1.

To construct a simulation, we will need to establish how to store a given configuration, how to produce an initial configuration given an input string, how to produce an output string given a final configuration, how to choose a possible transition and how to execute it. If we can do these things, simulation is trivial, simply being a matter of establishing an initial configuration and executing possible transitions until a final configuration is reached.

To make our lives easier we will assume a multi-tape Turing Machine, specifically 3 tapes. A configuration will be stored as follows:

- The first tape will contain a comma-delimited array, each element being a pair of a vertex name and the corresponding CTM's configuration. The read-write head is represented by a special character, to whose left is the character the head is currently over.

- The second tape contains the neighborhoods of each vertex that has participated in a transition - a comma-delimited array of pairs, where the first element is a vertex name and second is a list of its neighbors, ordered by their indices as per the orientation.
- The third tape is a scratchpad for other work.

To identify a possible transition, we need to check if any of the three elementary types of transition (standard, communication, failed-to-send) are possible. To check if a standard transition is possible, we may simply iterate through all CTM tapes, checking if any one is currently able to execute a standard transition by the character it is currently reading and its current state; executing the transition takes constant time, and all the information needed can be encoded in the TM simulating our network.

To check if a communication transition is possible, we iterate through all CTM tapes, gathering a list of vertices able to send a character to a vertex. We then iterate through this list, looking at each element's neighborhood in the second tape and seeing if any of its neighbors are willing to receive. If we find one such vertex, execute the involved steps for a communication transition to the neighbor, and empty the third tape. If the neighbor trying to be communicated with is not in the first tape, it has not been communicated with; add its empty tape to our first tape, compute its neighborhood in the second tape and execute the communication if it is able to receive such a message. If it is not able to receive the message, delete it again. If the sending vertex is trying to send to a neighbor of index larger than its neighborhood's size, the neighbor doesn't exist - execute a failed-to-send transition.

Producing an initial state given an input string is easy enough, as we merely create a single CTM tape with only the input string inside in the first tape, a list containing only the master node's neighbors in the second tape, and nothing in the third tape. Detecting acceptance or rejection requires us to observe the state of the master CTM; once this is done, we can delete everything but the output tape's contents, delete all characters not in the output alphabet and terminate.

Given this algorithm outline, we now aim to devise its complexity. We first investigate the complexity of a single transition. We first define the following notion, as we will need to factor in the size of vertex names present in our tape:

Definition 13. *For any network topology $\overline{G}$, the **name growth function** $\kappa_{\overline{G}} : \mathbb{N} \to \mathbb{N}$ is defined as*

$$\kappa_{\overline{G}}(n) = max(N_{\overline{G}}(n))$$

We also note that the length of the first tape after n transitions with an

input string of length x is bounded to $O(h(n))$, where the function h is defined as $h(n) = x + \gamma_{\overline{G}}(n)(\kappa_{\overline{G}}(n) + n)$. This is because it is bounded by the length of the list multiplied by the size of each element, where each element is a pair of a vertex name and a tape. The vertex name is evidently bounded above by $\kappa_{\overline{G}}(n)$ by definition, and the size of any tape is bounded above by n, the number of steps having occurred so far.

Similarly, the length of the second tape after n transitions is bounded by $O(k(n))$, where, given our topology's degree is Δ,

$$k(n) = \gamma_{\overline{G}}(n)\Big(\kappa_{\overline{G}}(n) + \Delta\kappa_{\overline{G}}(n+1)\Big)$$

ie. the number of vertices seen so far times the maximum size of any given list of neighbors, each of which contains a vertex name and a list of neighbors. This bound is given by the largest possible vertex name plus the largest possible neighbor; any vertex has a constant number of neighbors, so Δ can be ignored.

If we are simulating the TN $\mathcal{T} = \langle T, \overline{G}\rangle$ we note checking for and potentially executing a standard transition after n steps takes time $t_1(n) = O(h(n))$, ie. the size of the first tape. This bound holds because we merely need to traverse the first tape until we find a vertex able to perform a standard transition and do it, the latter step taking constant time.

To check for and execute a communication, we must produce the list of all potentially receiving and potentially sending vertices which takes $O(h(n))$ time, find any matches between these lists which takes $O((h(n) + k(n))^2)$ time, update the sender and receiver which take $O(h(n))$ time to reach on the first tape and constant time to update, and if not, try communing with a new vertex or fail a communication. To commune with a new vertex, for each of the $O(\gamma_{\overline{G}(n)})$ possibly sending vertices, it takes $O(h(n) + k(n))$ time to check its neighbors in the second tape, as well as the existence of each such neighbor in the first tape (there are a constant-bounded number of neighbors so we may pretend there is only one neighbor). Then, it will take $O(k(n) + h(n))$ to reach the end of both tapes and $O(\max_{v\in N(n+1),m} t_\phi(v,m))$ to compute this new neighbor's neighbors, where $t_\phi(v,n)$ is the time taken to compute $\phi(v,n)$, and $O(h(n))$ time to compute/undo this communication. Finally, $O(h(n) + k(n))$ time is needed to check for a communication with a nonexistent neighbor, ie. the time taken to traverse the list of candidate sending vertices and check each one's list of neighbors to see if the candidate neighbor exists.

Adding these all together and removing constant factors, the time taken to check for and execute a communication transition takes time

$$t_2(n) = O\Big((h(n) + k(n))^2 + \gamma_{\overline{G}}(n) \max_{v\in N_{\overline{G}}(n+1),m\in\mathbb{N}} t_\phi(v,m)\Big)$$

Therefore, the time to execute the n^{th} transition takes overall time $O(t_2(n))$, as it dominates $O(t_1(n))$ evidently.

From this, we may now deduce the overall time taken, as the time to construct the input string and then the output at the end are both $O(x)$ time for an input string of length x. There will also be a maximum of $\tau_{\mathcal{T}}(x)$ transitions clearly, each of which may involve all $\gamma_{\overline{G}}(n)$ vertices seen so far transitioning. Therefore, the overall time taken to simulate on a 3-tape Turing Machine is

$$\begin{aligned}
t_{\mathcal{T}}(x) &= O\left(x + \sum_{n=1}^{\tau_{\mathcal{T}}(x)} \gamma_{\overline{G}}(n) t_2(n)\right) \\
&= O\left(x + \sum_{n=1}^{\tau_{\mathcal{T}}(x)} \gamma_{\overline{G}}(n)\left((h(n)+k(n))^2 + \gamma_{\overline{G}}(n) \max_{v \in N_{\overline{G}}(n+1), m \in \mathbb{N}} t_\phi(v,m)\right)\right) \\
&= O\Bigg(\gamma_{\overline{G}}(\tau_{\mathcal{T}}(x))\tau_{\mathcal{T}}(x)\Big((h(\tau_{\mathcal{T}}(x)) + k(\tau_{\mathcal{T}}(x)))^2 \\
&\qquad + \gamma_{\overline{G}}(\tau_{\mathcal{T}}(x)) \max_{v \in N_{\overline{G}}(\tau_{\mathcal{T}}(x)+1), m \in \mathbb{N}} t_\phi(v,m)\Big)\Bigg)
\end{aligned}$$

The x may be removed at the end as all other terms will dominate it. We may finally expand $h(n)+k(n)$ and notice it reduces to $x + \gamma_{\overline{G}}(n)\kappa_{\overline{G}}(n+1)$. For the sake of simplicity, we also define the function

$$\Phi(n) = \max_{v \in N_{\overline{G}}(n+1), m \in \mathbb{N}} t_\phi(v,m)$$

which represents the longest time taken to compute a neighbor of anything of distance $n+1$ away from the vertex 1. Finally, we note that a 1-tape Turing Machine can emulate a 3-tape one in quadratic time [23, p. 293], giving us the final expression

$$t_{\mathcal{T}}(x) = O\left(\gamma_{\overline{G}}^2(\tau_{\mathcal{T}}(x))\tau_{\mathcal{T}}^2(x)\left(\left(x + \gamma_{\overline{G}}(\tau_{\mathcal{T}}(x))\kappa_{\overline{G}}(\tau_{\mathcal{T}}(x)+1)\right)^2 + \gamma_{\overline{G}}(\tau_{\mathcal{T}}(x))\Phi(\tau_{\mathcal{T}}(x))\right)^2\right)$$

(A similar bound for space complexity is given in Appendix A.4, along with some simple upper bounds derived from it.) We may now begin to use this to prove bounds on complexity.

3.3 Upper Bounds

Using the new function $t_{\mathcal{T}}$ for any given Turing Network $\mathcal{T}$ we have just derived, we may begin to observe upper bounds on complexity classes.

Firstly, a straightforward theorem immediately presents itself:

Theorem 4. *For any network topology $\overline{G}$ and $\mathbb{T}_{\overline{G}}(f)$ denoting the set of all TNs $\mathcal{T}$ with this topology such that $\tau_{\mathcal{T}}(n) = O(f(n))$,*

$$PARTIME_{\overline{G}}(f(n)) \subseteq DTIME(\max_{\mathcal{T} \in \mathbb{T}_{\overline{G}}(f)} t_{\mathcal{T}}(n))$$

All this theorem says is that if a problem is solvable in a given time in a Turing Network, we can simulate it running in this time on a sequential Turing Machine. This gives us a concrete upper bound on parallel time we can now exploit for more intuitive results.

We now investigate $P_{par}\langle\overline{G}\rangle$ for the rest of this section, as it constitutes all problems which are tractable given parallelism. Evidently, any TN $\mathcal{T}$ taking polynomial time to execute will have a simulation time of

$$t_{\mathcal{T}}(n) = O\Bigg(\gamma^2_{\overline{G}}(O(n^d))n^{2d}\Big(\big(n+\gamma_{\overline{G}}(O(n^d))\kappa_{\overline{G}}(O(n^d))\big)^2+\gamma_{\overline{G}}(O(n^d))\Phi(O(n^d))\Big)^2\Bigg)$$

In order to simplify our above theorem in this case, we will give a greater deal of scrutiny to the functions $\kappa_{\overline{G}}$, $\gamma_{\overline{G}}$ and t_ϕ, the major parameters we have not made sufficiently concrete. We note that $\gamma_{\overline{G}}(n) = O(2^n)$ always as any network topology's degree is constant by definition, greatly simplifying our analysis. Hence,

$$\begin{aligned} t_{\mathcal{T}}(n) &= O\Big(2^{O(n^d)}n^{2d}\big(2^{O(n^d)}\kappa_{\overline{G}}(O(n^d))^2+2^{O(n^d)}\Phi(O(n^d))\big)^2\Big) \\ &= O\Big(2^{O(n^d)}\big(\kappa_{\overline{G}}(O(n^d))^2+\Phi(O(n^d))\big)^2\Big) \end{aligned}$$

Note that the n^{2d} and n terms vanish as the ignored constant factors in $2^{O(n^d)}$ allow us to dominate them completely.

If we analyze at this level of granularity, it is evident that we cannot expect a better bound than EXPTIME at this point owing to the appearance of $2^{O(n^d)}$ within our expression. To achieve this bound precisely, we now investigate what bounds would be necessary on t_ϕ and $\kappa_{\overline{G}}$, the only two remaining properties of $\overline{G}$ influencing our complexity bound. This will reveal a class of network topologies which are unable to bring anything outside of EXPTIME back to polynomial time through unbounded parallelism.

Evidently, it is a necessary and sufficient condition that $\kappa_{\overline{G}}(\alpha n^d), \Phi(\beta n^d) = O(2^{O(n^d)})$ for some $\alpha, \beta \in \mathbb{N}$. The reader should be pleased to know that this requirement is not impossible to satisfy for κ - in fact, every network topology is isomorphic to one with this complexity of κ, as we can simply name the vertices in order of distance from the vertex 1. There are only up to $O(2^n)$ vertices of distance n away, so this naming will retain the property $\kappa_{\overline{G}}(n) = O(2^n)$ for our new topology $\overline{G}$, and hence $\kappa_{\overline{G}}(\alpha n^d) = O(2^{O(n^d)})$.

The question now remains whether this is possible to satisfy for Φ. Even if every topology is isomorphic to one with a sufficiently small κ, there is no guarantee that the complexity of the orientation will similarly decrease. We note that if $\max_{m \in \mathbb{N}} t_\phi(v, n) = O(f(v))$ for all $v \in V$, then $\Phi(n) = O(f(\kappa_{\overline{G}}(n+1)))$, ie. the time taken to compute a neighbor of the vertex with the largest name of distance n away from vertex 1.

In the worst allowable case of $\kappa_{\overline{G}}$, where this function is $O(2^{n^k})$ and thus $\kappa_{\overline{G}}(O(n^d)) = O(2^{O(n^{dk})})$, we must have then that $f(O(2^{n^d})) = O(2^{n^h})$ for an EXPTIME upper bound. Finding the maximal possible function f for this is nontrivial, however, we note that if $f(n) = O(2^{(\log n)^c})$ then

$$f(2^{n^d}) = O(2^{(\log 2^{n^d})^c}) = O(2^{n^{dc} \cdot (\log 2)^c}) = O(2^{O(n^k)})$$

for some constant k. This means that any orientation of quasi-polynomial complexity is sufficiently fast to be acceptable. However, if $f(n) = O(2^n)$ only, then $f(2^{n^d}) = O(2^{2^{n^d}})$, a bound lying outside of EXPTIME and in 2-EXPTIME instead. Hence, the upper bound on f lies somewhere between quasi-polynomial and exponential complexity.

We now have the following theorem:

Theorem 5. *For any network topology $\overline{G}$ such that $\kappa_{\overline{G}}(n) = O(2^{n^d})$ and $t_\phi(v, n) = O(2^{(\log v)^c})$,*

$$P_{par}\langle \overline{G} \rangle \subseteq \textit{EXPTIME}$$

Fundamentally, this theorem shows that in order to be able to make a problem beyond EXPTIME tractable, a network topology must implicitly contain the results of substantially difficult computations in its structure.

We have now shown a strong upper bound on a large class of network topologies. Some examples of network topologies which satisfy this are both the binary tree $\overline{B}$ and the path in our initial example. This suggests, for instance, that Presburger arithmetic cannot be made tractable by parallelism across a binary tree, as it is not within EXPTIME. [24] We identify some simple network topologies that satisfy this upper bound in Appendix A.5, including grids, all finite topologies and all constant-degree trees.

It is a desirable goal to find a PSPACE upper bound such that, coupled with our PSPACE lower bound in Section 3.1, we could find a class of network topologies in the second machine class. An attempt at this is given in Appendix A.4, which unfortunately fails to maintain our PSPACE lower bound in the process. Resolving this is left as future work.

4 Evaluation

After having considered this model's complexity theory in depth, we now take the time to return to the goals we originally posed for Turing Networks to fulfill in the project outline. Our first requirement is trivially met; our model is literally such a system. Our second requirement has been achieved throughout Chapter 3, showing we can define meaningful complexity classes and deduce substantial complexity bounds for a wide range of network topologies, especially for polynomial time parallel problems. We claim our third requirement is sated similarly, as the bounds we have found constrain tractable parallel problems between two complexity classes for Turing Machines. We have also provided an embedding of Turing Networks in Turing Machines in the process; the reverse simulation is trivial.

Our fourth requirement, however, has been left unconsidered thus far; we have not yet simulated other well-known models of parallelism with our own, a feat that would cement the relevance of our model by ensuring it can enact the behaviors that make other models so useful. It is now time we produce such simulations, namely of BSP and Boolean circuits; with this in place, we will have a sound case for the utility of Turing Networks.

4.1 Simulating BSP

BSP has found widespread use as a model for practical parallel computation [1] [13] [12], as it presents a useful and adaptable model for synchronization. To show our model for parallelism encapsulates other practical approaches, it is relevant for us to take BSP as a case study for simulation. We will show our model can simulate the two major traits given in [13] that identify BSP:

1. **Global Supersteps:** "The execution of a program proceeds in supersteps, in which concurrent components (normally called processes) compute asynchronously, with a global synchronisation at the end of each superstep." [13, p. 1]
2. **Global Communication:** "Processes communicate with one another asynchronously. All communication operations take effect at the synchronisation point." [13, p. 1]

We note that the major hurdle will be to emulate these global operations with local ones, as our model does not support global synchronization as a

primitive. We will assume that local communication is sufficient to emulate global communication, as messages can be transmitted across a network. Issues here can be relegated to network topology design.

Hence, we will attempt to emulate a system which performs local computations, awaits a global synchronization, transmits messages and then repeats *ad infinitum*, which we claim to be a sufficient simulation of the BSP model. In order to perform a global synchronization, we will place the master vertex in charge of coordination; it will signal for a global superstep to end, and it will decide when the next one begins. Broadcasts of these decisions will have to be designed such that a vertex done synchronizing does not interfere with those still doing so. We will assume the existence of special characters that will represent four special messages we will name SYNC, FINISH, ACK, and NEXTSTEP. A superstep will proceed as follows:

1. All vertices perform local computations, not sending messages.
2. The master vertex finishes its computations and sends SYNC to its neighbors, along with whatever data it wished to send to them.
3. Every slave, when done with its local computations, enters a waiting state, ready to receive SYNC from any neighbor. Once it receives one, it records the data this neighbor sent (possibly nothing), and enters a second state.
4. In this second state, each vertex now tries to send SYNC to all neighbors, along with whatever data it wishes to send them (possibly nothing). It also waits for such messages from all neighbors it has not received one from yet, recording data sent (possibly nothing). If a neighbor still in the slave starting state receives SYNC, it ignores it if no data is sent and enters the same waiting state otherwise.
5. Once every neighbor has been sent SYNC and has sent one as well, the vertex enters a second waiting state.
6. The master, on reaching this state, sends FINISH to every neighbor.
7. This message is broadcasted by all waiting vertices to produce a spanning tree; if a vertex has already received a FINISH message, it will reject all others and try and send FINISH to all neighbors similarly. All vertices who are done doing this wait again. A vertex in the slave starting state rejects such messages.
8. When no neighbor accepts a vertex's FINISH message, it knows itself to be a leaf. It sends ACK back to its parent; after all its children send it ACK, a non-leaf sends ACK to its parent. When the master receives ACK from all neighbors, all vertices are ready for the next superstep.
9. The master sends NEXTSTEP to all children in the spanning tree, which is broadcasted down the tree recursively. When a vertex has sent NEXTSTEP to all its children, it returns to step 1.

This algorithm works since the master vertex will not decide the superstep

ends until all vertices have told it so in step 8. Hence, all vertices will be in the same state between steps 8 and 9, having sent all communications for the superstep and ready to start the next. Vertices can then freely begin the next superstep, knowing their neighbors are either in this superstep or waiting to begin it by sending all NEXTSTEP messages they need to. Vertices in the latter case importantly won't accept SYNC messages yet, meaning they will definitely execute the next superstep's local computations.

The complexity of one superstep is proportional to the longest path in the currently visible vertices times the length of the longest message sent, which after n supersteps and message length x is $O(nx)$. This is far longer than the constant time taken in BSP, but we argue this to be irrelevant; as shown by the $\mathcal{LOCAL}$ model [11], locality is central to distributed systems, which are the most realistic model for a system that can grow to suit larger problems. Hence, constant-time global operations are unrealistic for a variable-sized system and a constant factor to ignore for one of fixed scale. Regardless, our simulation is polynomial-time and constant-space per vertex, showing we can simulate practical models of parallelism with identical restrictions to the Invariance Thesis. We could thus conclude our model to be 'reasonable' with respect to well-known models like BSP.

4.2 Simulating Boolean Circuits

It is known that Boolean circuits are an approximate simulation of parallelism [15, p. 118], hence classes like NC, the class of all problems solvable by a Boolean Circuit of polynomial size and polylogarithmic depth [8, p. 45] are of interest to parallel complexity theory. It is in our best interest to establish this connection to parallelism through our own model to show we can approximate parallelism in a similar manner.

Unfortunately, we quickly arrive at an impasse: our model requires the input to be present only at the master processor, whereas Boolean circuits assume each bit is available to a different logic gate immediately. We will find the linear-time complexity of transferring each bit of the input to a different processor to be insurmountable, and must therefore find a workaround in our complexity analysis.

A further issue arises since Boolean circuits are a model of *nonuniform computation* [15, p. 118], meaning different inputs are processed by different kinds of processors. In the context of Boolean circuits, this is because a given circuit can only accept a fixed input size, so an infinite family of circuits is required to deal with arbitrary input length. The proof given in [15] that Boolean circuits represent parallel computations assumes a similar kind of nonuniformity for parallel models as well, also assuming input is distributed across the network at the start.

To tackle both of these issues, we may choose some prefix of a derivation sequence to be the *setup computation*, where we send data to the correct positions to emulate some other model of parallelism that would have these at the required locations immediately. We can then subtract the time taken to do this from our time function, meaning many of our definitions can be reused. Of course, this will require the desired initial state to be reachable from our standard one, which is reasonable as an unreachable initial state would not arise in any constructible system.

This ignorance of a setup computation is reasonable as a Boolean circuit, in order to be used in any realistic setting, would have to be constructed itself and input appropriately provided before it could compute anything. Our model merely makes this requirement explicit, so we must label these computations appropriately. This reflects the proof provided in [15], which makes clear that these setup operations must be completed beforehand.

We now emulate a family of Boolean circuits $\{C_n : n \in \mathbb{N}\}$ which computes a function $f : \{0,1\}^* \to \{0,1\}^*$ with a single TN. We first define our topology, which will be an infinite rooted binary tree of root vertex 1, with extra augmentations. These augmentations will be added to each *layer* of the binary tree, meaning each set of vertices equidistant from vertex 1.

We wish to, for circuit C_n, reach the first tree layer of size larger than this circuit's size, then having the vertices in this layer each emulate a logic gate in the circuit, transmitting values between one another until the result is computed. This result will then be propagated back up to the master vertex. The augmentations we will add to each layer will comprise the interconnections necessary to transmit messages between all the vertices in a given layer in $O(\log^2 n)$ time. We will assume all gates have a constant-bounded fan-in, meaning no processor will receive more than a constant number of messages overall. This will be essential in our design of the interconnection network, owing to our requirement that all network topologies have constant degree.

We begin by noting we can derive a notion of 'left' and 'right' child of any vertex in our binary tree from our orientation. By doing this, any layer of size n can be implicitly indexed from 1 to n by left-to-right ordering. Henceforth, we assume such an indexing is in place for any layer of any tree.

For any given layer of the binary tree, say of distance $\log_2(n)$ to the root and thus of size n, we add to each vertex in the layer two new rooted binary trees with said vertex as their roots, each of whose bottom layer is of size n themselves. For vertex i, one of its trees will be titled the i^{th} *sending tree*, and the other the i^{th} *receiving tree*. We now add one more edge to each vertex in the bottom layer of each sending/receiving tree: for any $1 \leq i,j \leq n$, add an edge between vertex j of the bottom layer of the i^{th} sending tree and vertex i of the bottom layer of the j^{th} receiving tree.

We now give an algorithm to send a character from vertex i to j:

1. Send the character, along with the sender and intended recipient in binary (thus of size at most $2\log_2(n)+1$) down the send tree, navigating to the j^{th} vertex on the bottom layer of the sending tree.
2. Send the message from this vertex to the receiving tree vertex it is connected to, ie. vertex i at the bottom layer of the j^{th} receiving tree.
3. Send the message up the receiving tree until it reaches vertex j.

Overall, $2\log_2(n)$ vertices propagate a message of length $2\log_2(n)+1$, resulting in $O(\log^2 n)$ time to send one message. As no two vertices have the same sending tree nor connect to the same place on a receiving tree, all n vertices can enact steps 1 and 2 in parallel. Step 3 may lead to congestion as several messages propagate down the same vertex's receiving tree, but as we are assuming a constant bound on how many messages are sent to the same vertex, this leads only to a constant amount of extra time. Hence, the whole process takes $O(\log^2 n)$ time.

We name this entire network topology $\overline{\mathcal{B}}$. Clearly it has constant degree and 1 is in it, namely its root; as it is a countably infinite union of finite-sized layers, vertices can be made natural numbers. Thus, it is a valid topology.

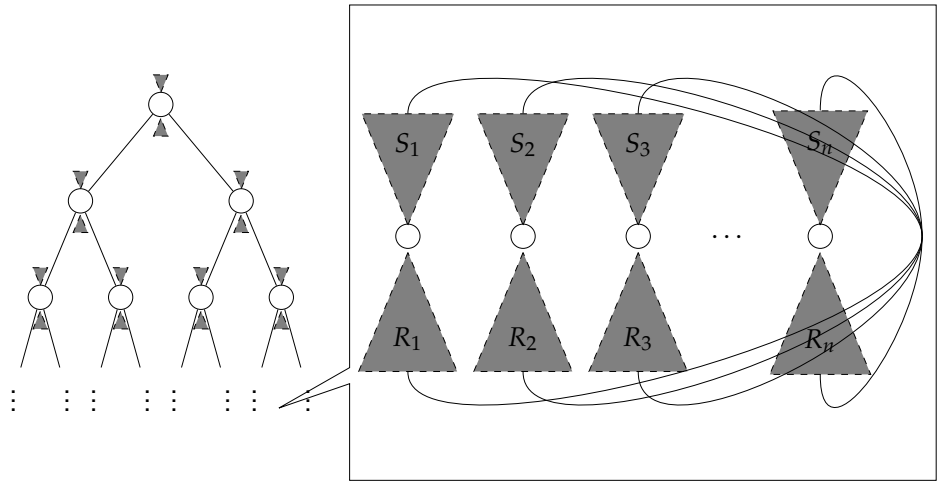

Figure 4.1: Diagram of $\overline{\mathcal{B}}$, with one layer of size n magnified. Note the send and receive trees S_i and R_i, connected at their leaves.

Now we provide a description of our simulation, starting with what we will define to be the setup computation. To begin, we compute the length n of the input provided at the master vertex and then compute the appropriate circuit description C_n, taking polynomial time. We then compute C_n's length N, writing it in base 2, and then write down the length of this expression plus 1; this gives $\lceil log_2 N \rceil$, the layer to which we will send our circuit. We then broadcast our circuit description along with this number down the binary tree; each node sends it further down and subtracts one from the number unless it is 1, in which case it is not propagated further. In these propagated

messages, we also include an extra number 0; when the message is sent to the left child, an extra 0 is added, otherwise an extra 1. By this number in the message, each vertex is indexed from left to right in the layer.

We can now send the message down, which we pad with enough 0's so it is of length $2^{\lceil \log_2 N \rceil}$, the size of the chosen tree layer. As we send this message down the tree, each vertex splits it down the middle, the lower half sent to the left child and upper half sent to the right. These are then received by the source vertices of the Boolean circuit [1]; non-source vertices await a 0 and discard it. Meanwhile, each vertex in the layer discards the entire circuit description except for the gate corresponding to its index. Each vertex knows where to send data to and receive data from by these indices as well. After this, each vertex sends a message to the vertex indexed as 1, which will then send acknowledgements to all non-source vertices to declare that computation is allowed to begin.

This marks the end of the setup computation. The vertex indexed as 1 now sends an acknowledgement to the first source vertex, who sends two acknowledgements to source vertices 2 and 3, and so forth like a binary tree, meaning all source vertices are activated in polylogarithmic time. Now, each vertex awaits all its inputs, computes the gate operation on them in constant time, and propagates the output to the destination until finally reaching destination vertices. This whole process takes $O(f(N)\log^2 N)$ time due to the extra time spent sending messages, where the circuit takes $O(f(N))$ time. The destination vertices then propagate the output back up the tree, the message being pieced back together in the inverse way the source was split up, finally arriving at the master vertex again in $O(m + m\log(N-m))$ time for output of length m. Hence, the time taken by our simulation, minus the setup computation, is

$$\tau_{\mathcal{T}}(n) = O(f(N)\log^2(N) + m(1 + \log(N-m)))$$

We note also that the setup computation time is a linearithmic function of the complexity of computing the circuit description. For NC, this means polynomial. We then gain the following theorem, since in NC we have $N = O(n^d), f(n) = O(\log^c n)$ and $m = 1$:

Theorem 6. *Any language in NC is decidable by a TN in polylogarithmic time, with setup computation taking polynomial time.*

As computing a circuit from NC takes polynomial time already, this confirms that Turing Networks are capable of simulating Boolean circuits efficiently, again within identical[2] bounds to the Invariance Thesis. This verifies our fourth requirement and shows Turing Networks can simulate both powerful theoretical and practical models of parallelism, confirming its own capacity on both fronts and demonstrating its range of applicability.

[1] We assume the source vertices are at the start of the circuit description.

[2] We claim our simulation is constant-space too; we leave the proof as future work.

5 Conclusion

In conclusion, we have managed to produce a model of message-passing parallelism which connects meaningfully to the sequential theory of Turing Machines, giving strict bounds on what improvements can be gained by moving from sequential algorithms to parallel message-passing ones. The model was also shown to be able to simulate both BSP and Boolean circuits, demonstrating it is able to efficiently encapsulate the behaviors of other well-known models of parallelism and thus transfer meaningful complexity results over to them. Thus, Turing Networks have been proven to be a useful and significant model of parallel complexity theory.

A number of limitations of this model are to be noted, namely that it assumed both a static network and one of constant degree. In modern distributed systems, for instance, dynamic networks are possible due to unreliable connections, as investigated by models like the π-calculus [25]. Hence, a more advanced model should take this into account. This could have been achieved by having the network topology gain or lose edges on each derivation, but this is beyond the scope of this project. Also, requiring a constant-degree network could be altered with a model that writes down an identifier for which neighbor it wishes to communicate with, but this was also considered beyond the scope of this project.

Another issue is that only time complexity was investigated, and nothing with respect to space or network usage. A full investigation of any such metric for complexity, however, could make for a dissertation in its own right, and is left as future work.

More future work is to be found in classifying network topologies by complexity bounds they introduce; such an explicit capacity to analyze individual topologies is a strength of this model. In particular, finding the classes of topology that fit into the first and second machine classes is a particularly interesting problem to investigate. Identifying these classes exactly is beyond the scope of this project.

Overall, this project has achieved its original goals, and it is hoped that the Turing Network model may find use in the future as a powerful tool for message-passing parallel complexity theory.

A Appendix

A.1 Network Topologies

This section contains miscellaneous information about network topologies.

Theorem 7. *The collection of all network topologies, Ξ, is a set. Furthermore, it is uncountable.*

To see this, every set of vertices is a subset of the natural numbers, and every set of edges is a subset of $\mathbb{N}^2$. An orientation function is also always a subset of $\mathbb{N}^3$. Hence, overall, we can produce an injection from Ξ to $\mathcal{P}(\mathbb{N}^6)$, making this collection a set.

To show Ξ is uncountable, we can produce an injection from $\mathcal{P}(\mathbb{N} \setminus \{1\})$ to Ξ by mapping $X \subseteq \mathbb{N} \setminus \{1\}$ to a linked list whose vertices are 1 and the elements of X in ascending order. □

We present a definition for network topology isomorphism below:

Definition 14. *Network topologies $\overline{G} = \langle G, \phi_G \rangle$ and $\overline{H} = \langle H, \phi_H \rangle$ are **isomorphic**, written $\overline{G} \simeq \overline{H}$, iff there exists a bijection $f : V_G \to V_H$ such that for all $v_n \in V_G, m \in \mathbb{N}$,*

$$f(1) = 1 \tag{A.1}$$
$$v_1 E_G v_2 \iff f(v_1) E_H f(v_2) \tag{A.2}$$
$$f(\phi_G(v_1, m)) = \phi_H(f(v_1), m) \tag{A.3}$$

We require (A.1) so that the master vertex does not change location in the graph, as this affects computations substantially. (A.2) simply requires this to be a standard graph isomorphism; recall that E_G and E_H, the edge relations of G and H, are indeed relations. (A.3) identifies the orientations with one another so the orientation remains the same structurally. Note that we do not need to ensure both orientations are defined on these inputs, as (A.2) and the definition of an orientation ensures this to be the case.

Theorem 8. *Network topology isomorphism is an equivalence relation on Ξ. Furthermore, $\Xi/\simeq$ is uncountable.*

We do not give a proof of $\simeq$ being an equivalence relation, as it is routine. To see the second point, note that we can construct an injection $(0,1) \to \Xi/\simeq$ by, for $x \in (0,1)$, converting x to its binary representation and creating a linked list graph. We encode x in the orientation; namely, if v_n is the n^{th} vertex to the right, we require $\phi(v_n, 1) = v_{n-1}$ if x's n^{th} decimal place in binary is 0, and v_{n+1} if it is 1. As no two such graphs corresponding to different numbers will be isomorphic, the set $\Xi/\simeq$ is uncountable. □

This means that there are an uncountable number of 'unique' topologies, since, as we saw from Theorem 1 in Chapter 3, two isomorphic network topologies will yield identical complexity classes. We can expect this for any other measure of complexity than time as well; a TN cannot access the names of its vertices, so there is no way for it to tell which topology in an equivalence class of $\simeq$ it is operating on.

We briefly present the construction of a Cartesian product of two network topologies, an extension of the Cartesian product of two graphs [26, p. 9] to be used in Appendix A.5:

Definition 15. *The **Cartesian product** of two network topologies $\overline{G}, \overline{H}$, written as $\overline{G}\Box\overline{H}$, is a network topology $\langle G_\Box, \phi_\Box \rangle$ such that $G_\Box = \langle V_\Box, E_\Box \rangle$ and*

$$v_{ij} = \frac{1}{2}(i+j)(i+j+1) + j \tag{A.4}$$

$$V_\Box = \{v_{ij} \mid i \in V_G, j \in V_H\} \tag{A.5}$$

$$v_{ij} E_\Box v_{kl} \iff (iE_G k \wedge j = l) \vee (jE_H l \wedge i = k) \tag{A.6}$$

$$\phi_\Box(v_{ij}, 2n) = v_{kl} \iff (j = l \wedge \phi_G(i, n) = k) \tag{A.7}$$

$$\phi_\Box(v_{ij}, 2n+1) = v_{kl} \iff (i = k \wedge \phi_H(j, n) = l) \tag{A.8}$$

(A.4) is simply the Cantor pairing function, here used to injectively map each pair of vertices to a unique new vertex, as in (A.5). The edge relation is standard; note that in the orientation in (A.7) and (A.8), we change so that even-numbered neighbors are now neighbors as per $\overline{G}$, and odd-numbered ones are neighbors as per $\overline{H}$. Notably, this choice means $\overline{G}\Box\overline{H} \not\simeq \overline{H}\Box\overline{G}$ in general, a potentially undesirable trait of our encoding. Fixing this is left as future work.

Our definition does, however, have reasonable complexity in the following sense:

Theorem 9. *For any network topologies $\overline{G}, \overline{H}$,*

$$\gamma_{\overline{G}\square\overline{H}}(n) = O(\gamma_{\overline{G}}(n)\gamma_{\overline{H}}(n))$$
$$\kappa_{\overline{G}\square\overline{H}}(n) = O((\kappa_{\overline{G}}(n) + \kappa_{\overline{H}}(n))^2)$$
$$t_{\phi_{\overline{G}\square\overline{H}}}(v_{ij}, n) = O(\max(t_{\phi_{\overline{G}}}(i, \lfloor n/2 \rfloor), t_{\phi_{\overline{H}}}(j, \lfloor n/2 \rfloor)))$$

Seeing each of these to be true is routine application of the definitions. This means we do not suddenly gain, for instance, exponential complexity by taking a Cartesian product of two topologies, which we should expect.

A.2 Computing Undecidable Problems

Below is given a proof that any decision problem, here stated as the characteristic function of a subset of the natural numbers, can be computed by some Turing Network. This shows that, given arbitrary network topologies, even undecidable problems can be solved by a Turing Network, indicating we must restrict our networks to those that are computable or even tractable.

Theorem 10. *Any decision problem $g : \mathbb{N} \to \{0, 1\}$ can be computed by some TN in $O(n^2)$ time.*

To prove this, consider some such function g. We use the same graph G as we used in our example in Section 2.1.1, namely a path of successive vertices starting with 1. However, our orientation ϕ is now different:

$$\phi(1, 1) = 2$$
$$\phi(n + 1, 1) = n + 2g(n) - 1$$
$$\phi(n + 1, 2) = n - 2g(n) + 1$$

This means neighbor 1 of vertex n will be $n - 1$ if $g(n - 1) = 0$, and $n + 1$ otherwise, the reverse holding for neighbor 2. To exploit this, we create a TN which broadcasts characters to every neighbor recursively, counting down until it reaches the node n steps away in a similar fashion to our earlier example. Messages sent back to a predecessor are ignored, which is necessary as we no longer know which neighbor is the successor by the orientation alone. When this node is reached, it sends a special character to neighbor 1; if this neighbor has sent it a message, neighbor 1 is the predecessor and this fact is broadcast until it reaches the master node. The inverse case is handled dually, and we have computed $g(n)$ in quadratic time by virtue of sending length n messages n times. □

A.3 Turing Machines

We include the definitions of standard Turing Machines we assume throughout this project here.

Definition 16. *A **Turing Machine** is a tuple*

$$T = \langle Q, \Sigma, \Gamma, q_0, h_a, h_r, \delta \rangle$$

where Q, Σ are nonempty finite sets, $\Sigma \cup \{\Lambda\} \subseteq \Gamma$, $q_0, h_a, h_r \in Q$, and $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$.

*We say that Q is the **set of states**, Σ is the **input alphabet**, Γ is the **tape alphabet**, q_0 is the **initial state**, h_a is the **accepting state**, h_r is the **rejecting state** and δ is the **transition function**.*

Our definition as above is equivalent to the standard one given in [27, p. 319], albeit differing in that the blank symbol Λ is implicitly in Γ, there is only one accepting state and there is an explicit rejecting state. Transforming between these two definitions is trivial.

We will henceforth paraphrase Turing Machine with TM. We will also assume TMs have an infinite tape in the right direction only, with the read-write head starting at the leftmost position in the tape.

We now define a configuration of a TM, ie. a 'snapshot' of the tape contents, the read-write head's position on the tape, and the current state of the TM. This is represented as a list of the tape contents to the left of the head, the character just below the head, the current state, and the contents to the right of the head that have been visited by the head before.

Definition 17. *A **configuration** of a TM T is a tuple $C \in \Gamma^* \times \Gamma \times Q \times \Gamma^*$.*

*We say T **transitions** from one configuration $C_1 = \langle r_1, s_1, q_1, t_1 \rangle$ to $C_2 = \langle r_2, s_2, q_2, t_2 \rangle$, written $C_1 \vdash C_2$, iff $\delta(q_1, s_1) = \langle q_2, s, X \rangle$ and*

$$\begin{aligned} X = S \Rightarrow \quad & r_1 = r_2, \qquad s = s_2, \quad t_1 = t_2 \\ X = L \Rightarrow \quad & r_1 = r_2 s_2, \quad t_2 = s t_1 \\ X = R \Rightarrow \quad & r_2 = r_1 s, \quad t_1 = s_2 t_2 \end{aligned}$$

The definition of a transition captures the concept of a read-write head reading a character, transitioning to a new state from its old one, writing a new character and moving left/right one step or staying still.

A.4 Space Complexity

We continue the calculations in Section 3.2 to produce a space complexity of our simulation of a TN $\mathcal{T}$. The reader should familiarize themselves with this section if they have not already.

The space required to simulate our 3-tape TM on a one-tape one is bounded by the sum of all tape sizes, ignoring some constant factor. We know the first two tapes are of size $O(h(n))$ and $O(k(n))$. At any point, our third tape contains the list of sending/receiving tapes or the intermediate state necessary to compute a vertex's neighborhood. Hence, the maximum space needed for transition n is $s(n) = O(h(n) + k(n) + \Phi_s(n))$. Here, we have defined a new function Φ_s, defined as

$$\Phi_s(n) = \max_{v \in N_{\overline{G}}(n+1), m \in \mathbb{N}} s_\phi(v, m)$$

where s_ϕ is the space complexity of our orientation function. Hence, Φ_s is the most space needed to compute any neighbor of any vertex within $n+1$ steps from vertex 1. To ascertain the overall space complexity, note that this complexity is simply that of the most space needed by any transition. Thus, the overall space required is

$$\begin{aligned} s_{\mathcal{T}}(x) &= O(h(\tau_{\mathcal{T}}(x)) + k(\tau_{\mathcal{T}}(x)) + \Phi_s(\tau_{\mathcal{T}}(x))) \\ &= O(x + \gamma_{\overline{G}}(\tau_{\mathcal{T}}(x))\kappa_{\overline{G}}(\tau_{\mathcal{T}}(x)+1) + \Phi_s(\tau_{\mathcal{T}}(x))) \end{aligned}$$

We now gain a simple theorem for space complexity:

Theorem 11. *For any network topology $\overline{G}$ and function $f : \mathbb{N}_0 \to \mathbb{N}_0$,*

$$PARTIME_{\overline{G}}(f(n)) \subseteq DSPACE\Big(n + \gamma_{\overline{G}}(f(n))\kappa_{\overline{G}}(f(n)+1) + \Phi_s(f(n))\Big)$$

In particular, we may test this bound with $P_{par}\langle\overline{G}\rangle$:

$$P_{par}\langle\overline{G}\rangle \subseteq DSPACE\Big(n + \gamma_{\overline{G}}(O(n^d))\kappa_{\overline{G}}(O(n^d)) + \Phi_s(O(n^d))\Big)$$

We note that this requires $\gamma_{\overline{G}}, \kappa_{\overline{G}}$ and Φ_s to all be within $O(n^d)$, meaning this must hold for s_ϕ as well since $\Phi_s(n) = O(\kappa_{\overline{G}}(s_\phi(n)))$. However, if $\gamma_{\overline{G}}(n) = O(n^d)$, note that we lack our condition placing $PSPACE$ within $P_{par}\langle\overline{G}\rangle$ from section 3.1. We obtain the following theorem:

Theorem 12. *For any network topology $\overline{G}$ such that $\gamma_{\overline{G}}, \kappa_{\overline{G}} = O(n^d)$ and $\phi \in PSPACE$,*

$$P_{par}\langle\overline{G}\rangle \subseteq PSPACE$$

A.5 Upper Bounds on Network Topologies

Continuing from Section 3.3, we present some simple laws that let us derive network topologies that satisfy Theorem 5. We first define this set of topologies:

$$T_{EXP} = \{\overline{G} \mid \kappa_{\overline{G}}(n) = O(2^{n^d}), t_\phi(v,n) = O(2^{(\log v)^c}), d, c \in \mathbb{N}\}$$

We now present some basic network topologies and constructions that preserve membership in this set.

Theorem 13. *Any finite network topology is in T_{EXP}.*

Clearly $\kappa_{\overline{G}}$ and thus $\gamma_{\overline{G}}$ are in $O(1)$ if $\overline{G}$ is finite. To see why this holds for the orientation, note that the entire orientation function can be represented as a finite set of ordered 3-tuples, each containing two arguments and their result. This can be seen as a finite string, and for any finite string, there exists a Turing Machine that only writes that string out in constant time and does nothing else. Hence, $t_\phi(v,n) = O(1)$.

Theorem 14. *Any rooted tree topology of constant degree is in T_{EXP}.*

Here, we define constant-degree rooted tree topologies similarly to the binary tree topology $\overline{B}$, but with n children rather than just 2. This includes the path topology we investigated in Section 2.1.1, a tree where each vertex has one child. It is routine to check this is true.

Theorem 15. $\overline{G}, \overline{H} \in T_{EXP} \Rightarrow \overline{G} \square \overline{H} \in T_{EXP}$

To see this, simply refer to Theorem 9 in Appendix A.1 and apply the constraints on T_{EXP}.

This suggests, for instance, that the grid topology, simply the Cartesian product of two path topologies, is in T_{EXP}.

Bibliography

[1] D. Skillicorn, J. M. D. Hill and W. F. McColl, 'Questions and answers about BSP', vol. 6, pp. 249–274, Jan. 1997. DOI: 10.1155/1997/532130.

[2] 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*, ser. PPOPP '93, San Diego, California, USA: ACM, 1993, pp. 1–12. DOI: 10.1145/155332.155333.

[3] J. Wiedermann, *Parallel Turing machines*. Department of Computer Science, University of Utrecht The Netherlands, 1984.

[4] P. Qu, J. Yan, Y.-H. Zhang and G. R. Gao, 'Parallel Turing machine, a proposal', *Journal of Computer Science and Technology*, vol. 32, no. 2, pp. 269–285, Mar. 2017. DOI: 10.1007/s11390-017-1721-3.

[5] J. Dean and S. Ghemawat, 'Mapreduce: Simplified data processing on large clusters', *Commun. ACM*, vol. 51, no. 1, pp. 107–113, DOI: 10.1145/1327452.1327492.

[6] D. Peleg, *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, 2000. DOI: 10.1137/1.9780898719772.

[7] S. Fortune and J. Wyllie, 'Parallelism in random access machines', in *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, ser. STOC '78, ACM, 1978, pp. 114–118. DOI: 10.1145/800133.804339.

[8] R. Greenlaw, H. J. Hoover and W. L. Ruzzo, *Limits to Parallel Computation: P-completeness Theory*. Oxford University Press, Inc., 1995, ISBN: 0-19-508591-4.

[9] V. E. with Robert van de Geijn and E. Chow, *Introduction to High Performance Scientific Computing*. lulu.com, 2011. DOI: 10.5281/zenodo.49897.

[10] G. E. Blelloch and B. M. Maggs, 'Algorithms and theory of computation handbook', in, M. J. Atallah and M. Blanton, Eds., Chapman & Hall/CRC, 2010, ch. Parallel Algorithms, pp. 25–25, ISBN: 978-1-58488-820-8.

[11] P. Fraigniaud, A. Korman and D. Peleg, 'Towards a complexity theory for local distributed computing', *J. ACM*, vol. 60, no. 5, 35:1–35:26, DOI: 10.1145/2499228.

[12] M. W. Goudreau, J. M. D. Hill, K. Lang, B. Mccoll, S. B. Rao, D. Stefanescu, T. Suel and T. Tsantilas, 'A proposal for the BSP worldwide standard library (preliminary version)', Feb. 1997. [Online]. Available: http://www.bsp-worldwide.org/standard/stand2.htm.

[13] H. Jifeng, Q. Miller and L. Chen, 'Algebraic laws for BSP programming', in *Euro-Par'96 Parallel Processing*, L. Bougé, P. Fraigniaud, A. Mignotte and Y. Robert, Eds., vol. 2, Springer Berlin Heidelberg, 1996, pp. 359–368. DOI: 10.1007/BFb0024724.

[14] L. G. Valiant, 'A bridging model for multi-core computing', *Journal of Computer and System Sciences*, vol. 77, no. 1, pp. 154–166, 2011. DOI: 10.1016/j.jcss.2010.06.012.

[15] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*, ser. IT Pro. Cambridge University Press, 2009. DOI: 10.1017/CBO9780511804090.

[16] P. van Emde Boas, 'Machine models and simulation', in *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., vol. A, Elsevier, 1990, ch. 1, pp. 1–66.

[17] W. J. Savitch and M. J. Stimson, 'Time bounded random access machines with parallel processing', *J. ACM*, vol. 26, no. 1, pp. 103–118, Jan. 1979. DOI: 10.1145/322108.322119.

[18] I. Parberry, 'Parallel speedup of sequential machines: A defense of parallel computation thesis', *SIGACT News*, vol. 18, no. 1, pp. 54–67, Mar. 1986. DOI: 10.1145/8312.8317.

[19] N. Blum, 'A note on the parallel computation thesis', *Information Processing Letters*, vol. 17, no. 4, pp. 203–205, 1983. DOI: 10.1016/0020-0190(83)90041-8.

[20] P. Fraigniaud, A. Korman and D. Peleg, 'Local distributed decision', in *Proceedings of the 2011 IEEE 52Nd Annual Symposium on Foundations of Computer Science*, ser. FOCS '11, IEEE Computer Society, 2011, pp. 708–717. DOI: 10.1109/FOCS.2011.17.

[21] N. Dershowitz and E. Falkovich-Derzhavetz, 'On the parallel computation thesis', *Logic Journal of the IGPL*, vol. 24, no. 3, pp. 346–374, 2016. DOI: 10.1093/jigpal/jzw008.

[22] 'A note on the size of prenex normal forms', *Inf. Process. Lett.*, vol. 116, no. 7, pp. 443–446, Jul. 2016. DOI: 10.1016/j.ipl.2016.03.005.

[23] J. Hartmanis and R. E. Stearns, 'On the computational complexity of algorithms', *Transactions of the American Mathematical Society*, vol. 117, pp. 285–306, 1965. DOI: 10.1090/S0002-9947-1965-0170805-7.

[24] M. J. Fischer and M. O. Rabin, 'Super-exponential complexity of Presburger arithmetic', in *Quantifier Elimination and Cylindrical Algebraic Decomposition*, B. F. Caviness and J. R. Johnson, Eds., Springer Vienna, 1998, pp. 122–135. DOI: 10.1016/j.jsc.2015.11.008.

[25] R. Milner, *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999, ISBN: 0-521-65869-1.

[26] J. Bang-Jensen and G. Z. Gutin, *Digraphs: Theory, Algorithms and Applications*, 2nd. Springer Publishing Company, Incorporated, 2008. DOI: 10.1007/978-1-84800-998-1.

[27] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation, Second Edition*. Addison-Wesley, 2000. DOI: 10.1145/568438.568455.