Relations between Space-Bounded and Adaptive Massively Parallel Computations

by

Michael Qi Yin Chen

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Pavan Aduri, Major Professor
Soma Chaudhuri
Christopher Quinn

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2023

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS

I would like to thank my family, Dr. Yang Yin Mae, Teresa Chen, and Jessica Chen, for their extended support. I would also like to thank Dr. Pavan Aduri and Dr. Vinodchandran Variyam for all the helpful discussions.

ABSTRACT

This thesis studies the class of languages solvable by Adaptive Massively Parallel Computations in constant rounds from a computational complexity theory perspective. A language L is in the class AMPC^0 if for every $\varepsilon \in (0,1)$, there is a deterministic AMPC algorithm running in constant rounds with a poly(n) processors, where the local memory of each machine $O(n^{\varepsilon})$. This thesis proves $\mathsf{L} \subsetneq \mathsf{AMPC}^0$ and then further improves it by showing $\mathsf{ReachUL} \subsetneq \mathsf{AMPC}^0$. The complexity class $\mathsf{ReachUL}$ lies between the well-known space-bounded complexity classes L and NL. We also show that $\mathsf{AMPC}^0 \subseteq \cap_{\varepsilon \in (0,1)} \mathsf{DSPACE}(n^{\varepsilon})$, which is stronger than $\mathsf{AMPC}^0 \subseteq \mathsf{SubEXP}$. We establish that it is unlikely that PSPACE admits AMPC algorithms, even with polynomially many rounds. We also establish that showing PSPACE is a subclass of (nonuniform) AMPC with polynomially many rounds leads to a significant separation result in complexity theory, namely PSPACE is a proper subclass of $\mathsf{EXP}^{\Sigma_2^{\mathsf{P}}}$.

CHAPTER 1. PRELIMINARY DEFINITIONS

1.1 Notation and Conventions

We start our natural numbers from 0, i.e., $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$. For $n \in \mathbb{N}$, $[n]$ denotes the set $\{1, \ldots, n\}$. n will typically be used to denote the input size. We use $\text{poly}(n)$ to denote the class of functions f such that f is a function from $\mathbb{N}$ to $\mathbb{N}$ and $f \in O(n^k)$ for some $k \in \mathbb{N}$.

We shall fix the alphabet to be $\Sigma = \{0, 1\}$ and use Σ^* to denote the set of all finite strings over Σ. For $n \in \mathbb{N}$, Σ^n represents the set of strings of length n over Σ. For a string $x \in \Sigma^*$, $|x|$ represents the length of the string.

We also assume the reader is familiar with the common notions of complexity theory from standard complexity theory textbooks such as Arora and Barak (2009); Sipser (1996).

1.2 Decision Problems/Languages

Definition 1. *Any subset of Σ^* is a decision problem/language.*

We shall deal with finite objects that are not strings; as such, we shall use $\langle \cdot \rangle$ to denote a binary encoding of that object. e.g. let G be a graph, then $\langle G \rangle \in \Sigma^*$ is the binary encoding of G.

We shall now define some of the decision problems that are required.

Definition 2 (D1-Reach)**.** D1-Reach *is the language consisting of tuples $\langle G, a, b \rangle$ such that*

(1) $G = (V, E)$ is a directed graph

(2) every vertex has an out-degree of at most 1

(3) there exists a directed path from a to b.

Let $G = (V, E)$ be a directed graph such that every vertex has an out-degree of at most 1. For $v \in V$, outdeg(v) denotes the out-degree of v in G. If outdeg$(v) = 1$, then let child(v) denote the unique vertex v points to.

Definition 3 (Reach-Unambiguous)**.** Reach-Unambiguous *is the language consisting of tuples* $\langle G, a, b \rangle$ *such that*

(1) $G = (V, E)$ *is a directed graph*

(2) for all $u \in V$ *there exists at most 1 directed path from* a *and* u

(3) there exists a directed path from a *to* b*.*

1.3 Turing Machine-Based Complexity Classes

We assume the reader is familiar with Turing Machines. We shall view Turing Machines as both deciders (machines that accept/reject strings) as well as computers (machines that compute some function from, say, Σ^* to Σ^*).

When defining complexity classes, we are typically interested in minimizing a resource. In the case of Turing Machines, it is usually space or time. In the case of circuits, resources to minimize would be size or depth.

1.3.1 Space-Bounded Complexity Classes

Definition 4. *A Turing Machine is said to be a log-space Turing Machine if the space used is* $O(\log n)$*.*

Definition 5. *Let* $s : \mathbb{N} \to \mathbb{N}$*. Define* $\mathsf{DSPACE}(s(n))$ *as those languages that are decided by a deterministic Turing Machine using* $O(s(n))$ *space.*

Note. To be technically correct we should write $\mathsf{DSPACE}(s)$ instead of $\mathsf{DSPACE}(s(n))$. A widely accepted convention is to use the notation $\mathsf{DSPACE}(s(n))$ to emphasize that the input to s is the length of the input, commonly denoted as n.

Definition 6. *Define the complexity class* L *as* $\mathsf{L} = \mathsf{DSPACE}(\log n)$*, i.e., those languages that can be decided by a deterministic log-space Turing Machine.*

1.3.2 Time-Bounded Complexity Classes

Definition 7. *Let* $t : \mathbb{N} \to \mathbb{N}$*. Define* $\mathsf{DTIME}(t(n))$ *as those languages that are decided by a deterministic Turing Machine using* $O(t(n))$ *time (or steps).*

Definition 8. *Define the complexity class* P *as* $\mathsf{P} = \bigcup_{k \in \mathbb{N}} \mathsf{DTIME}(n^k)$*, i.e., those languages that are decided by a deterministic polynomial-time Turing Machine.*

1.3.3 Reach-Unambiguous Complexity Classes

We assume the reader is familiar with non-determinism and unambiguous non-determinism. We state some of the definitions and results of reach-unambiguous Turing Machines. See Lange (1997); Allender and Lange (1998); Garvin et al. (2014) for more detailed discussions.

Definition 9. *A nondeterministic Turing Machine is called reach-unambiguous if for every configuration* C*, there is at most one path from the starting configuration to* C*.*

Note that the reach-unambiguous condition is stronger than the unambiguous condition. Reach-unambiguous Turing Machines require that every path (starting from the initial configuration) be unambiguous, whereas unambiguous Turing Machines require only the accepting path (if it exists) to be unambiguous.

Also, note that reach-unambiguous only poses the unambiguous restriction from the starting configuration to every other configuration. Imposing the unambiguous condition to every pair of configurations leads to strongly unambiguous complexity classes. Their configuration graphs are called mangroves.

Definition 10. *Let* $s : \mathbb{N} \to \mathbb{N}$*. Define* $\mathsf{RUSPACE}(s(n))$ *as those languages that are decided by a reach-unambiguous Turing Machine using* $O(s(n))$ *space.*

Definition 11. *Define the complexity class* ReachUL *as* $\mathsf{ReachUL} = \mathsf{RUSPACE}(\log n)$*, i.e., those languages that can be decided by a log-space reach-unambiguous Turing Machine.*

Observe that $\mathsf{L} \subseteq \mathsf{ReachUL} \subseteq \mathsf{NL}$, Where NL is the class of languages decided by a log-space non-deterministic Turing Machine. This follows directly from the definition of ReachUL.

1.4 Circuit-Based Complexity Classes

We assume the reader is familiar with circuits. For a circuit C, $\text{size}(C)$ denotes the number of gates in C, and $\text{depth}(C)$ denotes the length of the longest path from an input to an output. If a circuit has a single output, it can be viewed as a decider: an output of 1 means the circuit accepts the string, and an output of 0 means the circuit rejects the string. If a circuit has multiple outputs, it can be viewed as a computer: the output being the bit string generated by the outputs of the circuit.

Definition 12. *Let* $L \subseteq \{0,1\}^*$*, the characteristic function of* L *on inputs of length* n*, denoted* $\chi_L^{(n)} : \{0,1\}^n \to \{0,1\}$*, is defined as*

$$\chi_L^{(n)}(x) = \begin{cases} 1 & \text{if } x \in L \cap \{0,1\}^n \\ 0 & \text{otherwise} \end{cases}$$

for all $x \in \{0,1\}^n$

Definition 13. *Let* $L \subseteq \{0,1\}^n$ *and* C *be an* n*-input,* 1*-output circuit.* L *is said to be a computed by* C *if for all* $x \in \{0,1\}^n$

$$\chi_L^{(n)}(x) = C(x)$$

Definition 14. *For each* $i \in \mathbb{N}$*, an* NC^i *circuit is an infinite family of circuits* $(C_n)_{n\in\mathbb{N}}$ *whose fan-in is at most 2 with* $\text{size}(C_n) = O(\text{poly}(n))$ *and* $\text{depth}(C_n) = O(\log^i n)$*.*

Definition 15. *Let* $(C_n)_{n\in\mathbb{N}}$ *be an infinite family of circuits.* $(C_n)_{n\in\mathbb{N}}$ *is to log-space uniform if there exists a log-space Turing Machine* M *such that for every* $n \in \mathbb{N}$*,* M *on input* 1^n *outputs* $\langle C_n \rangle$

Uniformity. Circuits are non-uniform computation models; for each input length, a new circuit has to be defined. However, in Definition 15, we imposed uniformity by the condition that the description of each circuit should be generated by a log-space Turing Machine. Unless otherwise specified, NC^i represents the log-space uniform version of NC^i. See Mix Barrington et al. (1990) for a more detailed discussion on uniformity; the authors discuss other types of uniformity as well. Non-uniform circuits are very powerful in some cases, e.g., consider the undecidable problem, the unary version of the diagonal halting problem, K, defined as

$$K = \{1^n \mid n \in \mathbb{N} \text{ and } M_n \text{ halts on input } s_n\}$$

where $(M_n)_{n\in\mathbb{N}}$ is the standard enumeration of Turing Machines and $(s_n)_{n\in\mathbb{N}}$ is the standard enumeration of Σ^*. Observe that K belongs to non-uniform NC^1. Thus, distinguishing between uniform and non-uniform forms is very important; the reader should be able to disambiguate $\mathsf{NC}^1 \subseteq \mathsf{L}$ as uniform NC^1 is contained in L. As the non-uniform case would not hold true.

The following is an equivalent way of characterizing log-space uniform circuits.

Theorem 16. *Let $(C_n)_{n\in\mathbb{N}}$ be a circuit family. Define direct connection language for C, L_C, consisting of tuples $\langle t, a, b, 1^n\rangle$ such that a and b are nodes in C_n, b is a child of a, a is a node of type t and $n \in \mathbb{N}$. The following holds*

$$(C_n)_{n\in\mathbb{N}} \text{ is log-space uniform } \iff L_C \in \mathsf{L}$$

Definition 17. *For each $i \in \mathbb{N}$, define the complexity class NC^i as those languages L such that there exists an NC^i circuit, $(C_n)_{n\in\mathbb{N}}$ such that for every $n \in \mathbb{N}$, $L \cap \{0,1\}^n$ is computed by C_n.*

Definition 18. *Let $f : \mathbb{N} \to \mathbb{N}$. Define the complexity class $\mathsf{SIZE}(f(n))$ as those languages L such that there exists an family of circuits $(C_n)_{n\in\mathbb{N}}$ that decides L with $\text{size}(C_n) = O(f(n))$*

1.5 Computable Functions

Definition 19. *Let $i \in \mathbb{N}$ and $f : \Sigma^* \to \Sigma^*$. f is said to be NC^i computable if there exists a NC^i circuit that computes f.*

Definition 20. *Let $f : \Sigma^* \to \Sigma^*$. f is log-space computable if a log-space Turing Machine exists that computes f.*

The following is an equivalent way of characterizing log-space computable functions.

Theorem 21. *Let $f : \Sigma^* \to \Sigma^*$ and define $L_f = \{\langle i, f(x)_i \rangle \mid x \in \Sigma^* \wedge i \in \mathbb{N}\}$, where $f(x)_i$ is the i^{th} bit of $f(x)$. The following holds*

$$f \text{ is log-space computable } \iff L_f \in \mathsf{L}$$

We won't deal with the technicalities of functions vs. decision problems, but there is a distinction between L, languages decided by a log-space Turing Machine, and FL, functions computed by a log-space Turing Machine. For our purposes, we have access to polynomially many processors (in the MPC/AMPC model) so each bit of the output can be computed in parallel. Similarly, for NC^i and FNC^i and so on.

It is well known that $\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{P}$. Since a time-bound implies a space-bound, a polynomial-time Turing Machine can only write in polynomially many cells. Therefore, if $f : \Sigma^* \to \Sigma^*$ is a function computed by an NC^1 circuit or a log-space Turing Machine then there exists a $k \in \mathbb{N}$ such that $|f(x)| = O(|x|^k)$

1.6 Complete Problems

Complete problems are decision problems that are intuitively the "hardest" problem of some complexity class. Showing a complexity class is contained in another complexity class would require one to exhibit an "efficient" algorithm (efficiency depends on the complexity class you are trying to show where it is contained in) for every language in the complexity class. Finding an algorithm for every problem can be challenging. Instead, we solve the *hard* or *complete* problem to show inclusion. To solve any language in the complexity class, we first transform it to an instance of the *hard* or *complete* problem (the transformation process is called a reduction), and we then solve the new instance efficiently. This approach is what will be later referred to as the "standard reduction argument".

In many cases (but not all), solving a complete problem may imply that we have an "efficient" algorithm for every problem that reduces to it. However, in some cases, just solving the complete problem efficiently is not sufficient to show inclusion. The specifics depend on how costly the reduction is and what is the length of the reduction's output.

Definition 22. *Let $A, B \subseteq \Sigma^*$. A is said to log-space reduce to B, denoted $A \leq_{\mathsf{L}} B$, if there exists a log-space computable function f such that for all $x \in \Sigma^*$, $x \in A \iff f(x) \in B$*

Definition 23. *Let $A, B \subseteq \Sigma^*$. A is said to NC^1 reduce to B, denoted $A \leq_{\mathsf{NC}^1} B$, if there exists a NC^1 computable function f such that for all $x \in \Sigma^*$, $x \in A \iff f(x) \in B$*

Definition 24. *Let $B \subseteq \Sigma^*$. B is said to be L-hard under NC^1 reductions if for every problem $A \in \mathsf{L}$, $A \leq_{\mathsf{NC}^1} B$*

Definition 25. *Let $B \subseteq \Sigma^*$. B is said to be L-complete if*

1. *$B \in \mathsf{L}$*
2. *B is L-hard under NC^1 reductions*

It is well known $\mathsf{NC}^1 \subseteq \mathsf{L}$. When describing complete problems, we typically require the reduction to be weaker than the complexity class' power. e.g., allowing for log-space reductions for L will lead in undesirable results such as every language in L (except for $\emptyset$ and Σ^*) would be L-complete under log-space reductions.

Definition 26. *Let $B \subseteq \Sigma^*$. B is said to be $\mathsf{ReachUL}$-hard under log-space reductions if for every problem $A \in \mathsf{L}$, $A \leq_{\mathsf{L}} B$*

Definition 27. *Let $B \subseteq \Sigma^*$. B is said to be $\mathsf{ReachUL}$-complete if*

1. *$B \in \mathsf{ReachUL}$*
2. *B is $\mathsf{ReachUL}$-hard under log-space reductions*

Theorem 28 (Cook and McKenzie (1987))**.** D1-Reach *is L-complete under (uniform) NC^1 reductions.*

Theorem 29 (Lange (1997))**.** Reach-Unambiguous *is* ReachUL*-complete under log-space reductions.*

If unspecified, L-complete problems mean complete under (uniform) NC^1 reductions, and ReachUL-complete problems mean complete under log-space reductions.

CHAPTER 2. OVERVIEW

This thesis studies the computational complexity of the Adaptive Massively Parallel Computation (AMPC) Model defined by Behnezhad et al. (2019b). The AMPC model is an extension of the Massively Parallel Computation (MPC) Model defined by Beame et al. (2017). The MPC model is widely accepted as the standard theoretical model for distributed computation frameworks such as MapReduce, Spark, Hadoop, FlumeJava, Beame, Pregel, and Gigraph Behnezhad et al. (2019b); Charikar et al. (2020).

2.1 Massively Parallel Computation

For inputs of length n, the MPC model consists of $p = \text{poly}(n)$ identical machines, each with $s = O(n^{\varepsilon})$ local memory for some $\varepsilon \in (0, 1)$. Naturally, $p \cdot s \geq n$ must hold to store the input. The input is initially adversarially distributed, and computation occurs in rounds, and in each round, every machine performs computation based on its local data and then communicates with other machines with the constraint that the amount of communication by a process is equal to that of its local memory s. The goal is to minimize the total number of rounds. A salient feature of the MPC model is that no computational restriction is placed on the processor. Ideally, one would like to design an algorithm with constant rounds with a small number of processors. The MPC model has been extensively studied in the context of designing algorithms as well as its relationship with complexity classes in Andoni et al. (2018, 2019); Assadi et al. (2019); Behnezhad et al. (2019b); Beame et al. (2017); Behnezhad et al. (2019a); Roughgarden et al. (2018). We use the terms processor and machine interchangeably for an MPC/AMPC model.

In an algorithmic setting, it is typically desired that the total memory of an MPC/AMPC algorithm $p \cdot s$ to be $n \cdot \text{polylog}(n)$. However, to define a robust complexity class (closed under

reductions), we allow p to be poly(n) and require that for every $\varepsilon \in (0,1)$, there is an algorithm with $O(n^{\varepsilon})$ local memory per processor.

When discussing the round complexity of the MPC/AMPC model for a local memory $O(n^{\varepsilon})$, we hide the ε terms, e.g., $O(1/\varepsilon)$ is treated as $O(1)$. The reader may choose to view it as $O_{\varepsilon}(1)$; we choose not to use this notation for brevity's sake.

2.2 MPC Subroutines

This section discusses some of the common subroutines used in the MPC model. These can be extended to NAMPC and AMPC models as well. Note that broadcast trees are trivial in the NAMPC/AMPC models.

2.2.1 Broadcast Trees

Fix $\varepsilon \in (0,1)$. Consider an MPC model with $s(n) = O(n^{\varepsilon})$ local memory and $p(n) = \text{poly}(n)$ processors. Suppose machine 0 wants to share a bit, $\mathcal{B}$, with all other machines. Assuming that the case $p(n) > s(n)$, then a machine 0 cannot share a single bit with all the other processors in a single round due to communication restraints. However, we can achieve this goal by using broadcast trees in $O(1/\varepsilon)$ rounds. In the first round, machine 0 can broadcast $\mathcal{B}$ to machines $1, \ldots, s(n)$. In the next round, machine 1 can broadcast $\mathcal{B}$ to machines $s(n)+1, \ldots, 2s(n)$, machine 2 can broadcast $\mathcal{B}$ to machines $2s(n)+1, \ldots, 3s(n)$, so on till machine $s(n)$ can broadcast $\mathcal{B}$ to machines $(s(n)-1)s(n)+1, \ldots, s^2(n)$. Next round, each machine then broadcasts to $s(n)$ machines that do not know $\mathcal{B}$. Observe that at the end of R rounds, $O(s^R)$ machines know $\mathcal{B}$. Broadcasting to $p(n)$ machines takes, $O(\log_n p(n)/\varepsilon)$ rounds. Furthermore, from our assumption that $p(n) = \text{poly}(n)$, this procedure takes $O(1/\varepsilon)$ rounds. The resultant communication tree structure is called the *broadcast tree*.

2.2.2 Convergence Trees

We can "invert" the broadcast tree to get a *convergence tree*. A convergence tree can be used to converge multiple elements into a single element. e.g., suppose each machine has some number, and our goal is to compute the sum of all the numbers across all machines and store the resultant sum in machine 0.

Fix $\varepsilon \in (0, 1)$. Consider an MPC model with $s(n) = O(n^{\varepsilon})$ local memory and $p(n) = \text{poly}(n)$ processors. Using convergence tree machine 0 can obtain the sum of all these numbers in $O(1/\varepsilon)$ rounds.

2.2.3 Prefix Sum

Let $x_1, \ldots, x_n$ be a sequence of numbers. Our goal is to compute the prefix sum of this sequence, i.e., we want to compute $\sum_{i=0}^{q} x_i$ for all $q \in \{0, \ldots, n\}$. Observe that with the help of convergence trees, we can solve this problem in $O(1/\varepsilon)$ rounds. Furthermore, consider the more general problem of computing $\sum_{i=p}^{q} x_i$ for any $p, q \in \{0, \ldots, n\}$. With polynomially many processors, we can compute and store the sum $\sum_{i=p}^{q} x_i$ for all p, q in $O(1/\varepsilon)$ rounds.

2.3 Non-Adaptive Massively Parallel Computation

Although this thesis does not discuss the MPC model specifically, we propose a near[1] equivalent characterization of MPC using a shared memory called the Distributed Data Stores as defined in Behnezhad et al. (2019b). DDS was originally used to define the AMPC model. However, its characterization can be helpful in defining the Non-Adaptive Massively Parallel Computation (NAMPC) model, which may allow for simpler proofs in a complexity theory setting.

Definition 30. *A Distributed Data Store (DDS) is a shared (by processors) memory that contains data stored in a bit addressable manner, i.e., a collection of key-value pairs in the form of* $(i, i^{th}$ bit of the DDS$)$ *for* $i \in \mathbb{N}$.

[1] Equivalent in a complexity theory setting where we allow polynomially many processors, in an algorithmic setting, where number of processors are $n \cdot \text{polylog}(n)$, such an equivalence is unclear

Our definition of DDS is slightly weaker than Behnezhad et al. (2019b); namely, we view DDS as an indexed memory tape (akin to a Turing Machine's memory tape) rather than a collection of key-value pairs. Our definition doesn't allow a key to have multiple values, so we need to ensure that we write in a unique location for each unique bit we want.

Definition 31. *Let $p, s, r : \mathbb{N} \to \mathbb{N}$ be functions. An $\mathsf{NAMPC}[p(n), s(n), r(n)]$ algorithm for length n, is a collection of processors $M_{i,j}$ for $i \in [p(n)]$ and $j \in [r(n)]$ where each processor has a memory bound of $O(s(n))$. In addition to the processors, there is a collection of DDS denoted by $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{r(n)}$. The input string $x = x_1 \ldots x_n$ is stored in $\mathcal{D}_0$ in the form of $\{(i, x_i)\}_{i=1}^{n}$. The computation occurs in rounds. Fix a round $j \in [r(n)]$, for each $i \in [p(n)]$, the processor $M_{i,j}$ participates in the j^{th} round. At the start j^{th} round, $M_{i,j}$ is allowed to make $O(s(n))$ **fixed read queries** from D_{j-1}, after which it is not allowed to make any more read queries. It is then allowed to write $O(s(n))$ times to D_j. The computation stops after $r(n)$ rounds, and we say that the algorithm accepts string x if the value of key 1 is in $D_{r(n)}$ is 1.*

For a round j, we shall refer to D_{j-1} as the read DDS and D_j as the write DDS. We first describe the properties of this definition (which also applies to the AMPC model); then, we shall show its equivalence with the original NAMPC model.

Robustness of Word Size. NAMPC algorithms are defined to read/write only $O(n^\varepsilon)$ bits where $\varepsilon \in (0, 1)$. However, we can equivalently characterize the NAMPC model with words of size up to $O(\text{polylog}(n))$. Consider a NAMPC algorithm that reads/writes words each of length $O(\text{polylog}(n))$, then the amount of local memory is $O(n^\varepsilon \cdot \text{polylog}(n)) \subseteq O(n^\delta)$ for any $\delta \in (\varepsilon, 1)$.

Robustness of Number of DDS. Instead of allowing only one read DDS and one write DDS, we can also allow for multi-DDS NAMPC. For a given round j, we can associate a collection of read DDS $D_{j-1}^{(1)}, \ldots, D_{j-1}^{(k)}$ and a collection of write DDS $D_j^{(1)}, \ldots, D_j^{(k)}$. With a similar argument as the robustness of word size so long as $k \in O(\text{polylog}(n))$, this is an equivalent characterization of NAMPC.

Theorem 32 (MPC implies NAMPC)**.** *Fix $\varepsilon \in (0,1)$. Suppose we have an* MPC *algorithm that uses $p(n) = \text{poly}(n)$ processors per round, $s(n) = O(n^{\varepsilon})$ local memory per machine, and completes execution in $r(n)$ rounds then there exists an* $\mathsf{NAMPC}[p^2(n), s(n), r(n)]$ *algorithm.*

Proof. Consider an MPC algorithm that uses $p(n)$ processors per round, $s(n) = O(n^{\varepsilon})$ local memory per machine for some $\varepsilon \in (0,1)$, and completes execution in $r(n)$ rounds. We shall associate machines with numbers, let n the input length and $s = s(n), p = p(n), r = r(n)$. Define the NAMPC algorithm as follows:

When simulating the MPC algorithm, we first divide the DDS into chunks of size s so that each machine in the NAMPC algorithm knows exactly what its input is for that round. Since the MPC algorithm was designed to work even when input is distributed in an adversarial manner, this input arrangement works as well.

We simulate each round by introducing a $O(1)$ round overhead. Fix a round $j \in [r]$, consider machine $i \in [p]$ in the MPC model; it reads the input by making s queries to its respective chunk and proceeds to do the normal computation like in the MPC model. At the end of the round, it wants to send a collection of s messages, $m_1^{(i)}, m_2^{(i)}, \ldots, m_s^{(i)}$, each $m_k^{(i)}$ is of the form

$$\langle i, d_k^{(i)}, w_k^{(i)} \rangle$$

where i is the source machine, $d_k^{(i)} \in [p]$ is the destination processor, and $w_k^{(i)}$ is the word to be sent,

We introduce $O(1)$ rounds in the NAMPC algorithm so that at the end of these $O(1)$ rounds, the data in the DDS is in the form of chunks of size s so that the simulation for $j+1$ can proceed by each machine simply reading its respective chunk. Assume the DDS in the form of a 3-D array indexed $[1,\ldots,p][1,\ldots,p][1,\ldots,s]$. The NAMPC algorithm makes s writes to the respective cells, i.e., for each $k \in [s]$, it writes the message m_k into the index $[i][d_k^{(i)}][\star]$ where $\star$ depends on how many messages machine i wants to send to machine $d_k^{(i)}$, this ensures no collisions occur. Finally, the AMPC model can collect all the messages into chunks of size s (with the help of the MPC subroutines), and then the simulation of round $j+1$ can proceed. For prefix sums, the number of

processors required each round is $O(p^2)$ and round overhead is $O(\frac{\log_n p}{\varepsilon})$ which is $O(1)$ since we assumed $p(n) = \text{poly}(n)$. □

Theorem 33 (NAMPC implies MPC)**.** *Fix $\varepsilon \in (0, 1)$. Suppose we have an $\mathsf{NAMPC}[p(n), s(n), r(n)]$ algorithm where $s(n) = O(n^\varepsilon)$ and $p(n) = \text{poly}(n)$ then there is an MPC algorithm that uses $O(pn^{1+\varepsilon})$ processors, $s(n) = O(n^\varepsilon)$ local memory per machine and $O(r(n))$ rounds.*

Proof. Fix $\varepsilon \in (0, 1)$. Consider an $\mathsf{NAMPC}[p(n), s(n), r(n)]$ algorithm with $s(n) = O(n^\varepsilon)$ and $p(n) = \text{poly}(n)$. Observe that this NAMPC algorithm uses at most $s(n) \cdot p(n)$ bits in the DDS since each machine can write at most $s(n)$ bits each round.

Let n the input length and $s = s(n), p = p(n), r = r(n)$. Define the MPC algorithm as follows: Assume the input is of the form of the ordered pair (i, b_i), where i is the input index, and b_i is a bit. This input is distributed (in no particular order) across all the machines. In the first round, each machine sends each pair to the appropriate machine so that the input is in order, i.e., at the end of the round, the first machine has the first s bits of the input, $(1, b_1), (2, b_2), \ldots, (s, b_s)$. Similarly, the second machine has the next s bits, $(s + 1, b_{s+1}), (s + 2, b_{s+2}), \ldots, (2s, b_{2s})$, and so on. These machines now represent the DDS it wants to simulate; call it the pseudo-DDS. The number of bits in the DDS at any point of the NAMPC execution is $O(s \cdot p)$.

We simulate each round by introducing $O(1)$ round overhead. Fix a round $j \in [r]$, and consider machine $i \in [p]$ in the NAMPC model; it wants to make s non-adaptive queries. However, multiple machines might want to request the same bit: due to the limit on communication, this bit cannot directly be sent to every machine, so we duplicate the memory of the pseudo-DDS so that each machine has its own read-exclusive pseudo-DDS. We need to make p copies of $s \cdot p$ bits, via broadcast trees we need $O(\frac{\log_n p}{\varepsilon})$ rounds, which is $O(1)$ since we assumed $p(n) = \text{poly}(n)$. The maximum number of processors in a given round to duplicate the pseudo-DDS is $O(s^R) \cdot O(ps) = O(pn^{1+\varepsilon})$.

Now that each machine has its read-exclusive DDS, we can resolve queries by adding an additional round, the machine sending the bit request to the appropriate machine, and then in the

next round, the machine satisfies that request. Once this is achieved, the simulation of round $j+1$ can proceed. □

2.4 Adaptive Massively Parallel Computation

We now define the AMPC model; it is nearly identical to the NAMPC, except the algorithm can now make adaptive queries, i.e., its processor's execution path can depend on the query's output.

Definition 34. *Let $p, s, r : \mathbb{N} \to \mathbb{N}$ be functions. An $\mathsf{AMPC}[p(n), s(n), r(n)]$ algorithm for length n, is a collection of processors $M_{i,j}$ for $i \in [p(n)]$ and $j \in [r(n)]$ where each processor has a memory bound of $O(s(n))$. In addition to the processors, there is a collection of DDS denoted by $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{r(n)}$. The input string $x = x_1 \ldots x_n$ is stored in $\mathcal{D}_0$ in the form of $\{(i, x_i)\}_{i=1}^{n}$. The computation occurs in rounds. Fix a round $j \in [r(n)]$, for each $i \in [p(n)]$, the processor $M_{i,j}$ participates in the j^{th} round. At the start j^{th} round, $M_{i,j}$ is allowed to make $O(s(n))$ **adaptive read queries** from D_{j-1}. It is then allowed to write $O(s(n))$ times to D_j. The computation stops after $r(n)$ rounds, and we say that the algorithm accepts string x if the value of key 1 is in $D_{r(n)}$ is 1.*

Observe that every NAMPC algorithm is an AMPC algorithm.

Uniformity. The MPC/AMPC models are non-uniform in nature; namely, we can define each machine to execute a completely different algorithm, thus allowing the model to solve undecidable problems such as the halting problem. We impose uniformity by having each machine run the same algorithm. We assume that each machine is uniquely identified by some natural number that is given as input during execution. Unless otherwise specified, if not mentioned, MPC/AMPC algorithms are uniform.

Computing Log-space Uniform Circuits. An AMPC model can easily compute log-space uniform NC^1 reductions. Let $(C_n)_{n\in\mathbb{N}}$ be a log-space uniform NC^1 circuit generated by a log-space Turing Machine M. Via Theorem 16, the direct connection language of $(C_n)_{n\in\mathbb{N}}$ is decidable in

log-space. Fix $\varepsilon \in (0,1)$ and consider an AMPC machine with $s(n) = O(n^{\varepsilon})$ local memory. The AMPC model can compute the binary representation of the length of the input in $O(1)$ rounds using convergence trees. Observe that binary representation has length $O(\log n) \subseteq O(n^{\varepsilon})$. The binary representation of n is sufficient to decide the connection language. We can now distribute all possible pairs of nodes to $O(n^2)$ processors, and each of them can individually compute a part of the circuit and then upload it to the DDS and proceed to simulate the NC^1 circuit. Adaptive queries are not required, so this procedure also works for the MPC model.

Definition 35. *Define the complexity class* AMPC^0 *as those languages* L *such that for every* $\epsilon \in (0,1)$ *there exists a* $\mathsf{AMPC}[\text{poly}(n), n^{\varepsilon}, O(1)]$ *algorithm that decides* L*.*

Unless otherwise specified, AMPC^0 refers to uniform AMPC as described earlier.

2.5 Known Results

Behnezhad et al. (2019b) defined the AMPC model and exhibited a randomized algorithm to solve the L-complete problem, 1v2-Cycle, the promise problem of distinguishing whether a graph is an instance of a single cycle of n nodes or two cycles of $n/2$ nodes. The authors tackled the problem from an algorithmic perspective and showed that for any $\varepsilon \in (0,1)$, there is a $O(n)$ total space, $O(1)$ round AMPC algorithm using $O(n^{\varepsilon})$ local space AMPC algorithm to solve 1v2-Cycle. It is conjectured that the 1v2-Cycle cannot be solved in sublogarithmic MPC rounds.

Conjecture 36 (1v2-Cycle Conjecture)**.** *Any MPC algorithm that solves* 1v2-Cycle *using* $O(n^{\varepsilon})$ *local memory for some* $\varepsilon \in (0,1)$ *uses* $\Omega(\log n)$ *rounds. (even with polynomially many processors)*

1v2-Cycle can be reduced to more general connectivity problems such as graph connectivity, cycle detection, and so on. Thus, a $\Omega(\log n)$ MPC round lower bound is also implicitly conjectured for these general connectivity problems as well. The authors also showed that the equivalent 1v2-Cycle conjecture for the AMPC model does not hold. The authors also exhibit AMPC

algorithms that perform better (in terms of rounds) than currently known MPC algorithms for some other graph problems.

Frei and Wada (2019) showed that (uniform) NC^i circuits can be simulated in the MapReduce model of computation. Fix $\varepsilon \in (1/2, 1)$. The authors showed that for $i \geq 1$, NC^{i+1} circuits can be simulated by a MapReduce model in $O(\log^i n)$ rounds where each machine has $O(n^\varepsilon)$ local memory and runs in polynomial time, and the total memory is $O(n^{2\varepsilon})$. The authors also showed that NC^1 can be solved in $O(1)$ rounds, now ε can now be any value from $[1/2, 1)$.

Roughgarden et al. (2018) defined the s-shuffle model, an abstraction of the MPC model. Lower bounds on the round complexity of this model would apply to every computing platform similar to MapReduce. They also give a technique to separate NC^1 from P. Consider any model of parallel computation with the following properties:

1. The number of machines is polynomial in the input size n.
2. Computation proceeds in times steps. In each round, a machine can read $s(n) \geq 2$ bits from the input or from previously completed computations.
3. Each machine has enough power to evaluate a Boolean circuit with size at most $s(n)$ and depth at most $\log_2 s(n)$ in one step.

If some problem in P cannot be solved in $O(\log_{s(n)} n)$ steps in such a model, then $\mathsf{NC}^1 \subsetneq \mathsf{P}$ (The authors use the terminology "step", which is a round for the MPC/AMPC model). In proving their theorem, they justified that NC^1 can be efficiently simulated in $O(\log_{s(n)} n)$ rounds by parallel models of computations that satisfy the three conditions. In particular, the AMPC model satisfies such criteria. Therefore, for any $\varepsilon \in (0, 1)$, there exists an AMPC algorithm with $s(n) = O(n^\varepsilon)$ local space per machine to simulate NC^1 circuits in $O(\log_{s(n)} n) = O(1)$ rounds. We therefore have,

Theorem 37 (Roughgarden et al. (2018))**.** $\mathsf{NC}^1 \subseteq \mathsf{AMPC}^0$

Nanongkai and Scquizzato (2022) studied the computational complexity of the MPC model. The authors defined the $\mathsf{MPC}(f(n))$ as the class of languages solved by an MPC algorithm using

poly(n) machines and $O(n^\varepsilon)$ local memory per machine for some $\varepsilon \in (0, 1)$ and completes execution in $O(f(n))$ rounds. However, their definition of $\mathsf{MPC}(f(n))$ does not allow the standard reduction argument to prove inclusion. Let B be some L-hard problem. One of their main results (restated here in simpler terms) is proving the equivalence of the two propositions:

(1) $B \in \mathsf{MPC}(o(\log n))$

(2) $\mathsf{L} \subseteq \mathsf{MPC}(o(\log n))$

Clearly, (2) $\implies$ (1), but the proof of the converse, as presented, may not hold. NC^1 reductions result in a new problem instance that is polynomially larger than the original size. Thus, the standard reduction argument may not hold for their desired result. Namely, the memory requirements may exceed more than $O(n^\varepsilon)$. We summarise their proof approach and demonstrate the potential issue.

Proof approach. (1) $\implies$ (2) Assume $B \in \mathsf{MPC}(o(\log n))$, thus there exists an MPC algorithm that decides B in $o(\log n)$ using $O(n^\varepsilon)$ **for some** $\varepsilon \in (0, 1)$.

Since $\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{P}$, the length of the reduced instance of B could grow by a polynomial factor. Thus the new input size is larger; however, the communication constraint and local space remain the same. Formally, let $A \in \mathsf{L}$, and we wish to demonstrate a $O(n^\delta)$ local memory MPC algorithm for some $\delta \in (0, 1)$. Thus the limits of communication and local space are $O(n^\delta)$. Let x be an instance of A of length n that can be reduced to B via f, which is computed by an NC^1 circuit. Let $y = f(x)$ and $|y| = m$, we have $m = O(n^k)$ for some $k \in \mathbb{N}$. Naïvely trying to solve y using the same ε would result in the algorithm needing $O(m^\varepsilon) = O(n^{k\varepsilon})$ local memory, which would exceed our constraints if $k\varepsilon > 1$. Thus, such a δ may not exist.

CHAPTER 3. RESULTS

3.1 $\mathsf{L} \subsetneq \mathsf{AMPC}^0$

Our first result shows the strict inclusion of L in AMPC^0. We shall first prove Lemma 38, which will allow us to solve the L-complete problem, D1-Reach. After which, we prove inclusion using the standard reduction argument, strict inclusion follows from Theorem 47. Let $V = \{v_1, \ldots, v_n\}$ and let $f : V \to V$ be a function. For $k \in \mathbb{N}$, denote $f^{(k)}$ as f composed with itself k times.

Lemma 38 (Efficient Self Composition)**.** *Assume that f is stored explicitly in the DDS storage $\mathcal{D}_0$, as key-value pair $(v, f(v))$ for every $v \in V$. Then for every $t, r \in \mathbb{N}$, $f^{(t^r)}$ can be computed by an $\mathsf{AMPC}[n, t, r]$ algorithm.*

Proof. We prove the above by induction on r. The base case ($r = 0$) is clear from our assumption that f is already in the DDS. Inductively assume that $f^{(t^r)}$ computed in r rounds and is stored in DDS $\mathcal{D}_r$. We show how to compute $f^{(t^{r+1})}$ in one additional round by using additional n processors where each process makes $O(t)$ queries to $\mathcal{D}_r$. Fix $i \in [n]$, machine i requests for the $v_i \in V$. It computes $f^{(t^{r+1})}(v_i)$ by making following t adaptive queries to $\mathcal{D}_r$: $f^{(t^r)}(v_i), f^{(t^r)}(f^{(t^r)}(v_i)) \cdots$. Note that the answer to the t^{th} query is exactly $f^{(t^{r+1})}(v_i)$. The machine writes the key value pair $(v_i, f^{(t^{r+1})}(v_i))$ to $\mathcal{D}_{r+1}$. □

Fixing $\varepsilon \in (0, 1)$ as the local memory parameter. Due to query constraints, t can be at most $O(n^{\varepsilon})$. To solve D1-Reach in constant rounds, this is sufficient.

Theorem 39. D1-Reach $\in \mathsf{AMPC}^0$

Proof. Let $\langle G, a, b\rangle$ be a problem instance of D1-Reach with $G = (V, E)$ and $|V| = n$. Define $f : V \to V$ as follows

$$f(v) = \begin{cases} \text{child}(v) & \text{if outdeg}(v) = 1 \text{ and } v \neq b \\ v & \text{if outdeg}(v) = 0 \text{ or } v = b \end{cases}$$

For $k \in \mathbb{N}$ and $v \in V$, observe that $f^{(k)}(v)$ is the final vertex resulting by traveling a path of length k starting from v, provided the path is at least k long and the target vertex b has not been reached.

f is already present in the DDS as the E. Thus, instantiating Lemma 38 with $t = n^\varepsilon$ and $r = \lceil 2/\varepsilon \rceil$, we can compute $f^{(t^r)}$ in $O(1/\varepsilon)$ rounds. Accept the input if and only if $f^{(t^r)}(a) = b$. Since $t^r \geq n$, i.e., t^r is at least the diameter of G, therefore, if there exists a path from a to b, then the algorithm will detect it and accepts it. □

We now prove the inclusion of L in AMPC^0 by the standard reduction argument, i.e., to solve any problem in L, we reduce (via an NC^1 reduction) it to an instance of D1-Reach and solve the new instance instead.

Theorem 40. $\mathsf{L} \subsetneq \mathsf{AMPC}^0$

Proof. We shall show $\mathsf{L} \subseteq \mathsf{AMPC}^0$, the strictness of the inclusion follows from Theorem 47. Let $A \in \mathsf{L}$ and fix $\varepsilon \in (0, 1)$, we will exhibit an $\mathsf{AMPC}[\text{poly}(n), n^\varepsilon, O(1)]$ algorithm that decides A. Since $A \in \mathsf{L}$, via hardness of D1-Reach, Theorem 28, $A \leq_{\mathsf{NC}^1}$ D1-Reach via some g that is computable by an NC^1 circuit. Thus there exists a $k \in \mathbb{N}$ such that, $m = |g(x)| = O(|x|^k)$. By Theorem 37, there is a AMPC^0 algorithm that computes g.

Our algorithm for A works as follows: Let x be an input instance of A of length n, we invoke the AMPC^0 (with $s = n^\varepsilon$) algorithm to compute $g(x)$. Set $\delta = \varepsilon/k$, by Theorem 39 there is an AMPC^0 algorithm that checks whether $g(x)$ is in D1-Reach where the local memory of each processor is $O(m^\delta) = O(n^\varepsilon)$. The final algorithm is an AMPC^0 algorithm where each processor has local memory n^ε. Clearly, the algorithm runs in constant rounds and has polynomially many processors. □

Corollary 41. *Let f be a log-space computable function. For any $\varepsilon \in (0, 1)$, there is an $\mathsf{AMPC}[\text{poly}(n), O(n^\varepsilon), O(1)]$ algorithm to compute f.*

Proof. Let f be a log-space computable function. Let x be an input of length n, and $y = f(x)$ be of length m. We have $m = O(n^k)$ for some k. Define $L_f = \{\langle i, f(x)_i \rangle \mid x \in \Sigma^* \wedge i \in \mathbb{N}\}$, where $f(x)_i$ is the i^{th} bit of $f(x)$. Via Theorem 21 and Theorem 40, we have $L_f \in \mathsf{AMPC}^0$. Since we

have polynomially many processors and $L_f \in \mathsf{AMPC}^0$, the AMPC model can compute $\langle i, f(x)_i \rangle$ for each $i \in [m]$ in $O(1)$ by parallel checking what is each bit of the $f(x)$. □

Corollary 42 (AMPC^0 is closed under $\leq_{\mathsf{L}}$)**.** *Let $A, B \subseteq \Sigma^*$. If $A \leq_{\mathsf{L}} B$ and $B \in \mathsf{AMPC}^0$ then $A \in \mathsf{AMPC}^0$*

Proof. Assume the hypothesis. Since $A \leq_{\mathsf{L}} B$ there is a log-space computable function f such that $\forall x.\, x \in A \iff f(x) \in B$.

Fix $\varepsilon \in (0, 1)$ we define an AMPC algorithm to decide A. On input x of length n, let $y = f(x)$ of length m. Set $\delta = \varepsilon/k$, we have an AMPC algorithm to check if $f(x)$ belongs to B using $O(m^\delta) = O(n^\varepsilon)$ local memory. This algorithm uses polynomially processors and completes execution in constant rounds. □

3.2 $\mathsf{ReachUL} \subsetneq \mathsf{AMPC}^0$

We now show the strict inclusion of $\mathsf{ReachUL}$ in AMPC^0; we first show that Reach-Unambiguous $\in \mathsf{AMPC}^0$. We can then show inclusion using the standard reduction argument, namely, via Corollary 42, AMPC^0 is closed under log-space reductions. Strict inclusion follows from Theorem 47.

Let $\langle G, a, b \rangle$ be an input instance of Reach-Unambiguous where $G = (V, E)$ is a reach-unambiguous graph such that $V = \{v_1, \ldots, v_n\}$ and $a, b \in V$. Without loss of generality, we assume that the out-degree of each vertex is at most 2. For $u \in V$ and $s \in \mathbb{N}$, define $T_s(u)$ to be the tree resulting from a *Breadth First Search* (BFS) starting from u in G upto s nodes such that no node is partially visited, i.e., either all the children of any vertex are in the tree, or none of them are. We shall call every node other than u in $T_s(u)$ a descendent of u. The main ingredient in our proof is Algorithm 1 that constructs a compressed version of $T_s(u)$. This algorithm is based on the tree contraction idea Allender and Lange (1998). We note that recently tree contraction has been studied in the context of AMPC in Hajiaghayi et al. (2022). For any graph G, we often overload the notation G to also refer to the vertex set of the graph.

Definition 43. *Let $u \in V$, and v be a descendant of u in $T_s(u)$. v is said to be an intermediate vertex for $T_s(u)$ if there exists an edge $(v, w) \in E$ such that $w \notin T_s(u)$. Define $I_s(u) \subseteq T_s(u)$ as the set of vertices that are intermediate for $T_s(u)$. We say that $T_s(u)$ is* complete *if $I_s(u) = \emptyset$, otherwise $T_s(u)$ is incomplete.*

Intermediate vertices capture the idea of vertices that can still be explored. If v is an intermediate vertex for $T_s(u)$, that means v can still be further explored. But due to the BFS parameter s, it could not explore v any further. We shall assume for simplicity of the analysis that if a tree $T_s(u)$ is incomplete, the tree has exactly $s + 1$ vertices (in general, such a tree could have either s or $s + 1$ vertices). Thus the condition $|T_s(u)| < s + 1$ denotes the condition that $T_s(u)$ is complete.

Algorithm 1: Construct Algorithm

```
1  Function Construct(u, s):
2      Compute T_s(u) using at most O(s) queries.
3      if b ∈ T_s(u) then
           // b can be reached from u within s queries
4          T'_s(u) ← ({b}, ∅)
5      else if |T_s(u)| < s + 1 then
           // b cannot be reached from u
6          T'_s(u) ← ({u}, ∅)
7      else
           // b cannot be reached from u within s queries, need to explore
8          Compute I_s(u) using at most O(s) queries
9          T'_s(u) ← A complete binary tree whose leaves are exactly I_s(u)
10     Write T'_s(u) to the DDS
```

Let $T'_s(u)$ be the output of $Construct(u, s)$ in Algorithm 1. $T'_s(u)$ is a contracted version of $T_s(u)$. If $b \in T_s(u)$ or $|T_s(u)| < s + 1$, the search from u is completed, and we can contract the tree to a single node. Otherwise, the tree is contracted to a complete binary tree whose leaves are $I_s(u)$, which are precisely the candidates that can lead to b. Locally, it is possible that $T'_s(u)$ does not contract. Claim 44 shows that globally the contraction will occur.

Define the tree T' generated by starting with $T'_s(a)$, and recursively substituting every leaf $l \in T'$ with $T'_s(l)$. Continuing the process until substituting leaves does not change the tree. This graph has the property that

$\langle T', a, b\rangle \in$ Reach-Unambiguous $\iff \langle G, a, b\rangle \in$ Reach-Unambiguous since the only vertices that remain are those vertices that have the potential to reach b. For an AMPC model, T' need not be explicitly constructed since each tree is locally computed and is then updated in the DDS. We shall now show that the graph size reduces by a factor of $s/2$, which is sufficient to get the algorithm to halt in constant rounds by setting $s = O(n^{\varepsilon})$.

Claim 44. $|T'| \leq 2n/s$

Given $u \in V$, for analysis' sake construct $H_s(u)$ by making every descendant in $T_s(u)$ a child of u, i.e. $H_s(u)$ is a re-arranged version of $T_s(u)$ such that edges go from u to the descendants of u in $T_s(u)$.

We now construct an s-ary tree, H, such that it is always full (vertices either have out degree 0 or s). Start with $H_s(a)$, then for every leaf l whose parent is p such that $l \in I_s(p)$ substitute l with $H_s(l)$ if $T_s(p) = s + 1$. If the BFS search was incomplete, substitute it with b if $b \in H_s(l)$, otherwise, do nothing. Repeat the process until no more substitutions can be done. This construction leads to a full s-ary tree H, such that $|H| \leq n$. H represents a BFS traversal done in "batches" of size s, where only intermediate nodes are substituted with another s-ary tree. Exploring non-intermediate nodes would be redundant. Let i denote the number of internal nodes of this s-ary tree.

Proof of Claim 44. Since H is a full s-ary tree, $i = |H|/s \leq n/s$. And H is essentially a rearrangement of T' such that internal vertices in H correspond to intermediate vertices in T'. However, due to line 9 of Algorithm 1, we may be adding more vertices, but however, it is no more than twice. i.e. we have $|T'| \leq 2i$. Therefore we have $|T'| \leq 2n/s$ □

Theorem 45. Reach-Unambiguous $\in$ AMPC^0

Proof. Let $\langle G, a, b\rangle$ be a problem instance of Reach-Unambiguous with $G = (V, E)$ such that $V = \{v_1, \ldots, v_n\}$. Fix $\varepsilon \in (0, 1)$. Define the AMPC algorithm with $s = O(n^{\varepsilon})$ local memory. Assign $G_1 \leftarrow G$ and $R \leftarrow O(1/\varepsilon)$, for $i = 1, \ldots, R$ rounds do the following, for each $v \in G_i$, a machine executes $\mathit{Construct}(v, s)$ for G_i to get a new graph T' as described earlier. Assign $G_{i+1} \leftarrow T'$. After the rounds are complete, $|G_R| = \frac{n}{(s/2)^R} = O(1)$. Then a single machine can perform normal reachability on G_R, accepting the input if and only if there is a path from a to b. □

Theorem 46. $\mathsf{ReachUL} \subsetneq \mathsf{AMPC}^0$.

Proof. The containment follows from the fact that Reach-Unambiguous is complete for $\mathsf{ReachUL}$ under log-space reductions and by Corollary 42, AMPC^0 is closed under log-space reductions. The strict containment follows from the fact that $\mathsf{ReachUL} \subseteq \mathsf{NL} \subseteq \mathsf{DSPACE}(\log^2 n)$. Hence by Theorem 47, there is a language in AMPC^0 that is not in $\mathsf{ReachUL}$. □

3.3 Poly-Logspace vs AMPC^0

Theorem 47. $\forall c \in \mathbb{N}.\, \mathsf{AMPC}^0 \not\subseteq \mathsf{DSPACE}(\log^c n)$

Proof. Let $A \in \mathsf{DSPACE}(n^2)$ but $A \notin \mathsf{DSPACE}(n)$. We know such a problem exists due to the space hierarchy theorem. Fix $c \in \mathbb{N}$, consider a padded language B defined as

$$B = \{\langle x, y\rangle \mid x \in A, |x| = m, |y| = 2^{m^{1/c}} - m\}$$

Claim 48. $B \notin \mathsf{DSPACE}(\log^c n)$

Proof. Assume by contradiction, $B \in \mathsf{DSPACE}(\log^c n)$. We show that in that case, $A \in \mathsf{DSPACE}(n)$. Let x be an input instance of A of length m; we shall solve it by reducing it to an instance of B, by generating $z = \langle x, y\rangle$, where $y = 0^{2^{m^{1/c}} - m}$, we have $|z| = n = 2^{m^{1/c}}$. The instance z need not be explicitly stored; every bit can be computed on the fly. Thus the space used is $O(m)$. Then use algorithm for B to solve z using space $O(\log^c n) = O(\log^c 2^{m^{1/c}}) = O(m)$. Thus $A \in \mathsf{DSPACE}(n)$ a contradiction. Therefore, $B \notin \mathsf{DSPACE}(\log^c n)$. □

Now we show that B is in AMPC^0. Let $z = \langle x, y \rangle$ be a problem instance of B of length n. The crucial point to note is that the membership of z in B depends only on x, which has length $O(\log^c n)$. If x does not have this length, we can safely reject it. Fix an arbitrary $\varepsilon \in (0, 1)$. Consider the AMPC algorithm with just one machine and one round. Let $z = \langle x, y \rangle$ be an input of length n. The machine $\mathcal{M}$ reads x which has length $m = O(\log^c n) \subseteq O(n^\varepsilon)$, then checks if $x \in A$ using space $O(m^2) = O(\log^{2c} n) \subseteq O(n^\varepsilon)$, via our assumption that $A \in \mathsf{DSPACE}(n^2)$. If $x \in A$ then $\mathcal{M}$ accepts otherwise rejects. Thus we have exhibited a language B such that $B \in \mathsf{AMPC}^0$ but $B \notin \mathsf{DSPACE}(\log^c n)$. □

We can extend Theorem 47 to $\mathsf{AMPC}^0 \not\subseteq \mathsf{DSPACE}(n^{o(1)})$ with a similar padding argument.

Theorem 49. *For efficiently computable $f \in n^{o(1)}$ such that f^{-1} is also efficiently computable*

$$\mathsf{AMPC}^0 \not\subseteq \mathsf{DSPACE}(f)$$

The proof is similar for Theorem 47, for $A \in \mathsf{DSPACE}(n^2)$ and $A \notin \mathsf{DSPACE}(n)$ define $B = \{\langle x, y \rangle \mid x \in A, |x| + |y| = n, |x| = f(n)\}$, and similar arguments hold.

3.4 AMPC^0 in Sublinear Space

We now prove that AMPC algorithms can be simulated with sublinear space. We prove that AMPC algorithms that use $O(1)$ rounds can be solved in sublinear space. However, this result can be applied even when the algorithm uses sublinear rounds.

Definition 50. *Denote* $\mathsf{SubLINSPACE} = \cap_{\varepsilon \in (0,1)} \mathsf{DSPACE}(n^\varepsilon)$

Theorem 51. $\mathsf{AMPC}^0 \subseteq \mathsf{SubLINSPACE}$

Proof. Let $L \in \mathsf{AMPC}^0$. Fix $\varepsilon \in (0, 1)$, we shall demonstrate an $O(n^\varepsilon)$ space algorithm for L. Consider the $O(n^\varepsilon)$ algorithm that decides L in $r = O(1)$ rounds. Without loss of generality, assume that there is only one machine in the final round r, call it M_r, which is responsible for writing the output bit in the DDS. We shall simulate the algorithm in a bottom-up manner.

We start our simulation with M_r; this machine wants to read $s = O(n^\varepsilon)$ values (each associated with a key) from the DDS to decide whether to accept or reject the string. Let those keys be $k_1, k_2, \ldots k_s$. We shall focus on computing the value associated with k_1. An identical technique can be used to compute the other values. Some machine in the previous round is responsible for writing the key-value pair $\langle k_1, v \rangle$. Observe that if we knew exactly which machines in the previous round (and the rounds before) wrote which key-value pairs, an $O(n^\varepsilon)$ space simulation is clear. To get an associated value associated with some key, simulate in a depth-first manner till the first round is reached. At the end of the simulation, the machine writes $\langle k_1, v \rangle$

To avoid the problem of needing the information about which machine writes which key, we can simply exhaust all possible ways the writing process could occur in a depth-first manner. If a particular computation did not write the key-value pair that we wanted, we can discard it and try another computation. Since there are $r = O(1)$ many rounds, the memory required is $O(n^\varepsilon)$. □

Note that if we allowed for $n^{o(1)}$ rounds, the above simulation would still result in a sublinear space algorithm.

Remark. Improving Theorem 51 seems unlikely. Rephrasing its statement we have $\mathsf{AMPC}^0 \subseteq \mathsf{DSPACE}(n^{O(1)})$. However Theorem 49 states $\mathsf{AMPC}^0 \not\subseteq \mathsf{DSPACE}(n^{o(1)})$. So, it suggests that we cannot hope to improve the containment.

CHAPTER 4. DISCUSSION

We now discuss some properties and share some observations of the AMPC model (uniform and non-uniform). We first define another AMPC complexity class allowing for polynomially many rounds.

Definition 52. *Define the complexity class* $\mathsf{AMPC}^{\mathsf{poly}}$ *as those languages* L *such that for every* $\epsilon \in (0,1)$ *there exists a* $\mathsf{AMPC}[\text{poly}(n), n^{\varepsilon}, \text{poly}(n)]$ *algorithm that decides* L*.*

Observation 53. $\mathsf{AMPC}^{\mathsf{poly}} \subseteq \mathsf{PSPACE}$

Proof. Observe that we are restricted to $\text{poly}(n)$ processors, each having $O(n^{\varepsilon})$ local memory for some $\varepsilon \in (0,1)$. □

4.1 Uniform AMPC

We first show that AMPC algorithms using polynomially many rounds can be simulated in sub-exponential time, and then discuss some of its implications.

Definition 54. *Denote* $\mathsf{SubEXP} = \cap_{\varepsilon \in (0,1)} \mathsf{DTIME}(2^{n^{\varepsilon}})$

Definition 55. *Denote* $\mathsf{EXP} = \cup_{k \in \mathbb{N}} \mathsf{DTIME}(2^{n^k})$ *and* $\mathsf{EXP}^{\Sigma_2^{\mathsf{P}}}$ *as the languages decided by an exponential-time (*2^{n^k} *for some* $k \in \mathbb{N}$*) Turing Machine with access to* Σ_2^{P} *oracle. Where* Σ_2^{P} *is the existential variant of the second level of the polynomial-time hierarchy.*

Lemma 56. $\mathsf{AMPC}^{\mathsf{poly}} \subseteq \mathsf{SubEXP}$

Proof. Fix $\varepsilon \in (0,1)$. Consider an AMPC model with $s = O(n^{\varepsilon})$ local memory and $p = \text{poly}(n)$ processors. Simulating for $r = \text{poly}(n)$ rounds takes $O(2^s \cdot p \cdot r) = O(2^{n^{\varepsilon}} \cdot \text{poly}(n)) \subseteq O(2^{n^{\delta}})$ steps for any $\delta \in (\varepsilon, 1)$. □

As a consequence of this simulation, it is unlikely that NP or PSPACE would be contained in $\mathsf{AMPC}^{\mathsf{poly}}$, as it would imply a sub-exponential time solution for NP or PSPACE.

4.2 Non-Uniform AMPC

We now discuss some properties of the non-uniform AMPC model.

Theorem 57 (Miltersen et al. (1999))**.** *There exists a language in* $\mathsf{EXP}^{\Sigma_2^{\mathsf{P}}}$ *such that every circuit family that decides it has size* $\Omega(2^n/n)$.

Lemma 58. (non-uniform) $\mathsf{AMPC}^{\mathsf{poly}} \subseteq \cap_{\varepsilon \in (0,1)} \mathsf{SIZE}(\mathrm{poly}(2^{n^\varepsilon}))$

Proof. Consider an AMPC model with $O(n^\varepsilon)$ local memory for some $\varepsilon \in (0,1)$ and suppose $\mathrm{poly}(n)$ machines. Since there is a space-bound of $O(n^\varepsilon)$, that implies a time bound of $O(2^{n^\varepsilon})$. Therefore, each machine can be substituted with a circuit of size $\mathrm{poly}(2^{n^\varepsilon})$. The size of the entire circuit across all $\mathrm{poly}(n)$ rounds is going to be $\mathrm{poly}(2^{n^\varepsilon})$. □

Lemma 59. $\mathsf{EXP}^{\Sigma_2^{\mathsf{P}}} \not\subseteq$ (non-uniform) $\mathsf{AMPC}^{\mathsf{poly}}$

Proof. Follows directly from Theorem 57 and Lemma 58. Every non-uniform $\mathsf{AMPC}^{\mathsf{poly}}$ circuit results in a $\mathrm{poly}(2^{n^\varepsilon})$ which is not in $\Omega(2^n/n)$. □

Lemma 60. $\mathsf{PSPACE} \subseteq$ (non-uniform) $\mathsf{AMPC}^{\mathsf{poly}} \implies \mathsf{PSPACE} \subsetneq \mathsf{EXP}^{\Sigma_2^{\mathsf{P}}}$

Proof. Follows directly from Lemma 59. □

Thus, attempting to show PSPACE is contained in $\mathsf{AMPC}^{\mathsf{poly}}$ would be "difficult" as it would solve the open problem of separating PSPACE from $\mathsf{EXP}^{\Sigma_2^{\mathsf{P}}}$.

BIBLIOGRAPHY

Allender, E. and Lange, K.-J. (1998). RUSPACE($\log n$) $\subseteq$ DSPACE($\log^2 n/\log\log n$). *Theory of Computing Systems*, 31(5):539–550.

Andoni, A., Song, Z., Stein, C., Wang, Z., and Zhong, P. (2018). Parallel graph connectivity in log diameter rounds. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*.

Andoni, A., Stein, C., and Zhong, P. (2019). Log Diameter Rounds Algorithms for 2-Vertex and 2-Edge Connectivity. In *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 14:1–14:16.

Arora, S. and Barak, B. (2009). *Computational Complexity: A Modern Approach*. Cambridge University Press, USA, 1st edition.

Assadi, S., Sun, X., and Weinstein, O. (2019). Massively parallel algorithms for finding well-connected components in sparse graphs. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19.

Beame, P., Koutris, P., and Suciu, D. (2017). Communication steps for parallel query processing. *J. ACM*, 64(6).

Behnezhad, S., Dhulipala, L., Esfandiari, H., Lacki, J., and Mirrokni, V. (2019a). Near-optimal massively parallel graph connectivity. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1615–1636.

Behnezhad, S., Dhulipala, L., Esfandiari, H., Lacki, J., Mirrokni, V., and Schudy, W. (2019b). Massively parallel computation via remote memory access. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, page 59–68, New York, NY, USA. Association for Computing Machinery.

Charikar, M., Ma, W., and Tan, L.-Y. (2020). Unconditional lower bounds for adaptive massively parallel computation. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '20, page 141–151, New York, NY, USA. Association for Computing Machinery.

Cook, S. A. and McKenzie, P. (1987). Problems complete for deterministic logarithmic space. *Journal of Algorithms*, 8(3):385–394.

Frei, F. and Wada, K. (2019). Efficient Circuit Simulation in MapReduce. In Lu, P. and Zhang, G., editors, *30th International Symposium on Algorithms and Computation (ISAAC 2019)*, volume 149 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 52:1–52:21, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Garvin, B., Stolee, D., Tewari, R., and Vinodchandran, N. V. (2014). Reachfewl = reachul. *computational complexity*, 23(1):85–98.

Hajiaghayi, M., Knittel, M., Saleh, H., and Su, H.-H. (2022). Adaptive Massively Parallel Constant-Round Tree Contraction. In *13th Innovations in Theoretical Computer Science Conference (ITCS 2022)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 83:1–83:23.

Lange, K.-J. (1997). An unambiguous class possessing a complete set. In *Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '97, page 339–350, Berlin, Heidelberg. Springer-Verlag.

Miltersen, P. B., Vinodchandran, N. V., and Watanabe, O. (1999). Super-polynomial versus half-exponential circuit size in the exponential hierarchy. In Asano, T., Imai, H., Lee, D. T., Nakano, S., and Tokuyama, T., editors, *Computing and Combinatorics, 5th Annual International Conference, COCOON '99, Proceedings*, Lecture Notes in Computer Science.

Mix Barrington, D. A., Immerman, N., and Straubing, H. (1990). On uniformity within nc1. *Journal of Computer and System Sciences*, 41(3):274–306.

Nanongkai, D. and Scquizzato, M. (2022). Equivalence classes and conditional hardness in massively parallel computations. *Distributed Computing*, 35(2):165–183.

Roughgarden, T., Vassilvitskii, S., and Wang, J. R. (2018). Shuffles and circuits (on lower bounds for modern parallel computation). *J. ACM*, 65(6).

Sipser, M. (1996). *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition.