Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Scuola di Dottorato di Ricerca in Ingegneria dell'Informazione
Indirizzo: Scienza e Tecnologia dell'Informazione
XXVI Ciclo

Time Lower Bounds for Parallel Network Computations

Direttore della Scuola
Ch.mo Prof. Matteo Bertocco

Coordinatore d'indirizzo
Ch.mo Prof. Carlo Ferrari

Supervisore
Ch.mo Prof. Gianfranco Bilardi

Dottorando
Nicola Zago

Ai miei genitori
e a Francesca

Abstract

Direct Acyclic Graphs (DAGs) are a suitable way to describe computations, expressing precedence constraints among operations. Beyond the representation of the execution of an algorithm, a DAG can effectively represent the execution of a parallel network. This last kind of DAG has a regular structure, consisting in the repetition over time of the original network; these common representations suggest a possible uniform approach in the study of execution of algorithms and emulation of networks.

Both in parallel computing and computational complexity, DAGs have been extensively employed in the study of algorithmic features, as lower bounds for the execution/emulation time of algorithms/networks, the minimum quantity of memory needed for computing an algorithm or the minimum I/O complexity of an algorithm given a certain amount of fast memory cells. Developed techniques are quite different in their assumptions; one of the more fundamental differences is that some of them allow recomputation of intermediate results, while others disallow it, requiring the storage in memory of intermediate results for their further usages.

In nowadays computations the trade-off between data recomputation and data storing is important both in parallel and in local elaborations, since in the former we can increase the bandwidth and reduce the latency with whom data can be accessed (by computing the same data in several points of the network), while in the latter we can avoid to pay the latency of the access in memory to reload data, by recomputing them possibly loading fewer data or using data already present in memory.

So far it does not exist an universal technique able to foresee the strict lower bound for each execution of algorithm or emulation of network in each network

and the known results derive from several theorems. On the contrary there are a lot of cases for which it neither exists a tight result; among these there are also emulations of extensively studied networks, such as multidimensional arrays.

The first part of our thesis starts from this state-of-the-art: we propose a survey of several known lower bound techniques involving DAGs, followed by original theorems which clarify or solve open problems. In particular, in our survey we consider lower bound techniques for execution of algorithms and emulation of networks in parallel networks, showing their principles and their limits. In the discussion we show relationships among theorems, proving that no one of them is better of the others in general terms: there are counter-examples in which each theorem gives better bounds than others. We also exhibit examples where no bound among the considered techniques is tight. Moreover we generalize some theorems originally suited for network emulations, adapting them to execution of general DAGs in parallel networks, showing examples of their application. We also consider theorems for determining minimum I/O complexity, presenting similarities and differences with emulation theorems.

One of the main results of the thesis is a new general technique which provides lower bounds almost tight (except for a logarithmic factor) in a class of network emulations including multidimensional arrays. We improve previously better known results which have a polynomial gap between lower bound and actual emulation time. Our theorem considers emulations with recomputation, giving results valid in the most general context.

Finally we consider the role of recomputation in performance, trying to understand when it gives a real advantage respect to storing intermediate results in memory. In particular we introduce the problem in simple networks, showing a class of them in which recomputation can not improve I/O performance, ending in butterfly DAGs where recomputation can save a number of I/O accesses at least as big as the fast memory available during the computation. The approach used highlights the difficulty of exploit recomputation in executions of algorithms when their DAG representation exhibits an high bisection bandwidth.

Sommario

I *Direct Acyclic Graphs* (DAG, grafi orientati e aciclici) sono dei grafi che descrivono in modo semplice ed efficace le esecuzioni di algoritmi, e permettono di rappresentare graficamente le relazioni di precedenza tra le operazioni. Al di là dell'esecuzione di algoritmi, un DAG può anche rappresentare l'esecuzione di una rete parallela. Quest'ultimo tipo di DAG ha una struttura molto regolare, corrispondente alla ripetizione nel tempo della rete stessa; il fatto che l'esecuzione di algoritmi e di reti parallele abbiano questa rappresentazione comune ci suggerisce un possibile approccio unificato nel loro studio.

I DAG sono stati molto usati nello studio di caratteristiche di algoritmi, in calcolo parallelo e nello studio della complessità computazionale. Ad esempio sono stati impiegati per ottenere lower bound per il tempo di esecuzione di algoritmi e di emulazione tra reti, per la quantità minima di memoria necessaria al calcolo di un algoritmo e il numero minimo di accessi in memoria lenta durante l'esecuzione di un algoritmo con una quantità di memoria veloce predeterminata. Le tecniche sviluppate in questi studi partono da ipotesi diverse, una delle più importanti è la possibilità o meno di ricalcolare i risultati intermedi: se ciò non è possibile infatti è necessario salvarli in memoria per poterli usare in momenti successivi del calcolo.

Il trade-off tra ricalcolo e salvataggio in memoria dei dati è importante sia in ambito parallelo che nelle elaborazioni locali; infatti nel primo caso il ricalcolo può ridurre la latenza ed aumentare la banda con cui possiamo accedere ai dati in una rete di processori, calcolando gli stessi risultati in più punti della rete, mentre nel caso di elaborazioni locali il ricalcolo può evitare i problemi di latenza e banda nel recupero dei dati dalla memoria.

Ad oggi non esiste una tecnica universale in grado di fornire lower bound stretti per ogni algoritmo od emulazione di rete eseguiti in reti parallele, e i

risultati conosciuti derivano da diversi teoremi. Al contrario, ci sono molti casi in cui mancano risultati stretti, anche per reti molto studiate e relativamente semplici com gli array multidimensionali.

La tesi inizia da questo stato dell'arte: la prima parte propone una panoramica delle tecniche di lower bound per DAG note, e termina con la presentazione dei teoremi originali sviluppati con la tesi, che migliorano o risolvono alcuni dei problemi aperti noti. In particolare, nella panoramica consideriamo tecniche di lower bound per l'esecuzione di algoritmi e emulazione di reti da parte di reti parallele, mostrando le idee su cui si basano e i loro limiti. Nello svolgimento vengono messe in evidenza le relazioni tra i teoremi, mostrando che attualmente nessuno di essi dà in assoluto risultati migliori: è possibile infatti presentare controesempi in cui ciascun teorema fornisce risultati più stretti degli altri. È anche possibile mostrare esempi di coppie di reti dove il miglior bound tra le tecniche considerate non è stretto. Inoltre generalizziamo alcuni teoremi originariamente pensati per emulazioni di reti e che noi adattiamo all'esecuzione di DAG generici in reti parallele, mostrandone alcune applicazioni. Consideriamo anche teoremi per determinare la complessità minima di accessi alla memoria per il calcolo di un algoritmo, mostrandone similarità e differenze con i teoremi per le emulazioni.

Uno dei risultati più interessanti della tesi è una nuova tecnica generale che fornisce lower bound quasi stretti – eccetto per un fattore moltiplicativo logaritmico – in una classe di emulazione di reti che include gli array multidimensionali. Precedentemente il miglior risultato noto differiva di un fattore polinomiale dal miglior tempo di emulazione noto. Il nostro teorema ammette il ricalcolo durante l'emulazione, ponendosi nel contesto più generale possibile.

Infine consideriamo il ruolo del ricalcolo nelle performance, cercando di capire quando esso possa dare un reale vantaggio rispetto alla memorizzazione di risultati intermedi. Introduciamo il problema partendo da reti semplici, mostrando una classe di esse in cui il ricalcolo non migliora la complessità di accessi in memoria, terminando con i DAG a butterfly, dove il ricalcolo riesce a migliorare questa complessità di un termine almeno pari alla memoria usata durante il calcolo. L'approccio usato mette in luce la difficoltà di usare proficuamente il ricalcolo durante l'esecuzione di algoritmi che presentano un'elevata connettività.

Ringraziamenti

Dopo tanti anni e tanta fatica per completare la tesi, i ringraziamenti sono la parte più emozionante da comporre. Vorrei innanzitutto ringraziare Gianfranco per avermi seguito durante questo dottorato, trasmettendomi spunti e tecniche metodologiche che difficilmente i libri possono fornire. Assieme a lui vorrei ringraziare tutti i componenti dell'ACG, con cui ho trascorso tre anni di lavoro e di vita di cui conserverò senza dubbio un carissimo ricordo.

Oltre all'ambiente universitario, vorrei ringraziare anche tutti quelli che mi hanno supportato (e sopportato) durante il dottorato, per primi i miei genitori Umberto e Giuseppina, seguiti dal gruppo Giavera, Mauro, il gruppo Seisnet e soprattutto Francesca, mia futura moglie. Grazie a tutte queste persone gli ultimi quattro anni sono stati importanti quanto e forse più dei precedenti cinque che mi avevano portato alla laurea: ho completato la mia maturazione sia dal punto di vista professionale che come persona, partendo da studente universitario e arrivando all'inizio della mia età adulta. Arrivato a questo punto del mio percorso, con il bagaglio di esperienze maturate e accumulate negli ultimi anni, sono certo che le prossime pagine della mia vita che scriverò saranno le più importanti e le migliori.

Acknowledgements The last part of the acknowledgements is in English. In fact I would like to thank Markus for the hospitality and the interesting topics suggested during the three months that I spent at the ETH Zurich in the summer 2013. It has been an important experience in my professional and personal path, which gave me the opportunity to know the attitude of another nation. I would like to thank all the guys I knew in that period, expecially Daniele, Victoria, Georg, François, Alen, Marcela and Luca. Vielen Dank an Sie alle!

Contents

1 Introduction 1

1.1 General introduction . . . 1

1.2 State-of-the-art and purpose of the thesis . . . 7

2 Lower bounds for generic emulations 13

2.1 Background . . . 13

2.2 Distance-based lower bound . . . 16

2.2.1 State of the art . . . 16

2.2.2 Analysis and Critique . . . 19

2.2.3 Generalization of Theorem 1 . . . 22

2.3 Congestion-based lower bound . . . 27

2.3.1 State of the art . . . 27

2.3.2 Analysis and Critique . . . 30

2.4 Bandwidth-based lower bound . . . 34

2.4.1 State of the art . . . 34

2.4.2 Analysis and Critique . . . 36

2.4.3 Summary . . . 37

2.5 Parallel computing vs hierarchical memories . . . 38

2.5.1 State of the art . . . 40

2.5.2 Relations with parallel computing . . . 43

3 Lower bounds for specific networks 47

3.1 Lower bounds for multidimensional arrays emulations . . . 47

3.2 Mesh over linear array emulation . . . 52

3.3 Generalization to k-arrays over j-arrays . . . 54

3.4 Considerations . . . 57

4 Storing-recomputation trade-offs **59**

4.1 Basic facts . . . 60

4.2 Recomputation in tree DAGs . . . 61

4.3 Recomputation in butterfly DAGs . . . 62

4.3.1 Two general lower bounds . . . 62

4.3.2 Matching the lower bounds . . . 65

4.4 Recomputation in butterfly-like reduction DAGs . . . 71

5 Conclusions **75**

5.1 Summary and contributions . . . 75

5.2 Further research . . . 76

Bibliography **77**

Chapter 1

Introduction

1.1 General introduction

Computer evolution is too slow! Since the introduction of integrated circuits in the production of computer chips at the end of 1950s, the density of transistors has doubled every 18 months, according to Moore's law [M65]. The increase rates of many other hardware systems, e.g. storage devices' capability and bandwidth, follow similar laws, allowing for the availability of cheap electronic devices, which pervade almost every aspect of current life. This exponential evolution leads a constant improvement of the performance of these devices, although modern demand of data analysis grows with a rate greater than technological achievements.

Consider for example massive data analysis, coming from scientific fields as Physics, Astronomy or Biology, as well as related to new information technology applications, as social networks or search engines. Just technological increases would not be sufficient to deal with all these data, and computer scientists have to compensate with new algorithms, often characterized by randomization, since exact computations on such large inputs would take too much time and resources.

Algorithmic contribution may be crucial also in all day applications; consider for example the Fast Fourier Transform (FFT) in signal processing or the Merge Sort algorithm in the sorting. Classical trivial algorithms would be unsuitable for modern computation requirements, unless using them on hardware much more performable, while the enhanced algorithms already are up to the situation with current hardware. The main difference between these two classes of algorithms consists in their computational complexity, which is the relation among the size of the problem and the number of operations required to solve it. Usually, enhanced algorithms has a lower asymptotic growth, so the bigger will be the problem the better these algorithms will perform respect to classical ones.

Study of computer performance To predict algorithms performance, computational complexity can be studied in theoretical computational models. These models abstract computers capabilities, maintaining the main features and hiding useless details which would complicate the analysis, allowing us to give general results, valid in a wide range of machines. Early machine models, e.g. the Turing Machine [T36], appeared in 1930s but their objective was to study what is algorithmically computable. Models whose intent was to study the complexity execution in real computer, appeared from the beginning of the 1970s.

The most prominent among these is the Random Access Machine (RAM) model [CR73]; it considers a computer as composed of a sequential processor (with fixed program) operating on a countable number of cells, which can be accessed in a time independent from their number. This model is very simple still effective in determining the operational complexity of algorithms. Unfortunately, the assumptions of infinite random access memory is unrealistic, since a bit needs a minimum volume to be stored and there is a maximum velocity at which a bit of information can travel (principles of *maximum information density* and of *maximum information speed* [BP95]), so the wider is the memory and the bigger is the *latency* to access a cell far from the processor respect to access a cell near to it.

To model this feature, which between the end of the 1970s and the beginning of the 1980s has been sharpened by the introduction of cache memories, several theoretical memory models have been proposed, in particular the Hierarchical Memory Model (HMM) [AACS87], the Block Transfer (BT) model [ACS87] and the two-level memory model [AV88]. These models are very useful to understand how the program execution is slowed by the interaction with memory and to promote the development of algorithms exhibiting *temporal* and *spatial locality*, strategies which try to hide or at least reduce the latency effects. Taking into account the data movement introduces a metric different from the number of operations executed, in particular the metric which considers the number of accesses performed in the slow memories. Since actually the growth rate of speed of processors is greater than that of memory speed, for many problems this second metric is more important than that derived from the classical RAM model, since input and output operations (I/O operations) in slow memory often determine the bottleneck for the execution time. More recently, due to the large diffusion of mobile devices and in general to the increasing interest in the reduction of power consumption, also the energy aware computing has been extensively studied, both from architectural and programming points of view (e.g., [PSZ+02, PKK+04]).

In order to evaluate the performance of a given algorithm, it is useful to know which is the minimum number of operations or memory accesses required for the solution of the problem. This aspect of a computation is known as *computational lower bound*. The determination of a lower bound is usually an hard task and usually machine and memory models can not handle it, so we need to consider frameworks which explicitly target the study of lower bounds.

There are at least two kind of lower bounds: those involving a problem and those involving an algorithm, which is a particular way of solving a problem. The formers are very difficult to obtain, since we have to prove the bound for every possible strategy to solve the problem, and so far very few general and non-trivial results are known. Literature about lower bounds for algorithms is wider, also if there is not yet a general way to determine them. A typical example for this category is the lower bound of $\Omega(n \log n)$ operations in sorting n distinct number [CLR01]. It is quite general, since it involves the class of comparison-based algorithms, but it is no more valid for example if we use numbers as indexes (as in the Counting Sort [CLR01]); moreover one could argue than having n distinct elements means having at least $n \log n$ bits, and so the lower bound just matches the trivial one based on reading the whole input. Another example is algorithms for matrix multiplications using only scalar multiplication and addition, where a $\Omega(n^3)$ operations lower bound is known [K70], while in general case only the trivial $\Omega(n^2)$ lower bound is known.

As already noted above, nowadays it is fundamental to exploit the fast cache memories, so we could be interested in lower bounds on the minimum number of accesses in the slow memory (I/O complexity) needed in order to solve a problem. One of the seminal works in this field is [HK81], followed by [S95, EPR+13], which provide extensions of the original game respectively in hierarchical memories and in a context without recomputation. Exactly this aspect of computation is currently one of the more elusive: it is clear that there is a trade-off between recomputation and I/O accesses, still it is not yet fully understood. During a computation, if an intermediate result is used more than once, we can decide if temporarily store it or recompute it. According to the particular algorithm could be more convenient the former or the latter strategy; the determination of the optimal strategy usually is a very difficult problem, also for algorithms as regular as the FFT. This topic will be examined in depth in Chapter 4. A similar problem is present also in parallel computing, as will be discussed in next paragraph.

Birth and evolution of Parallel Computing So far we considered computation only from the sequential point of view, but since the 70s also *parallel computing* has been extensively studied. This term can refer to several forms of parallelism: *bit-level parallelism*, which consists in augmenting the number of bit elaborated per instruction increasing the word-size of the processor, *instruction-level parallelism*, which consists in conveniently designing the processor to pipeline the execute of instructions, obtaining an higher throughput and *task-level parallelism*, which consists in exploiting more processors during the execution of a program. The first two kinds of parallelism have been part of the evolution of sequential cores, so they have been transparent to developers. If we take an old sequential program and make it run on a computer with higher bit-level or instruction-level parallelism it almost automatically will improve its performance. Since the limit of VLSI technology is going to be reached, over the last ten years evolution of sequential processors is getting harder and harder and it is more advantageous to explicitly design processors with more computational cores respect to further improve the power of a single core. This kind of evolution is no more transparent to developers: we have to produce programs which explicitly use more cores since an old sequential program will just use one of them.

This third kind of parallelism is whom we usually refer to when we talk about parallel computing: the exploitation of several processing units at once, in order to reduce the time needed for the solution of a certain problem respect to sequential computing. This task is not so trivial, in fact the various processors need for communication and synchronization during the execution. Note that this problem heavily relies on the network joining the processors (every processor could be able to communicate with every other processor, or only with a part of them). The usage of parallel machines has been largely relegated to the research environment until few years ago, since, as we said above, before improving sequential cores was more advantageous. This leads to the fact that nowadays parallel computing has less established technological and programming standards.

Due to the higher number of freedom degrees of this field, the study of parallel machines did not follow exactly in sequential computing footsteps. Initially performance were not only studied in ideal parallel machine models, but also in machines with specific network configurations. Among the formers, the main model is the Parallel Random Access Machine (PRAM) [FW78, G78], consisting in a collection of processors, sharing a certain amount of memory cells. This framework allows to understand intrinsic difficulties of problems, but it ignores the communication complexity in which algorithms incur once they are implemented in a real machine, which also in this field is nowadays the real bottleneck for computations. Parallel algorithms are usually described by very high level languages (as the Work-Time Framework [Jaja92]), which aim to express the maximum parallelism and the minimum number of parallel steps reachable in the PRAM, and leave the communication problem to the particular implementations. This description for algorithms is very useful since we can easily see how their main features scale in different machines thanks to Brent's theorem [B74]. Conversely to the PRAM case, analysis performed on specific networks of processors are not enough general to highlight features of problems without being influenced by the considered networks. This motivates in the nineties the study of several high level, still realistic, models like BSP [V90], LPRAM [ACS90], LogP [CKP+96], D-BSP [DK96]. In these models we can study the communication complexity of problems without being misled by a particular network.

In parallel computing, maybe more than in sequential field, we are very interested in performance: in fact we are explicitly trying to exploit a new intrinsic feature of problems: the parallelism. The Amdahl's law [A67] limits our action field only to the fraction of the algorithm which can be parallelized, so if great part of the problem is strictly sequential, the improvement will be small. Once determined the leeway, we would like to know which is the minimum time required for the execution of a problem or of an algorithm with the available hardware. There is a natural bound on parallel time obtained from the ratio between the best sequential execution time and the number of processors available. When the number of processors is high respect to the problem size, this bound gets weaker and weaker, in fact it does not catch the complexity of the precedence constraints among the operations executed for the solution of the problem.

If we suppose to use a machine with infinite processors, executing each operation as soon as all its predecessors have been computed, we can execute a specific algorithm in the minimum parallel time possible for it: this quantity is the *depth* of the algorithm. The best sequential algorithm not necessarily provides also the lower parallel depth, so than often sequential algorithms must be rearranged or totally redesigned to exhibit a lower depth. In parallel computing we are interested in algorithms with the lowest depth possible, using a reasonable number of processor. Another optional yet attractive quality for these algorithms is to be *work optimal*, that is to have a number of operations similar to the best sequential algorithm. The determination of the lowest depth for a problem is a difficult task, also if for some problems it is feasible reasoning on how much data can be analyzed with the available processors. Important results of this kind are the $\Omega(\log n)$ time required for the computation of the OR function [CD82] and exact lower bounds for searching, merging and sorting [K83]. If we consider a specific network of processors, the task gets slightly easier, since we can point out lower bounds due to the data movement and possible communication bottlenecks. However this kind of lower bound is not general, since it is due to physical features of the networks, and it does not provide general information about the complexity of the problem.

As anticipated in previous paragraph, also in this field we should take into account the possibility of recompute intermediate results in order to avoid communication among distant processors and possibly obtain better algorithms. Given the generality of this approach, it is very difficult to obtain general lower bounds which can tightly catch the computational complexity when this strategy is allowed. This aspect will be extensively discussed in the rest of the thesis.

1.2 State-of-the-art and purpose of the thesis

This thesis makes a critical survey of the known lower bound techniques which target parallel computation of algorithms and completes some aspects which are still open.

Literature in this field can be divided according the considered network model, which can be an ideal PRAM, an high level but realistic model or a specific network, and according the consideration of recomputation, which can be allowed or not.

As for ideal PRAM, we can find the already mentioned [CD82], which proves a $\Omega(\log N)$ time lower bound for the computation of the OR function in a CREW PRAM (while a trivial constant time algorithm exists for CRCW PRAM). [PU87] proposes a trade-off between time and communication required for the computation of the diamond DAG, showing that the product of optimal lower bounds for time and communication considered singularly is strictly smaller than lower bound obtained considering jointly time and communication; this means that in some DAGs the PRAM can not match both the minimum time and the minimum communication with the same strategy. Finally [K83] present several tight lower bound for searching, sorting and merging in the PRAM.

In [VW85] the PRAM(m) model is introduced, which consists in a PRAM with only m cells of global memory, modeling a bandwidth limit of m data per time unit. It has been studied both with *exclusive read* and *concurrent read* memory access policies, leading to lower and upper bounds where the execution time take into account the bandwidth (e.g. [ABK95] for sorting, which also shows how the technique can be adapted to [CKP+96, V90], cited in the following lines).

Starting from [PU87], [ACS90] considers the LPRAM, a variation of PRAM which captures also communication costs, providing time and communication lower bounds for several problems as matrix–matrix multiplication, sorting, FFT. Beyond LPRAM, where we need to exhibit two distinct lower bounds for communication and time, among general realistic models there are the P-Log [CKP+96] and the *Bulk Synchronous Parallel* (BSP) [V90], theoretical machines which can describe a wide range of real networks thanks to their parametrized structure. Time lower bounds for these model contain in their formulas parameters for latency and bandwidth of the network which connects the processors, so that the time lower bound already expresses also the communication complexity. For relationships between the two models see [BHP+96]. BSP has been extensively studied and strict results are known in several fundamental problem [BSS12, SS14], It has been extended to take into account the locality in parallel computation in [DK96], where the Decomposable BSP is introduced. In this model bandwidth and latency are variable, and computations take less time for algorithms exhibiting communication among near processors. It is possible to prove which D-BSP can model processor networks more effectively than BSP, allowing emulations with only a constant factor slowdown for several networks, e.g. multidimensional arrays, see [BPP07] for a complete dissertation.

Note that there are several works, e.g. [CGG+95, LP93, FPP06, PPS06, MZ12], proposing cross-emulations or highlighting similarity among parallel and memory models, so that computational bounds in memory models are strictly related to the parallel ones. See section 2.5 for a deepened introduction to the argument.

Turning our attention on bounds for processor networks, we recall works on Universal Computer; an universal computer is a parallel machine with fixed communication network of bounded degree which aims to emulate as efficiently as possible any other bounded degree network. A N-nodes universal computer can emulate all N-nodes bounded-degree network with slowdown at least $\Omega(\log N/\log\log N)$ [M83], while if the machine has $N^{1+\epsilon}$ nodes it can emulate any N-nodes bounded-degree network with slowdown $O(1)$ [M86].

Other works consider different hypothesis, e.g. in [BP99] networks of processors are integrated with a local hierarchical memory and in general it target emulations respecting realistic physical constraints. In this paper no recomputation is allowed.

A powerful emulation technique is the definition of an embedding of the guest network into the host network; in Section 2.1 basic results about embedding are recalled. Embeddings allow optimal emulation of trees in multidimensional arrays [HKMU91], and are involved as subpart of other optimal emulation techniques. For example a plain embedding can not provide a constant slowdown emulation between the mesh network and the butterfly network, since any embedding of the former in the latter as at least dilation $\Omega(\log N)$, but embedding of sub-meshes in sub-butterflies are exploited in the $O(1)$-slowdown emulation proposed in [KLM+97].

In this last paper work-preserving emulations are considered; an emulation of a guest by an host is work-preserving if work performed by the host is similar to work of the guest, $W_H = O(W_G)$. In particular authors prove two theorems to determine lower bounds of the time required from a network to emulate the computation of another network when recomputation is allowed. These theorems are used to investigate the maximum size of an host to obtain a work-preserving emulation of a guest. In the same paper there is a survey of known emulations among networks, used to prove the optimality or looseness of lower bounds provided by the previous theorems. On the same line [KR94] proposes a lower bound technique for network emulations considering bandwidth of networks, imposing at most a constant level of recomputation. Also in this case the theorem is used to investigate maximum size of hosts to perform a work-preserving emulation of a guest and results reached are the same of [KLM+97] in several cases, but with a more intuitive approach.

Note that especially when recomputation is allowed, lower bounds not always are strict. For this reason can be useful to recall the already mentioned work of Hong and Kung [HK81], which deals with minimum number of accesses in slow memory when executing an algorithm with finite amount of fast memory and recomputation enabled. Note that saving I/O accesses replacing it with recomputation is related to save communication replacing it with recomputation in parallel environment. Also this field lacks fundamental proofs of the role of recomputation in performance.

Finding significant lower bounds for problems could also solve important questions in Computational Complexity field. For example if we could prove a polynomial lower bound for a problem in $\mathcal{P}$ (problems for which an algorithm polynomial in the size of the input is known), we could demonstrate that $\mathcal{NC} \subsetneq \mathcal{P}$, where $\mathcal{NC}$ is the Nick Class, the class of problems solvable in polylogarithmic time when a number of processors polynomial in the size of the input is available. This would means that not all the problems which are efficiently solvable in sequential way are also highly-parallelizable. Even more important, if we could exhibit a super-polynomial lower bound for some problem in $\mathcal{NP}$, we would know that $\mathcal{P} \subsetneq \mathcal{NP}$, where $\mathcal{NP}$ is the class of problems solvable in polynomial time with a non-deterministic Turing Machine. This would means that not every problem is tractable with actual computers, and that exist problems which are not computable in useful time also for small instances.

The rest of this thesis is divided in two parts: the first one includes Chapters 2 and 3 and it is about time lower bounds for emulations, while the second one includes Chapter 4 and is about trade-offs between recomputation and storing of intermediate results. In Chapter 2 we will consider lower bound techniques – allowing recomputation – valid for a wide gamma of networks from [KLM+97, KR94]. These techniques model an arbitrary pair of networks through some parameters which lead to a general formulation of lower bounds. Given a specific pair of networks we can determine the current value of the lower bound by just computing the parameters for the specific case. In the treatise, we show relationships among the techniques, highlighting their weaknesses and extending them to general DAGs. An important fact proved in the thesis is that among the considered techniques there is not one strictly more powerful than the others, in fact we can exhibit cases where each technique gives better bounds than the others. Moreover we highlight differences between theorems for lower bounds for I/O complexity and those for parallel computations.

Chapter 3 contains new results for lower bounds of specific emulations, in particular we consider emulations among multidimensional arrays: actually, the better known result has a polynomial gap with the better known emulation when arbitrary recomputation is allowed, while our theorem only suffers from a logarithmic gap. The proposed strategy can be extended to arbitrary networks but it is effective only for determinate classes of them.

Chapter 4 addresses the role of recomputation in the execution of algorithms from the point of view of I/O complexity. When recomputation is not allowed, partial results which have to be used again, must be stored in memory, while with recomputation in some case it is possible to save memory accesses by recomputing those results. The main result of this section is the proof that recomputation gives only a small advantage in the butterfly but asymptotically it has the same complexity and the same constants of the case without recomputation.

Finally, Chapter 5 concludes the thesis summarizing the contribution and proposing possible further researches.

I would like to remember that great part of results contained in Chapter 4 have been obtained in the three months that I spent at ETH Zurich, with Markus Püschel as advisor.

Chapter 2

Lower bounds for generic emulations

Given two processor networks, which is the fastest possible emulation of the first network performed by the second one? Starting from this question, in this chapter we will present a survey of known techniques to obtain time lower bounds for network emulations, showing the main ideas beneath them, the reciprocal relations and extending them to more general cases. We will show that actually there is not a single technique all-comprehensive, but we must consider several techniques to obtain tight lower bounds in networks with different features. Moreover we analyze weaknesses of the discussed theorems, giving a starting point for a possible all-comprehensive technique.

Some of the points left open in this chapter will be solved in Chapter 3, as for example a lower bound for emulations among multidimensional arrays.

2.1 Background

Before proceeding with the exposition, we recall some concepts extensively used in the rest of the work.

Definition 1. A *Parallel Random Access Machine* (PRAM) [FW78, G78] is an abstract parallel machine model, that consisting in a collection of P synchronous processors and M shared memory locations.

Beyond ideal machines, where every processor can communicate with each other in one step by the shared memory (*shared memory model*), we will consider machines with a given interconnection among their processors and where communication occurs by sending and receiving messages through edges of the interconnection network (similarly to the *message passing model*). In this second case, each processor has M local memory locations, accessible in one step only by processors which are joint to it by the network. A network is modeled by a undirected graph $G = (V, E)$, where vertices in V (also called nodes) represent processors and edges in E represent connections among processors. We refer to [KLM+97] for definitions of the topology of butterfly, k-dimensional array and tree networks.

During the thesis we will consider emulations of a network G by another network H, which consist in the reproduction of all intermediate results of the computation of G with H. One possible way to specify an emulation is by means of an embedding.

Definition 2. Consider a guest network $G = (V, E)$ and an host network $H = (W, F)$. An *embedding* of G in H consists in two functions, $\phi : V \to W$, which maps every node of V in a node of W, and $\psi : E \to \{\text{paths in } F\}$, which maps edges of G in paths in H. An embedding is characterized by its *dilation* d, which is the maximum length of a path $\psi(e)$, its *congestion* c, which is the maximum number of paths $\psi(e)$ passing on an edge of H and its *load* l, which is the the maximum number of nodes of G mapped to a node of H.

If there is an embedding of G in H with parameters c, d and l, then it exists an emulation of G by H with slowdown $O(c + d + l)$ [LMR88].

We will represent a computation as a *directed acyclic graph* (DAG) where, informally, nodes represent operations and edges represent the dependences among operations. Following definition is a variation of that in [BP01].

Definition 3. A DAG or *computation directed acyclic graph* (CDAG) is 4-tuple $C = (I, V, E, O)$ of finite sets such that:

1. $I \cap V = \emptyset$;

2. $E \subseteq (I + V) \times V$ is the set of arcs;

3. $G = (I + V, E)$ is a directed acyclic graph with *no* isolated vertices;

4. I is called the *input set*;

5. V is called the *operation set* and all its vertices have incoming arcs;

6. $O \subseteq I + V$ is called the *output set*.

Every node that can reach $u \in V$ with a path is a *predecessor* of u, in particular if a predecessor of u is directly connected to u than we call it *parent* of u. Similarly, every node reached by a path from u is a *successor* of u, in particular a successor of u directly connected to u is a *son* of u. Usually in this work we will consider nodes without predecessors as *inputs*, while nodes without successors as *outputs*.

Note that a DAG is given by a particular execution of an algorithm. Different executions of the same algorithm can give different DAGs. DAGs are very useful in the analysis of an algorithm, e.g., they can be used to study the space complexity [HWV77] or the I/O complexity [HK81], in both cases by pebble games. Next definition recalls the general idea behind pebble games, but games in [HWV77, HK81] are quite different among them.

Definition 4. A *pebble game* on a DAG G is the following game. The player has a certain number of pebbles which he can use to mark (to pebble) nodes of G, one in any step of the game. As long as there are available pebbles, an input can always be pebbled. Other nodes can be pebbled only if all their predecessors are currently pebbled and there are pebbles available. If there are no more available pebbles, we can unpebble some nodes (one per step of the game), freeing the pebbles used in them. The game ends when every output has been marked at least once.

Pebbles in the games represent memory locations used during the computation. Note that pebble game in [HWV77] aims to determine the minimum amount of memory required for execute an algorithm, while the game in [HK81], performed with pebbles of two colors, has the purpose of determine the minimum number of accesses in slow memory required by an algorithm. We can also think pebbles as parallel processors occupied in the execution of the DAG; in this game in the game we can perform as many moves per step as many pebbles are available.

A certain sequence for the creation of nodes of the DAG G is called *schedule*, in particular the *greedy schedule* assumes infinite pebbles and schedules nodes as soon as they are ready. The greedy schedule of G in the parallel case defines a partition $p_0, p_1, \dots, p_{D-1}$ of its nodes, where set p_0 contains inputs and set p_i contains nodes created at step i. This schedule provides useful features of G, in particular its *depth* D – the minimum number of steps required by a PRAM to compute the DAG – and its maximum parallelism $\max_i p_i$. We also define the *work* W of G as $W = |V_G| = \sum_{i=0}^{D-1} p_i$ and the *average parallelism* $p = W/D$. Note that if we limit the available processors to p, we can schedule G in $\sum_{i=0}^{D-1} \lceil p_i/p \rceil \leq \sum_{i=0}^{D-1} p_i/p + \sum_{i=0}^{D-1} 1 = 2D$.

2.2 Distance-based lower bound

2.2.1 State of the art

Notation

First, we will analyze two techniques from [KLM+97], which consider the emulation of an N_G-nodes guest network G, executing for T_G time steps, by an N_H-nodes host network H, which requires T_H time steps to perform it. Note that both the techniques are very general, and they are valid also if recomputation is allowed. The aim of the paper is to investigate in which case a network H can perform an efficient emulation of G, where the efficiency is the ratio among works performed by G and H. In particular, authors give two general methods to determine lower bounds of the *inefficiency*, defined as $I = \frac{T_H N_H}{T_G N_G}$. In an efficient emulation $\frac{T_H N_H}{T_G N_G} = O(1)$, and it is called *work preserving*. In this thesis we are interested in a more general target: the tightness of the bounds.

The computation of G is considered as a DAG, whom H has to pebble. In particular, nodes of G have a constant memory and in each step they can communicate their status to every neighbour. Note that the DAG considered, been produced by a network, has a very regular structure and its features are related to those of G. If G executes for T steps, the produced DAG has TN_G

nodes and can be thought as T levels, each with N_G nodes, one per node of G (Fig. 2.1). Level τ contains nodes of G at time step τ, and the only edges in the DAG are those from level i to level $i+1$, $i = 0, 1, \ldots, T-1$. We refer with (u, τ) to the node of the DAG representing the status of node u of G in the time step τ. Let $\delta(u, v)$ be the length of the shortest path between u and v in a given undirected graph G; then $B_G(u, i) = \{v \in V_G : \delta(u, v) \leq i\}$ and $b_G(u, i) = |B_G(u, i)|$. Node (u, τ) receives as input $(u, \tau-1)$ and $\{(v, \tau-1) : v \in B_G(u, 1)\}$, while it is an input for $(u, \tau+1)$ and for $\{(v, \tau+1) : v \in B_G(u, 1)\}$.

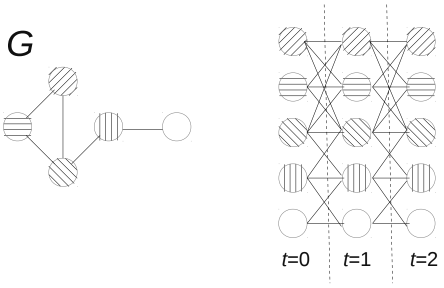

Figure 2.1: DAG produced by the execution of a network G in 3 time steps.

The Theorem

Theorem 1. *Let $G = (V_G, E_G)$ be an N_G-nodes guest network and $H = (V_H, E_H)$ be an N_H-nodes host network. Suppose that there are integers τ_G and τ_H such that*

$$\max_{u \in V_H} \sum_{i=1}^{\tau_H} b_H(u, i) < \min_{v \in V_G} \sum_{j=1}^{\tau_G} b_G(v, j). \tag{2.1}$$

Then any emulation of $T_G \geq \tau_G$ steps of G by H has slowdown

$$S > \frac{\tau_H + 1}{2\tau_G}.$$

Here we just sketch the main idea behind the theorem; see [KLM+97] for the complete proof. Given a node (v, t) of the DAG of G, it has $b_G(v, 1)$ predecessors at time step $t-1$, $b_G(v, 2)$ in time step $t-2$ and so on, until having $b_G(v, \tau_G)$ predecessors at time step $t - \tau_G$ (Fig. 2.2).

We are interested in the node v which has $P = \min_{v \in V_G} \sum_{j=1}^{\tau_G} b_G(v, j)$ prede-

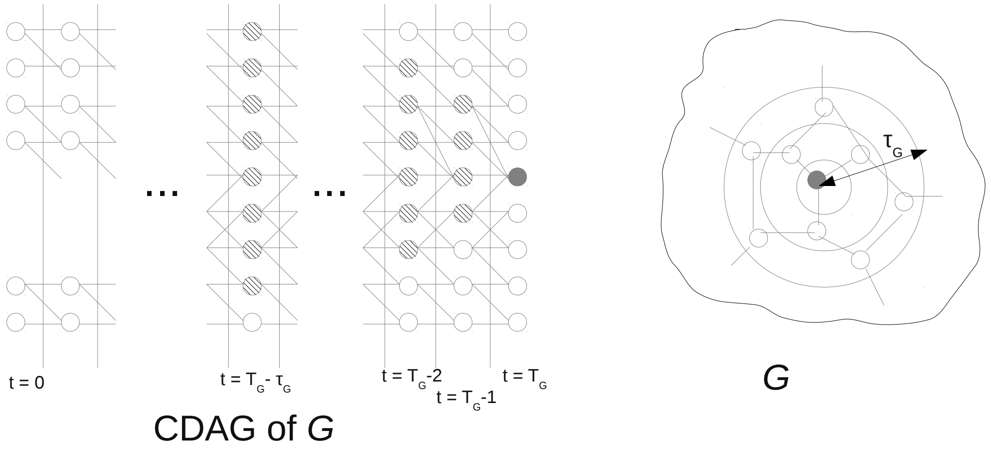

Figure 2.2: Quantity represented by $\sum_{j=1}^{\tau_G} b_G(v, j)$.

cessors, the minimum number of predecessors in the previous τ_G time steps (the "weaker" node of G). Similarly we can find out the node u in H, which in the previous τ_H steps has the maximum number of predecessors respect to nodes in H, but they are strictly less than P. This means that any node in G, in τ_G time steps, can be influenced by P operations, while any node in H needs strictly more than τ_H steps to be contacted by such a quantity of information.

Focus on first evaluation of (v, T_G) by a node u of H, at time T_H of the network H. There is at least one node v' in the region of the DAG of G influencing (v, T_G) during the time $[T_G - \tau_G, T_G]$ computed by H before $T_H - \tau_H$. We can repeat such a construction for node v', obtaining another region of G with predecessors at most in time step $T_G - 2\tau_G$ with a node v'' computed in H before time $T_H - 2\tau_H$ in H. Repeating this reasoning, we obtain $\Omega(T_G/\tau_G)$ regions of DAG of G, each requiring at least $\Omega(\tau_H)$ disjoint steps in the emulation performed by H, so that $T_H = \Omega(T_G \tau_H/\tau_G)$, or $S = \Omega(\tau_H/\tau_G)$.

In [KLM+97], Theorem 1 is used to prove the following fact.

Corollary 1. *For fixed k, any emulation of a complete binary tree G by a k-dimensional array H has slowdown at least* $\Omega((N_G/\log^k N_G)^{1/(k+1)})$.

In [KLM+97], authors stress the fact that a N-nodes k-array can perform a work-preserving emulation of a $(N^{(k+1)/k}/\log N)$-leafs complete binary tree. One should also note that the known embeddings of a tree on a k-dimensional array are optimal for each value of N_H. In fact when the k-dimensional array has diameter $\Omega((N_G \log N_G)^{1/(k+1)})$, it can match the lower bound in Corollary 1 with embeddings proposed in [HKMU91], while for smaller diameters, since an M-nodes k-dimensional array can emulate in work preserving fashion an N-nodes one ($M < N$), we can match the trivial load lower bound $\Omega(N_G/N_H)$ also provided by the theorem.

2.2.2 Analysis and Critique

Now we will highlight the aspects of the computation catched by the theorem. We will use as working example the emulation of a k-dimensional array G with a j-dimensional array H, $k > j$. If $N_H = O\left(N_G^{\frac{j}{k}}\right)$, H can perform an emulation of G with slowdown $O(N_G/N_H)$, while for bigger values of N_H, H can not perform a work preserving emulation of G and there is an inefficiency more than constant [KLM+97].

Given an N-nodes x-dimensional array A, we can prove that

$$\sum_{i=1}^{\tau} b_A(u,i) = \begin{cases} \Theta(\tau^{x+1}) & \text{if } \tau \leq \text{diam}_A \\ \Theta(N^{\frac{x+1}{x}}) + N(\tau - \text{diam}_A) & \text{if } \tau > \text{diam}_A \end{cases} . \tag{2.2}$$

If the execution time τ considered is smaller than the diameter of the network, the number of predecessors of u grows as $\Theta(\tau^{x+1})$, otherwise, if the execution time is long enough the growth is just linear respect to τ (since the whole network has already been reached after first diam_A steps).

If $\tau_G = \text{diam}_G = N_G^{\frac{1}{k}}$, right term of Equation 2.1 is $\Theta\left(N_G^{\frac{k+1}{k}}\right)$, while for the left term, depending on the size of H, we distinguish two cases, which lead to two different values for τ_H.

If $\text{diam}_H = \Omega\left(N_G^{\frac{k+1}{k(j+1)}}\right)$:

$$\Theta\left(\tau_H^{j+1}\right) = \Omega\left(N_G^{\frac{k+1}{k}}\right) \Rightarrow \tau_H = \Omega\left(N_G^{\frac{k+1}{k(j+1)}}\right),$$

while if $\mathrm{diam}_H = o\left(N_G^{\frac{k+1}{k(j+1)}}\right)$:

$$N_H\left(\tau_H - \mathrm{diam}_H\right) = \Omega\left(N_G^{\frac{k+1}{k}}\right) \Rightarrow \tau_H = \Omega\left(\frac{N_G^{\frac{k+1}{k}}}{N_H}\right).$$

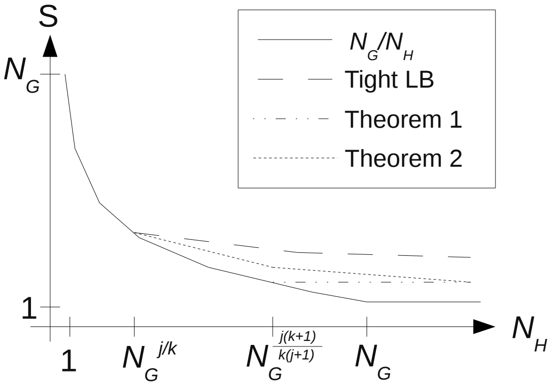

Figure 2.3: Comparison among tight lower bound, lower bound implied by load argumentation and lower bound due to Theorem 1 and 3. Tight lower bound is determined by ratio of bisection bandwidths for $N_H > N_G^{\frac{j}{k}}$ (see Theorem 4 and Chapter 3).

These values give respectively slowdown $S = \Omega\left(N_G^{\frac{k-j}{k(j+1)}}\right)$ and $S = \Omega\left(\frac{N_G}{N_H}\right)$. Note that the theorem considers a region of H with radius at most $O\left(N_G^{\frac{k+1}{k(j+1)}}\right) < N_G^{\frac{1}{j}}$. If H has more than $\Theta\left(N_G^{\frac{j(k+1)}{k(j+1)}}\right) < N_G$ nodes, these will not affect the lower bound, since they are too far to communicate with other parts of the network without slowing down the computation. Viceversa, if H is small, we just obtain the trivial slowdown due to ratio among the number of nodes of G and those of H (*average load* of H). Note that we are considering the class of j-dimensional array networks, which has a particular structure, in which the network A with diameter diam_A is a sub-network of the network B with diameter $\mathrm{diam}_B = \mathrm{diam}_A + 1$. When H belong to a class with such property, the technique emphatizes the diameter of the instance of network in which lower bound due to average distance of nodes can be improved by an argumentation based on average load of a region.

Moreover, lower bound obtained by the theorem in this case is not tight (see Figure 2.3 and proof of Proposition 4 in next subsection). The main issue which determines this gap is the fact that the theorem takes into account only the number of operations which a certain subregion of H can perform, without considering further communication problems among its nodes. The involved subregion is considered as an ideal PRAM, so that possible congestion problems in its edges are ignored.

Note also that for small value of τ_G, the lower bound could be not tight, since the theorem first considers the less connected part of G and the more connected part of H, which could lead to a weak bound. This fact is formalized in the following proposition.

Proposition 1. *Lower bound obtained by the application of Theorem 1 to networks G and H is not necessarily tight if*

$$\max_{v \in V_G} \sum_{j=1}^{\tau_G} b_G(v,j) = \omega \left(\min_{v \in V_G} \sum_{j=1}^{\tau_G} b_G(v,j) \right). \tag{2.3}$$

Proof. We will prove the theorem by counter-example. Consider the network G consisting in a $N/2$-nodes 3-dimensional array (a cube) joined with an $N/2$-nodes linear array by an edge from a vertex of the cube and a vertex of the array (it has diameter $N/2 + O(N^{1/3})$), and H consisting in an N-nodes 2-dimensional array. Network H can not emulate the cubic part of G without a certain amount of slowdown (we know from Theorem 1 that $S = \Omega(N^{1/9})$). Nevertheless, considering the whole G, the theorem does not highlight any slowdown: $\min_{v \in V_G} \sum_{j=1}^{\tau_G} b_G(v,j) = \Theta(\tau_G^2)$ for $\tau_G < O(N)$ and $\max_{u \in V_H} \sum_{i=1}^{\tau_H} b_H(u,i) = \Theta(\tau_H^{3/2})$ for $\tau_H < O(N^{1/2})$ give only pairs (τ_G, τ_H) such that $S = \Omega(\tau_H/\tau_G) = \Omega(1)$, since the theorem is biased by the weaker part of G. □

Proposition 2. *Lower bounds obtained by Theorem 1 are not necessarily tight.*

Proof. This result follows by Proposition 1. □

Lower bound in Corollary 1 is tight since binary tree is a loosely connected network and the optimal embedding in a k-dimensional array condenses the subtrees near the leaves in just one node of the array. In this case, increasing the load of some nodes of H does not increase the congestion of the network (the sub-trees communicate with just one node with the rest of the network).

Informally, the theorem is likely to be tight as long as the load and dilation (features of the emulation caught by the theorem) are bigger than the congestion (feature not considered by the theorem). Theorem 3 of Section 2.3 tries to solve this lack.

2.2.3 Generalization of Theorem 1

Theorem 1 considers the emulation of a network G by a network H modeling computation of G as a DAG. This subsection generalizes the theorem to target a general DAG F in place of the DAG obtained by the computation of a network.

As already noticed, DAG of computation of G for a certain time T_G, is very regular. Consider its greedy schedule: each level i has N_G nodes, corresponding to the evaluation of all nodes of G at time i; moreover level i has only edges to level $i+1$. Because of this regularity, once we identify $m_G = \min_{v \in V_G} \sum_{j=1}^{\tau_G} b_G(v,j)$ in the network G, we know that any node in the DAG of G has at least m_G predecessors in previous τ_G time step; in particular $b_G(v,j)$ counts the number of predecessors of v exactly j time steps before v. The theorem, as recalled in Subsection 2.2, identifies k nodes and relative predecessors in the DAG of G which imply $T_G < (k+1)\tau_G$, while H needs $T_H > k\tau_H$ time steps to emulate them.

Consider the greedy schedule of a general DAG F: it has T_F levels. In this case each level does not correspond to a time step of execution of a network, as well T_F is not the number of steps but simply the critical path of F. In analogy to Theorem 1, we can represent a node v of F as $v = (u,t)$, where t is its level in the greedy schedule and u is a unique name in F. Each level t could have a different number p_t of nodes and its nodes could be input also for nodes of levels other than $t+1$ (see Figure 2.4). Definitions of functions B_G and b_G, which in a network G counts neighbors of a node within a certain range, are valid also in a DAG F. Note that in this case F is the DAG, and nodes in the set $B_F(v,i)$ are all in levels preceding level of v; moreover the meaning of $\min_{v \in V_G} \sum_{j=1}^{\tau_G} b_G(v,j)$ in DAG of G in Theorem 1 is now assumed in DAG F

simply by $b_F(v, \tau_H)$. Let $P_F(v, i) = \{u : \delta(u, v) = i\}$ and $p_F(v, i) = |B_F(v, i)|$, then $b_F(v, \tau_H) = \sum_{i=1}^{\tau_H} p_F(v, i)$, similar to the notation for a DAG of a network, also if nodes in $p_F(v, i)$ are not necessarily in the same level. For example in Figure 2.4, $v \in B_F(u, 1)$, but v precedes u by three levels.

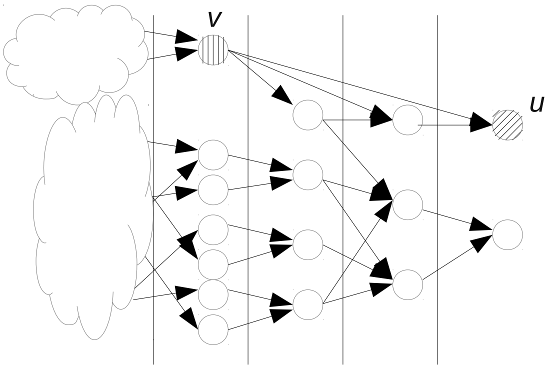

Figure 2.4: Greedy schedule of a general DAG: different levels can have different numbers of nodes and some node (e.g., v) can be used by several levels.

This invalidate the reasoning used in the original theorem to determine $T_G < (k+1)\tau_G$; in fact if we consider the last node $v = (u, T_F)$ of F compute by H, it is not true that it has m_F predecessors $(u', t), t \geq T_F - \tau_F$. In any case, if there is a limit Δ to the maximum distance in level between a node u and an its parent, we can prove the following theorem.

Theorem 2. *Let $F = (V_F, E_F)$ be a N_F-nodes DAG and $H = (V_H, E_H)$ be an N_H-nodes host network. Consider the greedy schedule of F; let T_F be the critical path and Δ be the maximum distance in levels between a node u and the furthest parent of it. Suppose that there are integers τ_F and τ_H such that*

$$\max_{u \in V_H} \sum_{i=1}^{\tau_H} b_H(u, i) < \min_{v \in V_F} b_F(v, \tau_F).$$

Then the number of time step of any computation of F by H is

$$T_H \geq \left(\frac{T_F}{\tau_F \Delta} - 1 \right) (\tau_H + 1).$$

Proof. Proof follows that of Theorem 1. Consider the last node $v_0 = (u_0, T_F)$ of F computed by H. Node v_0 has $b_F(v_0, \tau_F) \geq \min_{v \in V_F} b_F(v, \tau_F) = m_F$ predecessors between levels $T_F - \Delta\tau_F$ and $T_F - 1$ (we are considering predecessors at distance at most τ_F and each step could require up to Δ levels). Consider the node u of H which computes v_0 at host time T_H. A predecessor of v_0 created by a node at distance i from u must be created before time $T_H - i$ to influence the computation of u at time T_H, so from time $T_H - \tau_F$ to $T_H - 1$ at most $\max_{u \in V_H} \sum_{i=1}^{\tau_H} b_H(u,i) < m_F$ operations can influence computation of u and there is at least one node $v_1 = (u_1, t_1)$ computed by H at time $T_H - (\tau_H + 1)$ which occurs at or after level $t_1 \geq T_F - \Delta\tau_F$ in F.

We can repeat the reasoning on node v_1, pointing out a node $v_2 = (u_2, t_2), t_2 \geq T_F - 2\Delta\tau_F$ computed by a node of H before time $T_H - 2(\tau_H + 1)$. After k repetitions, we obtain a certain $t_k < \Delta\tau_F$, $t_k \geq T_F - k\Delta\tau_F$ computed by H at or before host time $T_H - k(\tau_H + 1) \geq 0$. We can merge $T_F < (k+1)\Delta\tau_F$ and $T_H \geq k(\tau_H + 1)$ to obtain:

$$T_H \geq \left(\frac{T_F}{\tau_F \Delta} - 1\right)(\tau_H + 1). \tag{2.4}$$

□

Asymptotically, Equation 2.4 becomes $T_H = \Omega(T_F \tau_H / (\Delta\tau_F))$. Note that in some important DAGs (e.g, tree, butterfly, diamond) $\Delta = 1$.

Application 1. Consider the execution of an N_F-leaves tree DAG F on an N_H-nodes k-dimensional array H. As for tree, when $\tau_F \leq \log N_F$, $b_F(v, \tau_F) = 2^{\tau_F + 1} - 1$, while in k-array when $\tau_H = O(\mathrm{diam}_H)$, $\max_{u \in V_H} \sum_{i=1}^{\tau_H} b_H(u,i) = \Theta(\tau_H^{k+1})$, and $\max_{u \in V_H} \sum_{i=1}^{\tau_H} b_H(u,i) = \Theta(N_H \tau_H)$ if $\tau_H = \omega(\mathrm{diam}_H)$.

Choosing $\tau_F = \log N_F$, if $\mathrm{diam}_H = \Omega(\sqrt[k+1]{N_F})$ we can use $\tau_H = \Theta(\sqrt[k+1]{N_F})$, in fact $\Theta(\tau_H^{k+1}) < 2N_F - 1$ which gives the lower bound $T_H = \Omega(\sqrt[k+1]{N_F})$. On the other hand, if $\mathrm{diam}_H = o(\sqrt[k+1]{N_F})$ from the inequality $\Theta(N_H(\tau_H - \mathrm{diam}_H)) < 2N_F - 1 \Rightarrow \tau_H = \Omega\left(\frac{N_F}{N_H}\right)$ which gives $T_H = \Omega(N_F/N_H)$. These lower bounds are strict, as we will show presenting a strategy with $T_H = \Theta(\sqrt[k+1]{N_F})$ when the k-dimensional array has $\mathrm{diam}_H \geq \sqrt[k+1]{N_F}$; when $\mathrm{diam}_H < \sqrt[k+1]{N_F}$ we exploit the fact that a M-nodes k-dimensional array can emulate a N-nodes k-dimensional array $(N > M)$ with $O(N/M)$ slowdown.

Note that a k-array with $\mathrm{diam}_G = \omega(N_F^{1/(k+1)})$ would not be useful since communication among nodes further than $O(N_F^{1/(k+1)})$ steps would slow down the computation.

Proposition 3. *A $N^{k/(k+1)}$-nodes k-dimensional array H can compute a N-leaves tree-DAG F in $O(N^{1/(k+1)})$ steps.*

Proof. First we show inductively Fact I:

> a k-array of $N^{k/(k+1)}$ nodes can compute a $N^{k/(k+1)}$-leaves tree in $O(N^{1/(k+1)})$ steps,

showing next how to extend the result to N-leaves trees. The **base case** for the induction is $k = 1$. This case considers a $\sqrt{N}$-nodes linear array H which compute a $\sqrt{N}$-leaves tree F in $\sqrt{N}-1$ steps with the following strategy $\mathcal{S}$. Node x_j of H has the jth node of level $(\log N)/2$ of F. In the first step nodes x_{2i} and x_{2i+1} cooperate to produce in x_{2i+1} the result of the second level of H (they need one communication step); in the second step nodes x_{4i+1} and x_{4i+3} cooperate to produce a result of the third level of H (they need two communication step) and so on, until nodes $i_{\sqrt{N}/2-1}$ and $i_{\sqrt{N}-1}$ create the output of H in $i_{\sqrt{N}-1}$, requiring $\sqrt{N}/2$ communication steps. The strategy requires $\sqrt{N} - 1$ parallel communication steps and $(\log N)/2$ parallel computational step.

Note that we can use this strategy to compute a whole N-leaves tree with a $\sqrt{N}$-nodes linear array in $O(\sqrt{N})$, by assigning nodes $\{i_{j\sqrt{N}}, \ldots, i_{(j+1)\sqrt{N}-1}\}$ of the $(\log N)$-th level of the tree to node x_j of H, so that it can compute sequentially a certain lower sub-tree L_j of F with height $(\log N)/2$ in $\Theta(\sqrt{N})$ steps (see Figure 2.5) and then applying strategy $\mathcal{S}$ for the computation of the upper part U_0 of the tree.

Now consider $k > 1$. The following **inductive hypothesis** holds: a $(k-1)$-array with $N^{\frac{k-1}{k}}$ nodes can compute a $(N^{\frac{k-1}{k}})$-leaves tree-DAG in $\Theta(N^{1/k})$. Consider a $N^{k/(k+1)}$-nodes k-array H formed by nodes $(i_1, \ldots, i_k), 0 \leq i_n \leq N^{1/(k+1)}, \forall n$ computing a $N^{k/(k+1)}$-leaves tree. Each node u of H contains a node v of level $k/(k+1)\log N$ of the tree, in particular node $(i_1, \ldots, i_k)$ contains node in position $\sum_{x=1}^{k} i_x N^{(x-1)/k+1)}$. H computes F in $O(N^{1/(k+1)})$ steps in this way: nodes $(u, i_2, \ldots, i_k), 0 \leq u < N^{1/(k+1)}$ use strategy $\mathcal{S}$ to compute a node v of level $\frac{k-1}{k+1}\log N$ in node $(N^{1/(k+1)} - 1, i_2, \ldots, i_k)$ in $N^{1/(k+1)} - 1$ steps. At this point

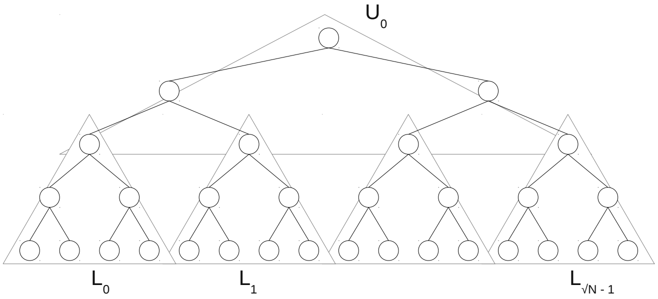

Figure 2.5: Division of tree DAG F in lower sub-trees $L_0, \ldots, L_{\sqrt{N}-1}$ and upper sub-tree U_0.

sub-array of H composed by nodes $(N^{1/(k+1)}-1, i_2, \ldots, i_k), 0 \leq i_x < N^{1/(k+1)}$ is a $k-1$-dimensional which has to compute the remaining $N^{(k-1)/(k+1)}$-leaves tree. By inductive hypothesis this can be done in $O(N^{1/(k+1)})$ steps, that jointly with $(N^{1/(k+1)}-1)$ steps of first part give a total of $O(N^{1/(k+1)})$ steps, so that Fact I is proved.

Note that similarly to base case, we can extend this general case to a N-leaves tree by allocating $N^{1/(k+1)}$ consecutive leaves of F per each node of H; each node computes sequentially a $(\frac{1}{k+1}\log N)$-levels sub-tree of F in $O(N^{1/(k+1)})$ steps, completing the last part of the computation with strategy used to prove Fact I. In the following we will refer to the first part of the process, which reduces the number of leaves of the tree from N to $N^{k/(k+1)}$ as *reduction step of strategy* $\mathcal{S}$. □

Note that this lower bound defers from Corollary 1, since it has a different meaning. In fact in Corollary 1 network H has to emulate the tree-shaped network G, while now network H is executing a tree-shaped DAG. Note also that executing $\log N$ steps of the optimal embedding of a tree in a k-dimensional array give $T_H = \Theta((N \log N)^{1/(k+1)})$, which does not match the lower bound.

Application 2. Consider the execution of an N_F-input butterfly by a N_H-nodes k-dimensional array H. We obtain the same lower bound $T_H = \Omega(\sqrt[k+1]{N_F})$ of the Application 1, since $B_F(v,i) = 2^{i+1}-1$. In fact each output of F is the root of a complete binary tree with N_F leaves, corresponding to inputs.

Despite the bigger complexity of the network, we can match the lower bound with a strategy similar to the previous one. Note that a $N^{k/(k+1)}$-nodes region

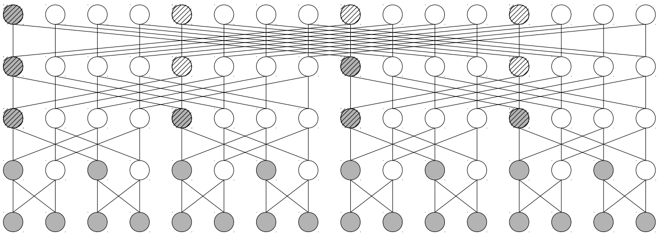

Figure 2.6: Every output of a butterfly is the root for a tree (gray nodes). Nodes of this tree in level $(\log N)/2$ can be thought as inputs of a $\sqrt{N}$-inputs sub-butterfly (the shaded nodes).

$\mathcal{R}$ of H, after the *reduction step of strategy* $\mathcal{S}$ described in Application 1 (which lasts $O(N^{1/(k+1)}))$ has in its nodes inputs of a $N^{k/(k+1)}$-inputs butterfly (Figure 2.6 shows the case with $N_F = 16$ and $k = 1$). $\mathcal{R}$ can compute it in $O(N^{1/(k+1)})$ time using the emulation of the descendant hypercube algorithm for butterfly-like algorithms.

If we consider $N^{1/(k+1)}$ regions of $N^{k/(k+1)}$ nodes of H, we can exploit $N^{1/(k+1)}$ times the previous observation to compute all N outputs in $O(N^{1/(k+1)})$ parallel steps. Note that this strategy use recomputation.

2.3 Congestion-based lower bound

2.3.1 State of the art

The theorem

The second theorem in [KLM+97], requires some more notation. Given an undirected graph $G = (V_G, E_H)$, we define the i-neighborhood of $U \subseteq G$, $\mathcal{N}_i(U) = (\cup_{u \in U} B_G(u, i)) \setminus U$, the set of nodes not in U within a distance i from a node in U.

A $(R, f(R))$-decomposition of a network $H = (G_H, E_H)$, is a partition of H in sets $\mathcal{R}_i$, such that each $\mathcal{R}_i$ (*region*) has $[R, 2R]$ nodes and $\mathcal{N}_1(\mathcal{R}_i) \leq f(R)$. Informally, we can partition H in regions of approximatively the same size and with a given upper bound to their bandwidth with the rest of the network.

The $(R, f(R))$-decomposition is used in the theorem to model H's features. As for G, it is modeled through the function $z_G(a, \varepsilon, c)$, where $a \leq c$ are integers in $[0, |V_G|]$ and $0 \leq \varepsilon < 1$. To understand the role of function z_G, consider the following situation: suppose that a region $\mathcal{R}$ of H has to emulate a set of a nodes of G for a particular time step t of the guest time. $\mathcal{R}$ has to import or compute predecessors of these nodes before computing them. In particular, if the nodes to compute are $X_0 \subset V_G$, it needs $Z_1 = X_0 \cup \mathcal{N}_1(X_0)$ for guest time $t-1$. If it imports Y_1 nodes among Z_1, it will have to compute nodes $X_1 = X_0 \cup \mathcal{N}_1(X_0) \setminus Y_1$ for time step $t-1$. We can iteratively define for time step $t-i$, the set $X_i = X_{i-1} \cup \mathcal{N}_1(X_{i-1}) \setminus Y_i$ of nodes which $\mathcal{R}$ will have to compute. Consider $i = k$ such that $|X_{k-1}| \leq c$ and $|X_k| > c$, with the constraint $\sum_{i=0}^{k} Y_i \leq \varepsilon a$ and k as big as possible. Function $z_G(a, \varepsilon, c)$ is an upper bound to k and it is defined so that it is non-increasing in a. Basically, it takes into account the expansion speed of network G considering how fast a region $\mathcal{R}$ of H, which has to compute a nodes of a guest time t and only imports εa nodes, must compute c predecessors of a previous guest time step.

Theorem 3. *Suppose that $H = (V_H, E_H)$ is an N_H-nodes host network with an $(R, f(R))$-decomposition, and that $G = (V_G, E_G)$ is an N_G-node guest network. Let*

$$\beta = \max\left\{ z_G\left(\frac{N_G}{4}, 0, \frac{3N_G}{4}\right), z_G\left(\frac{3N_G R}{8N_H}, \frac{1}{2}, \frac{N_G}{2}\right) \right\}.$$

Then for any emulation of G by H where $T_G > 3\beta$,

$$I \geq \min\left\{ \frac{R}{32\beta f(R)}, \frac{N_H}{192R} \right\}.$$

Again, we refer to [KLM+97] for the proof of the result, while here we just recall the main ideas. Consider the CDAG obtained by the execution of the network G for T_G time steps; the theorem partitions it in blocks of 3β consecutive steps and classifies each of it as *importer block* or *creator block*, depending if it can point out for some region a certain quantity of imported nodes or a certain quantity of nodes computed.

Note that each node $v \in V_G$ must be pebbled at least once for each $t \in [1, T_G]$, in particular in [KLM+97] the first pebble for v for time t created by H is called *t-primary pebble* of v. For each t, there are N_G t-primary pebbles and if we order them according the order in which they are created by H, we call the first $3N_G/4$

t-early pebbles and the last $3N_G/4$ *t-late* pebbles ($N_G/2$ t-primary pebbles are both t-early and t-late).

Focus on a block of the DAG from guest time $t-3\beta+1$ to t. The $(R, f(R))$-decomposition of H has at most N_H/R regions, so that in average every region computes $p = 3N_GR/(4N_H)$ t-early nodes. Consider the following two situations, which characterize an *importer block*: every region which produces $s \geq p/2$ t-primary nodes also imports at least $s/2$ predecessors of them of time steps between $t-2\beta$ and $t-1$; a region imports at least $3N_G/16$ nodes between $t-2\beta$ and $t-1$. In both cases we can determine a lower bound for the execution of the block (see [KLM+97]), due to the time needed to import $3N_G/16$ nodes by the at most N_H/R regions of H, which have a total bandwidth of $N_Hf(R)/R$ nodes per step: $T_{block} \geq 3N_GR/(16N_Hf(R))$. If half of the blocks are importer, we obtain a lower bound for the execution time of H:

$$T_H \geq \frac{T_G}{2\cdot 3\beta}\frac{3N_GR}{16N_Hf(R)} \qquad \Rightarrow \qquad S \geq \frac{N_GR}{32N_H\beta f(R)}.$$

If the previous hypothesis are not valid, it means that every region imports less than $3N_G/16$ nodes for guest steps $t-2\beta$ and $t-1$ and there is at least a region $\mathcal{R}$ which produces $s \geq p/2$ t-primary pebbles but imports less than $s/2$ for time steps $t-2\beta$ and $t-1$. With these limitations, we can show that $\mathcal{R}$ computes $\beta N_G/16$ pebbles for the considered block (the block is a *creator block*). Since $\mathcal{R}$ has at most $2R$ nodes, it needs at least $T_{block} \geq \beta N_G/(32R)$ steps to create the considered nodes. The creation of all these nodes are subsequent to creation of any $(t-3\beta)$-primary node, in this way the time required by the region for the creation of its t-primary nodes is disjoint by the time required by other blocks.

If half of the blocks are creator, we have the following lower bound:

$$T_H \geq \frac{T_G}{2\cdot 3\beta}\frac{\beta N_G}{32R} \qquad \Rightarrow \qquad S \geq \frac{N_G}{192R}.$$

When $T_G \geq 3\beta$, the two lower bounds found can be merged in

$$S \geq \min\left\{\frac{N_GR}{32N_H\beta f(R)}, \frac{N_G}{192R}\right\}, \tag{2.5}$$

and multiplying both members of inequality by N_H/N_G we obtain the proposition of the theorem.

In [KLM+97] the theorem is used to derive several results regarding the possibility of obtaining work preserving emulations of specific classes of networks by mean of other classes. For example, authors prove that k-arrays and butterflies can not perform work preserving emulation of expander networks, while a work preserving emulation of a butterfly G by a k-array H has at least slowdown $2^{\Omega(N_H^{1/k})}$.

Finally, they prove that a work-preserving emulation of a k-dimensional array G by a j-dimensional array H ($j < k$) has slowdown at least $\Omega(N_H^{(k-j)/k})$, so that such an emulation can not be work preserving ($S = O(N_G/N_H)$) if the j-dimensional array has more than $O(N_G^{j/k})$ nodes.

2.3.2 Analysis and Critique

Focus again on the emulation of an N_G-nodes k-dimensional array G by an N_H-nodes j-dimensional array H ($j < k$). In order to apply Theorem 3, we can obtain $\beta = O(N_G^{1/k})$ and $f(R) = R^{(j-1)/j}$. Replacing them in Equation 2.5, slowdown is

$$S \geq \min\left\{\frac{N_G^{(k-1)/k}R^{1/j}}{32N_H}, \frac{N_G}{192R}\right\};$$

in particular, lower bounds are both valid when

$$\frac{N_G^{(k-1)/k}R^{1/j}}{32N_H} = \frac{N_G}{192R} \qquad \Rightarrow \qquad R = \left(\frac{N_G^{\frac{1}{k}}N_H}{6}\right)^{\frac{j}{j+1}},$$

which holds to

$$S = \Omega\left(\frac{N_G}{\left(N_G^{\frac{1}{k}}N_H\right)^{\frac{j}{j+1}}}\right). \qquad (2.6)$$

Figure 2.3 represents the fact that for $N_H = O(N_H^{\frac{j}{k}})$ the lower bound is matched by an embedding and that Theorem 3 gives a tighter lower bound respect to 1 when $N_G > N_H$ (we can obtain this by comparing the lower bound in Equation

2.6 and those in previous subsection).

This fact is a counter-example which proves the following proposition.

Proposition 4. *Lower bounds obtained by Theorem 3 can be tighter than lower bounds obtained by Theorem 1.*

Now consider the emulation of a N_G-nodes tree G by a N_H-nodes k-dimensional array H. From the previous subsection we know that the optimal emulation slowdown is

$$S = \begin{cases} \Theta\left(\frac{N_G}{N_H}\right) & \text{if } N_H = o((N_G \log N_G)^{1/(k+1)}) \\ \Theta\left(\left(\frac{N_G}{\log^k N_G}\right)^{\frac{1}{k+1}}\right) & \text{if } N_H = \Omega((N_G \log N_G)^{1/(k+1)}) \end{cases} .$$

Parameter $f(R)$ for Theorem 3 still is $R^{(k-1)/k}$, while β is more tricky than in previous example. In fact $z_G\left(\frac{N_G}{4}, 0, \frac{3N_G}{4}\right) = O(\log N_G)$, while $z_G\left(\frac{3N_G R}{8N_H}, \frac{1}{2}, \frac{N_G}{2}\right) = O(N_G R/N_H + \log N_G)$, using a strategy to maximize β that targets a subtree as starting set and uses all importations available in its root.

Since depending on value of R, $1 \leq R \leq N_H/2$, the term $N_G R/N_H$ assumes values in $N_G/N_H \leq N_G R/N_H \leq N_G/2$, the inequality $N_G R/N_H = \omega(\log N_G)$ holds for every $N_H = o(N_G/\log N_G)$. In this case $\beta = O(N_G R/N_H)$ and Equation 2.5 becomes

$$S \geq \min\left\{\frac{N_G R}{32 N_H \frac{N_G R}{N_H} f(R)}, \frac{N_G}{192R}\right\} = \min\left\{\frac{1}{32 f(R)}, \frac{N_G}{192R}\right\},$$

and since in a connected network $f(R) = \Omega(1)$, we obtain

$$S = \Omega(1),$$

which is trivial. In this case the lower bound for the emulation of a tree by a k-dimensional array obtained by Theorem 1 is tight, while that obtained by Theorem 3 is not, so the following proposition is true.

Proposition 5. *Lower bounds obtained by Theorem 1 can be tighter than lower bounds obtained by Theorem 3.*

Now, we consider which elements make lower bounds of Theorem 3 weak.

Example 1. Consider the emulation of a $\sqrt{N} \times \sqrt{N}$ 2-dimensional mesh G by an N-nodes linear array. Theorem 3 gives $S = \Omega(N^{1/4})$ (obtained when $R = \Theta(N^{3/4})$), while the strict lower bound is $N = \Omega(N^{1/2})$, by Theorem 4. A $(R, f(R))$-decomposition of a linear array has $f(R) = O(1), \forall R$. If we consider the N-nodes network A in Figure 2.7, it has the same parameters of the $(N^{3/4}, f(N^{3/4}))$-decomposition of a linear array. Unlike linear array, it is easy to match the lower bound of Theorem 3 emulating G by means of A. In fact, we need just to consider one region, which has $N^{3/4}$ nodes, and perform the well-known work-preserving emulation of an N-nodes mesh on a $N^{3/4}$-nodes mesh.

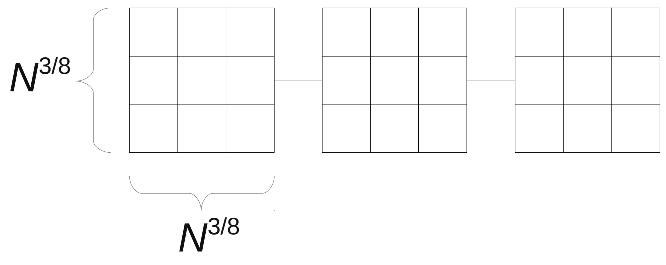

Figure 2.7: Consider an N-nodes "linear array of meshes" network: it consists of $N^{1/4}$ meshes $N^{3/8} \times N^{3/8}$ disposed as in picture. When $R = N^{3/4}$, the $(R, f(R))$-decomposition of this network and that of a N-nodes linear array are the same.

Example 2. Now consider network B in Figure 2.8, when $R = N^{1/4}$ its regions are $N^{1/4}$-nodes linear arrays and $f(R) = O(1)$, so its $(R, f(R))$-decomposition is similar to that of an $N^{3/4}$-nodes linear array, since their regions are linear arrays and have $O(1)$ edges with the rest of the network. Network B, unlike linear array, can emulate G in $O(N^{1/4})$: partition the mesh in sub-meshes $\sqrt[4]{N} \times \sqrt[4]{N}$, labeled $S_0, S_1, \ldots$ in row-major fashion, and then embed each of them in the corresponding linear array of network B, one column of the sub-mesh per node of the linear array. In this way we have an overall embedding with $l = c = d = O(\sqrt[4]{N})$ which implies an emulation slowdown $O(\sqrt[4]{N})$.

Example 3. Consider a N-nodes complete binary tree: when $R = O(\sqrt{N})$, we can partition it in $O(\sqrt{N})$ regions, one composed by the first $(\log N)/2$ levels of the tree and the others composed by the remaining sub-trees. In this case $f(R) = O(R)$, but only one region has this communication strength, while others just have one edge with the rest of the network. A network of $O(\sqrt{N})$ regions

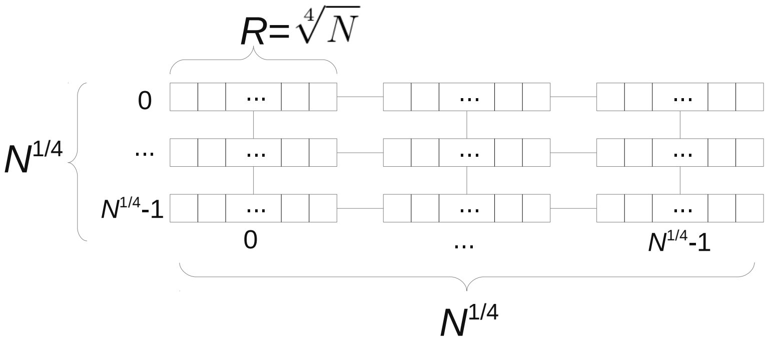

Figure 2.8: Consider an $N^{3/4}$-nodes "mesh of linear arrays" network: it consists of a $N^{1/4} \times N^{1/4}$ mesh of $N^{1/4}$-nodes arrays. disposed as in picture. When $R = N^{1/4}, f(R) = O(1)$.

consisting in complete graphs of $O(\sqrt{N})$-nodes, one communicating with each other, has the same $(R, f(R))$-decomposition, while by Theorem 1 we can prove that a binary tree can emulate it with $S = \Omega(\log N)$ (just consider $\tau_G = 3$ and $\tau_H = \log N - 1$).

These examples show how the $(R, f(R))$-decomposition does not describe adequately a network. In particular, it models weakly the following features:

Internal structure of a region. The internal structure of a region is summarized by its number of nodes R, without other elements to model the interconnection of these nodes. The part of lower bound provided by the computational load of the region, considers (asymptotically) just the trivial ratio N_G/R, modeling the region as a $\Theta(R)$-nodes PRAM, with an all-to-all interconnection among their nodes, while it could be as weak as a linear array, making the lower bound not strict (see Example 1).

Connection among regions Interconnections among regions are represented by the maximum number of edges $f(R)$ that there can be between a region and the rest of the network. This element hides the connection pattern, in fact there are several possibility, from a region communicating with $f(R)$ edges just with another region or regions communicating with $\Theta(f(R)R/N_H)$ edges with any region, with strong implications on the diameter of the network. (see Example 2)

Moreover it is also possible that only one region has $f(R)$ edges with the rest of the network, while other regions could have only as few as one edge (Example 3), so that $f(R)N_H/R$ is an overestimation of the real communication power of H.

Imported data In the part of lower bound derived by argumentations about the importation of predecessors, each region which computes $s \geq p/2$ nodes imports at least $s/2$ nodes or a region imports $\Omega(N_G)$ nodes, both in a time $O(\beta)$ (note that in several networks $\beta = \Omega(\text{diam}_G)$ because of term $z_G(N_G/4, 0, 3N_G/4)$ in its definition). In several emulations, the estimate of $\Omega(N_G)$ nodes for *total* communication traffic which H has to effort for the emulation of β step of G is weak, since each region has to accomplish it, with an overall bandwidth $f(R)$ in place of $N_H f(R)/R$.

When we are studying a lower bound with Theorem 3, to minimize S in Equation 2.5 we consider $R^* = \{R : R^2/f(R) = N_H\beta\}$, obtaining a particular S^*. A sufficient conditions for S^* to be tight is the existence of a work-preserving embedding of G in a region. In fact if this is true, the $\Omega(N_G/R)$ term in the lower bound matches with the $O(N_G/R)$ slowdown due to the work-preserving embedding. More in general, it suffices that the region can perform a general work-preserving emulation of G, also if this could be harder to find than an embedding.

2.4 Bandwidth-based lower bound

2.4.1 State of the art

The last technique which we are going to analyze is from [KR94]. Similarly to [KLM+97], [KR94] studies work-preserving emulations of fixed-connected networks, but it uses as main argument the relative communication power.

The paper considers *bottleneck-free networks* and *quasi-symmetric traffic distributions*, stating a lower bound corresponding to the ratio between bandwidths of the considered networks. We recall fundamental concepts to state the main result of [KR94], referring to the paper for the complete proof.

Definition 5. The *communication bandwidth* $\beta(M)$ of a machine M corresponds to the already introduced *bisection bandwidth* of M.

Definition 6. In a n-nodes machine M, a *symmetric traffic distribution* is a pattern of traffic where all source-destination message pairs have the same probability to occur.

Definition 7. In a n-nodes machine M, a *quasi-symmetric traffic distribution* is a pattern of traffic where $\Omega(n^2)$ of $n(n-1)$ source-destination message pairs have the same probability to occur, while the other pairs have zero probability.

Definition 8. A machine H is *bottleneck-free* if the average message delivery rate under any quasi-symmetric distribution on $m \leq |H|$ is at most a constant factor higher than the message delivery rate under the symmetric distribution (corresponding to $\beta(H)$).

The theorem consider bottleneck-free networks, which means networks where the communication power is "balanced"; in fact every sufficiently large subregion of the network is not fast more than a constant factor than the overall network in the exchange of an uniform communication pattern. For example a network obtained joining a $(N/2)$-nodes 2-dimensional mesh with a $(N/2)$-nodes linear array is not bottleneck-free (enabling only the mesh part, the networks has an average delivery rate of $\Theta(\sqrt{N})$ nodes per step, while the overall network is dominated by the linear array bottleneck $\Theta(1)$). The bandwidth of the host network H is viewed as a form of *communication capability* which during an emulation has to execute the *communication complexity* of the emulated network G.

Moreover the considered emulation of a network G has a minimum duration $\lambda(G)$, required to avoid strategies which through recomputation and local computation of predecessors have few or no communication for "short" emulations. Parameter $\lambda(G)$ depends on bandwidth of G. Note that classical networks (k-arrays, trees, butterflies, etc.) are bottleneck-free, and they have λ proportional to the diameter.

Theorem 4 (The Efficient Emulation Theorem)**.** *Any efficient emulation of a fixed degree guest graph G on host H has slowdown $S \geq \Omega\left(\frac{\beta(G)}{\beta(H)}\right)$ if: 1) the guest time τ satisfies $\tau \geq (1+\Theta(1))\lambda(G)$, and 2) H is bottleneck-free.*

Note that this work defines *efficient emulation* an emulation which produces a DAG D_H with at most a constant factor more nodes than the DAG D_G produced by G. In other words, the work W_H performed by H is at most a constant number of times greater than the work W_G of G, $W_H = O(W_G)$. This means that the result is specific for computations where each node of G is in average recomputed at most a constant number of time.

In [KR94] the theorem is used to study maximum sizes of networks to perform work-preserving emulation of other networks, matching the same bounds of [KLM+97], except for expander networks which can not be managed by the theorem, while working with parameters much more intuitive than those of Theorem 3.

2.4.2 Analysis and Critique

We want to explore the tightness of the bound provided by the theorem for the emulation between two network.

First of all, note that emulations where network G has the bandwidth lower than bandwidth of H, $S = \Omega(1)$ and the bound is probably going to be loose. For example in the emulation of a n-nodes binary tree G (which has $\beta(G) = 1$) on a k-array H, $S = \Omega(1)$, while Theorem 1 gives a tight bound (see Corollary 1). This example proves the following proposition.

Proposition 6. *Lower bounds obtained by Theorem 4 are not necessarily tight: lower bounds obtainend by Theorem 1 can be tighter than lower bounds obtained by Theorem 4.*

Now consider Theorem 3; unlike Theorem 4, it manages bottleneck-free host networks as the “linear array of meshes” of Figure 2.7 in Example 1 for which Theorem 3 gives a strict bound; on the other hand we can show that in some cases Theorem 4 can be tighter than the first two Theorems, at least in its domain of validity.

Proposition 7. *When considering computations where partial results are in average recomputed a constant number of times, lower bounds obtained by Theorem 4 can be tighter than lower bounds obtained by Theorems 1 and 3.*

Proof. Let G be a k-array and H a j-array, $k > j$, since $\beta(G) = \Theta(N^{(k-1)/k})$ and $\beta(H) = \Theta(N^{(j-1)/j})$, Theorem 4 gives

$$S = \Omega\left(\frac{N_G^{\frac{k-1}{k}}}{N_H^{\frac{j-1}{j}}}\right)$$

which is tight and can be matched by an embedding (see Proposition 10). We have already see above that Theorems 1 and 3 provide loose lower bounds in this case. □

This result matches the upper bound given by Proposition 10, but it is not valid when recomputation is allowed to be more than constant.

Finally we prove that for some network pairs the best lower bound among those provided by Theorems 1, 3 and 4 is not necessarily strict.

Proposition 8. *The tightest lower bound among those provided by Theorems 1, 3 and 4 for a network pairs G and H is not necessarily tight.*

Proof. Consider as guest network G a N-nodes 3-dimensional array and as host network H a N-nodes network consisting in a $N/2$-nodes linear array joined to a $N/2$-nodes two-dimensional mesh. Theorem 4 can not handle H since it is not bottleneck-free (enabling only source-destination pairs of the mesh part we have an expected average delivery rate asymptotically greater than when all H source-destination pairs are enabled). Theorem 1 gives a lower bound similar to the one it provides for the case G 3-dimensional array and H 2-dimesional array, while Theorem 3 gives a lower bound similar to the case G 3-dimensional array and H linear array. Both the lower bounds are weak, since it is possible to obtain tighter lower bounds (e.g., Section 3.3 gives $S = \Omega(N_G^{2/3}/(N_H^{1/2} \log N_G))$ if we consider only the 2-dimensional array part of H and this bound holds also considering the whole network).

□

2.4.3 Summary

We recalled three theorems from [KLM+97, KR94], which provide the three main lower bound techniques for network emulations when recomputation is allowed.

We proved that they are not mutually-inclusive, in the sense that we can exhibit pairs of networks where each technique gives a better bound than the others.

Hypotheses and domains of the Theorems 1 and 3 are the most general possible, while Theorem 4 is valid when H is bottleneck-free and the emulation work W_H is proportional to $O(W_G)$, so it does not really answer to the question stated at the beginning of the Chapter.

The union of results obtained by all the theorems gives tight lower bounds for most of the “classical” networks (see Table 2.1) but also if we consider them together, we will not obtain strict bounds for any network pairs (Proposition 8). In next chapter we present a new technique which considers an arbitrary level of recomputation and it can exhibit an almost strict lower bound for emulations among multidimensional arrays in every case.

2.5 Relation between parallel computing and hierarchical memories

We already hint at similarity between efficiency in memory hierarchies and in parallel computating. Modern efficient memory architectures and algorithms exploit mainly two features, *locality* and *concurrency* of memory accesses. Locality organizes computation to re-use data once it is moved in locations near the processor, in this way the first (possibly) slow access to a certain operand is amortize by the (possibly) subsequent fast accesses. Concurrency allows latency hiding through the overlap of accesses. Locality is extensively discussed and formalized in the Hierarchical Memory Model [AACS87], in the Block Transfer Memory [ACS87], which partially deals also with concurrency, and in the Two-Levels Disk Model [AV88, V98]; as for concurrency, the Pipelined Memory Model [LP93] is a good introductory study. Recently these two features have been considered jointly in [BEP09], which introduces a pipelined and hierarchical memory design, complying with physical constraints. The pipelined and hierarchical memory jointly with the SPE processor is able to match RAM complexity ($O(1)$ slowdown) on wide classes of programs, exploiting both concurrency and locality.

Concurrency and locality were already been extensively studied in parallel

Case	Lower bound T1	Lower bound T2	Lower bound T3[a]	Best known upper bound
$G = k$-array $H = j-$array	$S = \begin{cases} \Omega(N_G/N_H) & N_H \leq N_G^{\frac{j(k+1)}{k(j+1)}} \\ \Omega\left(N_G^{\frac{k-j}{k(j+1)}}\right) & N_H > N_G^{\frac{j(k+1)}{k(j+1)}} \end{cases}$	$S = \Omega\left(\frac{N_G}{\left(N_G^{\frac{1}{k}} N_H\right)^{\frac{j}{j+1}}}\right)$	$S = \Omega\left(\frac{N_G^{\frac{k-1}{k}}}{N_H^{\frac{j-1}{j}}}\right)^b$	Embedding $S = O\left(\frac{N_G^{\frac{k-1}{k}}}{N_H^{\frac{j-1}{j}}}\right)$ Proposition 10
$G =$ binary tree $H = k-$array	$S = \Omega\left(\sqrt[k+1]{\frac{N_G}{\log^k N_G}}\right)$	$S = \Omega(1)$	$S = \Omega(1)$	$S = O\left(\sqrt[k+1]{\frac{N_G}{\log^k N_G}}\right)$ [HKMU91]
$G =$butterfly $H = k-$array	$S = \Omega\left(\sqrt[k+1]{\frac{N_G}{\log^k N_G}}\right)$	$S = \Omega\left(\frac{N_G}{(N_H \log N_G)^{\frac{k}{k+1}}}\right)$	$S = \Omega\left(\frac{N_G}{N_H^{\frac{k-1}{k}} \log N_G}\right)$	

Table 2.1: Summary comparison among considered techniques: lower bounds are provided by Theorem 1 (T1), Theorem 3 (T2) and Theorem 4 (T3). Not reported some important emulations as k-array on butterfly and tree on butterfly, since they have a $O(1)$ emulation (see [KLM+97, GH91]) and all lower bounds are just $\Omega(1)$.

[a] Not valid if average recomputation is more the constant

[b] The fact that this lower bound matches the upper bound means that recomputation does not improve performance in this case.

computing, since the former allows independent executions among the processors and the latter limits communication among these. This fact is pointed out by several works which show how to effectively simulate parallel models in memory models, partially carrying the knowledge of parallel computing in this field. For example, in [CGG+95] and [LP93], general Parallel Random Access Machine (PRAM, the ideal parallel machine model) simulations are proposed respectively on DM and PM, deriving new upper bounds for some problems on these memory models exploiting previously known parallel results. In [FPP06] and [PPS06] is shown how to turn the submachine locality of the Decomposable Bulk Synchronous Parallel model (D-BSP, a parallel model where also communication and synchronization costs are considered) in locality of references for the HMM and in cache-oblivious algorithms in the Ideal Cache Model. Finally in [MZ12] algorithms for Work-Time Model [VW85] are adapted to the pipelined hierarchical memory model; in particular when work-optimal parallel algorithms are involved, the obtained pipelined hierarchical memory algorithm has the same performance of the ideal RAM algorithm.

All these works are one way: the parallel algorithmic knowledge is used to have an insight in hierarchical memory setup. In this section, after recalling the red-blue pebble game of Hong and Kung proposed in [HK81], a general method to model and obtain lower bounds for I/O accesses during the execution of algorithms, we show why general results for hierarchical memories do not give insights in parallel contexts.

2.5.1 State of the art

The red-blue pebbles game is a suitable way to represent a computation on a machine with a certain amount of fast memory. It is introduced in [HK81], together with the results recalled in this subsection.

The game is played on a DAG F with S red pebbles, representing fast memory cells and an infinite number of blue pebbles, representing the slow memory. At the beginning of the game, the inputs have a blue pebbles, while the other vertices have no pebbles. The game proceeds by rounds, in each of which we can perform one of the following moves:

R1. a red pebble can be placed in any vertex with a blue pebble; this move corresponds to copy a cell from slow memory to fast memory (input operation)

R2. a blue pebble can be placed in any vertex with a red pebble; this move corresponds to copy a cell from fast memory to slow memory (output operation)

R3. a red pebble can be placed in a vertex whose all immediate predecessors have a red pebble; this move corresponds to the computation of a vertex (compute operation)

R4. a red or blue pebble can be removed from any vertex; this move corresponds to the eviction of a vertex from fast/slow memory (delete operation)

The game terminates when all outputs have a blue pebble; we are interested in the game which minimize the number of input and output operations Q:

$$\begin{aligned} Q &= \text{minimum number of moves R1 and R2} \\ & \text{required by any computation} \end{aligned}$$

In the paper is also introduced the S-partitioning of a DAG and relationship between red-blue games and S-partitioning is proved.

Definition 9. Given a DAG $F = (V, E)$, the family of subsets $\{V_1, \ldots, V_h\}$ of V is a S-partition of F if

P1. $\{V_1, \ldots, V_h\}$ is a partition of V, or V_i are disjoint and $\cup_{i=1}^{h} V_i = V$.

P2. For each V_i, the exists a *dominator set* D_i that contains at most S vertices. A dominator set for V_i is a set of vertices in V such that every path from an input of F to a vertex of V_i contains some vertex in the set, which means that with vertices in D_i we can compute the whole V_i without further I/O operations.

P3. For each V_i, the *minimum set* M_i has at most S vertices.

P4. The is no cyclic dependence among vertex sets in $\{V_1, \ldots, V_h\}$, where V_i depends on V_j if there is an edge in E from a vertex of V_j to a vertex of V_i.

Theorem 5. *Let F be a DAG; any complete red-blue pebble game using at most S red pebbles is associated with a $2S$-partition of F such that*

$$S \cdot h \geq q \geq S \cdot (h-1),$$

where q is the I/O time required by the complete computation and h is the number of vertex sets in the $2S$-partition.

The relation formalized in Theorem 5 is important since it leads to the following lower bound:

Proposition 9. *For any DAG F, the minimum I/O time satisfies*

$$Q \geq S \cdot (P(2S) - 1),$$

where $P(S)$ is the minimum number of sets that any S-partition of F must have.

This proposition is thought to be the most general framework for I/O complexity lower bounds (it allows for recomputation of vertices of F) and it is the starting point of several further works (e.g. [S95, EPR+13]).

In [HK81], similarly to S-partition also *S-dominator partition* of F is defined, as a family of subset $\{V_1, \ldots, V_h\}$ of V which satisfy properties P1, P2 and P4, while property P3 is not necessarily fulfilled. Since the minimum number of vertex sets of a S-dominator partition $P_D(S)$ is smaller or equal to $P(S)$, Proposition 9 is valid also for this kind of partitioning.

This result is used to prove the asymptotically tight lower bound for I/O access in the FFT algorithm.

Theorem 6. *The minimum number of I/O accesses during the computation of a n-points FFT using S fast memory cells is*

$$Q = \Omega\left(\frac{n \log n}{\log S}\right).$$

Another useful definition introduced in the paper is the *information speed.*

Definition 10. Consider a DAG F where all inputs can reach all outputs through vertex-disjoint paths, each of these paths is called *line*. The *information speed function* for F is $\Omega(f(d))$ if for each pair (u, v) of nodes of F on the same line and with a distance of at least d edges one from each other, there are at least $f(d)$ vertices in F satisfying the properties:

1. vertices belong to different lines
2. each vertex belong to a path from u to v.

We can reverse the last definition, stating that if a dominator has S nodes, it can involve at most S lines, and we can not compute more than $O(f^{-1}(S))$ nodes for each line. The maximum number of vertices in a set computable starting from a dominator set of S vertices is at most $O(S \cdot f^{-1}(S))$, so there are at least $\Omega(L/S \cdot f^{-1}(S))$ sets, each requiring S I/O accesses. This leads to the following lower bound for I/O complexity based on information speed.

Theorem 7. *In any DAG F where all inputs can reach all outputs through vertex-disjoint paths, L is the number of vertices belonging to lines and the information speed is $\Omega(f(d))$, with f a monotone increasing, invertible function then*

$$Q = \Omega\left(\frac{L}{f^{-1}(S)}\right).$$

2.5.2 Relations with parallel computing

Thanks to the relation with the red-blue pebble game, a S-partition of F can be thought as a computation, topologically ordered, of sets V_i obtained by a repetition of the two steps: load in fast memory of the dominator set D_i, computation in fast memory of V_i (and possibly recomputation of some vertices of already computed V_j). If we consider nodes $S(D_i)$ which can be computed starting from D_i, every path from nodes in $S(D_i)$ to nodes in $S(D_i)$ contains only nodes in $S(D_i)$; we refer to this property as *set convexity*.

In suitable DAGs, we also know that a dominator set of S vertices can not support a computation for more that $f^{-1}(S)$ vertices in a line.

If we consider similar approaches in parallel computing, our task consist in mapping subsets of the S-partitioning of the DAG F in nodes or subregions of

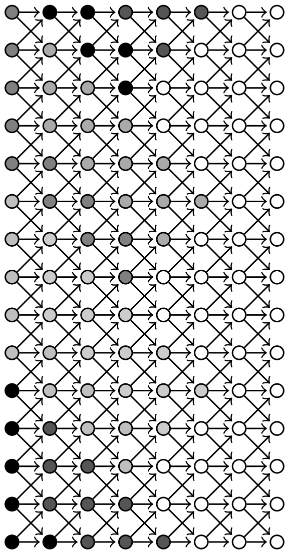

Figure 2.9: DAG produced by the execution of a ring, where nodes at a certain time step i are deployed in column (not represented wraparound edges among bottom nodes and top nodes). Subsets 1, 2 and 3 (respectively in dark, middle and light gray) are part of a S-partitioning where sets are convex but not topologically sortable. Dark nodes in a subset represent its dominator set.

a parallel network G. Each node/region imports dominator sets of the subsets which it has to compute, communicating minimum sets to other nodes/regions.

Subsets of the S-partitioning do not require properties P2 and P4, since more processors can concurrently proceed in the computation of various $S(D_i)$, communicating vertices of D_i not yet available when computation of a subset starts. In Figure 2.9 we can see for example the DAG F due to the execution of a ring, which can be S-partitioned by convex sets not topologically sortable. In fact vertices belonging to dominator sets corresponding to execution of nodes in instants $t > 0$ are not available at the beginning of the computation, and they can be loaded in advance unless we can compute them all concurrently. The partitioning gives information about the work and the imported nodes of a specific node or subregion of the parallel network.

Note also that without P2 and P4 the maximum size of a subset induced by a dominator set can not be predicted by information speed theorem. Consider for example the tree DAG case: in Hong and Kung partitioning each subset has a $O(S \log S)$ upperbound, while the maximum size without P2 and P4 is $\Theta(S^2)$, obtained selecting as dominator set a subtree of $S/2$ nodes at a specific time step of the DAG, jointly with the nodes corresponding to the evaluation of the root of the considered subtree in the following $S/2$ time step (this dominator set provides a subset of at least $S^2/4$ nodes).

Chapter 3

Lower bounds for specific networks

In the previous chapter we saw that in some cases state-of-the-art theorems are not tight because of their generality or are tight only in specific domains. In this chapter we present a new technique which is particularly effective in array emulations, exhibiting an almost tight lower bound for arbitrary levels of recomputation.

3.1 Lower bounds for multidimensional arrays emulations

Multidimensional arrays are well-known networks, often the first to be introduced in Parallel Computing courses. Nevertheless we still lack tight lower bounds for the slowdown of emulation of a multidimensional array emulated by another one with a smaller number of dimensions when the emulation has no particular constraints.

We recall that nodes of a j-dimensional array G can be considered as j-uple of coordinates in a j dimensional space: $(i_1, i_2, \ldots, i_{j-1}, i_j) : 0 \leq i_n < N^{1/j}, 1 \leq n \leq j$, where $u = (i_1, i_2, \ldots, i_{j-1}, i_j)$ has edges to all nodes which have exactly one coordinate that differs of one unit from the equivalent of u.

A M-nodes k-dimensional array can emulate a N-nodes k-dimensional array,

$M \leq N$, with slowdown $O(N/M)$ as also a M-nodes j-dimensional array H can emulate a N-nodes k-dimensional array G, $k > j$, $M \leq N^{\frac{j}{k}}$ [FF82]. By [KLM+97], if $M > N$ there are no work-preserving emulations between G and H, but the bounds provided in the paper do not match the best known emulations, namely embeddings. Similarly [KR94] matches the upper bound only if we limit recomputation to a constant rate.

In this chapter we analyze more specifically the range $N_G^{\frac{j}{k}} \leq N_H \leq N_G$, providing strict bounds under several hypothesis. First of all, we prove that a lower bound for any embedding is determined by the ratio of bisection bandwidths of G and H, and this lower bound can be matched.

Theorem 8. *Any embedding of an N_G-nodes k-array G in an N_H-nodes j-array H, $N_H = N_G^{(\frac{1}{k}+h)j}, 0 \leq h \leq \frac{k-j}{kj}$, $k > j$, has slowdown*

$$S = \Omega\left(\frac{N_G^{\frac{k-1}{k}}}{N_H^{\frac{j-1}{j}}}\right) = \Omega\left(\frac{N_G^{\frac{k-j}{k}}}{N_G^{h(j-1)}}\right).$$

Proof. A certain embedding e of G in H is given. Network H can be split in two sets a and b, in the following way. Set a contains $(0, 0, \ldots, 0)$, so that it emulates at least a certain number of nodes of G, mapped in $(0, 0, \ldots, 0)$. We continue to add nodes to a in lexicographical order as long as a emulates at least $N_G/2$ distinct nodes of G.

The bisection bandwidth of G is $\Omega\left(N_G^{\frac{k-1}{k}}\right)$, while the number of edges between a and b is $O\left(N_H^{\frac{j-1}{j}}\right)$, so that in embedding e at least $\Omega\left(N_G^{\frac{k-1}{k}}\right)$ edges of G pass through $O\left(N_H^{\frac{j-1}{j}}\right)$ edges of H giving a congestion

$$c = \Omega\left(\frac{N_G^{(k-1)/k}}{N_H^{(j-1)/j}}\right) = \Omega\left(\frac{N_G^{\frac{k-j}{k}}}{N_G^{h(j-1)}}\right).$$

□

Proposition 10. *The following embedding of a N_G-nodes k-array G in a N_H-*

nodes j-array H, $N_H = N_G^{(\frac{1}{k}+h)j}, 0 \leq h \leq \frac{k-j}{kj}$, $k > j$, has

$$S = O\left(\frac{N_G^{\frac{k-1}{k}}}{N_H^{\frac{j-1}{j}}}\right) = O\left(\frac{N_G^{\frac{k-j}{k}}}{N_G^{h(j-1)}}\right). \tag{3.1}$$

Proof. The proof is by construction: first we prove the statement when $k \leq 2j$ and then we exploit it in the general case.

Base Case Let $j < k \leq 2j$. Consider F_G^0, the j-dimensional surface of G defined as $\{(x_1, \ldots, x_k) : 0 \leq x_i < \sqrt[k]{N_G} \quad \forall 1 \leq i \leq j, x_i = 0 \quad \forall i > j\}$. Each point $(a_1, \ldots, a_j, 0, \ldots 0)$ of this surface has an associated $(k-j)$-array $(k-j \leq j)$ $A_{(a_1,\ldots,a_j)} = \{(a_1, \ldots, a_j, x_{j+1}, \ldots, x_k) : 0 \leq x_i < \sqrt[k]{N_G} \quad \forall i > j\}$.

Consider a partitioning of H in $N_G^{\frac{j}{k}}$ regions, each of which is a N_G^{hj}-nodes j-arrays. We label each region as a node of F_G^0, in particular region $R_{(a_1,\ldots,a_j)} = \{(x_1, \ldots, x_j) \in V_H : a_i N_G^h \leq x_i < (a_i + 1)N_G^h\}$ is in charge for the execution of $A_{(a_1,\ldots,a_j)}$ and it can do it with an embedding with $l = N_G^{\frac{k-j}{k}}/(N_G^h)^j, d = 1$ and $c = \Theta(N_G^{\frac{k-j-1}{k}}/N_G^{h(j-1)})$. This case is an embedding of a $(k-j)$-array in a j-array, where by hypothesis $k - j \leq j$; the same embedding is repeated in every $R_{(a_1,\ldots,a_j)}$.

This mapping of As in Rs determines a mapping of each processor of G in H, moreover edges among near $A_{(a_1,\ldots,a_j)}$ in G can be embedded in H with $d = 1$ and $c = \Theta(N_G^{\frac{k-j}{k}}/N_G^{h(j-1)})$. The overall embedding has $l = N_G^{(k-j)/k}/(N_G^h)^j, d = 1, c = \Theta(N_G^{\frac{k-j}{k}}/(N^h)^{j-1})$, with consequent slowdown $S = \Theta(N_G^{\frac{k-j}{k}}/(N^h)^{j-1})$, which respects Equation 3.1.

General Case We exploit the fact that a N_H-nodes j-array H can emulate a N_G-nodes x-array G, $x < k$, by embedding with $S = O\left(\frac{N_G^{\frac{x-1}{x}}}{N^{\frac{j-1}{j}}}\right)$ to prove the theorem.

Consider again surface F_G^0 of G, its associated $(k-j)$-dimensional array $A_{(a_1,\ldots,a_j)}$ and a partitioning of H in $N_G^{\frac{j}{k}}$ regions, each of which is a N_G^h-nodes j-arrays $R_{(a_1,\ldots,a_j)} = \{(x_1, \ldots, x_j) \in V_H : a_i N_G^h \leq x_i < (a_i + 1)N_G^h\}$ in charge for the execution of $A_{(a_1,\ldots,a_j)}$. Since $A_{(a_1,\ldots,a_j)}$ has $k - j < k$ dimensions, by the base case $R_{(a_1,\ldots,a_j)}$ can perform the task with an embedding with slowdown $S = O(N_G^{\frac{k-j}{k}}/(N^h)^{j-1})$. In each region we use the same embedding, so that we can merge them for the overall embedding of G in H; communication among adjacent $A_{(a_1,\ldots,a_j)}$ can be performed in systolic way with consequent $d = 1$ and $c = \Theta(N_G^{\frac{k-j}{k}}/N^{h(j-1)})$. □

These bounds show that emulation time among arrays is determined by ratio of bisection bandwidths when we are in the range $N_G^{\frac{j}{k}} \leq N_H \leq N_G$, while for $N_H < N_G^{\frac{j}{k}}$ the bound is determined by average load. The results hold only for embeddings, and they do not consider more general emulations, for examples those in which a node of G is not always computed from the same nodes of H.

In previous chapter, Theorem 4 from [KR94] shows that if a constant amount of recomputation is allowed (each node of G is computed at most a constant number of times by nodes of H) the same bound is still valid. Its proof is no more valid if recomputation is more than constant; in the following we provide an alternative proof which allows to understand why recomputing $\omega(1)$ times nodes of DAG of G could improve the speed of the emulation.

Theorem 9. *Any emulation of $T_G \geq diam_G$ steps of an N_G-nodes k-array G in an N_H-nodes j-array H, $N_G^{\frac{j}{k}} \leq N_H \leq N_G$, $k > j$, where any node of G is computed at most a constant number of times by nodes of H has*

$$S = \Omega\left(\frac{N_G^{\frac{k-1}{k}}}{N_H^{\frac{j-1}{j}}}\right). \qquad (3.2)$$

Proof. Consider the DAG DAG_G of G divided in blocks of $diam_G$ steps. Let focus on a specific block B starting at step t. In a way similar to Theorem 8, we partition H in n regions $\rho_0, \cdots, \rho_{n-1}$ where each region computes about $N_G/2$ distinct nodes of time step t of G and has a bandwidth of $O(N_H^{\frac{j-1}{j}})$. Regions have $\omega(N_H^{\frac{j-1}{j}})$ nodes, otherwise the load during B would imply the stated bound: a region of $N_H^{\frac{j-1}{j}}$ nodes requires $\frac{N_G/2}{N_H^{\frac{j-1}{j}}}$ steps for the computation of nodes of step t of G (excluding predecessors), so that average slowdown in $diam_G$ steps is the one in Equation 3.2. Since recomputation is constant, n is constant. It exists at least a region ρ that computes also $\Omega(N_G)$ nodes of time step $t + diam_G$, otherwise the sum of nodes of G computed by H for time step $t + diam_G$ would be less then N_G.

We call a *line* of B the set of nodes $\{(u, \tau) : u \in V_G, t \leq \tau \leq t + diam_G\}$, while (u, t) and $(u, t + diam_G)$ are the extreme nodes for a line. Consider the $\Omega(N_G)$ lines whose ρ computes the extreme node $(u, t + diam_G)$, they are of two kinds: lines entirely computed by ρ and lines for which ρ has to import at least a node. If the latter are $\Omega(N_G)$ a trivial lower bound

$$S = \frac{c}{bT_B} = \Omega\left(\frac{N_G}{N_H^{\frac{j-1}{j}} O(N_G^{1/k})}\right) = \Omega\left(\frac{N_G^{\frac{k-1}{k}}}{N_H^{\frac{j-1}{j}}}\right)$$

due to communication is valid for block B (c is the imported data, b the bandwidth of ρ and T_B the duration of B). Otherwise the same bound holds and the $\Omega(N_G)$ data is given by disjoint paths from extreme nodes $(u, t + diam_G)$ computed by ρ and extreme nodes (u, t) not computed by ρ.

If there are several blocks, $B_0, \cdots, B_{x-1}$, we repeat this reasoning on the even ones. Since no node of B_{l+2} can be computed until all nodes of B_l have been computed at least once, lower bounds of even blocks can be sum together, proving the theorem statement. □

If the initial redundancy of nodes were $\omega(1)$, maybe we could exploit it to save communication. Regions could import $o(N)$ data per block, keeping at least a copy of the state of each processor in the network and refreshing the redundancy only periodically. We are only speculating on the effectiveness of this strategy, to highlight that it is not obvious that the best possible emulations suffice of constant recomputation.

Lower bounds provided by Theorems 1 and 3 from [KLM+97] are valid for any emulation, but they have a polynomial gap from the upper bound of the embedding. In particular $S = \Omega\left(N^{\frac{k-j}{k(j+1)}}\right)$ when $N_H = N_G = N$; a complete graphical comparison among lower bounds is provided in Figure 2.3.

At this point it is not clear if recomputation can generate emulation strategies faster than Proposition 10 or if the known lower bounds are not strict. Following sections show a lower bound which differs only of a logarithmic factor in the denominator from the upper bound determined by embedding emulation.

3.2 Mesh over linear array emulation

In this section we are going to derive a lower bound for the emulation of an N-nodes two dimensional mesh G on an N-nodes linear array H valid for any rate of recomputation. Theorems 1 and 3 provide $S = \Omega(N^{1/4})$, which, as we will see, is not tight; Theorem 4 does not works under these conditions.

Consider G executing for $T_G = \text{diam}_G \frac{\log N}{2}$ time steps. During this execution, G performs $NT_G = \Theta(N\sqrt{N}\log N)$ work. Let $u_0, \ldots, u_{N-1}$ be the nodes of G labeled in row major order and $(u_i, t_j), 0 \leq i < N, 0 \leq j < T_G$, the nodes of the DAG D_G generated by the execution of G during the considered interval. As in Chapter 2, let $\mathcal{N}_G(u,i)$ be the nodes of G whose shortest path from u has length i, $\mathcal{N}_G(u,i) = \{v \in V_G : \text{dist}(v,u) = i\}$. Then in D_G, $(u, t-1)$ and $(v, t-1), \forall v \in \mathcal{N}_1(u,1)$ are the only inputs of (u,t), and (u,t) is an input for $(u, t+1)$ and $(v, t+1), \forall v \in \mathcal{N}_1(u,1)$.

At the beginning of the emulation, there is at least a copy of each $(u_i, 0)$ spreads in H, possibly also every node of H holds all $(u_i, 0)$. During the emulation, H produces every node of D_G, and the emulation terminates when every (u_i, T_G) has been produced at least once. Nodes (u,t) can be computed more than once (the emulation allows recomputation) in different nodes of H or in the same node.

The computation is featured by a certain maximum number c of messages that pass through an edge during the emulation. If $c \geq N/4$, the following trivial lower bound holds:

$$S \geq \frac{c}{T_G} = \frac{N/4}{\text{diam}_G \frac{\log N}{2}} = \frac{N}{2(2\sqrt{N}-1)\log N} = \Omega\left(\frac{\sqrt{N}}{\log N}\right).$$

Now consider $c < N/4$ and H divided in two subregions, $\rho_{1,0}$ and $\rho_{1,1}$, with the same number of nodes (or at most with a difference of a node). The first evaluation of at least $N/2$ outputs of D_G occurs in $\rho_{1,0}$ or in $\rho_{1,1}$. Let this subregion be $\rho_{1,0}$; because of the communication limit only set $c_1, |c_1| \leq c$, of predecessors can be imported and great part of the last diam_G steps of the DAG

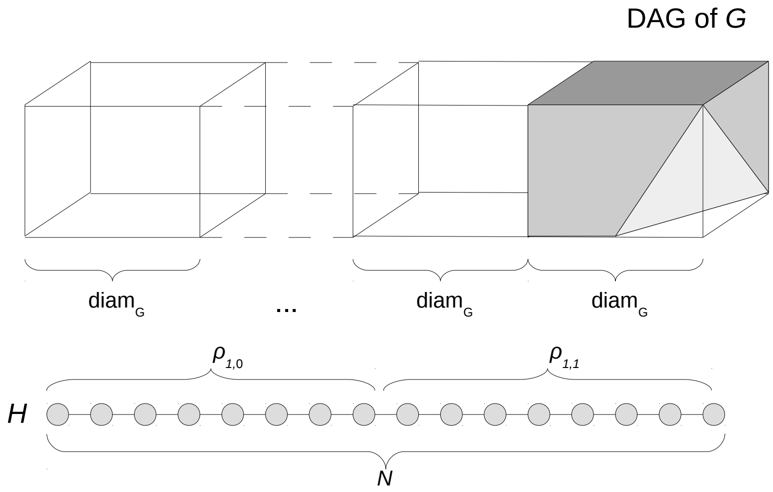

Figure 3.1: Sketches of the DAG of G and of the linear array H. DAG of G is a 2-dimensional version of DAG in Figure 2.9, where each section contains all nodes of G at a certain time of the computation. In gray the minimum volume of the DAG of G computed by $\rho_{1,0}$ if no communication is allowed.

must be computed by $\rho_{1,0}$; in particular, at least $(N - c)$ nodes $(u, T_G - \text{diam}_G)$.

In fact, without communication $\rho_{1,0}$ should compute all nodes $\{(u,t) : t \in [T_G - \sqrt{N}, T_G - \text{diam}_G]\}$ and with c_1 additional communication it can avoid the computation of at most $|c_1|$ nodes of a certain time step t preceding $T_G - \sqrt{N}$.

We can repeat this reasoning halving $\rho_{1,0}$ in $\rho_{2,0}$ and $\rho_{2,1}$, where at least one between them, without loss of generality $\rho_{2,0}$, computes the first evaluation (respect to subregion $\rho_{1,0}$) of at least $(N-c)/2$ nodes of $T_G - \text{diam}_G$. Due to the communication limit, at least $N - 2c$ nodes of $T_G - 2\text{diam}_G$ must be computed in $\rho_{2,0}$ (we don't know when exactly c_1 is imported, maybe it is imported with the set c_2 imported by $\rho_{2,0}$ in the computation of this block).

Again, $\rho_{2,0}$ can be split in $\rho_{3,0}$ and $\rho_{3,1}$, one of them must compute $(N-2c)/2$ nodes of $T_G - 2\text{diam}_G$ and at least $N - 2c$ nodes of $T_G - 3\text{diam}_G$ (a region can import at most $2c$ nodes since it has only 2 edges communication with the rest of the network).

Repeating $(\log N)/2$ this step, we eventually consider a $\sqrt{N}$-nodes region ρ which must compute $(N - 2c)/2 = \frac{N}{4}$ nodes of $t = \text{diam}_G$. Having $2c = N/2$ nodes imported during $[0, \text{diam}_G]$, ρ performs at least $W = \Theta(N\sqrt{N})$ during the $T_G = \text{diam}_G \log N$ step-long emulation. The reasoning highlights a work proportional to a block executed by a small subregion during the emulation:

$$S = \frac{W}{|\rho|T_G} = \frac{\Omega(N\sqrt{N})}{\sqrt{N}(\sqrt{N}\log N)} = \Omega\left(\frac{\sqrt{N}}{\log N}\right).$$

This reasoning proves the following theorem.

Theorem 10. *Let G be a two dimensional mesh of N nodes and H be a linear array of N nodes. Then any emulation of $T_G \geq diam_G \frac{\log N}{2}$ steps of G by H has slowdown $S = \Omega(\sqrt{N}/\log N)$.*

For computations of G longer than $\mathrm{diam}_G \frac{\log N}{2}$ steps, we just apply the theorem to blocks of $\mathrm{diam}_G \frac{\log N}{2}$ steps and sum the lower bounds obtained, similarly to Theorem 9.

Note that $\Theta(\log N)$ halving of H are needed to reach a region ρ polynomially smaller than the whole H. If the theorem would consider $x = o(\log N)$ block of diam_G steps, we would have a final ρ size $N/2^x > N^{1-\epsilon} \quad \forall \epsilon > 0$, while the work executed by ρ would remain the same $(\Theta(N\sqrt{N}))$.

The case with $N_H < N_G$ nodes is included in the generalization presented in next section.

3.3 Generalization to k-arrays over j-arrays

Let $G = (V_G, E_G)$ be a N_G-nodes k-dimensional array and $H = (V_H, E_H)$ an N_H-nodes j-dimensional array, $k > j, N_H \leq N_G$. We are going to adapt the strategy of Section 3.2 to the emulation of G in H. Again, D_G is the DAG of the computation of G and node (u, t) of D_G represents the computation of node u of G during the time step t of the computation.

If $N_H \leq N_G^{\frac{j}{k}}$ the emulation is work-preserving [KLM+97], so consider the case where H has $N_H = N_G^{(\frac{1}{k}+h)j}$ nodes, with $0 < h \leq \frac{k-j}{kj}$. Note that $\mathrm{diam}_H = jN_G^{\frac{1}{k}+h} - 1$ and its bisection bandwidth is $b_H \sim N_G^{(\frac{1}{k}+h)(j-1)}$. We consider $T_G \geq x\mathrm{diam}_G$, so that we have a sufficient number x of blocks of diam_G-steps to point out a region of H of $O(N_H/2^x)$ nodes, by halving the size of the considered region in each block when the communication limit is $c < N_G/(6j)$, similarly to Section 3.2.

The initial splitting of H is obtained removing the central edges from the first dimension. Let $u = (i_1, i_2, \dots, i_j), 0 \leq i_n < N_H^{1/j} \quad \forall 1 \leq n \leq j$, be nodes of H: edges involved in first splitting are those between nodes $(\lfloor (N_H^{1/j} - 1)/2 \rfloor, i_2, \dots, i_j)$ and $(\lceil N_H^{1/j}/2 \rceil, i_2, \dots, i_j), \forall i_n : n > 1$. Note that they are $N_H^{(j-1)/j}$ and the two regions $\rho_{1,0}$ and $\rho_{1,1}$, obtained removing the edges, have $N_H/2$ nodes if $N_H^{1/j}$ is even or $(N_H - N_H^{1/j})/2$ and $(N_H + N_H^{1/j})/2$ if $N_H^{1/j}$ is odd.

Consider the region, w.l.o.g. $\rho_{1,0}$, which first computes at least $N_G/2$ nodes (u, T_G). It must compute at least $N_G - c$ nodes $(u, T_G - \text{diam}_G)$ and it can be split again along the second dimension, removing edges between nodes $(i_1, \lfloor (N_G^{1/j} - 1)/2 \rfloor, i_3, \dots, i_j)$ and $(i_1, \lceil N_G^{1/j}/2 \rceil, i_3, \dots, i_j), \forall i_x : x \neq 2$.

In this case only about $N_H^{\frac{j-1}{j}}/2$ edges are involved (depending on the size of $\rho_{1,0}$) and again we focus on subregion $\rho_{2,0}$ which first computes at least $(N_G - c)/2$ nodes of $t = T_G - \text{diam}_G$ for $\rho_{1,0}$. It must compute at least $N_G - 2c$ nodes $(u, T_G - 2\text{diam}_G)$. After j similar steps, we have a region $\rho_{j,0}$ where all j dimensions are about $N_H^{1/j}/2$ and we continue this process starting again from the first dimension.

In each block the cut (edges removed) halves its size or it remains of the same size (when from dimension j we consider again dimension 1) so that $|cut| \leq N_H^{(j-1)/j} \quad \forall blocks$. Since ρs are j-arrays, at most $2j$ cuts participate to the importation of predecessors. The region ρ considered after x blocks has $O(N_H/2^x)$ nodes and it must compute $(N_G - c \cdot 2j)/2 > N_G/3$ nodes of time step $t = \text{diam}_G$, while the importation is of at most $c \cdot 2j < N_G/3$ nodes. This implies that ρ must compute $W = \Theta(N_G \text{diam}_G)$ nodes for steps $[0, \text{diam}_G]$ during the emulation, with a slowdown

$$S \geq \frac{W}{|\rho| T_G} = \frac{\Theta(N^{(k+1)/k})}{O(N_H/2^x) x \text{diam}_G} = \Omega\left(\frac{N_G 2^x}{N_H x}\right).$$

If more than $c = N_G/(6j)$ nodes pass through $N_H^{(j-1)/j}$ edges of H during the emulation, then

$$S \geq \frac{\frac{c}{|edges|}}{T_G} = \frac{N_G/6j}{N_H^{(j-1)/j} \cdot x \text{diam}_G}.$$

Asymptotically, the two lower bounds merge in:

$$\begin{aligned} S &= \Omega\left(\min\left\{\frac{N_G 2^x}{N_H x}, \frac{N_G}{N_H^{(j-1)/j}\cdot x\mathrm{diam}_G}\right\}\right) \\ &= \Omega\left(\min\left\{\frac{N_G 2^x}{N_G^{(\frac{1}{k}+h)j} x}, \frac{N_G}{N_G^{(\frac{1}{k}+h)(j-1)}\cdot x\mathrm{diam}_G}\right\}\right). \end{aligned} \tag{3.3}$$

Working on Equation 3.3, the lower bound is minimum when $2^x = N_G^h$, or $x = h\log N_G$, providing the lower bound

$$S = \Omega\left(\frac{N_G^{\frac{k-1}{k}}}{N_H^{\frac{j-1}{j}}\log N_G}\right).$$

This proves the following theorem who generalizes Theorem 10.

Theorem 11. *Let G be a k-dimensional array of N_G nodes and H be a j-dimensional array of $N_H = N_G^{(\frac{1}{k}+h)j}$ nodes, $0 \le h \le \frac{k-j}{kj}$. Then any emulation of $T_G \ge (h\log N_G)diam_G$ steps of G by H has slowdown*

$$S = \Omega\left(\frac{N_G^{\frac{k-1}{k}}}{N_H^{\frac{j-1}{j}}\log N_G}\right). \tag{3.4}$$

By Equation 2.6, we know that Theorem 3 provides

$$S = \Omega\left(\frac{N_G}{\left(N_G^{\frac{1}{k}}N_H\right)^{\frac{j}{j+1}}}\right),$$

which is asymptotic less tight than Theorem 11 $\forall h > 0$.

Note that in Theorem 11 for $h = 0$, $S = \Omega(N_G^{\frac{k-j}{k}})$, which is strict, while in general it differs only of a factor $1/\log N_G$ from the upper bound given by embedding of Proposition 10.

3.4 Considerations

The general ideas behind previous theorems are the following. The first part consider a certain amount of communication c which passes through a certain number of edges of H, b_H; this give an easy lower bound

$$S \geq \frac{c}{b_H T_G}.$$

Parameter b_H is an upper bound to the maximum number of edges we need to remove in a halving step of the second part of the theorem; in suitable networks, b_H is proportional to bisection bandwidth.

The second part considers a certain number x of blocks of diam_G steps in the computation of G. During the emulation of the last block, we split H in two subregions r_1 and r_2 approximatively of the same size, and we focus on the one which computes at least half of the outputs of the block. Also importing c nodes during this process, it must also compute great part of inputs of the block (at least $N_G - c$). The halving of the considered region of H keeps happening in each block of G DAG, and the nodes which can be imported are described by a function $f_G(c, x)$ depending on features of G, number of blocks x and the chosen quantity c. For example in j-arrays $f(x, c) = 2xc$ for $x < j$ and $f(x, c) = 2jc$ for $x \geq j$.

After x such blocks, we point out a small subregion of H (of about $N_H/2^x$ nodes) which must compute a number of nodes proportional to a block, $N_G\text{diam}_G$, during the emulation:

$$S = \frac{\Omega(N_G\text{diam}_G)}{\frac{N_H}{2^x}(x\text{diam}_G)}.$$

The lower bound of the theorem is given by taking the minimum of the two quantities.

$$S = \Omega\left(\min\left\{\frac{c}{b_H x\text{diam}_G}, \frac{N_G\text{diam}_G}{\frac{N_H}{2^x}x\text{diam}_G}\right\}\right). \tag{3.5}$$

From this parametric formula, we can see that both the terms suffer of a $1/x$ factor due to the number of block needed to select the subregion of H with an high computational load. Moreover value of c depends on features of G DAG.

The first fact suggest that Theorems 10 and 11 are not necessarily tight also of a factor $1/x$, which in particular for these theorems is $1/\log N$. The second fact suggests that the generalization of the theorem is unlikely to be tight with networks G with low bisection bandwidth, since in this case c must be small; e.g., consider the case $G =$ tree network and $H = j$-array, the first term of Equation 3.5 is trivially $\Omega(1)$.

Note that this theorem is quite different from Theorem 4 which provides a similar bound when recomputation is at most constant. Our theorem makes a trade-off between communication and computation: the less a region can import the more predecessors it will have to compute.

These consideration close the chapter, in the next one recomputation continues to have an important role, but this time it will be considered from the I/O complexity point of view.

Chapter 4

Storing-recomputation trade-offs

During a computation some intermediate results may be used more than once and we can choose if store them temporary in memory or compute them again when they are needed. Thus, according to the available memory, there is a trade-off between storing and recomputation of intermediate results.

This problem has several different but related applications in nowadays computations: in local computations, where the access to stored data could be slower than the recomputation of the needed result, in parallel computing, where importing data from a neighbor could be slower than the computation of the needed result, in energy-aware computations, where the cost of storing data could be greater than the cost of recomputing it. Note that in local computations recomputation occurs in subsequent times, while in parallel computations it can occurs contemporary in different places.

The topic has been studied for decades, from the already introduced [HK81], where the stress is on minimum number of I/O accesses in a framework considering recomputation, to nowadays works as [AM10, CLU12], where algorithms for automatic analysis of DAGs are proposed, in order to find optimal storing-recomputing strategies for minimizing computation costs in grid or cloud systems.

In the first section of this chapter we show some basic results about computations with and without recomputation, analyzing more in detail some important DAGs in the other sections. Arguments of this chapter have been developed during the visit period at ETH Zurich.

4.1 Basic facts

Let function $c_{IO}(\mathcal{S})$ count the number of I/O accesses occurring in strategy $\mathcal{S}$ and function $c_c(\mathcal{S})$ count the number of computational operations of $\mathcal{S}$.

Consider the pebbling strategy $\mathcal{S}_F^{R*}$ for a DAG F using recomputation where number of I/O accesses and computational operations is minimum, and the pebbling strategy $\mathcal{S}_F^{W*}$ for a DAG F with the same features of $\mathcal{S}_F^{R*}$ but where recomputation is not allowed. We define them respectively the *optimal strategy with recomputation* to pebble F and the *optimal strategy without recomputation* to pebble F.

A trivial fact is that $c_{IO}(\mathcal{S}_F^{R*}) \leq c_{IO}(\mathcal{S}_F^{W*})$, since $\mathcal{S}_F^{R*}$ uses recomputation only when is advantageous respect to $\mathcal{S}_F^{W*}$.

We recall the following definition in [BPD00], where access complexity of DAGs is studied from a space point of view.

Definition 11. A strategy $\mathcal{S}$ is a *parsimonious strategy* if outputs are pebbled exactly once and a pebbling of a node v is used to compute at least a son of v before another possible evaluation of v.

We similarly define a *parsimonious strategy with recomputation.*

Definition 12. A strategy $\mathcal{R}$ is a *parsimonious strategy with recomputation* if outputs are pebbled exactly once and a pebbling of a node v is used to compute at least a son of v before another possible recomputation of v.

Proposition 11. *The optimal strategy with recomputation $\mathcal{S}_F^{R*}$ is a parsimonious strategy with recomputation.*

Proof. Let $\mathcal{R} = \mathcal{S}_F^{R*}$. Suppose there is an output o_x computed at least twice. We can find a new schedule $\mathcal{R}'$ where o_x is computed just the first time. $\mathcal{R}'$ has less computational operations of $\mathcal{R}$, while it has at most the same I/O complexity, so $\mathcal{R}$ was not optimal, which is a contradiction.

If there is a node v recomputed, whose a particular pebbling v_y is not used before the following one, we can consider a new scheduling $\mathcal{R}''$ where v_y is not computed and v_{y+1} is used in place of v_y. $\mathcal{R}''$ has at most the same I/O complexity of $\mathcal{R}$ and less operations, obtaining again a contradiction with the hypothesis. □

As we'll see in Section 4.2 and 4.3, there exist both DAGs in which the optimal pebbling strategy does not use recomputation and DAGs where recomputation gives advantage respect to storing-based strategies.

4.2 Recomputation in tree DAGs

In this section we use the complete binary tree DAG to prove the existence of DAGs where recomputation does not give better results than strategies without recomputation. Moreover the result will be extended to the class of DAGs where each node has only a child.

Proposition 12. *Consider the complete binary tree DAG F where the root is the only output and leaves are the inputs. In the optimal strategy to pebble F every node is computed only once.*

Proof. Consider the optimal strategy with recomputation $\mathcal{S}$ to pebble F. By Section 4.1 it is a *parsimonious computation with recomputation*, that is the root is computed only once and a pebbling of a node v is used to compute at least a son of v before being possibly recomputed.

Since every node has only a child, we can show that there is no need to recompute any node. In fact the first time a node u is computed, it is used to compute its only child u', which must be used to compute its only child u'' before a possible recomputation and so on until we compute the root. If u is computed twice, we have to compute a second time also $u', u'', \ldots,$ the root, contradicting the fact that $\mathcal{S}$ is a *parsimonious computation.*

This means that in $\mathcal{S}$ every node is computed just once. □

In this case optimal strategy with recomputation for F does not really exploit recomputation and it coincides with optimal strategy without recomputation.

This fact can be generalized in the following proposition.

Proposition 13. *Consider a DAG F where each node has a single child. In the optimal strategy to pebble F all nodes are computed only once.*

Proof. F is a collection of disjoint trees not necessarily complete and with an arbitrary number of sons. With the same demonstration of Proposition 12 we can prove that computing twice a node implies computing twice an output, contradicting the fact that the strategy used is parsimonious. □

4.3 Recomputation in butterfly DAGs

In this section we prove that strategies with recomputation are more efficient that strategies without recomputation in the computation of butterfly DAGs.

A butterfly DAG F is a graph (V_F, E_F) where $V_F = \{(u_i, j) : 0 \leq i < N, 0 \leq j \leq \log N\}$ and, considering the bit representation of $u_i = u_i^{\log N-1} \ldots u_i^1 u_i^0$, each (u_i, j) has an edge to $(u_i, j+1)$ and to $(u_i^{\log N-1} \ldots u_i^{j+1}\overline{u_i^j}u_i^{j-1} \ldots u_i^0, j+1)$.

Nodes $L_j = \{(u_i, j), 0 \leq i < N\}$ are *level* j of F, while nodes $l_i = \{(u_i, j) : 0 \leq j \leq logN\}$ are *line* i of F. Level L_0 contains inputs of F, while $L_{\log N}$ contains outputs of F.

4.3.1 Two general lower bounds

Consider the pebbling of a N-inputs butterfly DAG F according to the Hong and Kung pebble game rules [HK81], using S red pebbles. We recall that in this model nodes of the DAG with a blue pebble can be thought as located in slow memory, while nodes with red pebbles can be thought as located in fast memory; the operation of pebbling with a red pebble a node with a blue pebble is equivalent to a load from slow to fast memory (input operation, or I) and the reverse operation, that is pebbling with a blue pebble a node with a red pebble, is equivalent to a writing from fast memory to slow memory (output operation, or O). In the following we exploit this analogy in our description. Note that since inputs are loaded from slow memory and outputs must be saved in slow memory, the minimum number of I/O accesses is at least $Q \geq 2N$.

A line is *covered* if at least one node of the line is in fast memory. When we pebble the first output of F at most $S-1$ lines are covered, so that at most $S-1$ lines can be computed and stored in slow memory with just one O operation, while the remaining $N - (S-1)$ lines need at least 2 I/O operations (an input or an intermediate node of the line have to be read again from slow memory and the output has to be saved in slow memory).

According to this, the following general lower bound holds:

$$Q \geq \underbrace{2N}_{\text{inputs and outputs}} + \underbrace{N-S+1}_{\substack{\text{lines uncovered}\\ \text{when computing } u}} = 3N - S + 1$$

This is a "warm-up" lower bound, presented to take confidence with this kind of reasonings. It can be strengthen as follow. We know that when first output is computed, at most $S-1$ lines are covered (set c) and their outputs can be computed without further memory accesses. Each of the remaining $N-S+1$ lines has at least to store an intermediate node of the line or it is uncovered. We partition these lines in those with nodes stored in secondary memory (set s) and uncovered lines (set u). If a line is uncovered it requires the load of some inputs (at least the one of the line) to be computed. With the proceeding of the computation its output can be computed without further memory accesses, an its intermediate node can be stored in slow memory or the line can become again uncovered before the computation of its output. When the first output of a line in u is computed, at most $S-1$ lines of u can be covered, so that lines which are still uncovered, require a further reading of their input to be computed or they have been stored in slow memory. We have distinguished three kind of line: $S-1$ line of set c which require at least two I/O operations, for the load of their input and the store of their output, at most $S-1$ lines of u which can possibly be computed with two reading of their input and the storing of their output (three I/O ops), and all the other lines which beyond the reading of the input and the storing of the output, require an intermediate storing or two other reading of their input (four I/O ops). These considerations imply the following lower bound:

$$Q \geq \underbrace{2(S-1)}_{\text{inputs and outputs of } c} + \underbrace{3(S-1)}_{\substack{\text{lines in } u \text{ which require}\\ \text{two reading of inputs}}} + \underbrace{4(N-2S+2)}_{\text{all the other lines}} = 4N - 3(S-1). \tag{4.1}$$

In this lower bound some nodes could be computed more than once; if recomputation is not allowed, then at least one pebble must stay on a line from when its node in second level is computed until an O operation occurs. The computation in that line will continue from that node only when it will be loaded again. Note that when two inputs are loaded to compute a node in the second level we can have three situations:

1. both sons are computed with that pebbles of inputs;
2. one son is computed, one parent is unload and then reloaded to compute the other son;
3. both parents are unload and reload before compute the other son.

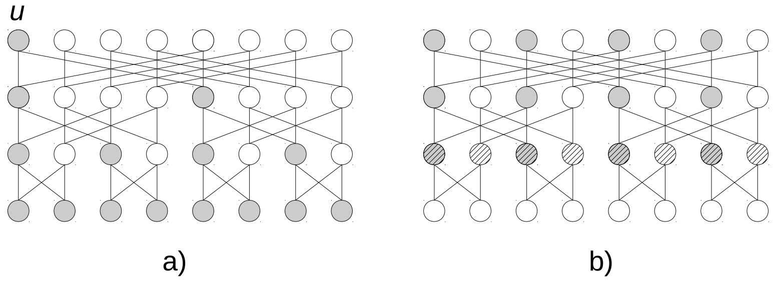

Figure 4.1: a) Tree of predecessors of u in F; b) Gray shaded nodes are inputs of sub-butterfly A (in gray), while white shaded nodes are inputs of sub-butterfly B (in white, excluding the input of the complete butterfly).

When pebbling the first output u, it means that the whole tree whose u is the root has been pebbled, in particular at least $N/2$ of the nodes of the second level of the butterfly (those on the tree whose u is the root, see Figure 4.1.a) and all the inputs.

Refer to sub-butterfly with inputs highlight in shaded gray in Figure 4.1.b as A and the one with shaded white inputs as B. Inputs of A and B are in the second level of F and two inputs of F are needed to compute the first node of a line of A or B. When computing u there are c_1 covered lines in A, while $x_1 = N/2 - c_1$ lines have an intermediate write in slow memory with a subsequent reload (once a node of a line is computed the first time, it can not be computed again, so the only way to avoid the recomputation is saving and reloading it; note that x_1 can be 0). During the computation of u, in sub-butterfly B there are c_2 covered lines and x_2 lines have been temporary saved in slow memory (each of them has already required an O operation and will require a I operation to continue its computation). Moreover, there are at least $N/2 - c_2 - x_2$ uncovered of B whose input has not yet been computed. These lines are of three kinds: t_2 of them have both inputs of F pebbled, i_2 have only one input pebbled and v_2 have no input pebbled (equations $c_2 + x_2 + v_2 + i_2 + t_2 = N/2$ holds). These lines require respectively at least zero, one and two I operations to start their computation. Sub-butterfly A does not need such distinction since all its inputs have already been computed.

Note that lines in B do not have inputs of F in common, moreover $c_1 + c_2 + i_2 + 2t_2 \geq S - 1$.

$$\begin{aligned}
Q &\geq 2N + 2(x_1 + x_2) + i_2 + 2v_2 \\
&= 2N + 2(x_1 + x_2) + i_2 + N - 2c_2 - 2x_2 - 2i_2 - 2t_2 \\
&= 3N + 2x_1 - i_2 - 2c_2 - 2t_2 \\
&= 3N + N - 2c_1 - i_2 - 2c_2 - 2t_2 \\
&= 4N - 2(c_1 + c_2) - i_2 - 2t_2
\end{aligned}$$

Since $2(c_1 + c_2 + i_2 + 2t_2) \geq 2c_1 + 2c_2 + i_2 + 2t_2$, also $-(2c_1 + 2c_2 + i_2 + 2t_2) \geq 2(S - 1)$ is true, and we obtain the lower bound

$$Q \geq 2N + 2(N - S + 1) = 4N - 2(S - 1) \tag{4.2}$$

which holds for computations with no recomputation in the second level of F.

4.3.2 Matching the lower bounds

We present two strategies, one with and one without recomputation, which almost match the previous lower bounds.

Strategy without recomputation If no recomputation is allowed, if the number of fast memory cells is $2^{\lceil \frac{\log N}{2} \rceil} + 2^{\lfloor \frac{\log N}{2} \rfloor} + 2 \leq S \leq N + 2$ we can consider the following strategy. The number S is such that a sub-butterfly of $2^{\lceil \frac{\log N}{2} \rceil}$-inputs can be computed, while at least $2^{\lfloor \frac{\log N}{2} \rfloor}$ nodes are kept in memory.

First consider the $2^{\lfloor \frac{\log N}{2} \rfloor}$ lower $2^{\lceil \frac{\log N}{2} \rceil}$-inputs sub-butterflies L_i of F (see Figure 4.2).

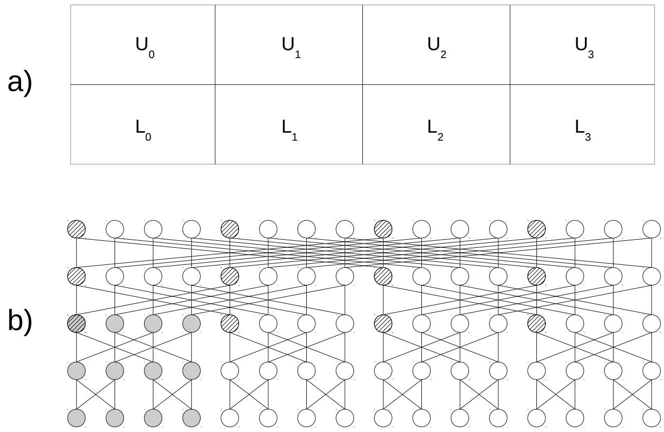

Figure 4.2: a) Division in L_i and U_i of a 16-inputs butterfly; in b) lower sub-butterfly L_0 (in gray) and upper sub-butterfly U_0 (shaded) are represented.

Each L_i is computed individually, requiring $2^{\lceil \frac{\log N}{2} \rceil}$ reading operations for the inputs. As for outputs, we save all of them but $(S - 2^{\lceil \frac{\log N}{2} \rceil} - 2)/2^{\lceil \frac{\log N}{2} \rceil}$, which are maintained in memory. In particular, inputs of upper sub-butterfly U_0 have the precedence in staying in memory, followed by U_1, U_2 and so on. Note that outputs of lower sub-butterflies are inputs for upper sub-butterflies and after this phase $N - S + 2$ outputs of the sub-butterflies L_i are stored temporary in memory, while $S - 2$ of them are already in memory.

Now we focus on the $2^{\lceil \frac{\log N}{2} \rceil}$ upper $2^{\lfloor \frac{\log N}{2} \rfloor}$-inputs sub-butterflies U_i of F. At least inputs of U_0 are in memory and we can compute it with no I/O computation but the storing of its outputs. With the $2^{\lfloor \frac{\log N}{2} \rfloor}$ pebbles freed after the computation of U_0 we can compute the other U_i, loading the inputs not yet in memory and saving the computed outputs.

This strategy requires $Q = 2N + 2(N - S + 2)$ I/O operation, differing of only 2 operations from the lower bound of Equation 4.2.

Strategy with recomputation We are presenting a pebbling strategy with recomputation using S red pebbles, which requires $\Omega(S)$ I/O accesses less than lower bound 4.2, valid for $\Omega(\sqrt{N}) < S < O(N)$.

Proposition 14. *Let F be a butterfly DAG F with N inputs, x, k_x and S_x be integers, $k_x = 2^x, 2 \leq x \leq \lfloor(\log N)/2\rfloor, S_x = \frac{N}{k_x} + (k_x - 1)\frac{N}{k_x^2} + k_x$. DAG F can be pebbled with a strategy with recomputation using $S = S_x + m, m < S_{x-1} - S_x$ pebbles which requires $\Omega(S)$ I/O accesses less than any strategy with recomputation.*

Proof. DAG F can be decomposed in k_x lower sub-butterflies $L_0, \dots, L_{k_x-1}$ with N/k_x inputs and N/k_x upper sub-butterflies $U_0, \dots, U_{N/k_x-1}$ with k_x inputs, where outputs of Ls are inputs of Us (in Figure 4.2 the case $N = 16, k = 4$ is represented).

A possible strategy $\mathcal{S}_1$ which satisfies the thesis consists in these steps:

1. Computation of L_{k_x-1}, maintaining its first N/k_x^2 outputs in fast memory, evicting all the others (N/k_x I operations).

2. Computation of $L_1, \dots, L_{k_x-2}$, maintaining in memory $(k_x-2)\frac{N}{k_x^2}+k_x-2+m$ outputs in memory (at least the first $\frac{N}{k_x^2} + 1$ inputs of each L_i, while the distribution of the remaining m is not important, saving the others in slow memory) ($(k-2)N/k_x$ I operations and $(k-2)N/k_x-(k-2)N/k_x^2-m-x_k+2$ O operations)

3. Computation of L_0 maintaining all input in memory (N/k_x I operations).

4. At this point there are only 2 red pebbles free and all inputs of $U_0, \dots, U_{N/k^2-1}$ are in memory, so that we can compute these U_is without further memory accesses except those needed for saving output of U_0 in slow memory. This phase performs N/k_x O operations and after it $N/k_x + 2$ red pebbles are free.

5. Computation of L_{k_x-1}, maintaining all outputs in memory (N/k_x I operations)

6. All inputs of U_{N/k_x^2} are in memory, so we can compute its outputs and save them in slow memory, freeing k_x red pebbles (k_x O operations).

7. Inputs of the remaining Us can be or not in memory, in any case we have sufficient memory to load all inputs of a U at once, computing it and storing in slow memory its outputs, for a total of $(k-2)N/k_x - (k-2)N/k_x - m - x_k + 2$ I operations (corresponding to the O operations of point 2) and $k_x(N/k_x - N/(k_x^2+1))$ O operations.

The number of I/O operations required by the strategy with $S = S_x + m$, $S_x \leq S < S_{x+1}$ pebbles is

$$\begin{aligned}\mathcal{Q}_S &= \underbrace{2N}_{\text{inputs and outputs}} + \underbrace{\frac{N}{k_x}}_{\substack{\text{second reading of}\\ \text{inputs of } L_{k_x-1}}} \underbrace{+2\left((k_x-2)\frac{N}{k_x} - (k_x-2)\left(\frac{N}{k_x^2}+1\right) - m\right)}_{\substack{\text{storing and second reading of}\\ \text{outputs of } L_1,\ldots,L_{k_x-2}\\ \text{not maintained in memory}}}\\ &= 4N - 5\frac{N}{k_x} + 4\frac{N}{k_x^2} - 2(k-2+m)\end{aligned}$$

By lower bound 4.2, strategies without recomputation require at least

$$Q_{NR(S)} \geq 2N - 2(N-S+1) = 4N - 4\frac{N}{k_x} + 2\frac{N}{k_x^2} - 2(k+m-1)$$

giving a difference of

$$\Delta Q = Q_{NR(S)} - \mathcal{Q}_S = \frac{N}{k_x} - 2\frac{N}{k_x^2} + 2.$$

Since by hypotesis $\frac{N}{k_x} + (k_x-1)\frac{N}{k_x^2} + k_x \leq S < \frac{N}{k_{x-1}} + (k_{x-1}-1)\frac{N}{k_{x-1}^2} + k_{x-1}$, $x \geq 2$ and $k_x \leq \sqrt{N}$, ΔQ is about in the range $[S/10, S/2]$, proving the thesis. □

This I/O saving is not the best possible: suppose $S = N/k_x + (k-1)yN/k_x^2 + (k_x - y - 1) + m + 2$, $m \geq 0$ stategy $\mathcal{S}_1$ can be improved in the following way.

1. Computation of $L_{k_x-y}, \ldots, L_{k-1}$, maintaining its first yN/k_x^2 outputs in fast memory, evicting all the others.

2. Computation of $L_1, \ldots, L_{k_x-y-1}$, maintaining in memory $(k_x - y - 1)(\frac{N}{k_x^2} + 1) + m$ outputs in memory (at least the first $y\frac{N}{k_x^2} + 1$ inputs of each L_i, while

the distribution of the remaining m is not important, saving the others in slow memory).

3. Computation of L_0 maintaining all input in memory.

4. At this point there are only 2 red pebbles free and all inputs of $U_0, \ldots, U_{yN/k^2-1}$ are in memory, so that we can compute these U_is without further memory accesses except those needed for saving output of U_0 in slow memory. This phase performs yN/k_x O operations and after it $yN/k_x + 2$ red pebbles are free.

5. Computation of $L_{k_x-y}, \ldots, L_{k-1}$, maintaining all outputs in memory.

6. All inputs of U_{yN/k_x^2} are in memory, so we can compute its outputs and save them in slow memory, freeing k_x red pebbles.

7. Inputs of the remaining Us can be or not in memory, in any case we have sufficient memory to load all inputs of a U at once, computing it and storing in slow memory its outputs.

The I/O accesses required by the strategy are

$$\begin{aligned}
\mathcal{Q}_S &= \underbrace{2N}_{\text{inputs and outputs}} + \underbrace{\frac{yN}{k_x}}_{\substack{\text{second reading of} \\ \text{inputs of } L_{k_x-y}, \ldots, L_{k-1}}} \underbrace{+2\left((k_x-y-1)\left(\frac{N}{k_x}-\frac{yN}{k_x^2}-1\right)-m\right)}_{\substack{\text{storing and second reading of} \\ \text{outputs of } L_1, \ldots, L_{k_x-y-1} \\ \text{not maintained in memory}}} \\
&= 4N-(3y+2)\frac{N}{k_x}+2\frac{y(y+1)N}{k_x^2}-2(k-y-1+m)
\end{aligned}$$

while the lower bound for strategies without recomputation is

$$\begin{aligned}
Q_{NR(S)} &= 2N+2\left(N-\left(\frac{N}{k_x}+(k-1)y\frac{N}{k_x^2}+k_x-y-1+m\right)\right) \\
&= 4N-(2y+2)\frac{N}{k_x}+2\frac{yN}{k_x^2}-2(k_x-y-1+m).
\end{aligned}$$

The difference is

$$\Delta Q = Q_{NR(S)} - \mathcal{Q}_S = y\frac{N}{k_x} - 2\frac{y^2 N}{k_x^2},$$

which in the case $N >> k_x >> y >> 1$ leads to $\Delta Q = \frac{y}{y+1}S$; this result is better of the one in Proposition 14 of a constant factor, but asymptotically has the same valence. This result almost much lower bound of Equation 4.1, except for a $S/(y+1)$ term.

Results of this section can be summarized as follows.

Proposition 15. *The pebbling of an N-inputs butterfly DAG with S red pebbles according to rules in [HK81] requires at least*

$$Q \geq 4N - 3(S-1)$$

memory accesses, and the lower bound can be refined to

$$Q \geq 4N - 2(S-1)$$

if no recomputation is allowed.

In particular these bounds are almost strict in the range $\Omega(\sqrt{N}) < S < N$ and the first one is almost matched (gap of $S/(y+1)$ I/O accesses) by strategy of Preposition 14, while the second one is matched (except for 2 I/O accesses) by the first strategy of Subsection 4.3.2.

In [EPR+13] there is an enhancement of pebble game thought for the case without recomputation, which allows to show the following theorem.

Theorem 12. *For the n-point FFT graph, the minimum I/O cost, Q, satisfies $Q \geq \frac{2n\log n}{logS}(1 - \epsilon_{n,S})$, where S is the number of red pebbles, and $\epsilon_{n,s}$ tends to 0 for large values of n, S and $\frac{n}{S}$.*

Our analysis inspects the range $\Omega(\sqrt{N}) < S < N$, where we show a separation between performance reachable with and without recomputation. The results suggest that a bound similar to Theorem 12 holds also for the case with recomputation, probably with a $\epsilon'_{n,s} > \epsilon_{n,s}$, but further studies in the range $1 \leq S < O(\sqrt{N})$ are required.

4.4 Recomputation in butterfly-like reduction DAGs

We introduce a class of DAGs where I/O complexity can take advantage from recomputation exploiting strategies similar to previous section. The results contained in this section are still incomplete, so the concepts and the proofs will be just sketched.

Definition 13 (Butterfly-like reduction DAG)**.** A (k, j, L)-*blr (butterfly-like reduction) DAG* is a DAG with the following construction:

- a $(k, j, 1)$-blr DAG as k inputs and j outputs. Each input is connected to each output.
- a (k, j, L)-blr DAG, $L > 1$, is constructed joining k $(k, j, L-1)$-blr DAGs $D_0, \ldots, D_{k-1}$ with an additional level as follows: fixed x, the set of outputs $(L-1, x)$ of all D_i are inputs for nodes $\{(L, xj^{L-1}) : 0 \leq x < j\}$ of the additional level.

Note the following features of the defined DAG:

- it has $L+1$ levels labeled from 0 to L; level 0 contains inputs of the network, while level L its outputs.
- nodes of level i have only input nodes from the level $i-1$
- nodes of level i are input only for nodes of the level $i+1$
- level i has $k^{L-i}j^i$ nodes, numbered from $(i, 0)$ to $(i, k^{L-i}j^i - 1)$

Moreover given suitable parameters k, j and L in (k, j, L)-blr DAG recomputation can improve I/O complexity respect to the case without recomputation (e.g. $k = j = 2$, the butterfly case), or can be not useful (e.g., $j = 1, \forall k$, k-ary trees case).

In Figure 4.3 there are some examples of blr DAG for small values of k and j.

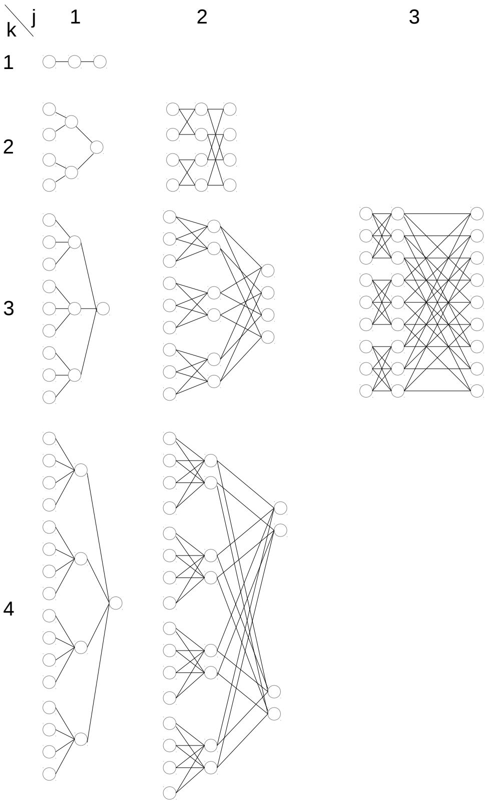

Figure 4.3: Examples of blr DAG, varying k and j, when $L = 2$.

Note that the $(2, 2, L)$-blr DAG is a butterfly with L levels, while a $(k, 1, L)$-blr DAG is a complete k-ary tree of height L.

Note also that (k, j, L)-blr DAG requires $Q \geq k^L + j^L$ I/O accesses in slow memory (to read inputs and to store outputs) and at least $S \geq k+1$ red pebbles are needed. Another interesting fact about these DAGs is that they can be pebbled without intermediate storings if S is proportional to the number of outputs (when $j > 1$) or to the depth of the DAGs (for trees).

Proposition 16. *DAG F, a (k, j, L)-blr DAG, can be pebbled in $Q = k^L + j^L$ I/O operations if $S \geq (k-1)\frac{j^L-1}{j-1} - 1 + k + j$ for $j > 1$ or $S \geq (k-1)L + 2$ if $j = 1$.*

Proof. Let indicate with $S_{k,j,L}$ the number of red pebbles to pebble F without intermediate node storings, with the additional property that outputs remain in fast memory once they have been computed. We establish a recurrence equation for $S_{k,j,L}$ which prove the proposition.

If $L = 1$ and $S_{k,j,1} = k + j$, we have trivially $Q = k + j$ since we must load all inputs and store all output after they have been computed. Note that all outputs can stay in fast memory after their computation.

We recall that a (k, j, L)-blr DAG consists in k $(k, j, L-1)$-blr DAGs and an additional level. With $S_{k,j,L-1}$ red pebbles a $(k, j, L-1)$-blr DAG can be pebbled with all its outputs in fast memory at the end of the process, so if we consider $S_{k,j,L} = (k-1)j^{L-1} + S_{k,j,L-1}$ we can compute all k the $(k, j, L-1)$-blr DAGs having their outputs in fast memory and at least $S_{k,j,L} - kj^{L-1} \geq 1$ available red pebbles to easily complete the last level of F. The process only requires the load of inputs and the store of outputs, then

$$Q = k^L + j^L$$

and

$$\begin{cases} S_{k,j,1} = k + j \\ S_{k,j,L} = (k-1)j^{L-1} + S_{k,j,L-1} \end{cases} \Rightarrow S = (k-1)\left(\sum_{i=2}^{L} j^{i-1}\right) + k + j$$

which closes the proof. □

In Section 4.3 we exploit the fact that in a butterfly DAG a lower sub-butterfly A of $\sqrt{N}$ inputs also has $\sqrt{N}$ outputs: in opportune strategies the computation of few outputs of A the first time that we read its inputs, with a consequent second reading of inputs, is advantageous respect to store and reload all the outputs of A. In particular, in the first case the computation needs $\sqrt{2N}$ input operations due to two loads of inputs of A, while the second case needs $\sqrt{N} + 2(\sqrt{N} - x)$, where x is the number of outputs of A that do not need temporary storing. Recomputation strategy pays off when $2\sqrt{N} < \sqrt{N} + 2(\sqrt{N} - x) \Rightarrow x < \sqrt{N}/2$, which means that the available memory must constrain the strategy based on storing to store and reload more than half outputs per sub-butterfly.

In (k, j)-blr DAGs when $j < k$ outputs are less than inputs, and by Proposition 16 we need approximately kj^{L-1} fast memory cells (the number of nodes of the penultimate level) to compute without intermediate storings the whole DAG. In these DAGs is more difficult to exploit recomputation to reduce I/O complexity; in fact we already know that in some cases it is not possible (e.g., for trees). If we try to apply the same reasoning of previous paragraph to a subregion of a (k, j)-blr DAG with i inputs and o outputs, recomputation requires $2i$ input accesses, while strategies without recomputation needs $i + 2(o - x)$ I/O accesses to read once the inputs and store the $o - x$ outputs, if x outputs can be used without intermediate storing. In this case the gain for strategies with recomputation occurs when $x < 2o - i$.

Since when $L = 1$ inputs and outputs are respectively $i = k$ and $o = j$, this inequality let us suppose that when $k \geq 2j$ the existence of strategies with recomputation more effective than strategies with only I/O accesses is unlikely. Probably the second term is even more near to k if we consider that the number of outputs decreases exponentially in L respect to inputs. Since when $k = j$ only S I/O accesses are saved, it would not be surprising if recomputation does not improve performance for any $j < k$.

On the other hand when $k < j$ recomputation has an easy task. For example consider the case $k = 1, j > 1$, which corresponds to a reverse j-tree, with the root as input and leaves as outputs. In this case $S = 3$ would suffices to compute all the DAG with the optimal 1+*number of leaves* I/O complexity, but it would involve an exponential computational complexity. This decreases very fast and $S = \log N$ is sufficient to require a linear work on the size of the DAG with no recomputation and optimal I/O complexity.

Note that when $k > j$ there is a sort of compression of the information, we need a lot of inputs to compute few outputs and in general it is more efficient to store few intermediate nodes instead of reload several inputs. On the contrary when $k < j$ few nodes allow to compute a lot of data and in general it is more convenient to load few inputs than store a lot of outputs.

Chapter 5

Conclusions

5.1 Summary and contributions

This thesis has pursued two distinct directions. The former is a critic overview of known results about lower bound for the emulation time among fixed-connected networks, with our original results for the topic. The latter considers the trade-off between recomputation and storing of intermediate results in a computation.

As for the first, we analyze theorems from [KLM+97, KR94], which are the best results known lower bounds for emulations, valid under the most general conditions; in particular they consider emulations where recomputation is allowed. In Chapter 2 we underline their features, weaknesses and the mutual relationships, showing actually there is not a theorem summarizing the others and also considering the tightest bound obtained by the three theorems we are not sure to obtain a strict lower bound. Theorem 1 has been extended to target general DAGs, and its results are tight in examples as tree DAG and butterfly DAG on k-array. We have also shown that theorems providing lower bounds for I/O complexity in hierarchical memory can not be easily adapted to the parallel case since the features of subsets in which we can decompose the computation are very different. In Chapter 3 we propose a new theorem to determine time lower bounds for emulation among multidimensional arrays, admitting arbitrary recomputation; our technique match the upper bound determined by embedding emulation unless for a logarithmic factor (improving the polynomial gap of Theorems 1 and 3, while Theorem 4 considers only at most constant average recomputation).

In the second part, in Chapter 4 we discuss about the trade-off between recomputation and storing of the intermediate results, in an offset similar to [HK81], and the main result is the demonstration that in N-input the butterfly DAG the use of recomputation gives strictly better performance than the case without recomputation, but only about S accesses are saved when $\Omega(\sqrt{N}) < S < N$. We also define (k, j)-blr DAGs to show how in DAGs which tend to "compress" data the exploitation of recomputation is difficult, while in DAGs which generate a lot of data from few inputs recomputation is advantageous from an I/O complexity point of view.

5.2 Further research

The first part of the thesis presents various strong and weak points of known lower bound theorems for network emulations and algorithm execution; it would be a great progress in the understanding of parallel computation to join all the highlighted elements, possibly managing the known results in an unique theory.

As for the second part, it would be interesting to find strict upper and lower bounds for butterfly DAG for S in the range $1 \leq S < O(\sqrt{N})$ to confirm or reject the hypothesis of a strict lower bound similar both in case of recomputation allowed or disallowed (end of Section 4.3). Moreover Section 4.4 lacks of exact lower and upper bounds for $k \neq j$. It would be interesting also the study of a more general framework to provide bounds when recomputation is or is not allowed in a general DAG, considering intrinsic features of it.

Bibliography

[A67] Amdahl, G.: Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. *AFIPS Conference Proceedings*, 30, 483–485, 1967.

[AACS87] Aggarwal, A., Alpern, B., Chandra, A. K., and Snir, M.: A model for hierarchical memory. In *Proceedings of the 19th ACM Symposium on Theory of Computing.* ACM, New York, 305–314, 1987.

[ABK95] Adler, M., Byers, J. W., Karp, R. M.: Parallel sorting with limited bandwidth, *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, p.129-136, June 24-26, 1995, Santa Barbara, California, USA.

[ACS87] Aggarwal, A., Chandra, A. K., and Snir, M.: Hierarchical memory with block transfer. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science.* IEEE Computer Society Press, Los Alamitos, 204–216, 1987.

[ACS90] Aggarwal, A., Chandra, A.K., and Snir, M.: Communication complexity of PRAMs. *Theoretical Computer Science*, 71:328, 1990.

[AM10] Adams, I. F., Madden, B. A.: Automating Analysis of the Computation-Storage Tradeoff, *Thesis.* UC Santa Cruz, 2010.

[AV88] Aggarwal, A. and Vitter, J. S.: The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[B74] Brent, R. P.: The Parallel Evaluation of General Arithmetic Expressions. *Journal of the ACM 21*, 2, 201–206, April 1974.

[BEP09] Bilardi, G., Ekanadham, K., Pattnaik, P.: On approximating the ideal random access machine by physical machines, *J. ACM*, 56(5), August 2009, 27:1–27:57, ISSN 0004-5411.

[BHP+96] Bilardi, Herley, K. T., Pietracaprina, A., Pucci, G., Spirakis, P.; BSP vs LogP, *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, p.25-32, June 24-26, 1996, Padua, Italy.

[BP01] Bilardi, G., Peserico, E.: A Characterization of Temporal Locality and Its Portability across Memory Hierarchies. *ICALP 2001*: 128–139, 2001.

[BP95] Bilardi, G., and Preparata, F.: Horizons of parallel computation. *J. Parall. Distrib. Comput.*, vol. 27, n. 2, pp. 172–182, 1995.

[BP99] Bilardi, G., and Preparata, F.: Processor–Time Tradeoffs under Bounded–Speed Message Propagation: Part II, Lower Bounds. *Theory Comput. Syst.* 32(5): 531–559, 1999.

[BPD00] Bilardi, G., Pietracaprina, A. and D'Alberto, P.: On the space and access complexity of computation dags. In *Proc. 26th International Workshop on Graph-Theoretic Concepts in Computer Science*, WG 2000, LNCS 1928, 47–58, June 2000.

[BPP07] Bilardi, G., Pietracaprina, A. and Pucci, G.: Decomposable BSP: A Bandwidth-Latency Model for Parallel and Hierarchical Computation, in *Handbook of Parallel Computing* (J. Reif and S. Rajasekaran Eds.), CRC Press, Boca Raton Fl USA, 2007.

[BSS12] Bilardi, G., Scquizzato, M. and Silvestri, S.: A Lower Bound Technique for Communication on BSP with Application to the FFT, *Euro-Par 2012*: 676-687.

[CD82] Cook, S. A. and Dwork, C.: Bounds on the Time for Parallel RAM's to Compute Simple Functions. STOC 1982: 231-233

[CGG+95] Chiang, Y., Goodrich, M. T., Grove, E. F., Tamassia, R., Vengroff, D. E., Vitter, J. S.: External-memory graph algorithms, *Proceedings of* *the sixth annual ACM-SIAM symposium on Discrete algorithms*, SODA 95, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1995, ISBN 0-89871-349-8.

[CKP+96] Culler, D.E., Karp, R., Patterson, D., Sahay, A., Santos, E., Schauser, K.E., Subramonian, R., and Eicken, T.V.: LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):7885, November 1996.

[CLU12] Cole-Mullen, H., Lyons, A., Utke, J.: Storing Versus Recomputation on Multiple DAGs, in *Recent Advances in Algorithmic Differentiation*, Lecture Notes in Computational Science and Engineering Volume 87, pp 197–207, 2012.

[CR73] Cook, S. A., Reckhow, R. A.: Time Bounded Random Access Machines. *J. Comput. Syst. Sci.*, vol. 7, n. 4, pp. 354–375, 1973.

[CLR01] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.: *Introduction to Algorithms*, second edition. MIT Press, 2001.

[DK96] P. De la Torre and C.P. Kruskal. Submachine locality in the bulk synchronous setting. In *Proc. of EUROPAR 96*, LNCS 1124, pp. 352–358, August 1996.

[EPR+13] Elango, V., Pouchet, L. N., Ramanujam, Rastello, F., J. and Sadayappan, P.: Data access complexity: The red/blue pebble game revisited, *Technical report*, OSU/INRIA/LSU/UCLA, Sept. 2013. OSU-CISRC-7/13-TR16

[EPR+15] Elango, V., Pouchet, L. N., Ramanujam, Rastello, F., J. and Sadayappan, P.: On Characterizing the Data Access Complexity of Programs, in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pp. 567–580, 2015. 1

[FF82] Fishburn, J. P., and Finkel, R. A.: Quotient networks, *IEEE Trans. Comput.*, C-31, 4 (Apr.), 288–295, 1982.

[FPP06] Fantozzi, C., Pietracaprina, A., Pucci, G.: Translating submachine locality into locality of reference, *J. Parallel Distrib. Comput.*, 66(5), May 2006, 633–646, ISSN 0743-7315.

[FW78] Fortune, S., and Wyllie, J.: Parallelism in Random Access Machines. In *Proceeding STOC '78 Proceedings of the tenth annual ACM symposium on Theory of computing*, pp. 114–118, 1978.

[G78] Goldschlager, L. M.: A Unified Approach to models of Synchronous Parallel Machines. In *Proceeding STOC '78 Proceedings of the tenth annual ACM symposium on Theory of computing*, pp. 89–94, 1978.

[GH91] Gupta, A., K. and Hambrusch, S., E.: Embedding Complete Binary Trees into Butterfly Networks, *IEEE Transactions on Computers*, vol. 40, pp. 853–863, 1991.

[HK81] Hong, J., and Kung, H. T.: I/O complexity: The red-blue pebble game. In *Proceedings of the 13th ACM Symposium on Theory of Computing*, pp. 326–333, New York, NY, USA, 1981. ACM.

[HKMU91] Heckmann, R., Klasing, R., Monien, B., Unger, W.: Optimal Embedding of Complete Binary Trees into Lines and Grids, *Proc. 17th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, WG91, 1991.

[HWV77] Hopcroft, J. E., Wolfgang, J. P., Valiant, L. G.: On Time Versus Space. *J. ACM* 24(2): 332–337, 1977.

[Jaja92] Jájá, J. F.: An introduction to parallel algorithms. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 1992. ISBN:0-201-54856-9.

[K70] Kerr, L. R.: *The Effect of Algebraic Structure on the Computational Complexity of Matrix Multiplication*, Ph.D. Thesis, Cornell University, New York, 1970.

[K83] Kruskal, C. P.: Searching, Merging, and Sorting in Parallel Computation. *IEEE Trans. Computers* 32(10): 942–946, 1983.

[KR94] Kruskal, C. P. and Rappoport, K. J., Bandwidth-based Lower Bounds on Slowdown for Efficient Emulations of Fixed-connection Networks, *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '94, 132–139, ISBN 0-89791-671-9, 1994.

[KLM+97] Koch, R. R., Leighton, F. T., Maggs, B. M., Rao, S., Rosenberg, A. L., Schwabe, E. J.: Work-preserving emulations of fixed-connection networks. *J. ACM* 44(1): 104–147, 1997.

[LMR88] Leighton, T., Maggs, B., Rao, S.: Universal packet routing algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science* (Oct.). IEEE, New York, pp. 256–271, 1988.

[LP93] Luccio, F., Pagli, L.: A Model of Sequential Computation with Pipelines Access to Memory, *Mathematical Systems Theory*, 26(4), 1993, 343–356.

[M65] Moore, G. E.: Cramming more components onto integrated circuits. *Electronics Magazine*, p. 4, 1965.

[M83] Meyer auf der Heide, F.: Efficiency of Universal Parallel Computers, *Acta Inf.* 3(19), 269–296, 10.1007/BF00265559, Springer-Verlag, 1983.

[M86] Meyer auf der Heide, F.: Efficient Simulations Among Several Models of Parallel Computers, *SIAM J. Comput.*, 15(1) , 106–119, 1986.

[MZ12] Milani, E. and Zago, N.: Exploiting Fine Grained Parallelism on the SPE. *ICTCS*, 2012.

[PKK+04] Parikh, A., Kim, S., Kandemir, M. T., Vijaykrishnan, N. and Irwin M. J.: Instruction Scheduling for Low Power. In *Journla of VLSI Signal Processing* 37(1): 129–149, 2004.

[PPS06] Pietracaprina, A., Pucci, G., Silvestri, F.: Cache-oblivious simulation of parallel programs, *20th International Parallel and Distributed Processing Symposium* (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece, IEEE, 2006.

[PSZ+02] Parikh, D., Skadron, K., Zhang, Y., Barcella, M., and Stan. M.: Power Issues Related to Branch Prediction. In *Proc. of the 2002 International Symposium on High-Performance Computer Architecture*, February, 2002, Cambridge, MA.

[PU87] Papadimitriou, C. H. and Ullman, J. D.: A Communication-Time Trade-off. *SIAM J. Comput.* 16(4): 639–646, 1987.

[S95] Savage, J.: Extending the Hong-Kung Model to Memory Hierarchies. In *Computing and Combinatorics*, v. 959 of *LNCS*, pp. 270–281. 1995.

[SS14] Scquizzato, M. and Silvestri, S.: Communication Lower Bounds for Distributed-Memory Computations. *STACS 2014*: 627–638.

[T36] Turing, A.M.: On Computable Numbers, with an Application to the Entscheidungs problem, *Proceedings of the London Mathematical Society*, 2 (42): 230–265, 1937.

[V90] Valiant, L.G.: A bridging model for parallel computation. *Communications of the ACM*, 33(8):103111, August 1990.

[V98] Vitter, J. S.: External Memory Algorithms, *Algorithms - ESA 98, 6th Annual European Symposium*, Venice, Italy, August 24-26, 1998, Proceedings (G. Bilardi, G. F. Italiano, A. Pietracaprina, G. Pucci, Eds.), 1461, Springer, 1998, ISBN 3-540-64848-8.

[VW85] Vishkin, U. and Wigderson, A.: Trade-Offs between Depth and Width in Parallel Computation, *SIAM Journal of Computing*, 14(2): pp. 303–314, 1985.