ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΕΦΑΡΜΟΣΜΕΝΩΝ ΜΑΘΗΜΑΤΙΚΩΝ ΚΑΙ
ΦΥΣΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

Παράλληλοι Υπολογισμοί για το Πρόβλημα των Αλγεβρικών Μονοπατιών

Μεταπτυχιακή Εργασία

Χριστίνας Καρουσάτου

Επιβλέπων:

Αριστείδης Παγουρτζής
Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2013

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΕΦΑΡΜΟΣΜΕΝΩΝ ΜΑΘΗΜΑΤΙΚΩΝ ΚΑΙ
ΦΥΣΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

Παράλληλοι Υπολογισμοί για το Πρόβλημα των Αλγεβρικών Μονοπατιών

Μεταπτυχιακή Εργασία

Χριστίνας Καρουσάτου

Επιβλέπων:
Αριστείδης Παγουρτζής
Επίκουρος Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή, 6 Μαρτίου 2013

....	....	....
Ε. Ζάχος	Α. Παγουρτζής	Δ. Φωτάκης
Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ε.Μ.Π.	Λέκτορας Ε.Μ.Π.

Περίληψη

Η παρούσα μεταπτυχιακή εργασία εκπονήθηκε στα πλαίσια του διατμηματικού προγράμματος μεταπτυχιακών σπουδών "Εφαρμοσμένες Μαθηματικές Επιστήμες" του Εθνικού Μετσόβιου Πολυτεχνείου.

Το πρόβλημα των αλγεβρικών μονοπατιών αντιπροσωπεύει μία γενικευμένη κλάση προβλημάτων. Στην κλάση αυτή υπάγονται αρκέτα προβλήματα από διαφορετικούς τομείς της θεωρητικής πληροφορικής, για παράδειγμα προβλήματα που σχετίζονται με την θεωρία γραφημάτων, με δίκτυα επικοινωνίας, προβλήματα σχετικά με πίνακες και τέλος προβλήματα που αφόρουν σε κανονικές γλώσσες. Επομένως, η εύρεση ενός αποδοτικού αλγορίθμου για την λύση του γενικού προβλήματος, θα αποτελούσε άμεσα μία αποδοτική λύση και για όλες τις κατηγορίες προβλημάτων που υπάγονται σε αυτό. Κάτι τέτοιο, βέβαια, δεν καθιστά ανώφελη την μελέτη των ειδικών κατηγοριών, κάθως, όπως θα δούμε στην συνέχεια, εξαιτίας των επιπλέον ιδιοτήτων που έχουν κάποια υποπροβλήματα μπορούμε να έχουμε αποδοτικότερους αλγόριθμους για τα προβλήματα αυτά. Στην εργασία αυτή αρχικά παρουσιάζονται σειριακοί αλγόριθμοι για το πρόβλημα των αλγεβρικών μονοπατιών και για συγκεκριμένα υποπροβλήματά του. Στην συνέχεια παρουσιάζονται παράλληλοι αλγόριθμοι για τα προβλήματα αυτά.

Η θεωρητική μελέτη παράλληλων υπολογισμών ξεκίνησε την δεκαετία του '70. Η τεράστια υπολογιστική ισχύς που μπορούμε να αποκτήσουμε μέσω των παράλληλων συστημάτων αποτελεί το ουσιαστικό κίνητρο για την διερεύνηση του πεδίου αυτού. Κατά την διάρκεια όλων αυτών των χρόνων διάφορα μοντέλα παράλληλων υπολογιστών έχουν προτάθει, κανένα όμως δεν κατάφερε να γίνει ευρέως απόδεκτο ως κυρίαρχο, όπως έχει συμβεί με το μοντέλο του von Neumann στους σειριακούς υπολογιστές. Το μοντέλο βάσει του οποίου μελετάμε και συγκρίνουμε τους παράλληλους αλγοριθμούς είναι ένα αρκετά διαδεδομένο μοντέλο, το BSP. Το μοντέλο αυτό είναι αρκέτα ρεαλιστικό και ταυτόχρονα επαρκώς απλό, με αποτέλεσμα η κατασκευή και η ανάλυση αλγορίθμων βασιζόμενων στο μοντέλο αυτό να είναι απλή και ευνόητη.

Ευχαριστίες

Ολοκληρώνοντας αυτήν την μεταπτυχιακή εργασία θα ήθελα να ευχαριστήσω αρχικά τον επιβλέποντα καθηγητή της εργασίας κ. Άρη Παγουρτζή, για την εμπιστοσύνη που έδειξε κατά την ανάθεσή της καθώς και για την καθοδήγηση που μου παρείχε καθ'όλη την διάρκεια εκπόνησής της.

Θα ήθελα, επίσης, να ευχαριστήσω τον κ. Ευστάθιο Ζάχο και τον κ. Δημήτρη Φωτάκη που μου έδωσαν την ευκαιρία να συνεργαστώ μαζί τους και να ενταχτώ στο εργαστήριο Λογικής και Υπολογισμών.

Ευχαριστώ όλα τα παιδιά του εργαστηρίου και κυρίως την Ματούλα Πετρόλια και τον Δημήτρη Σακαβάλα για την συνεχή βοήθεια που μου παρείχαν υπομονετικά.

Τέλος, θα ήθελα να ευχαριστήσω ιδιαιτέρως την οικογένειά μου για την διαρκή και εμπράκτη στήριξή τους.

Abstract

The current thesis has been elaborated in fulfillment of the thesis requirement for the inter-departmental postgraduate program "Applied Mathematical Sciences" of the National Technical University of Athens.

Algebraic path problem represents a generalized class of problems. This class consists of several problems, which fall under different fields of computer science, e.g. problems related to graph theory, communication networks, matrix problems, as well as regular language problems. Hence, designing an efficient algorithm for solving the general problem, would provide instantaneously an efficient solution to all the subcategories of the problem. However, this observation does not make useless the studying of several subcases, since, as we will present, due to some extra properties that some subproblems have, more efficient algorithms exist. In the current thesis, initially, are being presented sequential algorithms for the algebraic path problem, and for several subproblems as well. Next, we present parallel algorithms for solving the same problems.

The theoretical study of parallel computation begun in the decade of '70. The main motivation for studying such algorithms is the fact that through parallel computers we could obtain tremendous computing power. During the years a great variety of parallel models has been proposed in the literature, however, none achieved to be universally accepted as dominant, in the same way as von Neumann model is for sequential computing. The model on which parallel algorithms are compared and analyzed through this thesis is the BSP model. The proposed model is realistic enough and adequately simple as well, as a result design and analysis of algorithms, based on this model, becomes easier and more comprehensible.

Contents

1 Introduction **1**

1.1 Algebraic path problem 1

1.1.1 Definition of a semiring 2

1.1.2 Matrix approach 2

1.1.3 Graph approach 6

1.1.4 Examples of closed semirings 9

1.2 Parallel computing paradigms 11

1.2.1 Fine-grained parallelism: The PRAM model 12

1.2.2 Coarse-grained parallelism: The BSP model 14

1.2.3 Other models 18

2 Sequential algorithms **21**

2.1 Gauss-Jordan elimination algorithm 21

2.1.1 Elimination procedure 22

2.1.2 An alternative interpretation 26

2.1.3 Block decomposition method 28

2.2 Warshall's algorithm 29

2.2.1 Transitive closure version 30

2.2.2 Generic version 31

2.3 Dijkstra's algorithm 31

3 Parallel algorithms **33**

3.1 Blocks Gauss-Jordan 33

3.2 Transitive closure 35

3.2.1 Blocks Warshall 36

3.2.2 Leighton's algorithm 38

3.3 All pairs shortest path 42

3.3.1 Tiskin's nonnegative edge costs algorithm 42

3.3.2 Tiskin's general edge costs algorithm 46

Chapter 1

Introduction

Problems related to finding paths in networks as well as matrix computations are plentiful in computer science; they have been studied since the beginning of the Computer Age, and are at the heart of the vast majority of applications in the real world. Algorithms for many such problems were developed even earlier. The first known algorithms for matrix multiplication and matrix inverse were designed in the time of Gauss. Routing problems have been the subject of research since the mid 40's, initiated by the needs of telephone companies and travel agencies.

In this chapter, initially, algebraic path problem is presented and the two different approaches for solving the problem as well. Next, a variety of problems, which are special cases of the algebraic path problem, is analyzed. Finally, two most well known models of parallel computations are described (one of which will be used for analyzing parallel algorithms during this thesis), in order to investigate and evaluate the major differencies between parallel models. We conclude with a brief description of other significant models of parallelism.

1.1 Algebraic path problem

The algebraic path problem describes a general setting for solving various matrix and graph problems, as well as problems on regular languages. By developing a single algorithm for the specific generalized problem, a majority of problems can be solved. The problem is concerned with a special algebraic structure, called a closed semiring, with three generalized operations on the

elements of the set, and a unary operation called closure. These operations are defined on single elements, as well as on square matrices over a closed semiring. The *algebraic path problem* is the problem of finding the closure of a square matrix over an arbitrary closed semiring [Fin92].

1.1.1 Definition of a semiring

A *semiring* is a structure $(S, \oplus, \otimes, \bar{0}, \bar{1})$, where S is a set, $\oplus$ (a generalized addition) and $\otimes$ (a generalized multiplication) are binary operations on S, $\bar{0}$ and $\bar{1}$ are elements of S, and for all elements a, b and c of S the following properties hold:

A1) $(S, \oplus, \bar{0})$ is a commutative monoid

- a) $a \oplus (b \oplus c) = (a \oplus b) \oplus c$
- b) $a \oplus b = b \oplus a$
- c) $a \oplus \bar{0} = a$

A2) $(S, \otimes, \bar{1})$ is a monoid

- (d) $a \otimes (b \otimes c) = (a \otimes b) \otimes c$
- (e) $a \otimes \bar{1} = \bar{1} \otimes a = a$

A3) Generalized multiplication is distributive over generalized addition

- (f) $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$
  $(b \oplus c) \otimes a = (b \otimes a) \oplus (c \otimes a)$

multiplication has precedence over addition. The operations $\oplus$ and $\otimes$ are called generalized addition and multiplication respectively, because they may be different from the usual addition and multiplication in applications of the algebraic path problem to a particular graph problem. In subsection 1.1.4 are being presented several examples of semirings that correspond to certain graph problems.

1.1.2 Matrix approach

In the *matrix approach* to the algebraic path problem, suggested by Lehmann [Leh77], we define a new unary operation *, called *closure*, on the elements of a semiring such that

$$(\forall a \in S)\ a^* = \bar{1} \oplus a \otimes a^* = \bar{1} \oplus a^* \otimes a \text{ (closure property)}$$

Closure has precedence over the other operations. A semiring with a closure operation is called a *closed semiring*.

In the case we have a *partial closed semiring*, that is a closed semiring, as described above, where closure is a partial function satisfying the properties $(a) \ldots (f)$ whenever the closure is defined, it can be extended to a closed semiring. If S is a partial closed semiring, then $S \cup \{u\}$ can be transformed to a closed semiring by adding the following definitions:

$$
\begin{aligned}
&u \oplus a = a \oplus u = u \\
&u \otimes a = a \otimes u = u \\
&u^* = u \\
&a^* = u, \quad \text{if } a^* \text{ was not previously defined}
\end{aligned}
$$

for all elements $a \in S$. $S \cup \{u\}$ is called the completion of S.

Operations similar to generalized addition, multiplication and closure can be defined on $n \times n$ matrices over a closed semiring, that make this set nearly a closed semiring.

Let $A = (a_{ij})$ and $B = (b_{ij})$ be $n \times n$ matrices over a closed semiring S. We define

$$
\begin{aligned}
&A \oplus B = (a_{ij} \oplus b_{ij}) \\
&A \otimes B = (c_{ij}), \quad \text{where } c_{ij} = \bigoplus_{k=1}^{n} a_{ik} \otimes b_{kj}
\end{aligned}
$$

The *identity matrix* I over a closed semiring, shown in Figure 1.1, is a matrix whose all diagonal elements equal $\bar{1}$, and all other elements equal $\bar{0}$. It is straightforward to verify that generalized addition of matrices is commutative and associative, and generalized multiplication is associative and distributes over addition.

The closure operation on matrices is defined inductively on the size of the matrix by decomposing the matrix into four submatrices. The definition is correct because the computation of the closure of a matrix does not depend on the size of the submatrices.

Definition of the closure of a $n \times n$ matrix A:

$$\begin{pmatrix} \bar{1} & \bar{0} & \dots & \bar{0} & \bar{0} & \bar{0} \\ \bar{0} & \bar{1} & \dots & \bar{0} & \bar{0} & \bar{0} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \bar{0} & \bar{0} & \dots & \bar{1} & \bar{0} & \bar{0} \\ \bar{0} & \bar{0} & \dots & \bar{0} & \bar{1} & \bar{0} \\ \bar{0} & \bar{0} & \dots & \bar{0} & \bar{0} & \bar{1} \end{pmatrix}$$

Figure 1.1: Identity matrix

If $n = 1 \ (a)^* = (a^*)$

If $n > 1$ and $A = \begin{pmatrix} B & C \\ D & E \end{pmatrix}$ where, for some $0 < k < n$:

$B : k \times k, \quad C : k \times (n-k), \quad D : (n-k) \times k, \quad E : (n-k) \times (n-k),$

then,

$$A^* = \begin{pmatrix} B^* \oplus B^* \otimes C \otimes X^* \otimes D \otimes B^* & B^* \otimes C \otimes X^* \\ X^* \otimes D \otimes B^* & X^* \end{pmatrix}$$

for $X = E \oplus D \otimes B^* \otimes C$.

The proof that the closure matrix is well-defined boils down to computing the closure of a matrix with nine submatrices in two different ways:

$$\left(\begin{array}{c|cc} A & B & C \\ \hline D & E & F \\ G & H & I \end{array}\right) \quad \text{and} \quad \left(\begin{array}{cc|c} A & B & C \\ D & E & F \\ \hline G & H & I \end{array}\right)$$

and verifying nine identities. The verification is trivial using the properties of matrix operations, in particular, commutativity and associativity of matrix addition, associativity of matrix multiplication, and distributivity of matrix addition over matrix multiplication.

Lehmann has shown that the closure operation on matrices satisfies the closure property. That is, if A is an $n \times n$ matrix then:

$$A^* = I_n \oplus A \otimes A^* = I_n \oplus A^* \otimes A$$

where I_n is the identity matrix.

A class of closed semirings is *simple semirings*, in which the closure operation is simple to perform, $a^* = \bar{1}$. A closed semiring S is called simple if and only if

$$a \oplus \bar{1} = \bar{1} \quad \forall a \in S$$

Simple semirings are exactly the Q-*semirings* of Yoeli [Yoe61]. The *regular algebras* of Carre and Backhouse [BC75] are close to simple semirings. They do not assume $a \oplus \bar{1} = \bar{1}$, but assume $a \oplus a = a$. From this property a number of identities follow:

$$\begin{array}{l} \bar{1} \oplus \bar{1} = \bar{1}, \quad a^* = \bar{1}, \quad a \oplus a = a, \quad \bar{1} \oplus \bar{1} = \bar{1}, \\ a^* = \bar{1}, \quad a \oplus (a \otimes b) = a \oplus (b \otimes a) = a \\ (a \otimes b) \oplus (a \otimes b \otimes c) = a \otimes b, \quad \bar{0} \otimes a = a \otimes \bar{0} = \bar{0} \end{array}$$

From these properties arises the fundamental property of simple semirings from which derives that a simple semiring is a special case of the graph approach to the algebraic path problem. *Fundamental property* of simple semirings:
If $A = (a_{ij})$ is a $n \times n$ matrix over a simple semiring, $I_n = (d_{ij})$ the identity matrix, and $B = A^* = (b_{ij})$, then

$$b_{ij} = d_{ij} \bigoplus_{\substack{k_1,\dots,k_r \in [1,\dots,n] \\ k_l \neq k_m \neq i,j,\ \forall l,m \in [1,\dots,n]}} a_{ik_1} \otimes a_{k_1k_2} \otimes \dots \otimes a_{k_{r-1}k_r} \otimes a_{k_rj}$$

If we consider a $n \times n$ matrix as a weighted directed graph on n vertices, the fundamental property says that the $(i,j)-th$ element of the closure of a matrix is the generalized sum of the weights of all elementary paths from i to j. A path is called *elementary* if no vertices appear more than once in it. The property can be proved by using the inductive definition of A^*.

Lehmann has proved that if A is a $n \times n$ matrix over a simple semiring, then

$$A^* = I_n \oplus A \oplus A^2 \oplus \dots \oplus A^{n-1}$$

The proof is based on the fact that an elementary path has length less or equal to $n-1$ and the weights of all elementary paths of length r are terms in some element of A^r.

In simple semirings, the definition of the closure of a matrix becomes:

If $A = \begin{pmatrix} B & C \\ D & E \end{pmatrix}$ then
then,

$$A^* = \begin{pmatrix} X^* & X^* \otimes C \otimes E^* \\ E^* \otimes D \otimes X^* & E^* \oplus E^* \otimes D \otimes X^* \otimes C \otimes E^* \end{pmatrix}$$

for $X = B \oplus C \otimes E^* \otimes D$.

Futrhermore, there is a special case of simple semirings introduced by Dijkstra [Dij59]. A *Dijkstra semiring* is a simple semiring with the extra property

$$a \oplus b = \begin{cases} a \\ b \end{cases}$$

A Dijkstra semiring is totally ordered, where the order between two elements is defined by the rule: $a \succeq b \quad \text{iff} \quad a \oplus b = a$. The generalized addition then, can be viewed as a maximum operation in the ordered set.

1.1.3 Graph approach

The *graph approach* to the algebraic path problem, most clearly summarized by Rote [Rot85], is slightly less general than matrix approach, due to the fact that we assume two additional properties of a semiring, idempotence of addition and absorption rule:

i) $a \oplus a = a$
ii) $a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$

for all $a \in S$.
We consider a *weighted graph* $G = (V, E)$, which consists of a finite vertex set V, an edge set $E \subseteq V \times V$, together with a *weight function* $w : V \times V \to S$, where S is a semiring with additional properties i) and ii). Pairs of vertices that are connected by an edge, are assigned with non-zero weights otherwise, are assigned with $\bar{0}$. In the following we shall always assume that $V = \{1, 2, \ldots, n\}$. A *weighted path* from i to j in a weighted graph is an arbitrary sequence of vertices of the form $p = (i, k_1, k_2, \ldots, k_m, j)$, where k_i are vertices of the graph. The weight of a weighted path is defined as the generalized product of the weights of all edges of the path in order (generalized

multiplication is associative):

$$w(i, k_1, k_2, \ldots, k_m, j) = w(i, k_1) \otimes w(k_1, k_2) \otimes \ldots \otimes w(k_m, j)$$

We assume that the edges in the path are in order, due to the fact that generalized multiplication is not necessarily commutative.

A distinction should be made between empty paths, paths containing non-existing edges and loops. In the first case, an empty path, which begins and ends in the same vertex, containing no edges has weight equal to $\bar{1}$. In the second case, non-existing edges' weights equals to $\bar{0}$, hence, the weight of paths containing such edges equals to $\bar{0}$. In the latter case, loops also begins and ends in the same vertex, as in empty paths, but contain arbitrarily many edges, thus, loops may have non-zero weights.

In the algebraic path problem, what we want to compute is, in terms of the semiring, the $\oplus$-sum of the weights of all paths from i to j. Hence, we have the definition:

Given a weighted graph $G = (V, E, w), w : E \to S$, with weights from a semiring $(S, \oplus, \otimes, \bar{0}, \bar{1})$ find d_{ij} for all pairs of vertices i, j, where

$$d_{ij} = \bigoplus_{p \in P_{i,j}} w(p)$$

$P_{i,j}$ denotes the set of paths from i to j.

A solution to this problem need not exist, because the set of weights of the paths from i to j, over which the sum is taken, may be infinite. However, there may be semirings where such countable infinite sums may be defined in a consistent way, at least for some cases. To overcome this difficulty, we ask that the following axioms hold in addition to the existing:

Infinite distributivity

If A and B are countable, that is either finite or countably infinite, and the sums $\oplus_{a \in A} a$ and $\oplus_{b \in B} b$ are both defined, then the sum $\oplus_{a \in A, b \in B} a \otimes b$ is also defined, and

$$\bigoplus_{a \in A, b \in B} a \otimes b = (\bigoplus_{a \in A} a) \otimes (\bigoplus_{b \in B} b)$$

Infinite associativity

Let A be a countable subset of S, and $\{A_k | k \in K\}$ be a disjoint partition of

A. If for all A_k, $a_k = \oplus_{a \in A_k} a$ is defined, and if $\oplus_{k \in K} a_k$ is defined, then $\oplus_{a \in A}$ is also defined, and

$$\bigoplus_{a \in A} a = \bigoplus_{k \in K} a_k$$

Such semirings are called *partially complete*, or, if the sum is defined for every countable subset of S, *complete* semirings. These two axioms are sufficient for our purposes.

The algebraic path problem can also be formulated in a different way: With the weighted graph (V, E, w) we can associate a $n \times n$ *adjacency matrix* $A = (a_{ij})$, in which for all $(i, j) \in \{1, 2, \ldots, n\}$ the element a_{ij} contains the weight of the edge (i, j):

$$a_{ij} = \begin{cases} w(i,j), & \text{if}(i,j) \in E \\ \bar{0}, & \text{if}(i,j) \notin E \end{cases}$$

Figure 1.2 gives an example of a weighted graph and its adjacency matrix.

Generalized addition and multiplication of matrices over a semiring is defined as in matrix approach. We consider, now, the successive powers of square matrix A:

$A^2 = (a_{ij}^{(2)})$, where

$$(a_{ij}^{(2)}) = \bigoplus_{1 \le r \le n} a_{ir} \otimes a_{rj}$$

$A^3 = (a_{ij}^{(3)})$, where

$$(a_{ij}^{(3)}) = \bigoplus_{1 \le r_1, r_2 \le n} a_{ir_1} \otimes a_{r_1 r_2} \otimes a_{r_2 j}$$

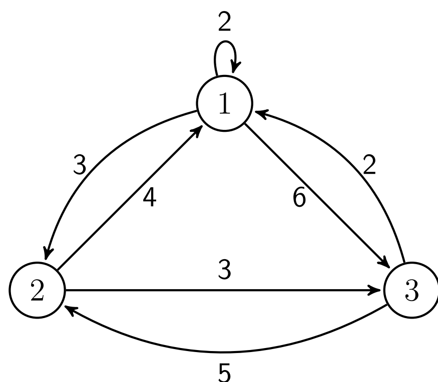
$\vdots$

$A^k = (a_{ij}^{(k)})$, where

$$a_{ij}^{(k)} = \bigoplus_{r_1, \ldots, r_{k-1} \in [1, \ldots, n]} a_{ir_1} \otimes a_{r_1 r_2} \otimes \ldots \otimes a_{r_{k-1} j}$$

From these relations we see that an element $a_{ij}^{(r)}$ of A^r contains the generalized sum of the weights of all r-edge paths from i to j. Therefore, if the matrix $D = (d_{ij})$ is the $n \times n$ matrix of the elements that are defined in algebraic path problem, we get, by the infinite associativity axiom, the matrix formulation of the algebraic path problem:

$$D = A^* = \bigoplus_{r \ge o} A^r = I_n \oplus A \oplus A^2 \oplus A^3 \oplus \ldots$$



(a) A weighted graph

	1	2	3
1	2	3	6
2	4	∞	3
3	2	5	∞

(b) Adjacency matrix

Figure 1.2: A graph and its corresponding adjacency matrix

1.1.4 Examples of closed semirings

We will consider now several examples of closed semirings in order to exhibit the great variety of problems that algebraic path problem generalizes.

Transitive closure
The problem of computing the transitive closure of a graph is, given a graph $G = (V, E)$, compute a new graph $G' = (V, E')$ with the same set of vertices as the given graph, and with a set of edges that satisfy the following property: $(i, j) \in E'$ if and only if there is a path from i to j in the initial graph.

The semiring for this problem has two elements, zero and unity, as well as disjunction and conjuction for generalized addition and multiplication respectively, $S = (\{0, 1\}, \vee, \wedge, *, 0, 1)$. Zero is the neutral element for disjunction, and unity the neutral element for conjuction. This semiring is called *Boolean semiring*. The closure operation, in this case, is easy to perform, since $0^* = 1^* = 1$. Table 1.1 displays the properties of the two operations.

To compute the reflexive-transitive closure of a graph, we define the following weight function

$$w(i,j) = \begin{cases} 1, & \text{if}(i,j) \in E \\ 0, & \text{if}(i,j) \notin E \end{cases}$$

We denote W the corresponding matrix. The closure of W is equivalent to the adjacency matrix of the transitive closure of the initial graph, since

$\vee$	0	1
0	0	1
1	1	1

$\wedge$	0	1
0	0	0
1	0	1

Table 1.1: Boolean operations

$d(i,j) = 1$ if and only if there is a path from i to j (where (d_{ij}) is defined in graph approach).

All pairs shortest path
In the problem of all pairs shortest path we are given a directed graph $G = (V, E)$, on every edge of which has been assigned some length (equivalent to weight), expressed as a non-negative real number. The length of a path equals to the sum of the lengths of its edges. The problem is to find for each pair of vertices (i, j) of the graph the length of their shortest path.

The semiring in this case is the set of non-negative real numbers including $+\infty$, with minimum operation min for generalized addition and usual addition for generalized multiplication, $S = (\mathbb{R}_+, \min, +, +\infty, 0)$. The neutral element of min is $+\infty$ and 0 is the neutral element of addition.

To solve this problem, we consider the graph approach to the algebraic path problem, hence, we define the following weight function

$$w(i,j) = \begin{cases} l(i,j), & \text{if}(i,j) \in E \\ +\infty, & \text{if}(i,j) \notin E \end{cases}$$

where $l(i, j)$ denotes the length of (i, j). The closure of matrix W, W is the corresponding matrix to the weight function, will be equivalent to the adjacency matrix we are looking for, since $d(i, j)$ equals the length of the shortest path from i to j.

Maximum capacity problem
This problem is similar to the all pairs shortest path problem. We are given a graph with some positive real valued cost assigned to every edge. The cost of a path is defined as the minimum of the costs of its edges. The problem is to find the path wich has maximal cost for every pair of vertices.

The semiring for this problem, as before, is the set of non-negative real numbers including $+\infty$, but in this case maximum operation corresponds to $\oplus$ and minimum operation corresponds to $\otimes$, thus, $S = (\mathbb{R}_+, \max, \min, +\infty, 0)$.

The weight function is the same with the previous one, with the only difference that $l(i, j)$ denotes the cost of (i, j). Due to the properties of operations in the semiring, once again, W^* corresponds to the correct solution to the problem.

Matrix inversion
The problem of matrix inversion is, given a real matrix A compute its multiplicative inverse A^{-1}, by satisfying: $A \cdot A^{-1} = A^{-1} \cdot A = I$, where I denotes the identity matrix.
The semiring that fits to this problem, is the set of real numbers with ordinary addition and multiplication for generalized addition and multiplication respectively, along with zero and unity as neutral elements, $S = (\mathbb{R}, +, \cdot, *, 0, 1)$.
The closure of a real number is given by:

$$x^* = \frac{1}{1 - x}$$

In this case, particularly, we have a partial closed semiring, where unity is the element of the semiring whose closure is not defined.
For matrices we can derive the formula for computing the inverse of the matrix from the relation $A^* = I \oplus A \otimes A^*$, that is:

$$\begin{aligned} A^* = I + A \cdot A^* &\equiv A^* - A \cdot A^* = I \\ &\equiv A^* \cdot (I - A) = I \\ &\equiv A^* = (I - A)^{-1} \end{aligned}$$

Now, instead of A, in the last formula, we substitude $(I - A)$, hence we get:

$$A^{-1} = (I - A)^*$$

For matrix inversion we used matrix appoach to the algebraic path problem.

1.2 Parallel computing paradigms

Over the years, various models of parallel and distributed computation have been proposed and studied in the literature. In this section we investigate two well known and representative parallel models of the two major categories of

parallel computations; memory-shared models and distributed memory models. Initially we examine the PRAM model, which for many years, had been the main model for studying parallel complexity theory. Next, we present the BSP model, a quite simple model, yet realistic enough to develop and analyze parallel algorithms. We conclude with a brief presentation of two other significant parallel models.

1.2.1 Fine-grained parallelism: The PRAM model

Shared memory parallel computers vary widely, but they all have in common the ability for each processor to be connected to one, shared, memory. The most representative shared memory model is the *Parallel Access Random Machine* (PRAM), originally introduced by Fortune and Wyllie, [FW78]. As its name indicates, the PRAM model is a straightforward parallel generalization of the Random Access Machine (RAM), [Pap95]. The PRAM model is an abstract model, which neglects various practical issues, such as communication and synchronization. This results to easier complexity and correctness analysis of algorithms, however, this level of abstraction might also lead to false estimations of practical situations. Another drawback of this model, as we will describe below, is the unrealistic assumption that the model consists of unbounded many processors (fine-grainess).

A PRAM consists of an unbounded global access memory, an unbounded collection of numbered processors $\{P_0, P_1, \ldots\}$, a set of input registers, and a finite program. Each processor knows its id, executes its own program, has an unbounded local memory, a program counter, its own accumulator, and a flag that indicates whether the processor is active or not.

Since PRAM is a parallel version of RAM model, a PRAM program is a finite sequence $\Pi = (\pi_1, \ldots, \pi_m)$ of instructions of the kinds READ, ADD, LOAD, JUMP, JZERO, STORE, SUB, HALT, with arguments standing for the contents of registers. The input of a PRAM program consisting of n bits, is placed at n designated shared memory cells. Initially, only processor P_0 is activated, it computes the number of required processors and places the number of processors in the designated shared-memory cell, then the corresponding processors start executing their programs. At each step, each processor executes the instruction defined by the program counter in one unit of time synchronously, it can access either a cell of the global memory, or a cell of its local memory. The only way for processors to communicate is by writing into and reading from shared memory cells. The problem that arises from

concurrent writing on the same shared memory cell will be discussed below. Computation proceeds until P_0 halts, at which time all active processors are halted. The output of the program consistng of n' bits is placed at n' designated shared memory cells.

The complexity analysis of a PRAM program is defined as follows: Let M be a PRAM. M computes in parallel time $t(n)$ with $p(n)$ processors if for every input $x \in \{0,1\}^n$, machine M halts within at most $t(n)$ time steps having activated at most $p(n)$ processors. M computes in sequential time $t(n)$ if it computes in parallel time $t(n)$ using a single processor.

At this point we can observe that the number of processors needed to solve a problem varies in function of the size of the input. This consists an unrealistic assumption, since at most cases the number of available processors is significant less than the size of the problem. For this reason we are more interested in models in which the number of processors is fixed. Therefore, in that case the goal would be to partition the data and opeations into blocks in order to distribute them equivalently to the available processors. Such a model is BSP, presented below.

Variations of the PRAM model

As stated previously, the operation of a synchronous PRAM can result in concurrent access of the same cell of shared memory by more than one processors. To be more precise, PRAM instructions are being executed in cycles consisting of three phases. In the first phase, each processor which has an instruction to read, reads from the shared memory. In the second phase, each processor performs a local computation, if it has been assigned any, and in the final phase if it has been assigned an instruction to write, writes to the shared memory. There have been developed several variants of the model to solve the possible conflicts, their description follows.

Exclusive Read Exclusive Write (EREW) PRAM
This variant does not allow any kind of simultaneous access neither read, nor write, to a common shared memory cell.

Concurrent Read Exclusive Write (CREW) PRAM
This variant allows concurrent reads but only one processor is allowed to write to a shared memory cell at every step.

Exclusive Read Concurrent Write (ERCW) PRAM
This variant does not allow simultaneous reading from a common memory

location, but permits to any processors to write to a single cell.

Concurrent Read Concurrent Write (CRCW) PRAM
This variant allows for concurrent reading as well as concurrent writing to a common shared memory cell.

Concurrent write has to be further constrained, in order to determine how the value written in a memory cell is selected among the many simultaneous writing attempts. We get the following submodels:

PRIORITY CRCW PRAM:
In this submodel the processors are assigned fixed distinct priorities and the processor with the highest priority writes in the cell.

ARBITRARY CRCW PRAM:
In this submodel one arbitrarily chosen processor is allowed to write in the common memory cell.

COMMON CRCW PRAM:
In this submodel concurrent write is allowed if and only if all the processors attempting to write to the same location have the same value to store.

1.2.2 Coarse-grained parallelism: The BSP model

The *Bulk-Synchronous Parallel* (BSP) model proposed by Valiant [Val90] is a unifying bridging model for parallel computation, in the same way as the von Neumann model is for sequential computation. The major purpose of BSP model is to provide a useful model that is simple, efficiently implementable, and acceptable to all parties involved: hardware designers, software developers, and end users. It is intended neither as a hardware nor as a programming model, but something in between, it is intended as a standard on which people can agree.

BSP computations

A *BSP computer* consists of a number of *components*, each with private memory, performing processing and/or memory functions, and a *router* that delivers messages point to point between pairs of components. Each component can read from or write to every memory cell in the entire machine. If the cell is local, the read or write operation is relatively fast. If the cell belongs to another component, a message should be sent through the router,

and this operation is slower. This implies that the router can be viewed as a black box, where the connectivity of the network is hidden in the interior. The router abstracts the underlying interconnection network of the components by introducing certain parameters in the model, described below. As users of a BSP computer, we need not to be concerned with the details of the interconnection network. We only care about the remote access time delivered by the router, which should be uniform. In the BSP computer is incorporated a barrier-style synchronization mechanism where all or a subset of the components are synchronized at regular intervals of L time units where L is the periodicity parameter.

A *BSP computation* consists of a sequence of *supersteps*. In each superstep, each component is allocated a task consisting of performing a sequence of operations on local data, message transmissions and message arrivals from other components, followed by global barrier synchronization. Consequently, a superstep consists of an input phase, a local computation phase and an output phase. In the input phase, a component receives data that were sent to it in the previous superstep; in the output phase, it can send data to other components, to be received in the next superstep. At the end of a superstep all components synchronize as follows. After each period of L time units, a global check is made to determine whether the superstep has been completed by all the components (or a subset), that is, each component checks whether all the local computations are finished and whether it has sent all messages that had to be sent and it has received all the messages that it had to receive. If it has, the machine proceeds to the next superstep. Otherwise, the next period of L units is allocated to the unfinished superstep. This form of synchronization is called bulk synchronization, because usually many computation or communication operations take place between successive syncronizations. There exist alternative synchronization mechanisms, for example, the system could continuously check whether the current step is completed, or synchronization methods, for example, pairwise synchronization; the results of the run-time analysis, however, will not change by more than small constant factors in the first case, and in the second case it would be in contrast to global barrier style synchronization.

BSP cost

The *cost function* of a BSP computation can be measured by summing the cost of its supersteps. We assume that in one time unit an operation can

be computed by a processing component on data available in memory local to it. The communication cost of a superstep is defined as follows. An *h-relation* is a superstep where each component sends and is sent at most h messages. Let h' denote the maximum number of messages received by each processor, h'' denote the maximum number of messages sent by each processor and $h = \max\{h', h''\}$ (another possible definition is $h = h' + h''$), then the communication cost of a computation which consists of S supersteps is

$$H = \sum_{s=1}^{S} h_s.$$

The local computation cost within a superstep is defined by w, where w is the maximum number of local operations performed by each processor. Hence, the local computation cost of a computation which consists of S supersteps is

$$W = \sum_{s=1}^{S} w_s.$$

As a result, the total cost of a BSP algorihtm is given by

$$W + H \cdot g + S \cdot l,$$

where g and l are machine-dependent parameters, defined below.

A BSP computer can be characterized by three basic parameters: p, g, and l. Here, p denotes the number of the components. Parameter g defines the basic throughput of the router when in continuous use (also called "bandwidth inefficiency" or "gap"). More precisely, g can be seen as the ratio between the number of local computational operations performed per second by all the components and the the total number of data units delivered per second by the router. Finally, l is the communication latency, which includes the startup cost of the h-relation realization, as well as the cost of synchronization (synchronization periodicity). In theory, parameters g and l can be bounded, but in reality the characteristic parameters of a computer are measured by computer benchmarking. The values of g and l depend on the underlying interconnection network and are nondecreasing functions of p. As a result, the larger the number of the components p, the more powerful the communication network must be to keep g and l low. The parameter g can be kept low, within limits, by using more pipelining or by having wider communication channels. Keeping g low or fixed, on the other hand,

p	number of components
g	communication throughput ratio
l	communication latency

Table 1.2: BSP Parameters

as the number of components p increases incurs extra costs. Hence, as the machine scales up, it may be preferable to spend more money on a better communication network than on faster components. Table 1.2 summarizes the BSP parameters.

We can predict the execution time of an implementation of a BSP algorithm on a parallel computer by theoretically analysing the cost of the algorithm and, independently, benchmarking the computer for its BSP performance [Bis04].

A representative BSP algorithm

The problem of matrix multiplication is one of the most studied problems in parallel computations. McColl and Valiant proposed a BSP coarse-grained algorithm for solving the problem, [McC95, McC96]; an algorithm which we will be using in chapter 3 for parallel matrix multiplications.

The problem consists in computing the matrix product of two $n \times n$ dense matrices $A \cdot B = C$ on p processors. The matrix product is computed by

$$C(i,j) = \sum_{k=1}^{n} A(i,k) \cdot B(k,j), \qquad 1 \leq i,j \leq n$$

To do so, initially we assume that A, B are distributed uniformly but arbitrarily across the p processors. Next, we partition matrices A, B and C into $p^{2/3}$ blocks of size $s \times s$ each, where $s = n/p^{1/3}$, i.e.

$$A = \begin{pmatrix} A[1,1] & \cdots & A[1,p^{1/3}] \\ \vdots & \ddots & \vdots \\ A[p^{1/3},1] & \cdots & A[p^{1/3},p^{1/3}] \end{pmatrix}$$

similarly for B and C. Then we have $C[i,j] = \sum_{1 \leq k \leq p^{1/3}} A[i,k] \cdot B[k,j]$. In this way we partition the initial problem into p distinct block multiplication

subproblems, which will be computed by all p available processors. To be more precise, we proceed with the description of the BSP algorithm.

In the first superstep of the computation each processor receives blocks $A[i,k], B[k,j]$ for which is responsible. Thus, the total cost of this superstep is equal to its communication cost which is equal to $n^2/p^{2/3}$. In the second superstep each processor computes the product of the blocks received during the first superstep, and sends each one of the $n^2/p^{2/3}$ resulting values to the unique processor which is responsible for computing the corresponding value in C. The computation cost of this superstep is n^3/p and the communication cost is $n^2/p^{2/3}$. During the final superstep each processor computes each of its n^2/p elements of C by adding the $p^{1/3}$ values received for that element. The cost of this superstep is equal to its computation cost which is $n^2/p^{1/3}$.

Hence, the total BSP cost of the algorithm is

$$W = O(n^3/p) \qquad H = O(n^2/p^{2/3}) \qquad S = O(1)$$

1.2.3 Other models

Systolic arrays

Systolic arrays were proposed by Kung and Leiserson as a formalization of parallel programming for VLSI, [KL78]. A systolic array is a grid-like planar mesh of processors, called cells. The main difference between this model and most other parallel models is that each processor may communicate only with neighbouring processors. This restriction makes the systolic architecture less flexible, however it does allow a more efficient hardware implementation. Each processor performs a sequence of operations on data that flows between them. In other words, we could say that every processor regularly "pumps" data in and out, hence the name systolic. Furthermore, each cell in the systolic array may keep some values in its own memory, but the size of its memory is $O(1)$. Systolic arrays is a scalable model meaning that the architecture can be easily extended to many more processors, since systolic arrays are built of cellular building blocks. Another advantage of the model is that it is capable of high communication throughput. Main disadvantages of the model are that it has an expensive implementation cost due to high bandwidth requirements, as well as the difficulty in building such arrays.

LogP

Based on the BSP model Culler et al. presented *LogP*, a distributed memory model for parallel computations, [CKP+93]. The main difference to the BSP model consists in being asynchronous, i.e. it features no barrier-style synchronization. The processors in this model communicate by point-to-point messages; pairwise messages are the exclusive means of synchronization of the processors. In this model, once again, processors are connected through an abstract interconnection network in which point-to-point communication is allowed. LogP uses parameters for modeling the performance of the communication medium. The parameters defining the LogP model are:

L: the latency, the delay of a sent message to reach its destination

o: the overhead, the time engaged for sending or receiving a message

g: the gap, minimum time among two message transmissions (per processor)

P: the number of processing units

We assume that each local operation costs one unit of time, called a processor cycle. The parameters L, o and g are measured as multiples of the processor cycle.

Chapter 2

Sequential algorithms

In this section we present sequential algorithms for solving the algebraic path problem and specific instances of its'. The first one is Gauss-Jordan elimination algorithm, the second one is Warshall's algorithm, and the third one is Dijkstra's algorithm. Warshall's algorithm solves the transitive closure problem, an instance of algebraic path problem, but we show that with a small modification to the algorithm we can derive a solution to the general problem. Dijkstra's algorithm computes the single source shortest path problem, another instance of our general problem.

2.1 Gauss-Jordan elimination algorithm

Rote presented an algorithm for the algebraic path problem [Rot90] which, eventually, corresponds to the Gauss-Jordan elimination algortihm of ordinary linear algebra, hence it carries this name.

Before describing the procedure, let us rewrite the equation that corresponds to the graph approach for the algebraic path problem.

$$x_{ij} = \bigoplus_{p \in P_{i,j}} w(p)$$

From this equation we can derive a new one, if we partition the set of all paths from i to j according to the first vertex of each path. This way we get:

$$x_{ij} = \bigoplus_{k=1}^{n} (a_{ik} \otimes x_{kj}), \qquad \text{for } i \neq j$$
$$x_{jj} = \bigoplus_{k=1}^{n} (a_{jk} \otimes x_{kj}) \oplus \bar{1}$$

where x_{ij} represents the sum of the weights of all paths from i to j. The quantities a_{ij} are the given weights of edges and x_{ij} are the unknowns. In matrix form we have $X = I \oplus A \otimes X$, where I is the identity matrix and A the adjacency matrix of the given graph.

We are looking now for a solution to the matrix equation. In order to find a solution for $n \times n$ matrices we first investigate the case of 1×1 matrices. Thus, we derive the *iteration equation*:

$$x = \bar{1} \oplus a \otimes x$$

If we repeatedly sustitute the expression for x into the right side of the equation, we have

$$\begin{aligned} x &= \bar{1} \oplus a \otimes x \\ &= \bar{1} \oplus a \otimes (\bar{1} \oplus a \otimes x) \\ &\vdots \\ &= \bar{1} \oplus a \oplus a^2 \oplus a^3 \oplus a^4 \oplus \dots \end{aligned}$$

If this sequence remains stable after a finite number of iterations, then the sum, which is equivalent to a^*, is a solution of the iteration equation. Correspondingly, the solution of the matrix form of this equation, is given by A^*. We can obtain a solution of a more general problem by multiplying iteration equation from the right side with any element $b \in S$, where if $x = a^*$ is the solution of the initial equation then $y = a^* \otimes b$ is a solution of $y = b \oplus a \otimes y$.

2.1.1 Elimination procedure

In order to describe the procedure, without loss of generality, firstly, we present the method by investigating a specific example.

Suppose our problem consists of a (4×4) matrix. If we examine the system of equations of the definition of the problem, we can observe that the

column index j of the unknown x_{ij} quantities is the same for all variables that occur in one equation. Hence, our initial system of equations consists of four decoupled systems of equations, one for each column of X, for example:

$$\begin{array}{lllllllllll} x_{13} & = & a_{11} \otimes x_{13} & \oplus & a_{12} \otimes x_{23} & \oplus & a_{13} \otimes x_{33} & \oplus & a_{14} \otimes x_{43} & & \\ x_{23} & = & a_{21} \otimes x_{13} & \oplus & a_{22} \otimes x_{23} & \oplus & a_{23} \otimes x_{33} & \oplus & a_{24} \otimes x_{43} & & \\ x_{33} & = & a_{31} \otimes x_{13} & \oplus & a_{32} \otimes x_{23} & \oplus & a_{33} \otimes x_{33} & \oplus & a_{34} \otimes x_{43} & \oplus & \bar{1} \\ x_{43} & = & a_{41} \otimes x_{13} & \oplus & a_{42} \otimes x_{23} & \oplus & a_{43} \otimes x_{33} & \oplus & a_{44} \otimes x_{43} & & \end{array}$$

This system is very much like an ordinary linear system of equations, with the difference that the unknowns appear on both sides; they appear explicitly on the left side and implicitly on the right side. We use the solution to the more general iteration equation we described earlier in order to solve the first equation, since it can be written: $x_{13} = a \otimes x_{13} \oplus b$, with $a = a_{11}$ and $b = a_{12} \otimes x_{23} \oplus a_{13} \otimes x_{33} \oplus a_{14} \otimes x_{43}$. If we assume that a^* exists, then we get that $x_{13} = a^* \otimes b$ is a solution of the above equation. This way we get an explicit expression for x_{13}:

$$\begin{array}{lll} x_{13} & = & a_{11}^* \otimes (a_{12} \otimes x_{23} \oplus a_{13} \otimes x_{33} \oplus a_{14} \otimes x_{43}) \\ & = & a_{11}^* \otimes a_{12} \otimes x_{23} \oplus a_{11}^* \otimes a_{13} \otimes x_{33} \oplus a_{11}^* \otimes a_{14} \otimes x_{43} \end{array}$$

Substituting this into the rest of equations and collecting terms, we get a new system:

$$\begin{array}{lllllllll} x_{13} & = & a_{12}^{(1)} \otimes x_{23} & \oplus & a_{13}^{(1)} \otimes x_{33} & \oplus & a_{14}^{(1)} \otimes x_{43} & & \\ x_{23} & = & a_{22}^{(1)} \otimes x_{23} & \oplus & a_{23}^{(1)} \otimes x_{33} & \oplus & a_{24}^{(1)} \otimes x_{43} & & \\ x_{33} & = & a_{32}^{(1)} \otimes x_{23} & \oplus & a_{33}^{(1)} \otimes x_{33} & \oplus & a_{34}^{(1)} \otimes x_{43} & \oplus & \bar{1} \\ x_{43} & = & a_{42}^{(1)} \otimes x_{23} & \oplus & a_{43}^{(1)} \otimes x_{33} & \oplus & a_{44}^{(1)} \otimes x_{43} & & \end{array}$$

where the new coefficients $a_{ij}^{(1)}$ are defined as follows:

$$\begin{array}{lll} a_{1j}^{(1)} & = & a_{11}^* \otimes a_{1j}, \qquad \text{for } j > 1 \\ a_{ij}^{(1)} & = & a_{ij} \oplus a_{i1} \otimes a_{11}^* \otimes a_{1j}, \quad \text{for } i \neq 1, j > 1 \end{array}$$

The four equations in the current system fall in two different categories; in the first one, the first equation has an explicit expression for x_{13}, and in the second one, the remaining three equations form an implicit system for the other three variables x_{23}, x_{33} and x_{43}. Our new system has the same structure as the initial system, but one variable less. This way, we repeat the elimination procedure in essentialy the same way as before. Firstly, we eliminate x_{23} from the second equation, assuming that $(a_{22}^{(1)})^*$ exists, and substitute this variable into the other three equations. The new system, looks like the last one, with the differnce that variable x_{23} does not appear on the right side and the number in the superscripts of the coefficients is 2 instead of 1 (second step). During the elimination of the variable x_{33}, though, there is a difference because of the $\bar{1}$ term on the right side of the third equation. This time, we have:

$$\begin{aligned} x_{33} &= (a_{33}^{(2)})^* \otimes (a_{24}^{(1)} \otimes x_{43} \oplus \bar{1}) \\ &= (a_{33}^{(2)})^* \otimes a_{24}^{(1)} \otimes x_{43} \oplus (a_{33}^{(2)})^* \end{aligned}$$

By substituting this into the other equations of the system, we get the following set:

$$\begin{aligned} x_{13} &= a_{13}^{(3)} \oplus a_{14}^{(3)} \otimes x_{43} \\ x_{23} &= a_{23}^{(3)} \oplus a_{24}^{(3)} \otimes x_{43} \\ x_{33} &= a_{33}^{(3)} \oplus a_{34}^{(3)} \otimes x_{43} \\ x_{43} &= a_{43}^{(3)} \oplus a_{44}^{(3)} \otimes x_{43} \end{aligned}$$

where the new coefficients of the system are defined, this time, as follows:

$$\begin{aligned} a_{33}^{(3)} &= (a_{33}^{(2)})^* \\ a_{i3}^{(3)} &= a_{i3}^{(2)} \otimes (a_{33}^{(2)})^*, && \text{for } i \neq 3 \\ a_{3j}^{(3)} &= (a_{33}^{(2)})^* \otimes a_{3j}^{(2)}, && \text{for } j > 3 \\ a_{ij}^{(3)} &= a_{ij}^{(2)} \oplus a_{i3}^{(2)} \otimes (a_{33}^{(2)})^* \otimes a_{3j}^{(2)}, && \text{for } i \neq 3, j > 3 \end{aligned}$$

Without loss of generality, we regard the constant terms as the third column of the coefficient matrix. The final elimination is of variable x_{43} from the last equation. Once again, by substitution, we arrive to the solution:

$$\begin{aligned} x_{13} &= a_{13}^{(4)} \\ x_{23} &= a_{23}^{(4)} \\ x_{33} &= a_{33}^{(4)} \\ x_{43} &= a_{43}^{(4)} \end{aligned}$$

with coefficients determined by:

$$\begin{aligned} a_{44}^{(4)} &= (a_{44}^{(3)})^* \\ a_{i4}^{(4)} &= a_{i4}^{(3)} \otimes (a_{44}^{(3)})^*, \quad \text{for } i \neq 4 \end{aligned}$$

The final step consists of substitutions of variables and applications of the semiring properties.

To sum up the procedure we have followed to derive the solution, for the specific case we have investigated, we formulate the equations we have obtained, in a general way. Hence, we get:

$$\begin{aligned} x_{il} &= \bigoplus_{j=k+1}^{n} a_{ij}^{(k)} \otimes x_{jl}, && \text{for } 0 \leq k < l, i \neq l \\ x_{ll} &= \bigoplus_{j=k+1}^{n} a_{lj}^{(k)} \otimes x_{jl} \oplus \bar{1}, && \text{for } 0 \leq k < l \\ x_{il} &= \bigoplus_{j=k+1}^{n} a_{ij}^{(k)} \otimes x_{jl} \oplus a_{il}^{(k)}, && \text{for } l \leq k \leq n \end{aligned}$$

where k denotes the step number. We assume that the elements $a_{ij}^{(0)}$ equals to the elements of the original coefficient matrix a_{ij}. In the example we illustrated above, we had $l = 3$. Finally, the equations for the coefficients are, for $k \geq 1$:

$$\begin{aligned}
a_{kk}^{(k)} &= (a_{kk}^{(k-1)})^* \\
a_{ik}^{(k)} &= a_{ik}^{(k-1)} \otimes (a_{kk}^{(k-1)})^*, && \text{for } i \neq k \\
a_{kj}^{(k)} &= (a_{kk}^{(k-1)})^* \otimes a_{kj}^{(k-1)}, && \text{for } j \neq k \\
a_{ij}^{(k)} &= a_{ij}^{(k-1)} \oplus a_{ik}^{(k-1)} \otimes (a_{kk}^{(k-1)})^* \otimes a_{kj}^{(k-1)}, && \text{for } i \neq k, j \neq k
\end{aligned}$$

We may observe now that the above recursions are the same for all columns l, as far as they overlap for different columns. Thus, for our problem, where we want to compute the whole matrix X, we get just the above recursions, and the final result is given by

$$x_{ij} = a_{ij}^{(n)}$$

The algorithm that we derive from the recursive procedure, described above, corresponds to the Gauss-Jordan elimination algorithm of linear algebra. The algorithm follows.

Gauss-Jordan elimination algorithm

1. **for** $k := 1$ **to** n **do**
2. $\quad a_{kk} = (a_{kk})^*$;
3. $\quad$ **for** $i = 1$ **to** n **with** $i \neq k$ **do**
4. $\quad\quad a_{ik} := a_{ik} \otimes a_{kk}$;
5. $\quad$ **for** $i = 1$ **to** n **with** $i \neq k$ **do**
6. $\quad\quad$ **for** $j = 1$ **to** n **with** $j \neq k$ **do**
7. $\quad\quad\quad a_{ij} := a_{ij} \oplus a_{ik} \otimes a_{kj}$;
8. $\quad$ **for** $j = 1$ **to** n **with** $j \neq k$ **do**
9. $\quad\quad a_{kj} := a_{kk} \otimes a_{kj}$;

The described algorithm has $O(n^3)$ time complexity.

2.1.2 An alternative interpretation

We shall give now an alternative interpretation to the equations studied in subsection 2.1.1. In contrast to the previous approach, where coefficients were derived by purely algebraic means, we will interpet the coefficients as generalized sums of path weights. Due to the fact that we are dealing with infinite sums, and since the purpose of this subsection is to provide a different insight to the aforementioned expressions, for simplicity, we assume that all infinite sums exist.

Let us define a family of sets of paths as follows: We assume that vertices are numbered $1, 2, \ldots, n$. In addition, for $1 \leq i, j \leq n$ and $0 \leq k \leq n$, $P_{ij}^{(k)}$ denotes the set of all paths from i to j whose intermediate vertices belong to the subset of vertices $\{1, 2, \ldots, k\}$. Furthermore, we have to take into consideration empty paths, in the case of an empty path (i) we count i as an intermediate node, hence, path (i) is contained in $P_{ii}^{(i)}$ and not in $P_{ii}^{(i-1)}$.

Let us begin with the formulas for the coefficients. We can now give the following interpretation of the coefficients $a_{ij}^{(k)}$, taking into account the previously stated notations. We have,

$$a_{ij}^{(k)} = \bigoplus_{p \in P_{ij}^{(k)}} w(p)$$

The correctness of the above expression can be verified by induction, using the set of recursive equations for the coefficients; starting from $k = 0$ we get the initial values of edges of the graph, since, $P_{ij}^{(0)}$ contains only the initial edge (i, j) (if it belongs to the graph), hence, $a_{ij}^{(0)} = a_{ij}$.

In the first expression we have $a_{kk}^{(k)} = (a_{kk}^{(k-1)})$. A path in $P_{kk}^{(k)}$ starts from vertex k and ends at vertex k; meanwhile, it can pass arbitrarily many times through k. We can get a more precise expression for this kind of paths, by cutting the path into pieces at these intermediate vertices k, thus, we get $r \geq 0$ subpaths which are contained in the set $P_{kk}^{(k-1)}$. For this reason, the formula $(a_{kk}^{(k-1)})^* = \bar{1} \oplus a_{kk}^{(k-1)} \oplus (a_{kk}^{(k-1)})^2 \oplus \ldots \oplus (a_{kk}^{(k-1)})^r \oplus \ldots$ accounts for every path in $P_{kk}^{(k)}$ in a unique way.

In the second formula we have $a_{ik}^{(k)} = a_{ik}^{(k-1)} \otimes (a_{kk}^{(k-1)})^*$, for $i \neq k$. A path in $P_{ik}^{(k)}$ can be divided in a unique way, by considering the first occurence of vertex k in the path. This way, we get that the first subpath is in $P_{ik}^{(k-1)}$, and the remaining subpath is accounted for by $(a_{kk}^{(k-1)})^*$. In the third formula the same argument holds, with the difference that we split the path according to the last occurence of k, instead of the first.

In the fourth and last equation, we have $a_{ij}^{(k)} = a_{ij}^{(k-1)} \oplus a_{ik}^{(k-1)} \otimes (a_{kk}^{(k-1)})^* \otimes a_{kj}^{(k-1)}$, for $i \neq k, j \neq k$. Following the same procedure, and with the same arguments holding, we split a path in $P_{ij}^{(k)}$ according to the first and last occurence of vertex k. In this case, though, we have to take into consideration the paths that do not go through k at all, hence, we add the term $a_{ij}^{(k-1)}$.

Using the same arguments as inductive steps from k to $k + 1$, we finally

get that $a_{ij}^{(n)} = x_{ij}$, because $P_{ij}^{(n)}$ denotes the set of all paths from i to j.

Consider, now, the set of equations of the terms x_{il}, which contain the coefficients $a_{ij}^{(k)}$ as well. In this interpretation there is no actual difference between the variables and the coefficients, since we consider them both as sums of path weights.

In the first expression we have $x_{il} = \bigoplus_{j=k+1}^{n} a_{ij}^{(k)} \otimes x_{jl}$, for $0 \leq k < l, i \neq l$. The left side represents all paths from i to l, for some $i \neq l$. Since l is greater than k, such a path must contain at least one intermediate vertex whose index is greater than k. Let j be the first intermediate vertex of the path whose index is greater than k's, and decompose the path into two subpaths according to this vertex. The first subpath from i to j contains no intermediate vertex greater than k, this is taken into account by the term $a_{ij}^{(k)}$. The second subpath can be any arbitrary path from j to l, and is represented be the term x_{jl}; this way, the product $a_{ij}^{(k)} \otimes x_{jl}$ is the sum of the weights of all paths whose first intermediate node which is greater than k is j. Concluding, we have that the right hand side of the equation represents every path from i to l in a unique way, because vertex j can be any vertex between $k+1$ and n.

In the second equation, $x_{ll} = \bigoplus_{j=k+1}^{n} a_{lj}^{(k)} \otimes x_{jl} \oplus \bar{1}$, for $0 \leq k < l$, the only difference from the first one is that $\bar{1}$ is added, due to the fact that we have to take into account the empty paths that are contained in $P_{ll}^{(n)}$.

Finally, the third equation where $x_{il} = \bigoplus_{j=k+1}^{n} a_{ij}^{(k)} \otimes x_{jl} \oplus a_{il}^{(k)}$, for $l \leq k \leq n$, differs from the first one due to the addition of the last term. Term $a_{il}^{(k)}$ is added because of the fact that a path from i to l does not need to contain an intermediate vertex j whose index is greater than k (that is k is the last intermediate vertex).

2.1.3 Block decomposition method

As in the case of real matrices, we can decompose a matrix into blocks and carry out the computations blockwise. As we described matrix decomposition in the matrix approach to the algebraic path problem, if we decompose matrix A into four blocks, we get: $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ where,

$A_{11} : k \times k$, $A_{12} : k \times (n-k)$, $A_{21} : (n-k) \times k$, $A_{22} : (n-k) \times (n-k)$, for some $0 < k < n$. Hence, equation $X = A \otimes X \oplus I$ changes to:

$$\begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \otimes \begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} \oplus \begin{pmatrix} I_{11} & 0 \\ 0 & I_{22} \end{pmatrix}$$

Next, we follow the elimination procedure, as described, for this block equation without any change. The only difference lies within computations, where during the elimination of X_{IJ} we use the iteration equation $X = A \otimes X \oplus B$ and the problem of computing A^* is of the same type as the original problem. However, the new problem has a smaller size, and this gives us the opportunity to develop recursive divide and conquer algorithms.

We apply the elimination algorithm to the above decomposition, this way we get the following:

$$\begin{array}{ll} A_{11}^{(1)} := (A_{11})^* & X_{22} = A_{22}^{(2)} := (A_{22}^{(1)})^* \\ A_{21}^{(1)} := A_{21} \otimes A_{11}^{(1)} & X_{12} = A_{12}^{(2)} := A_{12}^{(1)} \otimes X_{22} \\ A_{22}^{(1)} := A_{22} \oplus A_{21}^{(1)} \otimes A_{12} & X_{11} = A_{11}^{(2)} := A_{11}^{(1)} \oplus X_{12} \otimes A_{21}^{(1)} \\ A_{12}^{(1)} := A_{11}^{(1)} \otimes A_{12} & X_{21} = A_{21}^{(2)} := X_{22} \otimes A_{21}^{(1)} \end{array}$$

Hence, the solution is given by the following matrix:

$$X = A^* = \begin{pmatrix} A_{11}^* \oplus A_{11}^* \otimes A_{12} \otimes F^* \otimes A_{21} \otimes A_{11}^* & A_{11}^* \otimes A_{12} \otimes F^* \\ F^* \otimes A_{21} \otimes A_{11}^* & F^* \end{pmatrix}$$

for $F = A_{22} \oplus A_{21} \otimes A_{11}^* \otimes A_{12}$.

In section 3.1 we shall describe a recursive algorithm, for the above decomposition, using the divide and conquer technique.

2.2 Warshall's algorithm

As we presented in section 1.1.4 the problem of computing the transitive closure of a graph is an instance of the algebraic path problem. One of the most famous algorithms for solving the transitive closure problem is Warshall's algorithm [AHU74]. The method that Warshall has described [War62] can be used to solve the algebraic path problem as well (a property that not all algorithms for instances of algebraic path problem have). Furthermore, Floyd,

following the same procedure [Flo62], presented his well-known algorithm for the all pairs shortest path problem, as described in section 1.1.4.

2.2.1 Transitive closure version

In the case of the transitive closure problem we sum only simple paths, as opposed to the algebraic path problem where we sum all paths. *Simple paths* are paths in which no vertex appears twice, except possibly the first and the last one. The original algorithm is based on the following observation. Consider all the possible paths from vertex i to vertex j, for each pair of vertices $i, j \in V$, whose intermediate vertices belong to the subset $\{1, 2, \ldots, k\}$, for $1 \leq k \leq n$. Let p denote one of all the possible paths. Path p, now, may have one of the following two properties:

- Vertex k is an intermediate vertex of path p. In this case, we can decompose p into two subpaths p_1, p_2 according to k, that is, $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$, where subpaths p_1, p_2 consist of vertices that belong to the subset $\{1, 2, \ldots, k-1\}$. This property holds since we have assumed that p is a simple path, hence, vertex k appears only once in this path.

- Vertex k is not an intermediate vertex of path p. In this case, all intermediate vertices belong to the subset $\{1, 2, \ldots, k-1\}$. Thus, a path from vertex i to vertex j whose intermediate vertices belong to the subset $\{1, 2, \ldots, k-1\}$, may denote, as well, a path from vertex i to vertex j whose intermediate vertices belong to the subset $\{1, 2, \ldots, k\}$.

Based on the above properties of simple paths, Warshall presented the following algorithm:

Warshall's algorithm

---

1. **for** $k = 1$ **to** n **do**
2.   **for** $i = 1$ **to** n **do**
3.   **for** $j = 1$ **to** n **do**
4. $A(i, j) = A(i, j) \oplus A(i, k) \otimes A(k, j)$;

(recall that $\oplus$ denotes min and $\otimes$ denotes $+$). The time complexity of this algorithm is $O(n^3)$.

2.2.2 Generic version

Lehmann [Leh77] based on Warshall's algorithm for transitive closure problem [War62], Floyd's algorithm for all pairs shortest path problem [Flo62], and Kleene's proof that every regular language can be represented by a regular expession [Kle56], presented a generalized algorithm to compute the closure of a matrix, which he named after them. The description of the procedure follows:

WFK algorithm

1. **for** $k = 1$ **to** n **do**
2. **for** $i = 1$ **to** n **do**
3. **for** $j = 1$ **to** n **do**
4. $A(i,j) = A(i,j) \oplus A(i,k) \otimes A(k,k)^* \otimes A(k,j)$;

The proof of correctness of the algorithm is based, as in the previous cases, by induction on k. The time complexity of WFK - algorithm is, as before, $O(n^3)$.

2.3 Dijkstra's algorithm

The problem of single source shortest path is another instance of the algebraic path problem. In this case, we want to compute all the shortest paths, which have a specific starting vertex (source), of a given directed graph whose edges are assigned with nonnegative weights. The most well known algorithm for this problem was introduced by Dijkstra [Dij59]. In this case, unfortunately, we cannot derive a generalized algorithm for the algebraic path problem; we can get, however, an algorithm for the problem of all pairs shortest path, which is timewise equally efficient to the previously described.

Let s denote the starting vertex of the problem. In Dijkstra's algorithm [ZF12, CLRS01] a set T is used, in order to keep the indices of the already processed vertices. In the beggining of the procedure set T is empty, the cost of each node is set to ∞ except of the cost of s that is set to 0. The algorithm at each step of the procedure picks a vertex $u \in V \setminus T$ that has at the current step the lower cost. Next, the selected vertex is added to set T and the costs of the remaining vertices in $V \setminus T$, which are adjacent to the previously added vertex, are being recalculated. The procedure stops when

the set $V \setminus T$ has emptied. The description of the algorithm follows:

Dijkstra's algorithm

1. $l(s) := 0$;
2. **for all** $v \in V \setminus \{s\}$ **do** $l(v) := \infty$;
3. $T := \emptyset$; $Q := V$;
4. **while** $Q \neq \emptyset$ **do**
5. $\quad v := min(Q, l)$;
6. $\quad T := T \cup \{v\}$;
7. $\quad$ **for all** u adjacent to v **do**
8. $\quad\quad$ **if** $l(u) > l(v) + w(v, u)$ **then**
9. $\quad\quad\quad l(u) := l(v) + w(v, u)$;

The time complexity of the algorithm is $O(|V^2| + |E|) = O(|V^2|) = O(n^2)$.

As mentioned before, we can compute the all pairs shortest path problem using Dijkstra's method. In order to achieve this, we add the following extra statement at the beggining of the algorithm:

$1'$. **for all** $s \in V$ **do**

This way, we compute for each vertex s of the given graph the shortest paths which have s as a starting vertex. The time complexity of the modified algorithm is $O(n^3)$, since we execute n times Dijkstra's algorithm.

Chapter 3

Parallel algorithms

In this chapter we present coarse-grained parallel versions of algorithms, presented in chapter 2. In addition, we describe two parallel algorithms for the instance of algebraic path problem, all pairs shortest path. All the algorithms will be described and analyzed on the BSP model (section 1.2.2).

3.1 Blocks Gauss-Jordan

A BSP approach to the block version of Gaussian elimination, described in section 2.1.3, was proposed by Tiskin [Tis02]. The proposed recursive algorithm uses the divide and conquer technique, since the only parallelism that can be exploited, in the procedure, is within block operations.

In this algorithm, as described previously, we divide matrix A into square blocks, $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$, with the difference, that we specify the block size to be $(n/2 \times n/2)$. As a result, the expression of our problem changes to

$$\begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \otimes \begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} \oplus \begin{pmatrix} I_{11} & 0 \\ 0 & I_{22} \end{pmatrix}$$

Next, we apply block Gauss-Jordan elimination, thus, we get the system of equations:

$$
\begin{aligned}
&A_{11}^{(1)} := (A_{11})^* && X_{22} = A_{22}^{(2)} := (A_{22}^{(1)})^* \\
&A_{21}^{(1)} := A_{21} \otimes A_{11}^{(1)} && X_{12} = A_{12}^{(2)} := A_{12}^{(1)} \otimes X_{22} \\
&A_{22}^{(1)} := A_{22} \oplus A_{21}^{(1)} \otimes A_{12} && X_{11} = A_{11}^{(2)} := A_{11}^{(1)} \oplus X_{12} \otimes A_{21}^{(1)} \\
&A_{12}^{(1)} := A_{11}^{(1)} \otimes A_{12} && X_{21} = A_{21}^{(2)} := X_{22} \otimes A_{21}^{(1)}
\end{aligned}
$$

The procedure can be applied recursively to compute A_{11}^* and $(A_{22}^{(1)})^*$.

Initially, we assume that the matrices are distributed across the processors evenly, but otherwise arbitrarily. In the first step, all p processors are available to compute the block Gauss-Jordan elimination. There is no substantial parallelism between block decomposition and block operations in the derived system of equations. For this reason, we compute the recursion tree in depth first order. On each level of recursion block multiplications of the above set are performed in parallel by all processors available at current level. Each block decomposition is also performed in parallel by all processors available at the current level, assuming that block size is large enough. If blocks become sufficiently small, block Gauss-Jordan elimination is performed sequentially by a random processor. We can have a variation over the depth at which the algorithm switches from parallel to sequential computation. This variation allows us to trade off the costs of communication and synchronization in a certain range. Therefore, a real parameter α is introduced, which controls the depth of parallel recursion. The algorithm follows:

Tiskin's algebraic path computation algorithm

Parameters: integer $n \geq p$; real number α, $\alpha_{\min} = 1/2 \leq \alpha \leq 2/3 = \alpha_{\max}$

Description The computation is defined by recursion on the size of the matrix. Denote the size of the matrix at the current level of recursion by m, keeping n for the original size. Let $n_0 = n/p^{\alpha}$. The value n_0 is the threshold at which the algorithm switches from parallel to sequential computation.
On each step of recursion the matrix is divided into regular square blocks of size $m/2$. Then, computations of our set of equations are performed by the following schedule.

Small blocks: If $1 \leq m \leq n_0$, from all the processors that are currently available choose arbitrarily a single processor and compute block Gauss-Jordan elimination on this processor.

Large blocks: If $n_0 < m \leq n$, compute $A_{11}^{(1)}$ by recursion. Then compute $A_{21}^{(1)}$, $A_{22}^{(1)}$ and $A_{12}^{(1)}$ by McColl-Valiant algorithm. Next, compute $A_{22}^{(2)}$ by recursion. Finally, compute $A_{12}^{(2)}$, $A_{11}^{(2)}$, and $A_{21}^{(2)}$ by McColl-Valiant algorithm [McC95, McC96]. Each of these computations is performed with all available processors.

In the cost analysis of this algorithm, the values for the costs of computation, communication, and synchronization can be found from the following reccurence relations:

$$
\begin{aligned}
W(m) &= \begin{cases} 2 \cdot W(m/2) + O(m^3/p), & \text{if } n_0 < m \leq n \\ O(n_0^3), & \text{if } m = n_0 \end{cases} \\
H(m) &= \begin{cases} 2 \cdot H(m/2) + O(m^2/p^{2/3}), & \text{if } n_0 < m \leq n \\ O(n_0^3), & \text{if } m = n_0 \end{cases} \\
S(m) &= \begin{cases} 2 \cdot S(m/2) + O(1), & \text{if } n_0 < m \leq n \\ O(1), & \text{if } m = n_0 \end{cases}
\end{aligned}
$$

which gives the following BSP cost:

$$W = O(n^3/p) \qquad H = O(n^2/p^\alpha) \qquad S = O(p^\alpha)$$

For $\alpha = \alpha_{\min} = 1/2$ the BSP cost of Tiskin's algorithm is $W = O(n^3/p)$, $H = O(n^2/p^{1/2})$, $S = O(p^{1/2})$. On the other hand, for $\alpha = \alpha_{\max} = 2/3$ the total cost of the algorithm is $W = O(n^3/p)$, $H = O(n^2/p^{2/3})$, $S = O(p^{2/3})$. In the first case the synchronization cost is lower in comparison to the second case. In the latter case, for $\alpha = \alpha_{\max}$ the communication cost is lower in comparison to the first case. Therefore, this improvement in communication efficiency is offset by a reduction in synchronization efficiency. For large n, though, the communication efficiency of the described algorithm dominates the synchronization cost, and consequently the improvement of the communication cost should outweigh the loss of synchronization efficiency. Smaller values of α should be considered when the problem is moderately sized.

3.2 Transitive closure

In this section we present two BSP algorithms for the transitive closure problem. The first algorithm is a straightforward parallel coarse-grained imple-

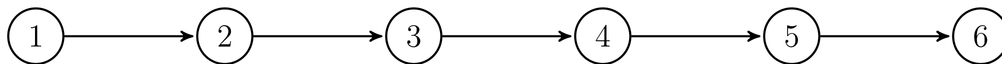


Figure 3.1: A graph with 6 vertices

mentation of Floyd's algorithm, which solves as well the algebraic path problem. The second one approaches the transitive closure problem with the use of dependency graphs. This approach allows us to exploit a large degree of independence.

3.2.1 Blocks Warshall

A straightforward parallel implementation of Warshall's algorithm was presented by Chan et al. [CGPR98] for the transitive closure problem, which also extends to the algebraic path problem.

Initially, a simplistic parallel implementation of the Warshall algorithm was attempted, which unfortunately was proved to be incorrect. In this "naive" approach we partition matrix A into p (p, the number of processors) blocks $\tilde{X}_{I,J}$ of size $(s \times s)$, where $s = n/\sqrt{p}$, and perform on blocks $\tilde{X}_{I,J}$ similar operations as for $A(i,j)$. The algorithm would be:

Naive blocks Warshall algorithm

---

1. **for** $K = 1$ **to** n/s **do**
2. **for all** $1 \leq I, J \leq n/s$ **in parallel do**
3. $\tilde{X}_{I,J} = \tilde{X}_{I,J} \oplus \tilde{X}_{I,K} \otimes \tilde{X}_{K,J}$;

We can prove that this algorithm is incorrect with a simple example. Consider the adjacency matrix A of the graph shown in Figure 3.1. We denote by A' the matrix that is computed if we use the naive algorithm for block size $s = 3$ and by A^* the expected result.

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} A' = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} A^* = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

We get the corrected version if we add an extra statement. We need to compute the closure of each diagonal block and ascertain that each processor

holds this information. The algorithm follows:
Blocks Warshall algorithm

---

1. **for** $K = 1$ **to** n/s **do**
2. **for all** $1 \leq I, J \leq n/s$ **in parallel do**
3. compute $\tilde{X}^*_{K,K}$;
4. $\tilde{X}_{I,J} = \tilde{X}_{I,J} \oplus \tilde{X}_{I,K} \otimes \tilde{X}^*_{K,K} \otimes \tilde{X}_{K,J}$;

We give an intuitive proof of correcteness of the Blocks Warshall algorithm. In the case of the sequential algorithm, alternatively fine-grained version, each path is computed according to the greatest intermediate vertex that it may contain, using the fact that no intermediate vertex may appear twice; since we compute only simple paths. In contrast, in the coarse-grained version in the fourth line of the algorithm is not computed a single path, but a set of paths, namely those that are contained in block $\tilde{X}_{I,J}$. This set of paths is not computed according to a single intermediate vertex, but according to a set of vertices, the vertices that belong to the set $\tilde{X}_K$. As a result, we have to compute all these additional subpaths. We achieve that by computing the closure of block $\tilde{X}_{K,K}$.

We may observe that this algorithm also computes the algebraic path problem, even though the initial fine-grained algorithm did not.

In the cost analysis of this algorithm, we have $n/s = p^{1/2}$ synchronization steps, hence $O(p^{1/2})$. Next, the communication cost of each step is equal to the size of the blocks that are sent to the processors, that is $4 \cdot (s \times s) = 4 \cdot (n/\sqrt{p} \times n/\sqrt{p}) = 4 \cdot n^2/p$. Hence, the communication cost over all steps is $O(n^2/p \cdot p^{1/2}) = O(n^2/p^{1/2})$. Finally, the computation cost of each step is defined by the multiplications between the blocks, which are computed by the McColl-Valiant algorithm, and the computation of the closure of block $\tilde{X}_{K,K}$, which we assume is computed by Warshall's algorithm. The resulting computation cost of each step is $3 \cdot n^3/p^{3/2}$, thus the total computation cost of the algorithm is $O(n^3/p^{3/2} \cdot p^{1/2}) = O(n^3/p)$. To sum up, the BSP cost of the Blocks Warshall algorithm is:

$$W = O(n^3/p) \qquad H = O(n^2/p^{1/2}) \qquad S = O(p^{1/2})$$

3.2.2 Leighton's algorithm

Another coarse-grained parallel algorithm was proposed by Pagourtzis et al. [PPR01] for the transitive closure problem. Initially a fine-grained parallel algorithm was created, which adapted the main idea of a systolic algorithm presented by Leighton [Lei92], by investigating the dependency graph of the computation; a dependency graph represents which operations must be executed before other operations. Next, the coarse-grained algorithm is derived from the fine-grained one.

Let us describe, at first, the main idea of Leighton's algorithm. This systolic algorithm is based on the concept of restricted paths. *Restricted path* is a path from vertex i to some other vertex such that all intermediate vertices have indices smaller than i. We define the following predicates.

$i \rightarrow j$: there is a single edge from i to j

$i \rightsquigarrow j$: there is a restricted path from i to j

$i \xrightarrow{*} j$: there is a path from i to j

We may observe that the transitive closure of a graph can be obtained by computing the predicate $i \xrightarrow{*} j$, for each i, j. For the above defined predicates hold the following two properties:

Property 1

$i \rightsquigarrow j \Leftrightarrow (\exists 1 \leq i_1 < \ldots < i_k < i)\ i \rightarrow i_1 \rightsquigarrow \ldots \rightsquigarrow i_k \rightsquigarrow j \quad \text{or} \quad i \rightarrow j$

Property 2

$i \xrightarrow{*} j \Leftrightarrow (\exists i < i_1 < \ldots < i_k \leq n)\ i \rightsquigarrow i_1 \rightsquigarrow \ldots \rightsquigarrow i_k \rightsquigarrow j \quad \text{or} \quad i \rightsquigarrow j$

Before we continue to the description of the fine-grained algorithm let us define the dependency graph. A *dependency graph* is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where the set of vertices $\mathcal{V}$ is a set of operations and each edge $(u, v) \in \mathcal{E}$ means that operation u should be executed before operation v. Next, we define two basic operations:

Procedure AtomicOperation(i, k, j)
$A(i, j) := A(i, j) \oplus A(i, k) \otimes A(k, j)$;

Procedure MainOperation(i, k)
for all j **in parallel do** AtomicOperation(i, k, j);

We define by $\mathcal{M}(n)$ the set of all main operations, that is all MainOperation(i, k),

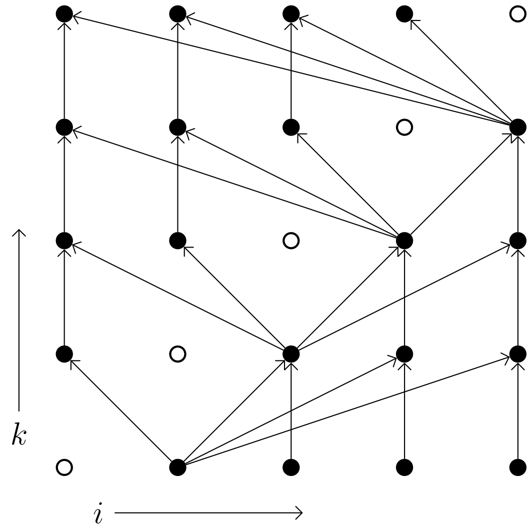


Figure 3.2: The dependency graph $\mathcal{G}(5)$. The black vertex at (i, k) is the operation MainOperation(i, k).

where $1 \leq i, k \leq n$, $i \neq k$. The dependency graph $\mathcal{G}(n)$, whose vertices are elements of $\mathcal{M}(n)$, i.e. each vertex (i, k) corresponds to MainOperation(i, k) is shown in Figure 3.2. We can make the following observations by investigating the derived dependency graph. The vertices below the diagonal correspond to the set of operations related to Property 1, and the nodes above the diagonal correspond to Property 2. Also, we have that for each vertex above the diagonal, except for the top row, there is an edge to its next upper vertex, and for each vertex strictly below the diagonal there are edges to all vertices in the next upper row. This property allows us to split the set of all MainOperations(i, k) into two passes. In the first pass all operations that correspond to vertices below the diagonal are being executed, and during the second pass the operations that correspond to the vertices above the diagonal are being executed. In other words, during the first phase we compute all restricted paths for which Property 1 holds, and during the second phase we compute all paths for which Property 2 holds. By splitting the operations into these two passes we can see that we gain a large degree of independence during the second pass. A property that is very useful in parallel computations, since processors may execute the tasks that are assigned to them without having the extra cost of synchronization. As a result the following fine-grained algorithm is derived:

Leighton's fine-grained algorithm

Pass I

1. **for** $k = 1$ **to** n **do**
2. **for all** $i > k$ **in parallel do** MainOperation(i, k);

Pass II

1. **for all** $i < n$ **in parallel do**
2. *Process* i : **for** $k = i + 1$ **to** n **do** MainOperation(i, k);

In order to obtain a coarse-grained algorithm, based on the fine-grained one, as before we initially divide matrix A into p blocks of size $(s \times s)$, where $s = n/\sqrt{p}$. Hence, we get $\sqrt{p}$ subsets of vertices $\{1, 2, \dots, n\}$ each containing s elements: $V_I = \{(I - 1) \cdot s + 1, \dots, I \cdot s\}$, $1 \leq I \leq n/s$. Once again, we denote by $\tilde{X}_{I,J}$ the block of A that contains all elements $A(i, j)$ such that $i \in V_I$ and $j \in V_J$.

As in the case of Warshall's algorithm, a straightforward transformation of the fine-grained algorithm to a coarse-grained one, i.e. by simply replacing elements $A(i, j)$ by blocks $\tilde{X}_{I,J}$, would not lead to a correct solution, with the same arguments holding. We need to take into account the closure of all the diagonal blocks. In this way, we define the block version of the previous procedures:

Procedure BlockAtomic(I, K, J)
$\tilde{X}_{I,J} = \tilde{X}_{I,J} \oplus \tilde{X}_{I,K} \otimes \tilde{X}_{K,J}$;

Procedure BlockMain(I, K)
if $I = K$ then $\tilde{X}_{K,K} = (\tilde{X}_{K,K})^*$;
for all J **in parallel do** BlockAtomic(I, K, J);

The dependency graph $\mathcal{G}'(n)$ related to BlockMain(I, K) operations is shown in Figure 3.3. The new dependency graph, in comparison to the previous one, has the additional edges from the diagonal element of a row to the rest elements of the row (on the right hand side), as well as, for each diagonal element, except for the first row, there is an edge from its previous lower neighbour. Despite the fact that we have additional edges in $\mathcal{G}'(n)$, we can split computations into two phases, as before, and have the same degree of independence during the second pass. The derived coarse-grained algorithm follows:

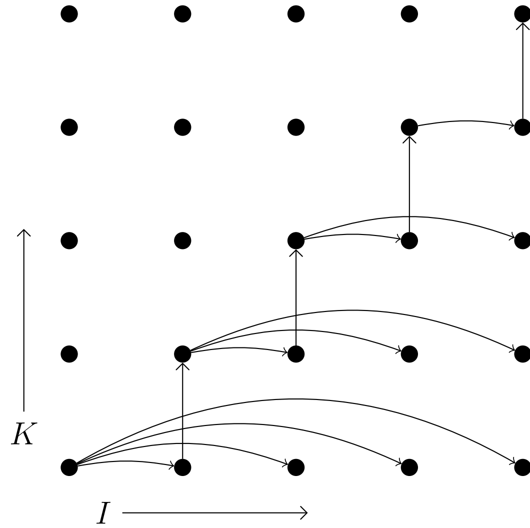


Figure 3.3: Additional edges of $\mathcal{G}'(5)$ (compared to $\mathcal{G}(5)$). The black vertex at (I, K) corresponds to block operation BlockMain(I, K).

Leighton's coarse-grained algorithm

Pass I

1. **for** $K = 1$ **to** n/s **do**
2. BlockMain(K, K);
3. **for all** $I > K$ **in parallel do** BlockMain(I, K);

Pass II

1. **for all** $I < n/s$ **in parallel do**
2. *Process* I : **for** $K = I + 1$ **to** N **do** BlockMain(I, K);

In the cost analysis of this algorithm during the first pass we have $2 \cdot n/s = 2 \cdot p^{1/2}$ synchronization steps, and during the second pass we have $n/s = p^{1/2}$ steps. Hence, the total synchronization cost of the algorithm is $O(p^{1/2})$. Next, the communication cost of each step of both passes is equal to the size of the blocks that are sent to the processors, hence, the cost of each step is $3 \cdot (n/p^{1/2} \times n/p^{1/2}) = 3 \cdot n^2/p$. The total communication cost of the algorithm is $O(n^2/p \cdot p^{1/2}) = O(n^2/p)$. Last, the computation cost of each step consists of the multiplication operations between the sub-matrices, which are computed by the McColl-Valiant algorithm, and the closure operation of the diagonal blocks, which we assume is computed by Warshall's algorithm. The

resulting cost of each step is $3 \cdot n^3/p^{3/2}$, thus, the total computation cost of the algorithm is $O(n^3/p^{3/2} \cdot p^{1/2}) = O(n^3/p)$. To summarize, the BSP cost of Leighton's coarse-grained algorithm is:

$$W = O(n^3/p) \qquad H = O(n^2/p^{1/2}) \qquad S = O(p^{1/2})$$

3.3 All pairs shortest path

In this section two BSP algorithms are being presented for the all pairs shortest path problem. The first algorithm solves a special case of the problem in which we assume that the costs assigned to the edges are nonnegative. The second algorithm extends the method of the first one to solve the general problem.

3.3.1 Tiskin's nonnegative edge costs algorithm

Tiskin proposed a coarse-grained parallel algorithm for the case of nonnegative edge costs [Tis01]. In this algorithm he combined two different methods for solving the all pairs shortest path problem that were proposed earlier.

The first method is the implementation of Dijkstra's algorithm (case of all pairs shortest path) in parallel, [Joh77]. Due to the use of Dijkstra's algorithm we ask for the edges to be assigned with nonnegative costs. To be more specific, each processor is assigned with n/p vertices (the sources), next, the processors apply Dijkstra's algorithm to compute the shortest paths originating in the assigned vertices. The synchronization cost of the resulting algorithm is $S = O(1)$, since, all computations are performed in one step. The communication cost is $H = O(n^2)$, due to the fact that matrix A is sent to all processors. Finally, the computation cost is $W = O(n^2 \cdot n/p) = O(n^3/p)$. The advantage of this algorithm is the low synchronization cost.

The second method implements the principle of path doubling. Since, no shortest path may contain more than n edges, we have that $A^* = A^n$. In path doubling we obtain matrix A^n by repeated squaring in $\log n$ steps, therefore, the synchronization cost of this algorithm would be $S = O(\log n)$. Assuming that matrix multiplications are performed by McColl-Valiant algorithm, the communication cost would be $H = O(n^2/p \cdot \log n)$ and the computation cost $W = O(n^3/p \cdot \log n)$. Tiskin, in the proposed algorithm, used a refined version of path doubling in combination with Dijkstra's algorithm. The description

of the algorithm follows.

Initially, without loss of generality, we assume that all edge and path costs are distinct, thus, all shortest paths are unique. We can achieve that by a small perturbation of edge costs. The main idea of Tiskin's algorithm is to perform path doubling, keeping track not only of path costs, but also of path sizes. *Path size* denotes the number of edges contained in a path. In order to keep the information of path costs and sizes as well, we define a data structure called path matrix. A *Path matrix* X is a matrix in which each entry $X[i, j]$ is either ∞, or corresponds to the cost of a simple path from i to j. In addition, for an integer k, $X(k)$ denotes the matrix of all paths in X of size exactly k. That is,

$$X(k)[i,j] = \begin{cases} X[i,j], & \text{if path } X[i,j] \text{ has size } k \\ \infty, & \text{otherwise} \end{cases}$$

Addition and multiplication of path matrices are defined in the natural way. Let $X(k_1, \ldots, k_s) = X(k_1) \oplus \ldots \oplus X(k_s)$. If the maximum path size of X is m, we have

$$X = X(0, 1, \ldots, m) = X(0) \oplus X(1) \oplus \ldots \oplus X(m)$$

To complete the properties of path matrices we have $X \leq Y$, if $X[i, j] \leq Y[i, j]$ for all i, j. Furthermore, we call an entry $X[i, j]$ trivial, if it is equal to ∞. Last, we call X and Y disjoint, if either $X[i, j]$, or $Y[i, j]$ is trivial for all i, j.

Let us consider the nonnegative all pairs shortest path problem defined by path matix A. In this matrix are contained all shortest paths of size 0, in the diagonal, and of size 1, in the off-diagonal elements. Next, we have that matrix A^k, for an integer k, contains all shortest paths of size at most k. Suppose now, that we have computed A^k for some $1 \leq k < n$. The next step would be to compute all shortest paths of size at most $3k/2$. In order to achieve this we decompose A^k into a disjoint generalized sum:

$$A^k = A^k(0, 1, \ldots, k) = I \oplus A^k(1) \oplus \ldots \oplus A^k(k)$$

We consider now, the upper half of this sum, that is, $A^k(k/2+1) \oplus \ldots \oplus A^k(k)$. In the matrices contained in this sum, the total number of nontrivial elements is at most n^2 (recall that the matrices are disjoint), therefore, the average

number of nontrivial entries per matrix is at most $2n^2/k$. We find a path size l, $k/2 < l \leq k$, with negligible cost of search, such that path matrix $A^k(l)$ contains at most $2n^2/k$ nontrivial elements. Let us return now to our goal, we want to compute all shortest paths of size at most $3k/2$. Consider any shortest path of size in the range $l+1, \ldots, 3k/2$, such a path consists of an initial subpath of size l, and a final subpath of size at most k. For that reason, all shortest paths of size at most $3k/2$ can be computed by $A^k \oplus A^k(l) \otimes A^k$, more precisely:

$$(I \otimes A^k) \oplus (A^k(l) \otimes A^k) = (I \oplus A^k(l)) \otimes A^k \leq A^{3k/2}$$

Hence, for the path doubling phase we want to compute the product $A^k(l) \otimes A^k$, for which we have that matrix $A^k(l)$ contains at most $2n^2/k$ nontrivial elements, therefore the matrix product requires at most $2n^3/k$ element multiplications.

In order to compute the sparse by dense matrix product efficiently, as before, we need to partition the problem into p subproblems. The difference in this algorithm is that Tiskin described a more sophisticated partitioning of the problem into p sparse by dense submatrix multiplications, where all the sparse arguments have an approximately equal number of nontrivial entries. The description of the partitioning follows.

Initially, we partition the set of rows in $A^k(l)$ into $p^{1/3}/k^{1/3}$ equal subsets, such that each subset contains at most $\frac{2n^2}{k^{2/3}p^{1/3}}$ nontrivial entries. This partitioning defines, up to a permutation of rows, a decomposition of the sparse matrix into $p^{1/3}/k^{1/3}$ equal horizontal strips. Each strip defines an $(\frac{n \cdot k^{1/3}}{p^{1/3}} \times n) \times (n \times n)$ sparse by dense matrix multiplication subproblem. Next, consider one of the defined subproblems. We partition the set of columns in the strip into $p^{1/3}/k^{1/3}$ equal subsets, such that each subset contains at most $\frac{4n^2}{k^{1/3}p^{2/3}}$ nontrivial entries. This partitioning defines, up to a permutation of columns, a decomposition of the strip into equal square blocks. Each block defines an $(\frac{n \cdot k^{1/3}}{p^{1/3}} \times \frac{n \cdot k^{1/3}}{p^{1/3}}) \times (\frac{n \cdot k^{1/3}}{p^{1/3}} \times n)$ sparse by dense matrix multiplication subproblem. Last, we partition the set of columns of the dense matrix into $p^{1/3} \cdot k^{2/3}$ equal subsets. This way, we obtain $p^{1/3} \cdot k^{2/3}$ sparse by dense matrix multiplication subproblems of size $(\frac{n \cdot k^{1/3}}{p^{1/3}} \times \frac{n \cdot k^{1/3}}{p^{1/3}}) \times (\frac{n \cdot k^{1/3}}{p^{1/3}} \times \frac{n}{p^{1/3} \cdot k^{2/3}})$.

The total number of sparse by dense matrix multiplication subproblems is p, in which the sparse argument of each subproblem contains at most $\frac{4n^2}{k^{1/3}p^{2/3}}$ nontrivial entries. As in the case of finding path size l, the cost of computing the partitioning is negligible.

The path doubling phase is stopped after at most $\log_{3/2} p$ steps, when matrix A^p has been computed. The BSP cost of each step during this phase consists of the synchromization cost $S = O(1)$, the communication cost $H = O(n^2/(k^{1/3} \cdot p^{2/3}))$, and the computation cost $W = O(n^3/(k \cdot p))$.

In the next phase Dijkstra's algorithm is adapted, the description of the implementation follows. For some q, $1 \leq q \leq p$, there exists some matrix $A^p(q)$ which contains at most n^2/p nontrivial elements. This matrix is being broadcasted to all processors. Next, each processor receives matrix $A^p(q)$, picks n/p vertices and computes all shortest paths originating in these vertices, by n/p independent runs of Dijkstra's algorithm. After the completion of this phase, the closure of matrix $A^p(q)$ has been computed. As a result, matrix $A^p(q)^*$ contains all shortest paths whose sizes are multiples of q; along with some other paths. The synchronization cost of this phase is $H = O(1)$, the communication cost is $H = O(n^2/p)$, and the computation cost is $W = O(n^2 \cdot n/p) = O(n^3/p)$.

In the final phase, one matrix product is being computed, based on the following observation. Any shortest path in A^* consists of an initial subpath of size that is a multiple of q, and a final subpath of size at most $q \leq p$ (recall that $1 \leq q \leq p$). Therefore, for the initial matrix A all the shortest paths can be computed by the matrix product:

$$A^* = A^p(q)^* \otimes A^p$$

Before we proceed with the summary of the algorithm and its cost analysis, we can achieve a reduction in the synchronization cost. We terminate the path doubling phase after fewer than $\log_{3/2} p$ steps, to be more precise, for $1 \leq r \leq p^{3/2}$ we can find a q such that matrix $A^r(q)$ contains at most n^2/r nontrivial elements. As a result, the communication cost of the second phase changes to $H = O(n^2/r)$.

To sum up the procedure:

Tiskin's all pairs shortest path algorithm (nonnegative case)

Parameters: integer $n \geq p$; integer r, $1 \leq r \leq p^{2/3}$

Description: The computation proceeds in three stages:

First stage
Compute A^r by at most $\log_{3/2} r$ rounds of path doubling.

Second stage
Select q, $0 < q \leq r$ such that $A^r(q)$ contains at most n^2/r nontrivial entries.

Broadcast $A^r(q)$ and compute the closure $A^r(q)^*$ by n independent runs of Dijkstra's algorithm, n/p runs per processor.

Third stage
Compute the product $A^r(q)^* \otimes A^r = A^*$.

In the cost analysis of this algorithm we have, for the first stage the computation and communication costs are dominated by the cost of the first step ($k = 1$), hence, $W = O(n^3/p)$, $H = O(n^2/p^{2/3})$, and the synchronization cost is $S = O(\log r)$. For the second stage the costs are $W = O(n^3/p)$, $H = O(n^2/r)$ and $S = O(1)$. Finally, the cost of the third stage is $W = O(n^3/p)$, $H = O(n^2/r)$ and $S = O(1)$ (McColl-Valiant matrix multiplication algorithm). Therefore, the total BSP cost of Tiskin's algorithm is

$$W = O(n^3/p) \qquad H = O(n^2/r) \qquad S = O(\log r)$$

In the case of this algorithm, parameter r allows us to trade off the costs of communication and synchronization.

Before we proceed to the description of a more general algorithm we present a modification in the second stage of the procedure. This variation will be useful in the following algorithm. During the second phase of the procedure instead of using matrix $A^r(q)$ with at most n^2/r nontrivial elements, we use matrix $A^r(r)$. In order to avoid broadcasting the whole matrix, we may represent it as the matrix product

$$A^r(r) = A^r(q) \otimes A^r(r - q)$$

For some q, $0 \leq q < r/2$, the disjoint sum $A^r(q) \oplus A^r(r-q)$ contains at most $2n^2/r$ nontrivial elements, otherwise matrix $A^r(r)$ should contain more than n^2 elements. As a result, the second stage of the algorithm can be replaced by broadcasting the disjoint sum of matrices $A^r(q)$ and $A^r(q - r)$, next, recovering their product $A^r(r)$, and finally, computing the closure $A^r(r)^*$.

3.3.2 Tiskin's general edge costs algorithm

Tiskin extended the previous algorithm to solve the all pairs shortest path problem with general edge costs, [Tis01]. This more general algorithm though, has an increase in its cost. In contrast to the previous case, we cannot use Dijkstra's algorithm due to the fact that edges may be assigned with negative costs. The proposed technique by Tiskin, to overcome this difficulty, is

to replace Dijkstra's algorithm during the second stage with an extra phase of sequential path doubling. The description of the generalized algorithm follows.

As in the previous case, the extended algorithm consists of three stages. In the first stage, we implement parallel path doubling in order to compute path matrix A^{p^2}. Hence, the procedure is stopped after $2\log_{3/2} p$ rounds of path doubling.

We introduce, now, a slightly different path matrix. Define

$$A^{p^2}((p)) = A^{p^2}(p, 2p, \ldots, p^2)$$

and

$$A^{p^2}((p) - q) = A^{p^2}(p - q, 2p - q, \ldots, p^2 - q)$$

To be more precise, path matrix A^{p^2} contains all shortest paths whose size is a multiple of p in the range $p, \ldots, p^2$.

In the second stage, we represent path matrix A^{p^2} as the product

$$A^{p^2}((p)) = A^{p^2}(q) \otimes A^{p^2}((p) - q)$$

We have that for some q, $0 \leq q < p/2$, the disjoint sum $A^{p^2}(q) \oplus A^{p^2}((p) - q)$ contains at most $2n^2/p$ nontrivial elements. As described in the previous subsection, we collect matrices $A^{p^2}(q)$ and $A^{p^2}((p) - q)$ in a single processor and recover their product $A^{p^2}((p))$. Next, for some l, $l \in \{(p/2) \cdot p, (p/2 + 1) \cdot p, \ldots, p^2\}$, matrix $A^{p^2}((p))(l)$ contains at most $2n^2/p$ nontrivial elements. We compute the closure of $A^{p^2}((p))$, for which holds that $A^{p^2}((p))^* = A^{p^2}(p)^*$, by sequential path doubling, in which the first step computes the generalized sum

$$\left(I \otimes A^{p^2}((p))\right) \oplus \left(A^{p^2}((p))(l) \otimes A^{p^2}((p))\right) \leq A^{3p^2/2}((p))$$

In the final stage, since any shortest path in A^* consists of an initial subpath of size that is a multiple of p and a final subpath of size at most p, it suffices to compute the matrix product

$$A^* = A^p(q)^* \otimes A^p$$

In this algorithm, early termination of the parallel path doubling stage is not considered; since the average number of nontrivial elements per matrix would be increased, leading to an increase of the communication cost, as well as to the computation cost of the second stage.

To sum up the algorithm:

Tiskin's all pairs shortest path algorithm (general case)

Parameters: integer $n \geq p$

Description: The computation proceeds in three stages

First stage
Compute A^{p^2} and $A^{p^2}((p))$ by at most $2\log_{3/2} p$ rounds of path doubling.

Second stage
Select q, $0 \leq q < p/2$ such that $A^{p^2}(q) \oplus A^{p^2}((p) - q)$ contains at most n^2/p nontrivial entries. Collect $A^{p^2}(q) \oplus A^{p^2}((p) - q)$ in a single processor and recover $A^{p^2}((p)) = A^{p^2}(q) \otimes A^{p^2}((p) - q)$. Compute the closure $A^{p^2}((p))^* = A^{p^2}(p)^*$ by sequential path doubling.

Third stage
Compute the product $A^{p^2}(p)^* \otimes A^{p^2} = A^*$.

In the cost analysis of Tiskin's extended algorithm we have, during the first stage the computation and communication costs are dominated by the cost of the first step ($k = 1$), hence, $W = O(n^3/p)$, $H = O(n^2/p^{2/3})$, and the synchonization cost is $S = O(\log p)$. For the second stage, once again, the computation cost is dominated by the cost of its first step, thus, $W = O(n^3/p)$, the communication cost is $H = O(n^2/p)$ and the syncronization cost is $S = O(1)$. Finally, the costs of the third stage are $W = O(n^3/p)$, $H = O(n^2/p^{2/3})$ and $S = O(1)$. Therefore, the total BSP cost of the algorithm is

$$W = O(n^3/p) \qquad H = O(n^2/p^{2/3}) \qquad S = O(\log p)$$

Tiskin's algorithm for the general case, computes correctly not only the all pairs shortest path problem, but also any instance of the algebraic path problem in which generalized addition ($\oplus$) is idempotent.

Conclusion

In this thesis we present the algebraic path problem and several of its instances. Initially, we describe sequential algorithms for solving the problem and special cases of it. Next, we present parallel coarse-grained algorithms in order to examine the benefits of parallelism for this problem, and compare their complexities. In Leighton's algorithm we observe that despite the large degree of independence that operations have, there is no impact on the BSP cost of the algorithm. This lack of exploitation leads to the need of further investigation of the algorithm. One possible direction would be to adapt the algorithm to other parallel models. Another direction would be to use different approaches for obtaining coarse-grained algorithms for the transitive closure problem [GPPR03]. Finally, one might try to use more refined methods of matrix multiplication as in Tiskin's algorithm, or to use fast matrix multiplication (current bound on $\omega < 2.3727$ [Wil12]).

Bibliography

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley series in computer science and information processing, Addison-Wesley Pub. Co., 1974.

[BC75] R. C. Backhouse and B. A. Carre, *Regular algebra applied to path-finding problems*, IMA Journal of Applied Mathematics **15** (1975), no. 2, 161–186.

[Bis04] R.H. Bisseling, *Parallel scientific computation: A structured approach using bsp and mpi*, Oxford scholarship online, OUP Oxford, 2004.

[CGPR98] Ken Chan, Alan Gibbons, Marcelo Pias, and Wojciech Rytter, *On the pvm computations of transitive closure and algebraic path problems*, Recent Advances in Parallel Virtual Machine and Message Passing Interface (Vassil Alexandrov and Jack Dongarra, eds.), Lecture Notes in Computer Science, vol. 1497, Springer Berlin Heidelberg, 1998, pp. 338–345.

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken, *Logp: towards a realistic model of parallel computation*, Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming (New York, NY, USA), PPOPP '93, ACM, 1993, pp. 1–12.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms*, 2nd ed., McGraw-Hill Higher Education, 2001.

[Dij59] E.W. Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik **1** (1959), no. 1, 269–271 (English).

[Fin92] Eugene Fink, *A survey of sequential and systolic algorithms for the algebraic path problem.*

[Flo62] Robert W. Floyd, *Algorithm 97: Shortest path*, Commun. ACM **5** (1962), no. 6, 345–.

[FW78] Steven Fortune and James Wyllie, *Parallelism in random access machines*, Proceedings of the tenth annual ACM symposium on Theory of computing (New York, NY, USA), STOC '78, ACM, 1978, pp. 114–118.

[GPPR03] Alan Gibbons, Aris Pagourtzis, Igor Potapov, and Wojciech Rytter, *Coarse-grained parallel transitive closure algorithm: Path decomposition technique*, The Computer Journal **46** (2003), no. 4, 391–400.

[Joh77] Donald B. Johnson, *Efficient algorithms for shortest paths in sparse networks*, J. ACM **24** (1977), no. 1, 1–13.

[KL78] H.T. Kung and C.E. Leiserson, *Systolic arrays for (vlsi)*, CMU-CS, Carnegie-Mellon University, Department of Computer Science, 1978.

[Kle56] S.C. Kleene, *Representation of events in nerve nets and finite automata*, Automata Studies (C.E. Shannon and J. McCarthy, eds.), Annals of mathematics studies, no. 34, Princeton University Press, 1956, pp. 3–41.

[Leh77] Daniel J. Lehmann, *Algebraic structures for transitive closure*, Theoretical Computer Science **4** (1977), no. 1, 59 – 76.

[Lei92] F.T. Leighton, *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes*, no. τ. 1, M. Kaufmann Publishers, 1992.

[McC95] W.F. McColl, *Scalable computing*, Computer Science Today (Jan Leeuwen, ed.), Lecture Notes in Computer Science, vol. 1000, Springer Berlin Heidelberg, 1995, pp. 46–61.

[McC96] William F. McColl, *Universal computing*, Euro-Par'96 Parallel Processing (Luc Boug, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, eds.), Lecture Notes in Computer Science, vol. 1123, Springer Berlin Heidelberg, 1996, pp. 25–36.

[Pap95] C.H. Papadimitriou, *Computational complexity*, Mathematics / a second level course, no. τ. 3-4, Addison-Wesley, 1995.

[PPR01] Aris Pagourtzis, Igor Potapov, and Wojciech Rytter, *Pvm computation of the transitive closure: The dependency graph approach*, Recent Advances in Parallel Virtual Machine and Message Passing Interface (Yiannis Cotronis and Jack Dongarra, eds.), Lecture Notes in Computer Science, vol. 2131, Springer Berlin Heidelberg, 2001, pp. 249–256 (English).

[Rot85] Gunter Rote, *A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion)*, Computing **34** (1985), no. 3, 191–219 (English).

[Rot90] G. Rote, *Path problems in graphs*, 155–189 (English).

[Tis01] Alexandre Tiskin, *All-pairs shortest paths computation in the bsp model*, Automata, Languages and Programming (Fernando Orejas, PaulG. Spirakis, and Jan Leeuwen, eds.), Lecture Notes in Computer Science, vol. 2076, Springer Berlin Heidelberg, 2001, pp. 178–189 (English).

[Tis02] A. Tiskin, *Bulk-synchronous parallel gaussian elimination*, Journal of Mathematical Sciences **108** (2002), no. 6, 977–991 (English).

[Val90] Leslie G. Valiant, *A bridging model for parallel computation*, Commun. ACM **33** (1990), no. 8, 103–111.

[War62] Stephen Warshall, *A theorem on boolean matrices*, J. ACM **9** (1962), no. 1, 11–12.

[Wil12] Virginia Vassilevska Williams, *Multiplying matrices faster than coppersmith-winograd*, Proceedings of the 44th symposium on Theory of Computing (New York, NY, USA), STOC '12, ACM, 2012, pp. 887–898.

[Yoe61] Michael Yoeli, *A note on a generalization of boolean matrix theory*, The American Mathematical Monthly **68** (1961), no. 6, pp. 552–557 (English).

[ZF12] E. Zachos and D. Fotakis, *Algorithms and complexity*, Lecture Notes, NTUA publications, 2012.