Computational Complexity of Sequential and Parallel Algorithms

Lydia Kronsjö

Library of Congress Cataloging in Publication Data:

Kronsjö, Lydia.
  Computational complexity of sequential and
parallel algorithms.

  (Wiley series in computing)
  Includes index.
  1. Electronic digital computers—Programming.
2. Mathematics—Data processing. 3. Algorithms.
4. Parallel processing (Electronic computers)
5. Sequential processing (Computer science)
I. Title. II. Series
QA76.6.K763 1986 001.64'2 85–9566

ISBN 0 471 90814 2

British Library of Cataloguing in Publication Data:

Kronsjö, Lydia
  Computational complexity of sequential and
  parallel algorithms.—(Wiley series in computing)
  1. Algorithms
  I. Title
  511'.8 QA9.58

ISBN 0 471 90814 2

Printed and bound in Great Britain

Preface

In the 1930s the notion of an algorithm was formulated as a precise mathematical concept. The Turing machine was conceived and with remarkable simplicity captured the mechanism of problem solving; for the first time the notion of an algorithmically computable function was formalized. Now, 50 years later, the science of computing is a well-established and vigorously researched field; we have algorithms which solve large and complex problems on our powerful computers in seconds, we are aware of the existence of those problems which cannot be solved by a computer, and we have encountered problems which we cannot solve in realistic time limits in spite of having very powerful computers. Until recently, algorithms were constructed under the assumption that a computer executes the algorithm's instructions in sequential order, one after another. However, the latest advances in computer technology have brought about a new generation of computing machines—parallel computers—which are able to perform several computing tasks simultaneously, and all the indications are that the future belongs to these new machines because they offer computing power which is higher than any sequential machine may physically be capable of. For the science of algorithms it means that a new, conceptually different, approach is needed in the design of algorithms. At the same time, undoubtedly, the wealth of knowledge which has been acquired by the era of sequential algorithms should prove very useful in the new developments.

This book consists of two parts. In Part One some conceptually important achievements in the design methodology of sequential algorithms are presented. In Part Two new solutions in the field of parallel algorithms are introduced. Whenever possible both sequential and parallel algorithms for the same class of problems are treated. The author's purpose was to present a concise treatment of the important results from the theory and applications of sequential algorithms and, in parallel, give an introduction to the fast developing area of parallel algorithms. Such presentation of material in one book is aimed at giving the reader an opportunity to compare, contrast, and appreciate unique features of the two kinds of computing environment. A comprehensive list of references is included which may be of service to the reader intending to go further.

The book can be recommended for second- and third-year undergraduate and postgraduate students in mathematics, computer science, and software engineering. Chapters 1 and 9 are introductory to Parts One and Two respectively; apart from these chapters, the book can be followed in any selective order as the chapters are essentially self-contained.

I would like to warmly acknowledge the work of several of my students who contributed to the material of the book by developing examples which enhanced the algorithmic exposition and, in some cases, by devoting detailed analysis to some arguments of our discussion. Among them, Martin Edge worked with Pan's algorithms, Matthew Levin carried out an excellent project on asynchronous algorithms for solving non-linear systems, Nilden Eminer researched into echo algorithms, and Stephen Lee produced a crisp project on parallel sorts. My colleague Dennis Parkyn kindly read most of the manuscript. The sustained interest and encouragement of my close friends is most gratefully appreciated. To my son Tim, husband Tom, and his father Erik go the last but probably deepest thank you for their always being close and understanding.

LYDIA KRONSJÖ
Birmingham, 2 February 1985

Contents

Part One

Sequential Algorithms

1

Introduction

Computational complexity addresses itself to the quantitative aspects of the solution of computational problems. Its concern is with the development of techniques for bounding the amount of computational resources which are necessary and sufficient to solve certain classes of problems algorithmically.

A typical problem in computational complexity may be characterized as follows.

We have a class of similar computational tasks, a problem P, e.g. inversion of matrices, parsing of strings, sorting of files, where the parameter n is a measure of an instance of P, which can be the order of a matrix to be inverted, or the length of a string to be parsed, or the size of the file to be sorted in order; a measure typically 'measures' the number of inputs, x_1, x_2, $\ldots$, x_n.

We also have a basic set of operations for carrying out the computation task in hand, e.g. the set of arithmetic operations $+$, $-$, $*$, $/$, the operation of comparison of two elements, the operation of fetching a record from computer memory, etc.

A step-by-step procedure for solving the problem is called an algorithm. A typical pattern of research in computational complexity is the design and development of algorithms which solve the problem with a reduced number of basic operations and the establishment of arguments which allow us to determine bounds on the possible extent of that reduction. The latter is a particularly difficult problem, and for many present-day problems a gap remains between the requirements of the existing algorithms and those of the minimum bounds for solving the problem.

At a certain level of detail we can usefully distinguish between the algorithm—an abstraction describing a process of computation that can be implemented on many computers—and the computer program, which is an implementation of the algorithm for a particular machine. Algorithms are abstractions whose generality is intended to transcend the specifics of any implementation. We shall not develop Pascal programs, but rather strive to describe algorithms in a form unencumbered by any preordained syntax or semantics, since the algorithms are aimed at thinking readers rather than computers.

There are often several algorithms which solve P, and we may ask the following questions:

(a) What is a 'good' algorithm for solving P?

To answer this question we may seek to compare the relative efficiencies of different algorithms.

(b) What is the minimum number of basic operations required to solve P?

We may then seek to find specific algorithms that solve the problem with the minimum number of operations.

(c) Is the problem perhaps such that no algorithm will solve it in a practically reasonable time?

The problems with a positive answer to this question are called *intractable*. The idea of an intractable problem was independently introduced by Cobham (1964) and Edmonds (1965). It helped to divide all problems for which there exist algorithms, into two distinct classes: efficient or practically solvable algorithms which usually correspond to some significant structure in the problem, and inefficient algorithms which often reflect a brute-force search and whose execution time explodes into exponential and superexponential growth.

In order to make an informed choice between algorithms which solve the same problem, systematic information about the algorithms' performance is needed.

1.1 Measures of Complexity

One way to compare the algorithms' performance is to compute some measure of their efficiency. To be useful, this measure should be machine independent. Good algorithms tend to remain good if they are expressed in different programming languages or run on different machines.

The two most useful measures are the time required to execute the algorithm and the memory needed by the algorithm. These measures are generally expressed as a function of the problem size.

The time complexity function is the computational time requirements which in practice are often calculated as the number of times a particular computer operation occurs, e.g.

```
(i)   for $i := 1$ to $n$ do
          $x_i := x_i + 1$              is $O(n)$—read as 'of order n'.
(ii)  for $i := 1$ to $n$ do
          for $j := 1$ to $n$ do
              $x_{ij} := 2x_{ij}$       is $O(n^2)$.
```

The space complexity function is the space requirements of the algorithm. This may be for the storage of matrices, intermediate data, etc. and the function represents the peak amount required.

On the program level, other measures, such as the program length, which is indicative of computation time, and the depth of the program, i.e. the number of layers of concurrent steps into which the problem can be decomposed, are useful. Depth corresponds to the time that the program would require under parallel computation.

The time of the algorithm and storage space are important measures and are particularly appropriate if the algorithm (program) is to be run often. The time of the algorithm is the factor that restricts the size of problems which can be solved by computer and the program length in some sense measures the simplicity of an algorithm. Tarjan (1978) calls an algorithm a simple algorithm if it has a short program and a short correctness proof, and an elegant algorithm if it has a short program and a long correctness proof. The program length as a measure of the algorithm's efficiency is most appropriate if programming time is important or if the program is to be run frequently.

1.2 Computer Model

Any specific complexity measure, i.e. execution time, storage space, program length, program depth, assume some model of an effective 'step' in a computation process, that is, a computer model for measuring the computation time/space of an algorithm.

The fame of the first computer model is attributed to the celebrated Turing machine of the mid-1930s, which was conceived to be an automaton equipped with an infinite supply of paper tape marked off in square regions, and capable of just four actions: it could move the tape one square, left or right, it could place a mark in a square, it could erase a mark already present, and at the end of a calculation it could halt. These operations were to be performed according to a sequence of instructions built into the internal mechanism. The machine was a conceptual device for automatically solving problems in mathematics and logic. With his machine, Turing for the first time rigorously formalized the notion of an algorithmically computable function.

Reduced to its essentials the Turing machine is a language for stating algorithms as powerful in principle as the more sophisticated languages now employed for communicating with computers. Several computer models have been proposed since. Following the Turing concept, many computer models assume a sequential (linear tape) storage media. The simplicity of such models makes them useful in theoretical studies of computational complexity, but serial storage media implies artificial restrictions from the point of view of efficient implementation of algorithms. Real computers have random access memories, so a better computer model would be a random-access machine. The question seems to be how better to conceive such a model.

One model of a random access machine (RAM), due to Cook and Reckhow (1973), is an abstraction of a general-purpose digital computer. The memory of the machine consists of an array of n words, each of which is able to hold a single integer. The storage words are numbered consecutively from 1 to n. The number of a storage word is its address.

The machine also has a fixed finite set of registers, each able to hold an integer. For problems involving real numbers, storage words and registers are allowed to hold real numbers. In one step, the machine can transfer the contents of a register to a storage word whose address is in a register, or transfer to a register the contents of a storage word whose address is in a register, or perform an arithmetic operation on the contents of two registers, or compare the contents of two registers.

A program of fixed finite length specifies the sequence of operations to be carried out. The initial configuration of memory represents the input data, and the final configuration of memory represents the output.

Various modifications and expansions to the basic model were introduced by Aho *et al.* (1974). Other machine models were suggested by Kolmogorov (1953), Kolmogorov and Uspenskii (1963), Knuth (1968), Schönhage (1973), and Tarjan (1977). Knuth, Schönhage and Tarjan first developed and subsequently improved the so-called pointer machine. As distinct from a RAM machine (which has a finite fixed memory and a set of registers' structure) a pointer machine consists of an extendable collection of nodes, each divided into a fixed number of named fields. A field holds a number or a pointer to a node. In order to carry out an operation on the field in a node, the machine must have in a register a pointer to the node. The operations which the machine can perform are fetching from a node field, storing into a node field, creating a node, and destroying a node; but there is no address arithmetic on a pointer machine and hence algorithms that require such arithmetic, e.g. hashing, cannot be implemented on such a machine. A pointer machine is a less powerful model than a RAM machine, but nevertheless it is useful for the lower bound studies of a great many algorithms, notably for the list processing algorithms (Tarjan, 1983).

A more recent Schönhage's storage modification machine (Schönhage, 1980) is organized in such way that it can be viewed either as a Turing machine that builds its own storage structure or as a unit cost (step) RAM machine that can only copy, add, or subtract one, or store or retrieve in one step.

All machine models described share two properties: they carry out one step at a time, i.e. they are sequential, and the future behaviour of the machine is uniquely determined by its present configuration. In later sections we shall discuss extensions to these machine models which incorporate the concepts of non-determinism and parallel computation. Non-deterministic machine models are particularly useful in theoretical studies of computational complexity (Aho *et al.*, 1974; Garey and Johnson, 1979). On the other hand, the novel machine architectures which have been made possible by very large scale integration (VLSI) give rise to a new type of algorithms, parallel algorithms, which are becoming more and more important.

Part One of the book deals with the ideas and achievements in the field of sequential algorithms, stressing the importance of non-determinism in coping with computationally difficult problems. In Part Two we discuss several problems for which efficient and interesting parallel algorithms have been developed.

2

Arithmetic Complexity of Computations

The arithmetic complexity of computations concerns the algorithms where the set of basic operations required for carrying out the task of computing the solution consists of arithmetic operations, $+$, $-$, $*$, $/$, and perhaps extended to include $\surd$, max or min.

The two major questions of arithmetic complexity—what is the minimum number of arithmetic operations needed to perform the computation and how can we obtain a better algorithm when improvement is possible?—pertain to any computation which may be carried out using the set or subset of arithmetic operations, $+$, $-$, $*$, $/$.

The major classes of arithmetic problems include algebraic processes, such as solution of linear systems, matrix multiplication and inversion, evaluation of the determinant, evaluation of a polynomial, at a point or at n points, and evaluation of the polynomial derivatives, iterative computations, e.g. the root-finding problem, solution of linear systems, and multiplication of two integers.

More recent problems in arithmetic complexity are explicitly concerned with the binary representation of a problem within a computer and relate the requirements of the numerical accuracy of computations. Pan (1980) has formulated the following matrix problem. The entries of matrices are numbers given with a certain precision p in binary form. Find the lower and upper bounds on the number of bit operations involved in the evaluation (with another given precision q) of the product of the two matrices.

2.1 Discoveries of the 1960s

The 1960s are one of the major milestones in the history of computational complexity. During this decade three very surprising algorithms were discovered—for the multiplication of two integers, for computing the discrete Fourier transform, and for the product of two matrices. As Winograd (1980) has aptly noted, all three computational problems have been known for a very long time, and the discovery of more efficient algorithms for their execution was an exciting result which encouraged further investigations of the limits on the efficiency of algorithms.

Today, the importance of these discoveries is not only historic; the new algorithm for computing the Fourier transform has had indeed a profound impact on the way many computations are being performed. Perhaps one of the most remarkable uses of the Fast Fourier transform (FFT) algorithm is in medical analysis, the process known as computerized tomography. We shall briefly outline the three historical results.

2.2 Product of Integers

In 1962 two Russian mathematicians, Karatsuba and Ofman, published a paper in which a new way was given for multiplying two large integers. Using the usual method, as we know it, the product of two n-digit numbers is obtained by performing n^2 multiplications of single digits and about the same number of single-digit additions. The method of Karatsuba and Ofman computes the product in $O(n^{\log 3})$ time and is based on the following idea.

Let x and y be two n-digit numbers, where $n = 2m$, an even number. If b denotes the base then

$$x = x_0 + x_1 b^m, \qquad y = y_0 + y_1 b^m , \tag{2.2.1}$$

where x_0, x_1, y_0, and y_1 are m-digit numbers. The product w is then expressed as

$$\begin{aligned} z = xy &= (x_0+x_1b^m)\,(y_0+y_1b^m) \\ &= x_0y_0 + (x_0y_1+x_1y_0)b^m + x_1y_1b^{2m} \end{aligned} \tag{2.2.2}$$

and its computation can be viewed as consisting of four steps: (1) compute x_0y_0; (2) compute $x_0y_1+x_1y_0$; (3) compute x_1y_1; (4) perform $n = 2m$ single-digit additions.

Multiplication by the power of 2 in the binary environment represents a simple shifting operation and is ignored in this analysis. Thus $4M$ (M = multiplication) of m-digit numbers and $2A$ (A = addition) of $2m$-digit numbers is required to obtain w.

This is the same as the estimate on the number of operations in the usual method. The key to the new algorithm is the way in which x_0y_0, $x_0y_1+x_1y_0$, and x_1y_1 are computed. Their computation is based on the identities

$$\begin{aligned} x_0y_0 &= x_0y_0, \\ x_0y_1+x_1y_0 &= (x_0-x_1)(y_1-y_0)+x_0y_0+x_1y_1, \\ x_1y_1 &= x_1y_1, \end{aligned} \tag{2.2.3}$$

from which it follows that to obtain the left-hand-side values of the identities one needs to find three products of two m-digit numbers and carry out some additions and subtractions. If the addition or subtraction of two single-digit numbers with the possibility of carry or borrow is taken as a unit of addition then the product of two n-digit numbers uses $3M$ of m-digit numbers and $4n$ units of addition.

The same scheme can be used to obtain each of the products x_0y_0, $(x_0-x_1)(y_0-y_1)$, and x_1y_1. This leads to the result that for $n = 2^s$, a power of 2, the product of two $n = 2^s$ digit numbers is obtained using 3^sM of single-digits and $8(3^s-2^s)$ units of addition, assuming the initial condition that for $m = 2^0 = 1$ only one single digit M and no units of addition are needed.

When n is not a power of 2, the numbers can be 'padded' by adding enough leading zeros. So the formulae are still valid if we take $s = \lceil \log_2 n \rceil$ for any n.

In summary, the method shown for computing the product of two n-digit numbers uses at most $3*3^{\log_2 n} = 3n^{\log_2 3} = 3n^{1.59}$ single-digit M and $8*3*3^{\log_2 n} = 24n^{\log_2 3} = 24n^{1.59}$ units of addition.

The more detailed analysis can reduce the constants 3 and 24, but the order of the function on the number of operations will remain $n^{\log_2 3}$. Since the Karatsuba and Ofman paper the result has been improved by several authors. The currently fastest asymptotic performance time for the number product is $O(n \log n \log \log n)$ and was devised on a multitape Turing machine by Shönhage and Strassen (1971) using FFT.

2.3 The Fast Fourier Transform

Given is a set of n points

$$a_0, a_1, a_2, \ldots, a_{n-1}.$$

The discrete Fourier transform (DFT) on these points is a set of n points

$$A_0, A_1, A_2 \ldots, A_{n-1},$$

where A_p's are computed by the formula

$$A_p = \sum_{q=0}^{n-1} w^{pq} a_q, \qquad p = 0,1,\ldots, n-1, \tag{2.3.1}$$

and where w is the nth root of unity, that is $w = e^{2\pi i/n}$, $i = \sqrt{-1}$.

Straightforward computation of each A_p uses $n-1$ complex M and $n-1$ complex A, thus yielding a computational procedure of complexity $O(n^2)$.

In 1965 Cooley and Tukey observed that whenever $n = rs$ is a composite number a more efficient method of computation exists. Each p in $0 \leq p \leq n-1$ can be written uniquely as

$$p = p_1 + p_2 r, \qquad 0 \leq p_1 < r, \qquad 0 \leq p_2 < s, \tag{2.3.2}$$

an each q in $0 \leq q \leq n-1$ as

$$q = q_1 s + q_2, \qquad 0 \leq q_1 < r, \qquad 0 \leq q_2 < s. \tag{2.3.3}$$

Therefore

$$\begin{aligned} A_{p_1+p_2 r} &= \sum_{q_1=0}^{r-1} \sum_{q_2=0}^{s-1} w^{(p_1+p_2 r)(q_1 s+q_2)} a_{q_1 s+q_2} \\ &= \sum_{q_1=0}^{r-1} \sum_{q_2=0}^{s-1} w^{p_1 q_1 s} w^{p_1 q_2} w^{p_2 q_1 rs} w^{p_2 r q_2} a_{q_1 s+q_2} \\ &= \sum_{q_1=0}^{r-1} \sum_{q_2=0}^{s-1} (w^s)^{p_1 q_1} w^{p_1 q_2} (w^r)^{p_2 q_2} a_{q_1 s+q_2}, \end{aligned}$$

since

$$w^{rs} = w^n = 1,$$

$$= \sum_{q_2=0}^{s-1} (w^r)^{p_2 q_2} \left[w^{p_1 q_2} \sum_{q_1=0}^{r-1} (w^s)^{p_1 q_1} a_{q_1 s+q_2} \right]. \tag{2.3.4}$$

Hence to compute each A_p we need to

(i) Compute the DFT on r points,

$$b_{p_1,q_2} = \sum_{q_1=0}^{r-1} (w^s)^{p_1 q_1} a_{q_1 s+q_2}. \tag{2.3.5}$$

(ii) Compute the factor

$$c_{p_1,q_2} = w^{p_1 q_2} b_{p_1,q_2}, \tag{2.3.6}$$

(iii) Compute the DFT on s points,

$$A_p = \sum_{q_2=0}^{s-1} (w^r)^{p_2 q_2} c_{p_1,q_2}. \tag{2.3.7}$$

This requires $(r+s-1)$ M and $(r+s-2)$ A, and to compute n values A_p we shall need $n(r+s-1)$ M and $n(r+s-2)$ A.

If r and s are themselves composite numbers we can use the same idea to compute the DFT of r or s points. In particular, when $n = 2^s$ is a power of 2 this computation process uses $2^s s = n \log_2 n$ M and $2^s(s-2) = n(\log_2 n - 2)$ A. Thus the new algorithm, known as the FFT, is of complexity $O(n \log n)$, a staggering reduction as compared with $O(n^2)$.

As can be observed, the FFT takes advantage of the periodic nature of sine and cosine functions, i.e. $w^{\alpha i} = \sin\alpha + i \cos\alpha$, to greatly reduce the number of multiplications required in evaluating the DFT.

The tremendous reduction in computing time of the DFT which the FFT algorithm offers makes the DFT one of the most powerful mathematical tools used in the solution of many important practical problems. Of all the applications, perhaps the greatest effect on the world at large has been in the area of diagnostic medicine: the so-called computerized tomography (CT) has revolutionized radiology.

The Computerized Axial Tomography (CAT) Scanning and the FFT

In CT the FFT algorithm is used in reconstruction of an image of human body cross-section from data collected by measuring the intensity of X-ray beams transmitted through the cross-section, see Fig. 2.3.1(a). A tomogram is a picture of a slice—a display of an anatomical plane sectioning the body at a given orientation. In the CT method the information from different views