ON THE PARALLEL IMPLEMENTATION OF THE LEHMAN FACTORING ALGORITHM

DAVID JOHN PATERSON HARE

A Thesis Submitted to the Faculty of Science
of the University of Glasgow

for the degree of
Doctor of Philosophy

July 1985

ProQuest Number: 13834209

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 13834209

Published by ProQuest LLC (2019). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Acknowledgments

It gives me great pleasure to express my gratitude to Professor D. C. Gilles, the Head of the Department of Computing Science, for the opportunity to pursue this research in the first place, and for all his advice and patience when supervising my work.

As co-supervisor of my work, Dr. M. K. N. Nair, of the Mathematics Department, gave generously of his time and proved repeatedly to be both friend and counsellor, as well as having done much to stimulate my interest in this field of study during undergraduate days, and to him I am most grateful for it all.

I would also like to thank Professor W. D. Munn, the Head of the Mathematics Department for all his encouragement and consideration, and to his predecessor, Professor I. N. Sneddon, whose interest went far beyond the bounds of academic duty, and whose friendship has meant more than I can ever express.

In the Department of Computing Science I have many friends, especially the System Manager, Mr. Z. Podolski, who never seemed to tire answering questions; Doctors A. C. Allison and R. R. W. Poet who helped in the initial stages of the work and Doctors M. J. Jamieson and L. M. Mackenzie and Mr. W. Findlay who proof-read parts of the thesis and made some helpful comments. To them all I extend warmest thanks. Without the friendship and company of the postgraduate students in both departments, this would have been a lonely struggle. To all of them I say "thank-you", especially Mr. N. Nei, for his specialist knowledge and help with the graphs.

Mrs Anne Donnelly and Mrs Maxine Ramsay made an excellent job of decyphering my handwriting when typing the major part of the thesis. Sincere thanks are due to them also.

Without the generous assistance of the Science and Engineering Research Council this work would not have been possible. Professor D. Parkinson, the Head of the DAP Support Unit at Queen Mary College, London, was most generous with his time, and for several helpful discussions I am very grateful.

Last, but by no means least, to my parents, whose understanding of me made up for their lack of understanding of my research, and whose encouragement and help did much to keep me at it, I wish to express heartfelt thanks.

Table of Contents

Chapter		Page
0	Preface	1
1	Parallel Processing : An Historical Survey	4
2	The ICL DAP	19
3	The Lehman Algorithm	35
4	The Serial Implementation	51
5	The DAP Implementation	58
6	Assessment of Results	89
7	The Generalised Lehman Algorithm	104
8	An Alternative Technique for Detecting Divisors	128
9	Primes and Primality Testing	141
10	Conclusions	149

Erratum - page 36, last paragraph, lines 6-9

According to [CACM 1984] it was, in fact, the Quadratic Sieve algorithm which was used by Simmons and his team.

Note re. page 30, last line

The appropriate reference manual is the ICL Technical Publication 6918 (Third Edition, April 1981), which is entitled "DAP: FORTRAN Language".

Table of Illustrations

between pages 103 and 104

Plate

1	Step 1
2	Step 2
3	VAX - DAP comparison for 46-bit prime

between pages 127 and 128

Plate

4	T = 2589 with 11,111,111,111,111,111
5	comparison for 11,111,111,111,111,111
6	further comparison for 11,111,111,111,111,111
7	(k,d) pairs within "easy reach"

Chapter 0 Preface

There must be few people who, ten years ago, could have imagined the tremendous interest now being shown in finding the prime factors of large integers. What used to be just another part of the Mathematics taught in schools that most pupils forgot after entering the outside world, namely that every positive integer (> 1) is the product of a unique set of primes, has now become a focus of attention. Several factors have contributed to this, but undoubtedly one reason lies in recent developments in cryptography.

Several years ago a scheme was announced called the RSA Public Key Encryption system [Rivest, Shamir and Adleman 1978]. This is a method for coding information with the advantage that the method by which a message is encoded can be made public (hence the name "Public Key") while only the information necessary for decoding need be kept secret. The mathematics involved centres round a particular congruence modulo N , say, where the integer N is the product of two large primes (each having probably more than 40 decimal digits) chosen beforehand, and at present, the only way known (or at least published!) to crack this is to find these two primes. This is not a simple task. Certainly, the factorisation of very small integers poses few problems, but to find the factors of, say, a 100-digit decimal integer would probably take many years of computer time, even using the fastest algorithm available! Hence the considerable amount of effort now being applied to this problem. In the related area of primality testing, recent theoretical advances have resulted in tests that can ascertain whether a given integer is prime or not in a very short period of time. For example, the work involved for a 100-digit number would typically require less than one minute of cpu processing! Unfortunately this has not happened with factorisation, and the main achievements in the field are now coming from the efficient implementation of existing algorithms on the very fast computers now available.

It has not just been the great progress made in chip technology that has produced these high-speed computers. Much research effort has also been directed to the actual design of machines themselves, and how best to make use of the devices available. Thus, many novel computer architectures have been suggested and built, and it is these that have proved extremely useful to those mathematicians trying to solve the problems associated with integer factorisation.

Just three years ago, it was widely believed that numbers with 50 digits represented the limit of computational feasibility. But such is the progress in this area, made possible by the use of these advanced computers, that only last year a team at Sandia National Laboratories managed to factor the 69-digit Mersenne composite $2^{251} - 1$ in only 32·2 hours of computer time. This was such a significant achievement, demonstrating as it does, the advances made in numerical computing, that it was the subject of the plaque presented to the IEEE on the occasion of their centenary, by the ACM. In fact, the Sandia team have gone on to larger numbers, factoring, for example, a 71-digit integer in only 6·45 hours of cpu time. The leader of this group, Gustavus Simmons, is quoted by Kolata [1983] as saying: "I'm convinced now that large-scale computational problems such as factoring depend as much on the architecture of the machine as on its brute-force speed. If you can modify the architecture you can make enormous progress .... The exploitation of machine architecture is a whole new way of doing mathematics."

Simmons' work has been on a CRAY-1 computer, which is typical of one approach to "supercomputer" design. However, there are other possibilities, and these are the subject of Chapter 1. All the designs have one feature in common though, namely, the use of parallelism to increase the speed of the machine. The survey begins by showing how parallelism was first introduced to computer architectures at the bit level, with the development of word-based machines, before the demand for ever faster processors prompted its incorporation into other areas of machine design, with the result that now computers have been built with up to thousands of processing elements working in parallel.

As part of this research has involved the use of one particular supercomputer, namely the ICL Distributed Array Processor (DAP), we then go on to describe this machine in some detail. Mention is also made of the high-level language available on the computer, and examples are given and techniques discussed, to illustrate the problems associated with programming such a processor array. It is interesting to note that, after trying to implement the Continued Fraction factoring algorithm on the DAP in only a couple of months, Wunderlich is on record [Kolata 1983] as saying, "It takes a gigantic effort to put a new algorithm on a large machine."

Chapter 3 begins with an outline of the mathematical idea which underlies most of the modern factoring algorithms along with a brief survey of some of the better-known current work, before the method chosen for consideration in this study, namely the Lehman factoring algorithm, is presented, and proved. During the discussion which follows, various short-cuts that can be incorporated into a computer implementation are mentioned, including one modification to the algorithm which reduces the constant implied in the running time being $O(N^{1/3})$.

In Chapter 4 we give a brief description of an implementation of this algorithm on a VAX 11/780, before going on to discuss in detail, the parallel version on the DAP. The problem of performing trial divisions in parallel on a processor array is analysed and we show that, unless just prime divisors are used (which is impractical because of the storage required), the best processor utilisation figure that can be obtained (on the DAP) is only 80%. Also discussed fully is a binary algorithm for finding square roots which is particularly suited to the architecture of the DAP.

Assessing the relative performance of machines (especially if they have fundamentally different architectures, as here) is very difficult. Thus, the beginning of Chapter 6 is devoted to a brief discussion of this complicated problem, before the results obtained from this research are presented and analysed. For various reasons, which are examined, the speed-ups over the VAX, obtained by using the DAP, are sometimes more than would have been expected.

In Chapter 7 we give an example of when an even greater speed-up was recorded, and show mathematically why this was the case. Further analysis of the algorithm follows, leading up to the statement of a generalisation of the Lehman algorithm. We also discuss how certain extra knowledge can be used to reduce the amount of work required by the algorithm.

An alternative technique to division for identifying divisors of an integer is presented and analysed in Chapter 8 and, in addition to an examination of how one could generate lists of consecutive primes on the DAP, Chapter 9 contains a discussion of a topic related to factorisation, namely primality testing. Two tests suitable for use on the DAP are presented (from the literature), along with a running time analysis which shows that the algorithm concerned runs in polynomial time.

Chapter 10 contains a summary of the conclusions which this research led to, as well as an indication of further work that could be done.

Chapter 1 Parallel Processing : An Historical Survey

With the advent of electronic computers, scientists found themselves able to do in minutes what had previously taken them days, or even longer. It is surprising to discover, though, that there were those who felt that even faster machines would not be required. In a summary of a talk he gave in 1949, von Neumann [1949] is reported as making the following point.

> "A major concern which is frequently voiced in connection with very fast computing machines, particularly in view of the extremely high speeds which may now be hoped for, is that they will out-run the planning and coding which they require and, therefore, run out of work."

Of course, this was not the case. Von Neumann went on to point out, in the same talk, how the size of the problems that had been tackled up till then, had been limited by the speed of the machines available, and how the desire to solve larger problems would bring pressure to bear on designers to produce faster computers. Five years later, when speaking at the first public showing of the IBM Naval Ordnance Research Calculator [von Neumann 1954], he emphasised the importance of following the example of the US Navy and IBM, and "to write specifications simply calling for the most advanced machine which is possible in the present state of the art". As can be seen from what follows, his advice was heeded.

The early electronic computers were "bit-serial" machines (i.e. arithmetic and logical operations, as well as data transfers, were performed serially, one bit at a time) and so were comparatively slow. An example of such a machine was the Pilot ACE (short for Automatic Computing Engine), and its commercial derivative, the English Electric DEUCE [Wilkinson 1953]. An obvious (looking back!) improvement was to process more than one bit of a number or operand simultaneously, and so n-bit machines (where $n > 1$ is an integer) were born. Bit-parallel arithmetic, as it is called, became possible with the advent (and availability) of static random-access memories from which all the bits of a word could be read in parallel. In 1952, at the Institute of Advanced Studies, the first experimental machine to use such arithmetic was completed, while 1953 saw the appearance of the first commercial computer to employ parallel arithmetic, the IBM 701.

Another feature of early machines which limited their performance was that all input/output operations involved the use of a register in the arithmetic unit, thus halting

arithmetic activity for relatively long periods of time, due to the slowness of the type of I/O equipment being used (e.g. on-line card reader: 150-250 cards per minute ; card punch: 100 cards per minute ; line printer: 150 lines per minute; and, of course, paper tape reader: up to 1000 characters per second). Even the introduction of magnetic tape drives as the primary I/O medium with the provision of off-line card-to-tape and tape-to-printer facilities via another computer, failed to solve the problem, since even a tape speed of 15,000 characters per second was approximately 1,000 times slower than some processors (e.g. IBM 704).

This problem was at least partially overcome with the use of a separate computer (called a data channel) whose only job was to transfer data to and from (slow) peripherals and the main memory of the computer. Thus, once initiated by the main control unit, data transfers could proceed independently of the main processor, leaving the arithmetic unit free for more useful tasks. Of course, one was not limited to a single channel. In 1958 six data channels were added to the IBM 704, which produced the IBM 709, though this machine had a short life due to the use of "out-dated" technology (i.e. the use of valves when the transistor had become a reliable component).

The use of data channels is an example of what could be called "functional parallelism", which is the provision of several independent units for performing different functions (such as I/O, logic, addition, or multiplication), which are capable of operating simultaneously on different sets of data. This represents one of the four ways of introducing parallelism into the architecture of computers, namely pipelining, functional parallelism, multiprocessing, and processor arrays.

1. Pipelining

This method makes use of the principle of the production line. The assembly of a car, for instance, can be divided into many smaller tasks, each independent of the others, and all capable of being performed simultaneously. There are many tasks in computing that can be similarly split up. For example, the addition of two floating-point numbers could be decomposed into the following three stages:

- prenormalise
- arithmetic operation
- postnormalise

all of which can be carried out at the same time on (obviously) different data, thus forming what would be called a three-stage pipeline.

If pipelining is to be used in the design of a computer, several factors have to be taken into consideration. For one thing, it is desirable that the operation whose execution is to be pipelined be divided into stages that take approximately the same time to complete, otherwise some stages will involve a (comparatively) large amount of waiting for others to finish, resulting in a certain degree of inefficiency. It is also important to keep the pipeline as full as possible (i.e. We want each stage to have operands to process for as much of the time as we can.). Gaps in the pipeline (as stages without operands are called) occur when the sequence of operations is being set up (i.e. in the filling of the pipeline), and when the sequence is being terminated (i.e. when the pipeline is, as they say, being flushed). Unfortunately, there is really nothing one can do about this. However, once the pipeline is full, a certain amount of ingenuity is required on the part of the designer to keep it that way. Such contingencies as conditional branches and operand dependency can, unless properly dealt with, cause a (wasteful) gap in the pipeline.

Despite these problems, this technique has been used effectively to speed up various arithmetic operations. For example, the CRAY-1 [Russell 1978] has twelve functional units, all pipelined, to perform such tasks as floating-point addition, multiplication and reciprocal approximation, as well as logical operations and shifts on vector operands (which are a feature of this very high-speed machine). The CDC 7600 and IBM 360/91 [Anderson et al 1967] also provide pipelined functional units, while in the AMDAHL 470 V/6, the entire processing of instructions is pipelined, with instruction execution divided into twelve suboperations. A new instruction can be taken every two clock cycles (or every 64ns), and so, at any given time, up to six instructions can be in various phases of execution. Thus, to a certain extent, this machine, and the others like it, may be considered parallel processors.

While the CRAY-1 computer, as already noted, features many special-purpose pipelines, other machines, like the CYBER 205 (as Hockney and Jesshope [1981] report) incorporate a number of general-purpose ones. Another such machine was the TIASC [Watson 1972] which had either one, two or four identical general-purpose pipelines.

The efficacy of pipelining was demonstrated in the early 1960s in Manchester, where the ATLAS computer was developed. This was a single processor machine in which pipelining was employed in the executing of instructions. It proved very effective because, for example, in a series of floating-point additions, the average time per operation was reduced from 6.0 μ s for the sequential execution of the program, to 1.6 μ s with pipelining [Hockney and Jesshope 1981].

2. Functional Parallelism

As already mentioned, the term "functional parallelism" refers to the provision of separate units for performing different tasks, each independent of the others, and all capable of operating simultaneously on different data. One of the earliest computers to include this feature was the pilot ACE [Wilkinson 1953], which permitted one out of each of the following classes of operation to be performed in parallel:

- add, subtract, fetch, store
- multiplication, division
- memory transfer to and from drum
- input, output

The ATLAS computer is another example, since it provided a separate autonomous 24-bit adder for index calculations, in addition to the main fixed- and floating-point arithmetic unit which worked on 48 bits. However, the first computer to employ functional parallelism as a major design feature was the CDC 6600 [Thornton 1964] which had ten separate functional units. The IBM 360 series machines also used this idea in that they had separate execution units for floating-point and integer address calculation which could operate in parallel.

Before discussing multiprocessing and processor arrays, it might be constructive to see where parallel processors fit into the whole spectrum of computer architecture. A useful categorization of machine design was developed by Flynn [1966, 1972]. He classified computers into four groups according to the number of instruction and data streams which can be processed simultaneously, as follows.

(1) SISD single instruction stream/single data stream. This is the conventional serial von Neumann [von Neumann et al 1946] computer in which there is one processor executing one stream of instructions, using a single stream of data. Whether pipelining is used or not is irrelevant for the purposes of the categorization. Examples of such machines are: CDC 6600 (unpipelined); CDC 7600 (pipelined arithmetic); AMDAHL 470 V/6 (pipelined instruction processing).

(2) SIMD single instruction stream/multiple data stream. Such a machine could be called a vector processor since each instruction operates on a vector of operands, rather than on a single data item. The individual elements of each vector could be considered members of different data streams - hence the classification name. One could argue that this group should contain the pipelined uniprocessors which can process vector instructions (like the CRAY-1), although nowadays the term SIMD is used to refer

to an array of processors working under common control, examples of which will be given later.

(3) MISD multiple instruction stream/single data stream. In this type of computer, each operand would be operated on by several instructions simultaneously. However, at present, there is no system that is purely MISD, and it is hard to imagine what advantages such a machine would have. Thus, this grouping is only included to make the categorization symmetric.

(4) MIMDmultiple instruction stream/ multiple data stream. The computer described here is formed from at least two independent processors (independent in the sense that each is capable of operating on its own data stream, using its own instruction stream) connected together in some way. This group will shortly be described under "multiprocessing".

As can be seen, in this classification, nothing is said about the architecture of the processors involved, or about how they are connected together (in the cases of SIMD and MIMD machines), and so, to a certain extent, it is too broad. However, it will suffice for our purposes. (An example of a more detailed categorization is Shore's taxonomy which Hockney and Jesshope [1981] describe, in addition to a very complicated system of their own.)

3. Multiprocessing

As the name implies, a multiprocessor is a computer incorporating more than one processor. Since 1959, when Holland (as Hockney and Jesshope [1981] report) presented what could be considered to be the first large-scale multiprocessor design, several such machines have been produced, and from the standpoint of high-speed computation, the area seems a promising one. Unlike a processor array, which is composed of a large number of identical processors working in lockstep, this type of computer is made up of usually a smaller number of general-purpose processors, each capable of executing different instruction streams. The processors share global memory, and so one of the most important components of the system is the processor-memory interconnection network. In the designs so far produced, many different types of interconnection schemes have been used.

The most popular design has been that of a crossbar switch, which connects every processor to every memory module. It was used by Burroughs in their 5000 series machines, as well as in their D825 command and control computer [Anderson et al 1962]. The C.mmp [Wulf and Bell 1972] developed at Carnegie-Mellon University is another example, as more recently, is the S-1 multiprocessor [Widdoes and Correll 1979]. Developed at the Lawrence

Livermore Laboratory, University of California, under the auspices of the U.S. Navy, this machine was designed to have a performance ten times that of a CRAY-1! The S-1 Mark IIA, as it is called, consists of 16 specially developed single processor machines (called S-1 Uniprocessors), each with about the computational power of a CRAY-1, connected to 16 main memory banks via a crossbar switch which provides high-bandwidth, low-latency interprocessor communication. (There is also a shared bus whose function, among other things, is to transmit interrupts and small data packets from one uniprocessor to any subset of the others.) Early results seem encouraging, and there are plans to (re)implement the whole system using VLSI techniques. (That the idea of connecting together very powerful computers is still current in research, can be seen from a recent paper which describes a distributed system, called LCAP [Clementi et al 1984], being developed by IBM. Designed initially for applications in computational chemistry, it consists of two IBM-4341's and one IBM-4381 as front end processors, and ten FPS-164 attached array processors. This parallel system was claimed, in June of last year, to have a peak performance of 120MFlops (i.e. millions of floating point operations per second), which, it was hoped, would have risen to 550MFlops by the end of 1984).

There is, however, a major snag with the crossbar network. The cost of a such a switch grows as the product of the number of processors and memory modules. So, for a large number of processors with the corresponding memory, the price of the required switch could dominate the cost of the entire system!

A second computer developed at Carnegie-Mellon University, called Cm* [Swan et al 1977a, b], uses a less expensive interconnection method. The main distinguishing feature of this machine is that, instead of the shared main memory being separated from the processing elements, it is spread throughout the whole system. One processor (in this case an LSI11) and a unit of memory form the basis of what is called a "computer module". Up to 14 of these modules may be grouped together, along with a mapping processor (called a Kmap) to form a "cluster". The Cm* is basically a collection of such clusters. Communication in the system is performed using a three-tiered bus structure. Within each computer module, communication is via the LSI11 processor's bus. To enable modules in the same cluster to communicate with each other, every cluster contains a bus connected to all the LSI11 buses via a switch (or Slocal) in each module. The Map Bus (as it is called) is also connected to the Kmap which performs the required routing. If, however, a processor wishes to communicate with a processor in a different cluster, then this can be done through an intercluster bus connecting all the Kmaps, which again controls the routing of the data or message. Packet Switching, rather than Circuit Switching, is used in an attempt to achieve good bus utilization while minimising bus contention.

The cost of such a system is roughly linear with respect to the number of computer modules, and this makes possible the inclusion of a large number of processors (Jones [Jones and Schwarz 1980] reports the use of 50.). However, this interconnection scheme is comparatively slow, especially when intercluster communication is required. Thus, a certain amount of planning is required to try and ensure that very rarely does a processor need an operand contained in a memory unit within a different cluster.

A good compromise between the costly crossbar switch and the slow asynchronous bus would seem to be the network used in the Burroughs' design for a Flow Model Processor (FMP) [Lundstrom and Barnes 1980]. This was an architecture designed for NASA, with a planned performance of 1000Mflop/sec, targeted to be the FMP for the Numerical Aerodynamic Simulator. While this machine was not actually built, extensive simulation and analysis showed that, for certain favourable aerodynamic applications, the required performance level would have been achieved. The connection network between the 512 processors and 521 memory modules (Hockney and Jesshope [1981] state that this number was chosen because having a prime number of memory elements which is greater than the number of processors reduces memory conflict.) in the computer was considered an essential element in the design. What was required was a connection scheme that would allow all the processors to request simultaneously, and attend to their needs "fairly" and with little delay. The designers chose what is called the "baseline" network. This has a complexity of $O(PlogP)$ (where P is the number of processors), which compares favourably with the crossbar switch ($O(P^2)$ complexity) from the point of view of cost, and with the asynchronous bus system with respect to time.

While the design of this class of machines makes them inherently more flexible than processor arrays, there are very real obstacles in the way to achieving high performance. Such problems as allocating tasks to processors, synchronization of computations on different processors, and the sharing of resources, giving precedence to certain processes over others, as well as the design of interprocessor connection networks, are still topics requiring further research. (The Denelcor HEP has an interesting solution to the problem of resource allocation incorporating a queueing system [Snelling 1984].)

It is worth noting that most of the multiprocessors commercially available do not incorporate as many processors as some of the research machines mentioned above. Two or four processors have been the more common numbers to be used. For example, the CRAY XMP has only two processors, but with each of them being equivalent to a CRAY-1, it is one of the most powerful computers ever built.

4. Processor Arrays

Here we are referring to the provision of an array of identical processing elements working under common control, designed to perform the same operation simultaneously, but on different data stored in their private memories (i.e. operating in lockstep). These should be distinguished from array processors, since the latter term is used to refer to any computer that has special facilities for the processing of arrays, and so can include pipeline machines as well as processor arrays [Robinson 1979].

Hockney and Jesshope [1981] report that the idea of having a connected array of processors was first developed by von Neumann, who demonstrated that a two-dimensional array of computing elements with 29 states could simulate a Turing machine and so perform all operations. Unger [1958] followed this theoretical work by producing a design for a "spatial computer" consisting of a rectangular array of processing modules under the direction of a single control unit that broadcast instructions to all the modules. The major milestone in the introduction of this type of architecture was the SOLOMON computer, first described in 1962 [Slotnick et al 1962].

The Simultaneous Operation Linked Ordinal Modular Network (or SOLOMON for short) was originally a two-dimensional array of 32×32 processing elements (or PE's), each with its own two memory frames, again under the control of a single control unit which processed a single stream of instructions, and an input-output unit. However it appeared from a later paper [Gregory and McReynolds 1963] that it was possible to add or remove modules of 256 PE's, and their associated memory frames, from the system, so that the configuration could vary in size from a minimum of 32×8 PE's up to a maximum of 32×64. Each PE had 4096 bits of core storage in addition to a bit-serial arithmetic unit, and was able to communicate with its four nearest neighbours, if they existed. Gregory and McReynolds state that by specifying alternate connections between (only) the edge elements, a number of different geometric configurations were available to the programmer, namely: a horizontal cylinder; a vertical cylinder; a torus; a horizontal circle; and a vertical circle, as well as the normal planar configuration. The network control unit had two major functions: that of decoding and organising the execution of instructions, and controlling memory addressing. In addition, a broadcast option was provided, whereby the control unit could simultaneously supply the same operand (e.g. some required constant) to any (probably large) subset of the PE's, thus acting as a sort of "fifth nearest neighbour".

While this actual design was never implemented, it did "pave the way" for many of the processor arrays so far constructed. Machines like the ILLIAC IV, PEPE, STARAN, DAP, and the MPP all exhibit the influence that the SOLOMON computer had on the architecture of this type of machine. Before discussing these examples more fully, it would be

worthwhile to consider some of the main design issues which have to be resolved in the design of a processor array.

Firstly, one has to decide how many processors are required for the array. Or perhaps a better question would be "How many processors can be afforded?"! The answer to this problem will be influenced very much by one's decision on the second point, namely: "How complex should the processors be?". (As mentioned before, all the processors in the array will be identical.) "Do we want a large number of single bit processors (like the DAP), or a smaller number of 64-bit floating-point units (as in the ILLIAC IV), or are we going to try and seek a compromise somewhere in between?" is a question that has to be answered. It is interesting to note that, in terms of a processor array, 64 PE's constitute a small number! The amount and type of memory (i.e. associative or not), and how much of it will be common to all, rather than local to each PE, has also to be considered. Possibly the most important decision to be made concerns how the PE's will be interconnected (if at all - c.f. PEPE, later). It would be very undesirable if the potential increase in computation speed gained from the number of processors was lost, or not fully realised, because of routing delays. There have been several communication networks proposed for processor arrays [see Kuck 1977, Siegel 1979], three of which (namely mesh, cube and perfect shuffle) are described by Dekel and Sahni [1981] as follows.

(1) Mesh Connected Computer (MCC).

Here we consider the PE's to be logically arranged as a k-dimensional array which we will denote by $A(n_{k-1},n_{k-2},\cdots,n_0)$, where there are n_i processors in the ith dimension, and the total number of processors is equal to $n_{k-1}n_{k-2}\cdots n_0$. The PE at location $A(i_{k-1},\cdots,i_j,\cdots,i_0)$ is connected to the PE's at location $A(i_{k-1},\cdots,i_j+1,\cdots,i_0)$ and $A(i_{k-1},\cdots,i_j-1,\cdots,i_0)$, $0 \leqslant j < k$, provided they exist. This interconnection scheme, however, requires (at most) $2k$ connections per PE ($<2k$ for the cases where not all the "neighbours" exist).

(2) Cube Connected Computer (CCC).

This interconnection scheme is only possible if the number of PE's is equal to a power of 2 (i.e. $p = 2^q$, say). Using Dekel and Sahni's notation, if we let $i_{q-1}i_{q-2}\cdots i_0$ be the binary representation of i, for $i \in [0,p-1]$, and $i^{(b)}$ be the number whose binary pattern is $i_{q-1}\cdots i_{b+1}\overline{i_b}\, i_{b-1}\cdots i_0$, where $\overline{i_b}$ is the complement of i_b, and $0 \leqslant b < q$, then in this network, PE(i) is connected to PE($i^{(b)}$), $0 \leqslant b < q$. Thus we have a model requiring $\log_2 p$ connections per PE.

(3) Perfect Shuffle Computer (PSC).

Let p, q, i and $i^{(b)}$ be as above, and let $i_{q-1}i_{q-2}\cdots i_0$ be the binary representation

of i. Define $SHUFFLE(i)$ and $UNSHUFFLE(i)$ to be, respectively, the integers with binary representations $i_{q-2} i_{q-3} \cdots i_0 i_{q-1}$ and $i_0 i_{q-1} \cdots i_1$. Then, in this scheme, PE(i) is connected to PE($i^{(0)}$), PE($SHUFFLE(i)$) and PE($UNSHUFFLE(i)$), the three connections being called exchange, shuffle and unshuffle respectively. It is not hard to see that this connection network requires only three connections per PE.

As most communication networks are more suited to certain applications than others (due to requirements for data flow in the algorithms), the choice of interconnection scheme is usually influenced by the proposed uses of the machine. So it is important to decide what one wants the machine for, before building it. Admittedly, it would be preferable if there was a design which was independent of a particular application and efficient in the solution of a wide range of problems. However, this achievement is still a matter for research. Of course, another avenue for investigation is the development of algorithms for many different applications, all suited to a certain type of interconnection network , as is taking place with, for example, the ICL DAP .

As with the previous class of machines, many computers have been built which conform to the basic SIMD pattern - too many for them all to be listed here. Thus, in the descriptions that follow, mention is made only of the more significant or representative machines.

ILLIAC IV, PEPE, BSP

In 1966 the University of Illinois was awarded a contract by the U.S. Department of Defense's Advanced Research Projects Agency to design a computer based on the SOLOMON proposal. This machine became known as the ILLIAC IV [Slotnick 1967, Barnes et al 1968]. The original design was for a machine made up of four quadrants connected by a highly parallel I/O bus, and using a large disk for secondary storage. Each quadrant was to contain 64 processors (PE's) under the direction of a control unit executing a single stream of instructions, with each PE connected to its four nearest neighbours. Slotnick reported that each PE was provided with three 64-bit arithmetic registers and high-speed adders for full 64-bit floating- and fixed-point operations, in addition to 2000 64-bit words of thin film memory. Such a machine, it was planned, would achieve a maximum processing rate of 1Gflop/s, and was to be used for the solution of partial differential equations. However, due to many problems, ranging from the too slow development of the intended technology (ECL) to uncertainty over a location for this huge machine (see [Falk 1976] for further details), the

design as planned was never constructed. One quadrant, though, was built by Burroughs and delivered to NASA Ames Research Center, California, in 1972. It took a further three years of testing and replacing of faulty parts (e.g. 110,000 resistors had to be replaced because of unreliability) before it was fully operational. While the performance was not even a tenth of what had been proposed, the ILLIAC IV had a profound influence on the development of computer design, software, and especially technology. It was one of the first machines to use semiconductor memory chips for all its main memory, due to the fact that there was not enough room on the circuit boards for the thin-film memory originally intended. In the excellent historical survey at the beginning of their book , Hockney and Jesshope [1981] mention that the ILLIAC IV also "pioneered the use of 15-layer circuit boards and computer-aided layout methods that proved necessary to wire them".

As already mentioned, Burroughs were the main contractors of the ILLIAC IV (from 1969 - 73). In fact, they were involved with three parallel machines in the 1970's, the second of which was the PEPE computer (Appendix A of [Enslow 1974]). The Parallel Element Processing Ensemble (PEPE, for short) was a special-purpose machine designed to control a ballistic missile defence system of radar detectors and missile launchers for the U.S. Army. It grew (Hockney and Jesshope [1981] state) out of research at Bell Laboratories, Whippany, into content-addressable distributed logic memories combined with floating-point processing . PEPE consisted of 288 PE's working together in lockstep under the direction of three control units. Each PE actually contained three processors (one each for input of radar signals, processing of data, and output of control signals) - hence the need for three control units, one for each type of processor. Hockney and Jesshope report that "When operating, each target that was identified became the responsibility of one PE and, because there were no ordered connections between the targets, no direct connections were provided at all between the PE's. When necessary, communication between the PE's took place via the memories of the control units. The array of processors was then said to be unstructured and the word ensemble was coined for this arrangement.". Theoretically, the maximum computing rate of PEPE was 288Mflop/s (1Mflop/s per PE). However, a more realistic estimate was 100Mflop/s.

The third machine was a commercial venture called the Burroughs Scientific Processor (or BSP) [Stokes 1977]. Its design benefited from the experience gained from the construction of the ILLIAC IV. The aim was to provide a computer using standard technology (rather than pioneering new methods) which was capable of sustaining a high percentage of its maximum performance (in this case, sustain 20-40Mflop/s and have a maximum performance of 50Mflop/s), something the ILLIAC IV failed to do, and be programmed exclusively in a high-level language. The BSP consists of 16 serially organised floating-point

processors (each over twice as powerful as those in the ILLIAC IV) connected via a crossbar network to 17 memory banks, under the direction of a central processor. Pipelining is used in the execution of instructions. While the first design principle was not adhered to strictly, since the massive file memory of the BSP was constructed using charge coupled logic (CCL) technology, Hockney and Jesshope [1981] list timings which show that the desired performance was achieved. However, the BSP was withdrawn in 1980 before any had been sold.

DAP, MPP

This type of machine exemplifies a completely different approach to high-speed computation via arrays of PE's. Rather than the powerful processors used in machines like the BSP, here, large numbers of simple processors are involved in the design. The ICL Distributed Array Processor (DAP) [Flanders et al 1977] is an example of such a machine, and since much of this dissertation concerns it, a more detailed description will follow in the next chapter. But, for the sake of completeness, a brief outline is included here to show where the DAP fits into the spectrum of parallel computers. First delivered in 1980, it is a 64×64 two-dimensional array of single bit processors, each connected to its four nearest neighbours, working under the control of what is called the Master Control Unit. A feature of this machine is that 16 PE's and the associated memory are mounted on the same circuit board, in contrast to the von Neumann concept of a computer in which logic and memory are completely different (both conceptually and materially). Thus the logic could be said to be distributed throughout the memory - hence the name of the computer. Hockney and Jesshope give performance figures for this machine ranging from 15Mflop/s for matrix inversion to 48Mflop/s for Poisson solution which uses a number theoretic transform which optimises the use of the hardware.

The Massively Parallel Processor (MPP) [Batcher 1980] was the result of a contract awarded to Goodyear Aerospace by the NASA Goddard Space Flight Center in December 1979 for a machine capable of the very high-speed image processing required to process satellite photographs. It comprises a two-dimensional array of 128 × 128 PE's, again with nearest neighbour connections, and a single array control unit. (There is also an additional rectangle of 128×4 PE's that is used to reconfigure the main PE array in the event of a PE fault.) Single bit processors were chosen to enable the efficient processing of operands of any length. The results achieved are truly astonishing. Batcher gives a table showing speeds of typical operations which range from 216 million operations per second for the multiplication of two 32-bit floating-point numbers to 6553 million operations per second for the addition

of 8-bit integers.

STARAN, LUCAS

These machines are similar to the DAP and MPP in that they also comprise arrays of single bit processors. The main difference lies in their use of associative memory. In this type of memory, an item is accessed by its contents, rather than by an address (hence its alternative name: Content Addressable memory.). Usually a comparison is performed between a certain part, or field, of each memory word and a given pattern or mask, and the data item is accessed if a match is obtained. (Some associative memories also have a conventional addressing system.) Various machines based on associative memories have been produced [see Thurber and Wald 1975, Yau and Fung 1977], the best known of which is the STARAN [Batcher 1974] processor array produced by Goodyear and completed in 1976. Hockney and Jesshope summarise the features and uses of the computer as follows.

> "The STARAN typically comprised four array modules, each with 256 one-bit PE's and between 64Kbits and 64Mbits of total storage, controlled by a sequential PDP-11. Unlike the SOLOMON, however, the storage was not assigned to specific PE's; instead, a flexible 'FLIP' network was interposed between the PE's and the memory. A slice of 256 bits was selected from memory in a pattern specified, under program control, by a 256-bit code. The pattern selected may, for example, have treated the store as a multidimensional array with a varying number of dimensions, or shuffled the data in the manner required by the fast Fourier transform and other important numerical algorithms. Connections between the PE's were achieved by passing the 256-bit slices of data through the FLIP network, thus achieving in minimum time a highly flexible effective interconnection pattern that could be varied from problem to problem by the programmer. The STARAN, like other bit-oriented computers, was most effective when performing logic and short word-length integer arithmetic. A particularly suitable application is the digital processing of pictures, in which the image is divided into millions of pixels (picture elements) each of which is represented by 6 - 12 bits. The first STARAN was delivered to the Rome Air Force base for such an application and it has also been proposed for air-traffic control."

A more recent, if less well known, machine is that developed at the University of Lund in Sweden. LUCAS (Lund University Content Addressable System) [Ohlsson and Svensson 1983] was developed as a research tool to investigate the applicability of associative processors. It consists of an array of 128 identical single bit processors interconnected using the Perfect Shuffle/Exchange network (see above). Apart from STARAN, whose FLIP

network can resemble the Perfect Shuffle connection, LUCAS is the only machine at present known to have this network, and so the findings of this research will be especially interesting. Early results seem promising.

It should be said before concluding this section, that all the above machines normally require a host computer, and so act as special-purpose attached processors. For example, the ILLIAC IV was connected to a PDP-10, the PEPE to a CDC-7600, the DAP to an ICL 2980, and the STARAN at Rome Air Development Centre is attached to an I/O channel of a Honeywell HIS-645 computer.

Other approaches

All the machines mentioned above follow von Neumann's idea of sequential instruction execution. However, architectures are being proposed which do not operate on this principle. One alternative is the data flow concept which Dennis [1979] describes as follows:

"In a data flow computer, an instruction is ready for execution when its operands have arrived - there is no concept of 'control flow', and data flow computers do not have program location counters. A consequence of data-activated instruction execution is that many instructions of a data flow program may be available for execution at once. Thus highly concurrent computation is a natural accompaniment of the data flow idea."

However, the extent to which such potential concurrency can be exploited is limited when only a single processor is being used (although such techniques as pipelining can be employed). Thus, a natural extension is to connect together many data flow processing elements to form what Dennis calls a "data flow multiprocessor system". In fact, he goes on to describe a possible architecture for such a machine, that has been developed at MIT. Various other designs have also been proposed, including a 20 processor system currently under construction at Manchester University [Gurd 1984]. However, this class of computers is still at the development stage, with much more work to be done if this type of machine is to become commercially widely available, and whether they will ever achieve the performance figures quoted for other high-speed machines remains to be seen.

Another approach to high-speed computation, though specifically for use in special-purpose machines, has been developed recently at Carnegie-Mellon University. Kung [1982] describes the basic idea involved in "systolic architectures", as they are called, as follows.

"A systolic system consists of a set of interconnected cells, each capable of performing some simple operation. Because simple, regular communication and control structures have substantial advantages over complicated ones in design and implementation, cells in a systolic

system are typically interconnected to form a systolic array or a systolic tree. Information in a systolic system flows between cells in a pipelined fashion, and communication with the outside world occurs only at the 'boundary cells'. For example, in a systolic array, only those cells on the array boundaries may be I/O ports for the system."

Kung goes on to explain that this class of machine was proposed in order to speed-up those computations where multiple operations are performed on each data item in a repetitive manner (e.g. matrix multiply, convolution problems). By replacing a single processing element with an array of systolic cells, it was envisaged that a higher computation throughput could be achieved, without increasing memory bandwidth. Since the cells used are very simple processing elements, such machines can be implemented in VLSI at relatively low cost. Versions of systolic processors are being designed and built by several industrial and governmental organisations, and it will be interesting to see if, as Kung hopes, they result in "cost-effective, high-performance special-purpose systems for a wide range of problems."

Related areas in research

The study of Parallel Processing is not limited to machine architecture. The development of algorithms, and languages to express them, which exploit the potential of these machines is also of great importance. Work has been done in the writing of compilers which can detect parallelism in a serial program. But, for reasons that will be discussed later, for the most part, it is still up to the programmer to identify the possible parallelism beforehand, and to express it using one of the available parallel languages like TRANQUIL, GLYPNIR, ACTUS and CFD FORTRAN, which were all developed for the ILLIAC IV, or DAP FORTRAN, to name but a few. As can be seen, the above languages are specifically designed for certain machines. However, in the interests of even just portability, it would be desirable to have a "universal" language for the writing of programs for parallel machines. It had been hoped that the new language Ada would have proved suitable. However, this seems unlikely to be the case.

It seems certain that this whole area will be the subject of much study in the future, for as Stone says [Stone 1975]:

> "To achieve even faster computers in the future we must take new approaches that do not depend on breakthroughs in device technology, but rather on imaginative application of the skills of computer architecture.".

Chapter 2 The ICL DAP

In the previous chapter we briefly mentioned that the DAP, with its 64×64 array of bit processors operating in lockstep under the control of the Master Control Unit (mcu), is typical of one of the approaches to parallel processing. We start this chapter by giving some of the reasons which motivate the design of this class of machine.

It is well-known that there are many algorithms which require basically the same operation(s) to be performed on many different numbers (e.g. matrix manipulation, finite difference methods for solving Partial Differential Equations, Lattice Gauge Theory). On a serial machine, this involves the use of DO loops (or the equivalent), where the same sequence of instructions has to be repeated over and over again, but on different data. If, however, we were able to allocate a separate processor to each data item, then we could perform all the required operations in parallel. Often, as in the case of finite difference methods, this might require the use of thousands of processors, This, in turn, places certain constraints upon them; for example, the processors would have to be physically small, of low power consumption, and be relatively simple. Conventional 8- or 16-bit processors are not suitable; for one thing, they (rightly) have an instruction decoder. But this facility is unnecessary in an SIMD computer, since all the processors will obey the same instructions which, therefore, need only be decoded once (by the mcu), and not 4096 times. Thus it is not surprising that most of the SIMD machines designed to date, have used specially-designed, bit-organised processing elements.

The DAP is no exception. Its processing elements (PE's) are identical, and have only three registers, each one bit long, together with an arithmetic logic unit, and some memory. Originally, each processor had 4K bits of memory, but in the DAP at Queen Mary College, extra has been added, to bring the total up to 16K bits per processing element. As one would expect, there is an Accumulator (Q), and a Carry register (C).However, the third register is more unusual. Called an Activity bit (and denoted by A), this register provides the ability to effectively "turn off" (and "turn on") processors as we wish. Since the processors work in lockstep, they all obey the same sequence of instructions, broadcast from the mcu. Of course, they will probably not be operating on the same data! However, if for some reason, we only wanted some subset of the PE array to perform a certain task, we are forced to have the necessary work carried out by all the processors. This could cause problems, as will be shown later. But, among other things, the Activity bits allow us to select which processors

will actually store the result of such an operation, and which will not. Thus, while all the processors have to do the specified work, not all of them need to record the effect. Hence, we can think of those PE's which do not perform the assignment part of the instruction as being "switched off". This technique (known as masking) is one of the most powerful features of this machine, and so will be described in more detail later.

Normally, (though, of course, it will depend on the application), the processors use data from their own local memories. But there are times when information is needed from "outside", so to speak, and there are two ways in which this can be supplied. As well as telling the processors what to do, the mcu can broadcast a given scalar value to all the processors at once. Although it is not possible for different values to be made available globally simultaneously (instead, several broadcast instructions would need to be used, along with masks to select which processors received which data), this facility can prove very useful. In addition, the processors can communicate with each other by means of the interconnection network. The PE's are connected to their nearest neighbours (i.e. to the processors which are immediately above, below, to the right, or to the left - denoted, in an obvious notation, by NSEW), and data can be passed between them on these row and column highways (as they are called). There is no restriction here that all the values be the same. It is quite possible for each processor to receive a different number. However, there is a limitation; namely, that every processing element receives a value, and that all the values travel the same distance, in the same direction. In other words, it is not possible for one processor to receive a piece of data from the PE which is three rows below it, but in the same column, while another processor is passed a value from the PE to its left. Every processor must receive information from the same corresponding source, with masking used to indicate which processors are not to store the value they receive.

It is because of this interconnection scheme of rows and columns, that we can conceive of the processors as being arranged in an array. Of course, physically, this is not true. The PE's are placed on circuit boards (16 per board) which are, in turn, arranged side by side in cabinets. Nevertheless, the conceptual idea of an array is a useful one to have, as it can help us visualize the parallel activity that we are trying to arrange. It also means that we can refer to a certain processor easily, by stating which row and column it is in. Thus PE(i , j) will lie at the intersection of row i and column j , where both i and j take values between 1 and 64 inclusive.

The hardware also provides connections between the elements at the edge of the array. The processors on the left edge (i.e. in the first column) take as their left neighbour, the element in the same row in the last column, while the PE's in the first row are connected to the corresponding elements of the last row. Thus the array can be thought of as both a

vertical, and a horizontal cylinder. A more unusual system of edge connections, supported by software, provides a link between the bottom elements of the first 63 columns with the top elements of the successive columns. Thus, for example, PE(1,64) is (conceptually) joined to PE(2,1), PE(2,64) is connected to PE(3,1), and so on. How such a scheme can prove useful will be seen later in this chapter.

Since the processors are capable of receiving information only one bit at a time, the connections described above are just one bit wide. Thus, a 64-bit register could supply data to all the row or column highways simultaneously. The mcu has 8 such registers attached to the row and column highways, which can be used, not only to transmit data to all the processors in a given row or column, but also to receive data from selected processors.

As has already been mentioned, the DAP does not stand alone, but requires a host computer, through which all communication with "the outside world" takes place. The DAP itself has no input or output facilities, and so any required data must be read in by the host, which must also write out any necessary results. Programming the DAP is done in a specially-designed high-level language called DAP-FORTRAN, and the instructions are executed after a call of the DAP from a main program in the host. Thus there are two distinct parts to any job that requires the use of the DAP: a host section, written in FORTRAN IV or FORTRAN 77, consisting of a main program, and possibly some host subroutines; and the DAP code, which is a collection of subroutines, written in DAP-FORTRAN. At least one of these subroutines must be, what is called, an "entry subroutine".

The entry subroutine is a parameter-less subroutine which is called by the host program as if it were written in normal FORTRAN. However, all its instructions are for execution on the DAP, and so, on its call, control is passed from the host to the DAP mcu, which then takes over the running of the program. If the programmer wishes, further subroutines may be called once the DAP has been entered. These will be internal to the DAP section of the job, and can have parameters, if required, which will be passed by address, rather than by value. It is only the entry subroutine which must have no parameters. Instead, values of variables are passed in and out of the DAP by means of FORTRAN named COMMON blocks. The way data is stored in the host is not the same as in the DAP, and so, before any values in a COMMON block can be used, they have to be converted to the DAP format (There are routines supplied for doing this, which have only to be called, usually from the entry subroutine, with the appropriate names as parameters.). Similarly, any output from the DAP must be converted to the host mode (again using standard routines) before the DAP section can be completed, and control passed back to the host.

The DAP may be entered (and left) as many times during the running of a job as the programmer wishes, provided that one is prepared to accept the overhead in time that this will require. But, in the interests of speed of execution, it is better to limit leaving and re-entering the DAP, to as few occurrences as possible. Indeed, with each of the programs that will be described in this thesis, there is only one entry subroutine, which is just called once. This does not mean, however, that we never re-enter the DAP, as we will now explain.

In a serial machine, a "tried and trusted" way of debugging a program is by tracing, i.e. to insert (possibly many!) "write statements" in the code, to print out the values of relevant variables, so that the programmer can discover if the program does what it was meant to. This technique can sometimes prove vital in finding inconspicuous errors in a piece of code. However, this cannot be done easily on the DAP. Values of variables can only be printed out by the host, and these would normally be passed from the DAP in a COMMON block, once the work of the entry subroutine, and any routines it may call, had been completed. Thus, to find the intermediate values of certain variables, we would be faced with writing a whole series of entry subroutines, each one containing a few more instructions than the previous one, running them in succession, and then comparing the results returned in the COMMON blocks - a very cumbersome procedure!

In order to avoid this, DAP-FORTRAN provides a facility for tracing the intermediate values of variables. Called, not surprisingly, a TRACE statement, it is used in exactly the same way as a "write statement", except that it causes the DAP to be left, the host entered, the required value(s) printed out, and then the DAP to be re-entered, with execution of the DAP program restarted at the instruction immediately following the TRACE statement. While such statements may be inserted anywhere in the DAP code, they have one disadvantage which will be mentioned later. Even so, this feature is extremely useful, as it makes less difficult what is, perhaps, the hardest part of DAP programming, namely, debugging a program and getting it to run! Of course, since TRACE statements necessitate leaving the DAP, they should be used sparingly in programs which are known to work correctly.

When a machine has such a definite structure as the DAP, this must be taken into account when producing programs for it, otherwise the computer's potential will, almost certainly, not be realised. Certainly, this is nothing new. For example, optimising compilers are written for a specific machine so that advantage can be taken of the features of its design, without the programmer having to know about them. Programming bit processors in parallel is not that simple, however. The problem lies in there being two different areas that must be considered before optimal performance can be reached, or even approached. The first concerns the parallelism of the machine, to make use of which normally involves the

re-ordering or unrolling (or both) of loops, so that blocks of iterations can be performed in parallel. For many years now, such problems and their solutions have been discussed in the literature [Kuck 1980], and compilers have been written to detect pieces of code which could be performed in parallel, and then arrange for this to be done, with varying degrees of success. For example, an optimising compiler (called CFT) has been written for the CRAY-1 computer, which "vectorizes" innermost DO loops to take advantage of this particular machine's vector registers [Russell 1978].

However, it is in the second area that real problems stand in the way of compiler-based solutions. As we will discuss more fully later, one of the consequences of performing many tasks in parallel using bit-processors is that all the arithmetic involved must be done bit-serially, which opens up a whole new field for optimisation and improvement. Certainly one could just perform in parallel exactly the same sequence of instructions as would be used on a conventional word-based machine. But this would mean ignoring much of the potential of the DAP. There are many algorithms which, while being relatively time-consuming on word-based computers are relatively quick on the DAP (finding square roots is an example of this which will be discussed later). In addition, an array of bit processors provides the programmer with the scope for using bit-manipulating algorithms, which as we will also show later, can prove very effective. Such a simple approach to parallel programming as described above, will also mean the neglecting of the interconnection network, and the possibilities it makes available, like the technique of recursive doubling (again, see later). From the above, it should be clear that to get the best out of a machine like the DAP will certainly require much careful thought and planning, and probably involve different algorithms and techniques to those employed on a word-based machine. For this reason, at the present time, it is not practical to write a compiler which would produce from an existing serial program, very efficient code for the DAP, and the programmer has to decide how to organise the performing of the required tasks in a way that will capitalise on the DAP's form. To do this, one must be able to express the parallelism required.

To meet this need in a high-level language, a version of FORTRAN has been written specifically for the DAP, namely DAP-FORTRAN. One of the advantages in using a high-level language is that, while the programmer will be aware of the general architecture of the computer (e.g. that it has 4096 processors), the low-level features of the design (e.g. that the processors are bit-serial, and are connected in an array) can be hidden. Of course, to use the processors most efficiently will probably require some knowledge of the latter, and even then, as we will describe later, DAP-FORTRAN can be used to express low-level concepts.

The decision to extend an existing language (in this case, FORTRAN IV) rather than design a new one, was made because it was envisaged that most of the users of the DAP would be members of the scientific computing community, where in the past, FORTRAN has been the most popular language and that it would be easier for them to adapt to the new machine if they did not have to learn a completely new language.

The main extension provided by DAP-FORTRAN is in the data modes. Conventional FORTRAN has only one : namely, scalar mode. DAP-FORTRAN has, in addition to this, two new modes: vector and matrix. A vector, of a certain type, is the same as a 64-element array of scalars of that type, except that all the elements of the former will be operated on simultaneously. Similarly, a matrix is equivalent to a two-dimensional array (with 64 elements in each dimension) of scalars of the same type, where, again, all 4096 elements of the matrix can be processed at the same time. Thus, instead of using the following loops:

$$DO\,10\ I\ = 1{,}64$$

$$DO\,10\ J\ = 1{,}64$$

$$10\ C(I,J)\ = A(I,J) + B(I,J)$$

to add the contents of corresponding locations of two 2-dimensional arrays (each of size $64{\times}64$), we can simply say

$$C = A + B$$

where A, B, C are declared to be matrices of the desired type. This could also be written as

$$C(,) = A(,) + B(,)\ ,$$

where the two dimensions of each operand have been left empty, showing that they are what are called "constrained dimensions", and to be operated on in parallel. The inclusion of the constrained dimensions is optional, unless one is dealing with an array of matrices, in which case they must be included. For example, an array of INTEGER matrices, with 3 components, TEMP say, would be declared as

$$INTEGER\quad TEMP(,,3)$$

(The same convention applies to vectors, except that there is only one constrained dimension.).

Thus the iterations required can be expressed in terms of a single instruction. The same is true for the other arithmetic operations: subtraction, multiplication and division. It is worth mentioning that when you multiply two DAP-FORTRAN matrices together (using the arithmetic operator *), what is performed is not the conventional matrix multiplication algorithm, but the forming of the product of the corresponding elements of the two

operands. The other arithmetic operations : + , - and / , are similarly performed pointwise.

It is through the use of these two data modes that the programmer controls the parallelism of the machine. A matrix of type, say INTEGER, refers to 4096 scalars of that type, each one lying in the memory of a different processor, but all at the same address. The same is true for matrices of the two other numerical types in the language: REAL and LOGICAL. The fact that all the elements have the same address (in different memories) is very important, as will now be shown.

We have already mentioned how we imagine the processors to be arranged in a 64×64 array. Let us imagine further, that all of the 16K bits of memory local to PE(1,1) are placed one on top of the other, resulting in a column, as it were, rising out of the processor. If we were to do this for every PE, then we could consider the DAP to be the shape of a cuboid, with the processors at the bottom, and the memory arranged in planes (each 1 bit deep) above them. Now, since all the elements of a matrix have the same address (but lie in different memories, and hence, columns in our analogy), we can imagine a matrix to be stored as a contiguous set of these store planes (hence the use of the term "vertical storage" to describe the representation of matrices).

The DAP can process only one store plane at a time, and so all arithmetic has to be performed by system-supplied software routines. One consequence of this is that we can choose the precision to which we work, and similarly, we can choose how many store planes each matrix will be allocated. However, as we have already remarked, if we require some processors to perform a task, then all the PE's have got to do it : we can only be selective about the storing of results. This is because the DAP cannot process just part of a store plane. It is a case of "all or nothing". So, if we had a matrix of integers, one of which had 33 bits in its binary representation, while all the others had less, then all the elements would have to be allocated 33 bits (i.e. the matrix would require 33 planes of the DAP store). In fact, in a DAP-FORTRAN program, we would need to set aside 40 bit planes for storing such a matrix because there is a (software) limit to the variety of precisions that DAP-FORTRAN INTEGER variables can have.

Normal FORTRAN supports at most two precisions: single and double, which, on a machine with a word size of 32 bits, would correspond to 32 and 64 bits, respectively, being allocated to integer variables. In addition to these lengths, DAP-FORTRAN offers the choice of 16, 24, 40, 48 and 56 bit integers (denoted by INTEGER * n , where the integer is to be allocated $8n$ bits). A similar situation exists with the type REAL, but not with LOGICAL variables.

It is one of the important features of DAP-FORTRAN that LOGICAL variables only occupy a single bit of memory. This is far more efficient than the method often used in computers with word lengths of more than one bit, where up to 32 binary digits (i.e. one machine word) are used to store either a .TRUE. or a .FALSE.. (Of course, having a machine word larger than one bit does have certain advantages when performing arithmetic operations, as will be mentioned later.) The DAP's storage arrangement means that a LOGICAL matrix will only require one bit plane to hold its values. Hence, a one-dimensional array of LOGICAL matrices, with 32 components will be stored in 32 contiguous store planes, and thus be indistinguishable from a matrix of type INTEGER * 4. From this we see that the EQUIVALENCE statement of normal FORTRAN can take on a whole new significance here. By making an array of LOGICAL matrices (of the right size) equivalent to, say, an INTEGER matrix, we can gain access to the bit patterns of the latter's elements.

The same is true for scalars and vectors. Scalars are stored horizontally (i.e. in a single plane), with the bit pattern (of length up to 64 bits) spread across consecutive processors in the same row, with the least significant bit stored in the rightmost processor. A LOGICAL vector is also stored in a single row, and so, by means of an EQUIVALENCE statement, the bits of the scalar's binary pattern can be manipulated in terms of the elements of the LOGICAL vector. (If the scalar is not of 8-byte precision, then it should be converted to this length, to match the size of the vector.) INTEGER or REAL vectors, however, are stored one element per row (as scalars above), and so require a whole store plane (since each has 64 elements). Thus, for a vector of type INTEGER, the 64th column of the plane will contain the least-significant bits of all 64 elements. Access to the binary digits can be obtained via a LOGICAL matrix, after any necessary length change, and the appropriate EQUIVALENCE statement.

Thus, by means of an EQUIVALENCE statement, we can have access to the binary patterns of numbers (although they are expressed in terms of the logical values .TRUE. and .FALSE. rather than the digits 1 and 0) directly from our high-level language. This allows the programmer to implement bit-manipulating algorithms (which are, of course, very suited to an array of bit processors) without having to resort to the complications of assembly code programming. As we will illustrate later, this feature has been made use of in the implementation of the Lehman algorithm on the DAP .

Returning to our analogy of the cuboid, we can see another use for LOGICAL matrices. We have considered the DAP to be 16,384 bit planes, stacked on top of the array of processing elements. But, since each PE contains 3 bit registers, we could think of there being 16,387 bit planes, with the lowest 3 planes containing the contents of the 3 registers,

one register per plane. Thus all 4096 Activity bits (designated the A plane) could be assigned by means of a LOGICAL matrix. This is indeed possible; how it can be done is shown in the following short piece of a program to obtain the square roots of those elements of the real matrix NUMBERS which are positive.

```
LOGICAL  MASK(,)
REAL*5   NUMBERS(,)
REAL*3   ROOTS(,)

MASK = NUMBERS .GE .0
ROOTS(MASK) = SQRT(NUMBERS)
ROOTS(.NOT. MASK) = 0
```

We require the square roots of the elements of the real matrix NUMBERS, but have no guarantee that the latter are all non-negative. Trying to find the square root of a negative real number (even just one out of 4096) will cause a machine error; but only if an assignment of the "bogus" root is attempted. Thus, by identifying those processors with negative values (indicated by a .FALSE. in the LOGICAL array MASK), we can block or "mask out", the assignment of the root in these processors, and so avoid trouble. The masking takes place during the instruction

```
ROOTS(MASK) = SQRT(NUMBERS)
```

which causes the following course of action to take place: (a) all the elements of the matrix NUMBERS have their square roots taken but (b) only where the corresponding element of the A plane (which holds the matrix MASK) contains a .TRUE. will the value of the root be stored in ROOTS. We can also negate the mask so that all the locations not assigned above may be given some other value; in this case, 0.

This technique is extremely useful as, for instance, it allows us to implement "IF (condition) THEN (task 1) ELSE (task 2)" branches, where a mask would be set with the condition in the IF statement, and used in masked instructions to perform task 1. Then task 2 could be carried out, again using masked assignments, but with the negative of the previous LOGICAL matrix.

One of the features of the DAP is that, once the masks have been obtained, using them to select processors for an assignment like this, is virtually "free", for a masked assignment takes only 1 μsec more than the equivalent unmasked instruction. There is an important principle of DAP programming here: "The DAP loves to make decisions in parallel", so to

speak. Of course, this is the exact opposite of a pipeline machine, where conditional branches can cause a flush and refill of the pipeline, thus wasting time. Admittedly, with the DAP, both branches of a conditional may have to be performed, as there could be some processors with values necessitating one course of action, while the other processors required the alternative. But this is a consequence of the SIMD approach, and cannot be avoided. However, it cannot be overemphasised that boolean operations cost so little, (For example, on a CRAY, a floating-point multiply is only equivalent to a few logical operations, whereas on a DAP, depending on the precision required, it is equivalent to up to 1,000 boolean operations.) as this is one of the key features in favour of arrays of single bit processors.

We have shown above, how the binary patterns of the elements of a matrix can be made available to the programmer by means of LOGICAL matrices, and now we have noted how effective the DAP is at logical operations. Thus, through the high-level language of DAP-FORTRAN, it is possible for the user to take advantage of the bit-processing nature of the DAP. However, as we will discuss later, the gain from this approach is not as great as might be hoped, due to the overhead of all the indexing involved in manipulating arrays.

Before considering some other features of DAP-FORTRAN, it is worth mentioning that there are facilities provided for converting between data modes. For example, a matrix (with identical columns or rows, respectively) can be formed from a vector by setting either each row or each column of the former, equal to the latter. There also exist functions to expand a scalar to a vector or matrix (all of whose elements would be equal to that scalar), but they are not often used, because the desired expansion happens (conceptually) "automatically" when, say, adding a scalar to a vector or matrix. In fact, at times, something slightly more efficient happens. For example, in the case of adding a scalar to (each element of) a matrix, rather than broadcasting the value to each processor, bit by bit, and having it transferred to the DAP store as a series of bit planes, which then have to be "brought back down" to the processors so that the addition can be performed, each bit of the scalar could be broadcast to every PE, and added to the relevant bit of the existing element, and the sum stored immediately. Tricks like this have been implemented in some of the software routines, while in others, due probably to time constraints when the system software was being written, no shortcuts are taken.

A vector may be obtained from a matrix by summing its columns or its rows (also a scalar from a vector or matrix again by summing, while from a LOGICAL matrix, a scalar can be produced by using the function SUM. When used with a LOGICAL matrix (or vector) this function forms the sum of the 4096 elements of the matrix, counting a .TRUE. as a 1, and a .FALSE. as zero (similarly for LOGICAL vectors). Such a facility is very useful, as it supplies the number of elements which have the value .TRUE. in a mask. Thus,

if a matrix had been set according to some expression, the function SUM could be applied to the mask to find how many elements had satisfied the test. In the example above, for instance, we could assign to the INTEGER scalar, NUM, the number of operands which were greater than or equal to zero, by including the instruction

$$NUM = SUM(MASK)$$

in the code.

If one simply wished to know if any bits at all, were set in a mask, then the function ANY could be used, which logically OR's all the elements of the matrix. On the other hand, that all the bits were set, could be discovered by the function ALL, which performs a logical AND on all the matrix elements.

Another useful function is FRST which, when applied to a LOGICAL matrix, returns another such matrix, all of whose entries are .FALSE. except for one, corresponding to where the first .TRUE. occurs in the former. (How we can talk about a "first" .TRUE. element in a matrix will be explained later.) Of course, if no bits are set in the operand, then all the elements of the resulting matrix will also be .FALSE.. If, on the other hand, one needed to know where in a LOGICAL matrix, or vector, the first .TRUE. element was, then this could be found with the use of the function ELN which, when given a LOGICAL matrix or vector, returns the index of the first .TRUE. entry (or zero, if all the bits are .FALSE.). To show how these functions should not be used (but often are, by the beginner), we give the following example.

Suppose we wished to find the element in the INTEGER matrix NUMBERS which corresponded to the first .TRUE. element in the LOGICAL matrix MASK, then we might be tempted to write:

$$INDEX = ELN(MASK)$$

$$TEMP = NUMBERS(INDEX)$$

where TEMP and INDEX are both INTEGER scalars. (How we can use a single number to refer to a matrix element will be explained shortly.) While this code would perform the required task, it is very clumsy and inefficient, since extra calculations (hidden from the programmer) have to be performed to find where, in the matrix, the required element is. Instead, we should use:

$$TEMP = NUMBERS(FRST(MASK))$$

since this is much quicker (and "better" style) as it uses a LOGICAL matrix with only one bit set, to indicate which is the required element, (another example of a use for LOGICAL masks). The equivalent can be done with vectors also. While the former suggestion would

probably be more natural to a programmer used to serial machines, it is the latter way of thinking that has to be adopted (and cultivated) if the potential of the DAP is going to be used.

The snag with using TRACE statements that we referred to earlier, is that one cannot trace just a selection of elements from, say, a vector or a matrix, but must output all the values (similarly for arrays of any type and mode). However, as we have just shown, there are various ways of selecting elements from matrices and vectors, and values thus obtained could be assigned to scalar variables (of the appropriate type and length) for printing with a TRACE statement.

In the above we used a single index to select an element from a matrix. We can do this because, as well as thinking of a matrix as a two-dimensional array, DAP-FORTRAN permits the programmer to imagine it to be a one-dimensional array with 4096 components. This is what is called "long vector" format, in which we consider the columns of the matrix to be joined "nose to tail", with the successor of the 64th element of one column being the 1st element in the next column to the right.

Processors communicate with each other in DAP-FORTRAN by means of shifting matrices or vectors. Matrices can be shifted up or down (i.e. move whole rows up or down), and to the right or left (by shifting columns). This poses the following problem. Suppose we shift a matrix one column to the left (i.e. column 1 is replaced by column 2 which, in turn, is replaced by column 3, etc.). What is the new value of column 64? In DAP-FORTRAN we can specify two different "geometries": planar, in which case the last column will be filled with zeros; or cyclic, where the values shifted out of column 1 are transferred to the last column (considering the DAP to be a vertical cylinder).

In addition, a matrix can be considered to be a long vector, and so shifted as a vector (either to the left or to the right). In other words, columns can be moved up or down, with the bits shifted out being fed in at the bottom of the previous column, or at the top of the next, respectively - hence the unusual connection scheme, supported by software, which was described earlier. As before, we can specify what happens at the ends of the vector, by choosing which geometry is required.

There are many other functions in DAP-FORTRAN which have not been mentioned above, since the aim of this section is to emphasise those aspects of the machine which have proved useful in the implementation of the Lehman algorithm (hence too, the emphasis on matrix, rather than vector, mode). A complete description of the language and all the built-in functions can be found in the appropriate ICL reference manual.

In addition to the built-in functions of DAP-FORTRAN, there is a collection of many other useful routines called the DAP Subroutine Library. A full description of the contents (along with some local additions) can be found in a handbook document from QMC. We use one of these subroutines, called

X05_LONG_INDEX

which, when given a matrix and a starting value, n say, considers the matrix to be a long vector, and assigns to the kth element , the value ($k - 1 + n$). Rather than the 4096 assignments that a serial program would need, on the DAP this can be done in only 12 operations (an operation being a shift and an add) after an initial assignment, by means of a technique known as recursive doubling. By way of explanation, we demonstrate this method for a "vector" of length 8 bits, and a starting value of 1.

(i) initialise all the elements to 1	1 1 1 1 1 1 1 1
(ii) shift right 1 place	1 1 1 1 1 1 1
(iii) add	1 2 2 2 2 2 2 2
(iv) shift right 2 places	1 2 2 2 2 2
(v) add	1 2 3 4 4 4 4 4
(vi) shift right 4 places	1 2 3 4
(vii) add	1 2 3 4 5 6 7 8

and we are done. Thus at each stage, we double the number of locations correctly assigned. This technique is not limited to assigning vectors, but can be used in many algorithms to replace, say, N operations with only $\log_2 N$.

Before leaving this chapter, it must be noted that one of the consequences of using an array of bit processors to enable the parallel execution of tasks previously done one after the other, is that some of the operations which would have been done in parallel on a conventional machine now have to be performed serially. For example, a machine with a

word length of 32 bits (e.g. a DEC VAX 11/780) can add two 32-bit integers together in one instruction (in addition to the fetching of the operands and the storing of the result). But the DAP, on the other hand, when dealing with matrices, can only handle one bit of each operand (albeit there are 4096 pairs of such operands) at any given moment. Thus, we cannot expect a speed-up factor of 4096 when comparing the DAP to conventional serial machines - a fact that will be discussed later, in Chapter 6. All arithmetic operations on matrices have to be performed one bit plane at a time by software routines, and so the time taken for a given operation is proportional to the precision being used. Addition and subtraction varies with the length of the operands, division and multiplication with the square of the length (since in DAP-FORTRAN, one is restricted to both operands being of the same length). This makes the DAP very suitable for problems involving the manipulation of integers of arbitrary length since it gives the flexibility of only doing as much work as is necessary on each operation. However, unless one is prepared to store the integers concerned in contiguous sets of bit planes, and write one's own arithmetic routines in terms of logical operations, then the full advantage of the DAP's flexibility is not available to the programmer through DAP-FORTRAN. Unfortunately, if one wishes to add an 8-bit integer to a 32-bit one, the former must be converted to the latter's length (done automatically) before the addition is performed. The same is true for the other arithmetic operations. Even more of a drawback is that if, for example, you wished to multiply two 32-bit integer matrices together, and even just one entry of the answer is larger than $2^{31} - 1$ then overflow will occur, and the program terminated. To avoid this, the whole multiplication would have to be performed in the precision needed by the largest result. It is left to the programmer to do this, probably by making use of the LENGTH statement in DAP-FORTRAN which converts a given operand (the first parameter) to the length specified (which is the second parameter). Unless the maximum size of the numbers concerned was known beforehand, the programmer would have to "play safe" and choose 8-byte precision.

Of course, if a programmer is so concerned about efficiency that he is worried about problems like those mentioned above, then one might ask why he or she chose to compromise by using a high-level language, when they would have been better off writing their program in assembly language. As we will show later, shortcuts which make use of binary patterns can be implemented in DAP-FORTRAN through LOGICAL arrays, but obviously, the improvements in running-time so achieved will not be as large as they would have been, had the code been written in APAL, the DAP's assembly language. However, since programming in a high-level language is so much easier than in assembly code, thus allowing ideas to be developed quickly, it was decided to implement all the routines for this implementation in DAP-FORTRAN. It is worth noting though, that the DAP, when

programmed in APAL, will still be far more flexible than a word-based machine running its assembly language because the latter is still forced to operate on a fixed number of bits per instruction whereas the DAP is not, and so with the processor array, one can take advantage of (known) low-precision numbers, as well as handling much larger integers effectively.

Among the consequences of doing arithmetic one bit at a time are that data movement is fast relative to floating point operations (a fact that is important since it is often thought that the cost of data routing could dominate execution times), and, as we have already noted, logical operations are very fast relative to arithmetic. Parkinson [1980] gives the following table which illustrates these, and other points. The figures on the right are approximate relative operation times for a single DAP processing element, and translate approximately to microseconds on the basis of the 200 nanosecond (approximately) cycle time of the machine.

Single bit Boolean operation	(L1.AND.L2)	1
16 bit fixed point addition	(I = J + K)	10
16 bit data movement	(I = J)	6
32 bit fixed point addition	(I2 = J2 + K2)	20
32 bit floating point addition	(X = Y + Z)	180
32 bit floating point multiplication	(X = Y * Z)	280
32 bit floating point square root	(X = SQRT(Y))	250
64 bit floating point multiply	(X2 = Y2 * Z2)	1000

Perhaps the most surprising fact to be observed in the above is that square root-taking is quicker for matrices than multiplication (and also division). This is because both tasks have to be performed by software and, as we will describe later, the method used for finding roots, is relatively simple, requiring little arithmetic, but mostly shifting. Of course, this fact would not be true of conventional serial machines with their advanced floating-point multiplication units. Thus, because the relative magnitude of certain operations on a bit processor is completely different from that on a word-based computer with a hardware floating-point unit, we find that operations which would normally be considered time-consuming and avoided if possible, are relatively quick on the DAP, and so can form the basis of efficient programs. We will see an example of this in connection with Step 2 of the Lehman algorithm.

In addition to operation times varying with the length of the operands, they also change according to the mode being used. As a rough "rule of thumb", working with vectors and matrices takes respectively, 2 and 10 times as long as the equivalent scalar operations, since the latter are performed by the mcu (The different times for vector and

matrix work are due to the different way in which they are stored in the machine.). This difference in time between scalar and matrix manipulations is taken into account (as it should be!) in the implementation of Step 2 of the Lehman algorithm (see Chapter 5).

One final point worth noting is that it is not possible to get an exact execution time for a program on the DAP at Queen Mary College. Instead, "pseudo-time" (called DAP-time) is used, where an approximate value is calculated on the basis of the number of machine instructions used. However, the author has been assured by staff there that the figures given are accurate, in general, to within 5%.

Chapter 3 The Lehman Algorithm

Factoring integers has long been of interest to mathematicians. In fact, most of the modern algorithms for this task are based on a technique that Fermat used in 1643. However, as commented earlier, despite centuries of work, relatively few breakthroughs have been made. As Lenstra [1982] put it :

"Suppose, for example, that two 80-digit numbers p and q have been proved prime; this is easily within reach of the modern techniques [for primality testing]. Suppose further, that the cleaning lady gives p and q by mistake to the garbage collector, but that the product pq is saved. How to recover p and q ? It must be felt as a defeat for mathematics that, in these circumstances, the most promising approaches are searching the garbage dump and applying mnemo-hypnotic techniques."

Nevertheless, considerable progress has been made in the last 60 years, much of it stimulated by a British army officer, Lt.-Col. Allan J. C. Cunningham. In 1925, he and H. J. Woodall published a volume of tables containing the factorizations of numbers of the form $b^n + 1$ and $b^n - 1$ for $b = 2,3,5,7,10,11,12$, for various high powers of n. By drawing on others' work, in addition to their own, the authors had, in effect, presented a summary of what was then known in the area and, in so doing, had shown how much was still to be done.

During the years that followed, D. H. Lehmer [1933] built several sieve machines (both mechanical and electrical) which he used, to some effect, for factorization and primality testing. However, progress became more rapid with the advent of electronic computers, and hence the availability of surplus cpu hours. As is reported in the historical survey in [Brillhart et al 1983], many factorizations were obtained by programs running at low priority on University machines.

Even so, as Kolata [1983] reports, it was widely thought that the limit of computational feasibility had been reached in 1982. But, as has already been described, by making use of the "supercomputer" architectures now available, further substantial progress has been made possible.

The basis of most of the modern factoring algorithms, as well as the method Fermat used, is the following.

Let N be the positive integer we wish to factor (if $N < 0$, then factorise $-N$), and suppose we have found two integers A, B such that

$$A^2 \equiv B^2 (mod\ N)$$

Then

$$N \mid (A^2 - B^2)$$

i.e.

$$N \mid (A + B)(A - B)$$

and one of these four cases applies:

(i) $N \mid (A + B)$, but $N \nmid (A - B)$;

(ii) $N \mid (A - B)$, but $N \nmid (A + B)$;

(iii) $N \mid (A + B)$ and $N \mid (A - B)$;

(iv) $N \nmid (A + B)$ and $N \nmid (A - B)$, but since $N \mid (A + B)(A - B)$, we must have that "part of N" $\mid (A + B)$ and "part of N" $\mid (A - B)$.

So, if $A \not\equiv B (mod\ N)$ and $A \not\equiv -B (mod\ N)$, then case (iv) applies, and the calculation of $gcd(A + B, N)$ will yield a non-trivial factor of N. Note that

$$gcd\ (A + B, N) = 1$$

implies that $N \mid (A - B)$, which contradicts (iv), and

$$gcd\ (A + B, N) = N$$

implies that $N \mid (A + B)$, which again gives a contradiction. Thus, the integer found will be a non-trivial factor of N, (and so worth finding!)

To date, the two most popular methods have been the Continued Fraction [Morrison & Brillhart 1975] and the Quadratic Sieve [Pomerance 1982, Gerver 1983] algorithms. The former searches for such squares (i.e. for an A as above) among the continued fraction convergents to $\sqrt{N}$, while the latter searches among consecutive integers, starting at $[\sqrt{N}]$. However, due to the very recent publication of the Quadratic Sieve algorithm, it is the Continued Fraction method that has been the means of most of the published results. It was this technique that the team at Sandia Laboratories implemented on a CRAY-1 (and were reported [CACM 1984] to be moving to the CRAY XMP at Los Alamos National Laboratory last year), to achieve the results given previously. Wagstaff and his colleagues have designed and built their own machine to perform this algorithm. One of the features of their computer is the ability to perform trial divisions in parallel. After attempting to

implement the Continued Fraction algorithm on the DAP (while some lessons were learnt [Parkinson & Wunderlich 1984] the programs were never completed), Wunderlich is reported [Kolata 1983] to have transferred his work to the MPP. His research was planned with the ILLIAC IV in mind, but because of its eventual destruction, he moved to these other SIMD machines. However, Kolata quotes him as saying: "But my experience on the ILLIAC said to me that this is really the way to do factoring."

Apart from some work by Lehmer on the ILLIAC IV no other research seems to have been carried out into the suitability of the SIMD type of architecture to number theory problems and in particular, to factorisation and primality testing - hence this dissertation.

The factorisation algorithm chosen for consideration was developed by R. Sherman Lehman [1974], who combined the above idea about squares with trial division, to produce a completely deterministic $O(N^{1/3})$ algorithm. This last point is an important one. The Continued Fraction and Quadratic Sieve methods are probabilistic algorithms. By that is meant that there is no guarantee that they will work (i.e. find the required squares). However, when they work (in practise, most of the time), they do so much quicker than the deterministic ones. However the Lehman algorithm is inherently parallel, as Voorhoeve [1982] noted, and so it seemed worthy of further study. The following discussion down to the end of the proof of the Theorem is based on Voorhoeve's statement and proof of the algorithm.

Lehman algorithm:

Step 1:

Trial divide N up to $[N^{1/3}]$. If no factors are found, N is either prime, or the product of two prime factors p, q, with

$$N^{1/3} < p \leqslant q < N^{2/3}$$

Step 2:

For $k = 1, 2, \ldots, [N^{1/3}]$, $d = 0, 1, \ldots, \left[\frac{N^{1/6}}{4\sqrt{k}} + 1\right]$, test whether

$$([\sqrt{4kN}] + d)^2 - 4kN$$

is a perfect square. We will show below, that if any pair (k, d) produces a square, then we have found numbers A and B as described above in (iv).

Thus, the idea is to try and find small prime factors first (if there are any). Then, if there are no small factors, one looks for two larger prime factors via a system of "educated guesses".

That this method exhausts all the possible prime factors of N which are $> N^{1/3}$ and $< N^{2/3}$ is proved in the following Theorem.

Theorem:

If we find no factors in this way, then N is prime. i.e. if N is composite, this method will yield the required factors.

Proof:

We wish to show that if N is not prime, then there exists a pair (k, d) such that $k \in \{1, 2, \cdots, [N^{1/3}]\}$ and $d \in \{1, 2, \cdots, [(N^{1/6}/\ 4\sqrt{k}) + 1]\}$ and

$$([\sqrt{4kN}] + d)^2 - 4kN$$

is a square, and hence Step 2 yields a non-trivial factor of N.

So, suppose that N is composite. Then, since N has no prime factors $\leqslant [N^{1/3}]$ (after Step 1), $N = pq$, where p, q are prime with $N^{1/3} < p \leqslant q < N^{2/3}$.

We assert (and will prove in Appendix A) that there exist $r, s \in Z^+$ (where Z^+ denotes the set of positive integers) with $rs < N^{1/3}$ and $| pr - qs | < N^{1/3}$.

Put $k = rs$. (Note that $k < N^{1/3}$ and $k \in Z^+$ imply that $k \leqslant [N^{1/3}]$, and so $k \in \{1, 2, \cdots, [N^{1/3}]\}$.)

Then we have that

$$4kN = (pr + qs)^2 - (pr - qs)^2$$

which implies that

$$(pr + qs)^2 - 4kN = (pr - qs)^2.$$

Putting $d = pr + qs - [\sqrt{4kN}]$, we have that

$$([\sqrt{4kN}] + d)^2 - 4kN = (pr + qs)^2 - 4kN$$

is a square, as we require for Step 2.

All that remains to show is that $d \in \{0, 1, \cdots, [(N^{1/6}/\ 4\sqrt{k}) + 1]\}$. Now, since $p, q, r, s, [\sqrt{4kN}] \in Z^+$, d is certainly an integer. Putting $A = pr + qs$, $B = | pr - qs | < N^{1/3}$ we have that

$$A^2 = 4kN + B^2$$

which implies that

$$A^2 < 4kN + N^{2/3}$$
$$< \left(\sqrt{4kN} + \frac{N^{1/6}}{4\sqrt{k}} \right)^2$$

i.e.

$$A < \sqrt{4kN} + \frac{N^{1/6}}{4\sqrt{k}} .$$

Since A is an integer, and $[x + y] \leqslant [x] + [y] + 1$ for all $x , y \geqslant 0$, it follows that

$$A \leqslant [\sqrt{4kN}] + \left[\frac{N^{1/6}}{4\sqrt{k}}\right] + 1$$

i.e.

$$d = A - [\sqrt{4kN}] \leqslant \left[\frac{N^{1/6}}{4\sqrt{k}}\right] + 1 .$$

Also, since $B^2 \geqslant 0$,

$$A^2 \geqslant 4kN$$

i.e.

$$A \geqslant [\sqrt{4kN}] .$$

Hence $A = [\sqrt{4kN}] + d$, where $0 \leqslant d \leqslant \left[\frac{N^{1/6}}{4\sqrt{k}}\right] + 1$ (It will be shown later how one can obtain $d \geqslant 1$.).

Thus we have proved that if N is not prime, then Step 2 yields a pair of squares congruent modulo N . Hence, if we find no such squares via Step 2, we can conclude that N is prime. Q.E.D.

Why both $gcd(A + B , N)$ and $gcd(A - B , N)$ yield non-trivial factors of N , for N greater than some limit, can be seen from the following.

$$A^2 = 4kN + B^2$$

implies that

$$A^2 < 4N^{4/3} + N^{2/3}$$
$$= N^2 - (N^2 - 4N^{4/3} - N^{2/3})$$
$$= N^2 - N^{2/3}(N^{4/3} - 4N^{2/3} - 1)$$

$$= N^2 - N^{2/3}((N^{2/3}-2)^2-5).$$

Thus, $A^2 < N^2$ provided that $N^{2/3}((N^{2/3}-2)^2-5) \geqslant 0$, i.e. provided that $N^{2/3} \geqslant 2+\sqrt{5}$.

In other words, $A < N$ provided that $N \geqslant 9$.

This implies that $A+B < N + N^{1/3} < 2N$, so that (since $A+B \geqslant p+q > 1$) $A+B \equiv 0 (mod\ N)$ can only occur if $A+B = N$, and hence $A-B = 4k$. However, this would imply that $2A = N + 4k$ which is a contradiction, since N is odd. Thus, for $N \geqslant 9$, $A+B \not\equiv 0 (mod\ N)$ and so (since $A+B > 1$) $gcd(A+B, N)$ is a non-trivial factor of N.

Also, $A-B = ps + qr - |ps - qr|$ which equals either $2qr$ or $2ps$, depending on whether $ps \geqslant qr$ or not. In both cases we have that $A-B > 2$. In addition, for $N \geqslant 9$ (since $A < N$), $A-B < N$ and so $gcd(A-B, N)$ is also a non-trivial factor of N (for $N \geqslant 9$).

Furthermore, since 8 is not divisible by two distinct primes and 7 is prime, the lower bound for N can be reduced to $N \geqslant 6$, which is the figure that Voorhoeve states.

It was claimed earlier that this algorithm is of order $O(N^{1/3})$. Clearly Step 1 involves less than $N^{1/3}$ operations (in this case, divisions), and so it only remains to show that Step 2 is also $O(N^{1/3})$. The number of operations (where an "operation" consists of processing a (k, d) pair) required in Step 2 is at most

$$\sum_{k=1}^{[N^{1/3}]} \left(\left[\frac{N^{1/6}}{4\sqrt{k}} + 1 \right] + 1 \right)$$

$$\leqslant \sum_{k=1}^{[N^{1/3}]} \left(\frac{N^{1/6}}{4\sqrt{k}} + 2 \right)$$

$$= \frac{N^{1/6}}{4} \sum_{k=1}^{[N^{1/3}]} \frac{1}{\sqrt{k}} + 2[N^{1/3}].$$

Since the function $f(k) = \frac{1}{\sqrt{k}}$ is monotonically decreasing and positive for $k \geqslant 1$,

$$\sum_{k=1}^{[N^{1/3}]} \frac{1}{\sqrt{k}} \leqslant \int_0^{[N^{1/3}]} \frac{1}{\sqrt{k}}\, dk ,$$

and so the number of operations required in Step 2 is

$$\leqslant \frac{N^{1/6}}{4} \int_0^{[N^{1/3}]} \frac{1}{\sqrt{k}}\, dk + 2[N^{1/3}]$$

$$\leqslant \frac{N^{1/6}}{4}\int_0^{N^{1/3}} \frac{1}{\sqrt{k}}\,dk \;+\; 2N^{1/3}$$

$$= \frac{N^{1/6}}{4}\left(\left|2k^{1/2}\right|_0^{N^{1/3}}\right) \;+\; 2N^{1/3}$$

$$= \frac{N^{1/6}}{4}(2N^{1/6}) \;+\; 2N^{1/3}$$

$$= \frac{1}{2}N^{1/3} + 2N^{1/3}$$

$$= \frac{5}{2}N^{1/3}.$$

Hence, the algorithm is indeed of order $O(N^{1/3})$.

General Considerations

Before we go on to consider the implementations of the algorithm, it would be appropriate to mention some of the important decisions, regarding strategy, that have to be made before designing and writing the code.

Step 1

(a) The algorithm requires trial division up to $[N^{1/3}]$ and so we must first calculate an upper bound for the divisors. The only condition that this bound must satisfy is that it be greater than or equal to $[N^{1/3}]$. Of course, if we were to adopt an integer much larger than necessary, then we could waste a considerable amount of time performing needless trial divisions. (The only occasion that these extra trial divisions will not be wasted time, is when one of the two remaining factors of N (p, say) is so much smaller than the other (i.e. is so close to $N^{1/3}$) that it is less than the upper bound we have calculated, and so the factor is found before Step 2 is entered. However, unless we tested what was left of N for primality, we would still have to perform all the iterations of Step 2 (for $k \leqslant [q^{1/3}]$) before we discovered that we had finished, and this might take longer than Step 2 would have taken to produce both factors! Since this is such a rare occurrence, it is not worth trying to capitalise on its happening.) Therefore it is worth taking some care over choosing our limit.

As will be described more fully later, in the serial case we use an "integer-only" version of the Newton-Raphson algorithm, whereas for the parallel implementation,

we use a method more suited to the DAP's architecture.

(b) The next problem to be dealt with concerns how we obtain the trial divisors. Obviously, the best possible solution is to use only primes. However as there are in the region of 600,000 primes $\leqslant 10^7$ and all of these would have to be found and stored beforehand (since there is no formula for the nth prime), it is probably not feasible to adopt this method (and definitely not feasible for N much larger than 10^{21}). Another method has therefore to be used instead. What we want is an easily computable series of integers which contains all the primes and as few composite numbers as possible. (Because we use the divisors in ascending order, it is wasting time to divide N by a composite number, since the latter's prime factors, if originally present in N, will already have been removed (see later).)

Our first thought might be to eliminate all multiples of 2 (except 2 itself), by choosing the set of odd numbers (with 2 added). But, if we list this set

(2), 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, ...

it soon becomes clear that one out of every three numbers is always composite, and so $\frac{1}{3}$ of the trial division time would certainly be wasted. It would be sensible, therefore, to eliminate multiples of 3 (except 3 itself) also, leaving us with the set

(2),(3), 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, ...

which, apart from the first two elements, consists of the integers congruent to 1 or 5 modulo 6 (i.e. integers of the form $6k+1$ or $6k-1$, for some k).

However, viewing this set as residues modulo 30, we find that, after the initial irregularity, we are generating the set of integers which are congruent modulo 30 to one of the following 10 residues:

1, 5, 7, 11, 13, 17, 19, 23, 25, 29

But, since 5 divides 30, those integers which are congruent to either the second or the ninth residues are clearly composite, and so represent wasted time. Removing these two residues leaves us with 8 residues (modulo 30) of interest to us.

Thus, having started by looking at 1 out of every 2 integers, then considering 2 out of every 6, we are now using only 8 out of every 30 integers.

This process can obviously be continued. For example, using the relevant residues modulo 210 will remove multiples of 7 (except 7 itself of course!); residues modulo 210 will cause the removal of multiples of 11, and so on. The following table summarises the position, ignoring the initial slight irregularity.

prime multiples removed	proportion of integers used	successive improvement
2	$\frac{1}{2} = 50\%$	
2, 3	$\frac{2}{6} = \frac{1}{3} \approx 33 \cdot 3\%$	$\frac{1}{6}$
2, 3, 5	$\frac{8}{30} = \frac{4}{15} \approx 26 \cdot 7\%$	$\frac{1}{15}$
2, 3, 5, 7	$\frac{48}{210} = \frac{8}{35} \approx 22 \cdot 9\%$	$\frac{4}{105}$
2, 3, 5, 7, 11	$\frac{480}{2310} = \frac{16}{77} \approx 20 \cdot 8\%$	$\frac{8}{385}$
2, 3, 5, 7, 11, 13	$\frac{5760}{30030} = \frac{192}{1001} \approx 19 \cdot 2\%$	$\frac{16}{1001}$

Thus it is clear that as we remove the multiples of more and more small primes, we are tending to a limit (namely, that of using all primes). As it is impractical to use this limit, we have to stop the process somewhere, and, as will be justified later, we use the residues modulo 30 (and also modulo 210 in the DAP implementation) to generate our set of trial divisors.

However, since small prime factors are much more common than large ones, (e.g. $\frac{1}{2}$ of all the integers are divisible by 2, whereas only $\frac{1}{3023}$ of them have 3023 as a prime factor), it is important to try all the small primes as soon as possible (i.e. avoid wasting time by dividing by composite numbers so early on). It is probably preferable that the first 4,096 primes be stored in a file in the machine, and we start our trial divisions using only these primes, and afterwards, use the residues modulo 30 to generate the subsequent trial divisors. This solution also has the advantage of getting over the initial irregularities with the residues mentioned above.

(c) Once a factor, say p, is found, it should be removed from N (i.e. replace N by $\frac{N}{p}$. This involves no extra work, since the quotient $\frac{N}{p}$ has already been calculated during the initial trial divisions, and has the possible advantage of reducing the amount of work in the subsequent trial divisions, since the new N might require fewer multiple-precision digits.

A further, more substantial, gain can be made as follows. Once a factor, say p, is found, we could, in effect, start the whole algorithm again, this time to factor $\frac{N}{p}$. Thus our upper limit will now be $\left[\left(\frac{N}{p}\right)^{1/3}\right]$ (or slightly greater), which could be much less than $[N^{1/3}]$. Of course, we do not need to re-try all the earlier divisors, but can start the divisions with the next trial divisor after p (unless multiple occurrences of p were not checked for before, in which case we should re-start with p). This involves the calculation of $\left[\left(\frac{N}{p}\right)^{1/3}\right]$ either from scratch or by considering $\left[\frac{N^{1/3}}{p^{1/3}}\right]$. As will be seen later, in the serial version, we use the former method so as to avoid the problems with round-off error which using the latter could cause, whilst in the parallel version we again use a method more suited to the DAP's architecture. (In the case when the factor found, p, occurs to a power > 1, it is obviously more efficient to wait until all the occurrences of p have been removed before we recalculate the upper bound.)

Step 2

(a) Again we need to obtain a value (to approximate) for $[N^{1/3}]$ to serve as the upper limit for the outer loop, and this is found using the methods previously mentioned . In addition, we need a bound for the counter in the inner loop.

For a given k, the algorithm requires d to take values up to and including $[N^{1/6}/(4\sqrt{k}) + 1]$, and so we have to calculate $[N^{1/6}/(4\sqrt{k})]$ (since $[N^{1/6}/(4\sqrt{k}) + 1] = [N^{1/6}/(4\sqrt{k})] + 1$). Now,

$$\left[\frac{N^{1/6}}{4\sqrt{k}}\right] = \left[\left(\frac{N^{1/3}}{16k}\right)^{1/2}\right]$$

$$= \left[\left(\frac{[N^{1/3}]}{16k} + \frac{\epsilon}{16k}\right)^{1/2}\right]$$

where $0 \leqslant \epsilon < 1$

$$= \left[\left(\frac{[N^{1/3}]}{16k} \right)^{1/2} \right],$$

and it is this last expression which we use for the upper bound, since we already have (an approximation to) the value of $[N^{1/3}]$.

In Step 1, performing a few extra trial divisions (because our upper limit is too large) is not a great disaster, while in Step 2, if N is composite, we are guaranteed the discovery of two squares congruent modulo N before k exceeds $[N^{1/3}]$, and so, again it does not matter if our upper limit for k is slightly too large.

The only case (say, for Step 2) in which such inaccuracy could cause extra work is if N is prime. However, there are much quicker ways of showing that an integer is prime than by using this algorithm, which is primarily a factoring one, and so the use of a primality test at the start of the program would solve this problem. In fact, the use of such a test would eliminate the need for an upper bound for the outer loop in Step 2 altogether, and a suitable one will be presented in Chapter 9.

Now, since the inner loop is performed possibly many times, for each value of k, it is important that our upper limit for d be as small as possible. It has been shown above how the exact value for this bound can be calculated in terms of $[N^{1/3}]$. But, as will be mentioned later, the "integer-only" version of the Newton-Raphson algorithm returns a value for $[N^{1/3}]$ which could be 1 or 2 too large. Even so, it would be an extremely rare occurrence if this resulted in a d loop bound being 1 too large.

(b) Of course, of far more importance than finding upper bounds, is ensuring that the calculations required inside the loops are performed as efficiently as possible.

One trivial point is that of calculating the product $4kN$, and the integer part of its square root, outside the d loop (i.e. once for each value of k) rather than calculating it afresh for each value of d. For the serial version, the value of $[\sqrt{4kN}]$ is calculated using the Newton- Raphson iterative method (again, an "integer-only" version), while, in the parallel implementation, a method which exploits the bit-processing capabilities of the DAP is used.

The addition, squaring operation and subtraction, required by the inner loop, are

performed using the traditional arithmetic algorithms (see [Knuth 1981] for examples of these), and it is the final task, that of ascertaining whether or not we have found a perfect square, that requires the most consideration. Because this will have to be done possibly many thousands of times, it is vital that it be implemented as efficiently as possible. As will be discussed later, we again find that the best serial method is not the best choice for use on the DAP.

In the serial implementation, possibly many multiplications can be avoided by adding $4N$ (which has already been calculated when $k = 1$) to the product $4kN$, to obtain the value of $4(k+1)N$. Similarly, since $(X+1)^2 = X^2 + 2X + 1$, the value of $([\sqrt{4kN}\,] + 2)^2 - 4kN$ can be found by adding $2[\sqrt{4kN}\,] + 1$ to the previous difference, $(([\sqrt{4kN}\,] + 1)^2 - 4kN)$, thereby avoiding further, time-consuming, multiple-precision multiplications. Versions of both these ideas can also be used in the parallel program.

(c) Finally, whenever $d = 0$, the expression formed in the inner loop $([\sqrt{4kN}\,]^2 - 4kN)$ is either negative, or zero. So it is only in the latter case that we need to look for, and hence find, a perfect square. But,

$$[\sqrt{4kN}\,]^2 - 4kN = 0$$

if and only if $|[\sqrt{4kN}\,]| = |\sqrt{4kN}\,|$

if and only if $4kN$ is a perfect square

if and only if kN is a perfect square

if and only if both k and N are perfect squares

The validity of the last equivalence can be seen from the following.

An integer is a perfect square if and only if the exponents of the primes in its prime decomposition are all even. So, if kN is a perfect square, but $N = pq$, say, is not (i.e. the primes p and q are different), then p, q must be factors of k, each appearing an odd number of times in the set of its prime divisors. But this implies that $k \geqslant pq = N$, which contradicts $1 \leqslant k \leqslant [N^{1/3}]$. Hence N must be a perfect square. That k must also be a perfect square, and that the second-last equivalence holds, can be seen from a similar argument.

Thus, if N is not a perfect square, the $d = 0$ loop is unnecessary, since it can never produce a perfect square. On the other hand when N is a perfect square, there is an

easier way to detect this than by calculating $[N^{1/3}]$, $[(N^{1/6}/4)]$ and $([\sqrt{4kN}]^2 - 4kN)$! So, we can omit the $d = 0$ iteration, preferring instead to start Step 2 by testing for N being a perfect square.

This can also be seen from the proof given earlier, for the $d = 0$ case corresponds to $B = 0$. However, the latter equality implies that $pr = qs$. Now, if N is not a perfect square, $(p, q) = 1$, and so we must have that $q \mid r$. But, this is impossible since $r \leqslant rs = k < N^{1/3} < q$. Hence, unless $p = q$, in which case N is a perfect square (and this can be detected by a test before Step 2), the $d = 0$ iteration is unnecessary.

The above improvement has the effect of reducing the constant implicit in the algorithm being $O(N^{1/3})$, since now the number of operations in Step 2 is at most:

$$\sum_{k=1}^{[N^{1/3}]} \left[\frac{N^{1/6}}{4\sqrt{k}} + 1 \right] \leqslant \frac{3}{2} N^{1/3}.$$

by an argument similar to that given previously to obtain the constant $\frac{5}{2}$.

The constant $\frac{3}{2}$ above can be improved further as follows. We require a bound for the sum

$$S = \sum_{k=1}^{[N^{1/3}]} \sum_{d=1}^{\left[\frac{N^{1/6}}{4\sqrt{k}} + 1\right]} 1 .$$

This is equal to the number of integer points under the graph of $y = \frac{N^{1/6}}{4\sqrt{x}}$ for $1 \leqslant x \leqslant N^{1/3}$. In the previous working we approximated this value by the area under the graph for $0 < x \leqslant N^{1/3}$. However, by changing the order of summation, we can obtain a better bound for S.

$$S \leqslant \sum_{k=1}^{N^{1/3}} \sum_{d=1}^{\frac{N^{1/6}}{4\sqrt{k}}+1} 1$$

$$= \sum_{d=1}^{\frac{N^{1/6}}{4}+1} \sum_{k=1}^{\min\left\{N^{1/3}, \frac{N^{1/3}}{16(d-1)^2}\right\}} 1$$

$$= \sum_{d=2}^{\frac{N^{1/6}}{4}+1} \sum_{k=1}^{\frac{N^{1/3}}{16(d-1)^2}} 1 + N^{1/3}$$

$$= \sum_{d=2}^{\frac{N^{1/6}}{4}+1} \frac{N^{1/3}}{16(d-1)^2} + N^{1/3}$$

$$= N^{1/3} + \frac{N^{1/3}}{16} \left(\sum_{\delta=1}^{\frac{N^{1/6}}{4}} \frac{1}{\delta^2} \right)$$

(where $\delta = d - 1$)

$$\leqslant N^{1/3} + \frac{N^{1/3}}{16} \left(\sum_{\delta=1}^{\infty} \frac{1}{\delta^2} \right)$$

$$= N^{1/3} + \frac{N^{1/3}}{16} \left(\frac{\pi^2}{6} \right)$$

by the well-known result that $\sum_{r=1}^{\infty} \frac{1}{r^2} = \frac{\pi^2}{6}$.

Thus

$$S \leqslant (1 + \frac{\pi^2}{96})N^{1/3} \approx (1 \cdot 1028) N^{1/3}.$$

We will show later how this bound can be improved even further.

It is important to note that the above working simply involves counting the maximum number of (k, d) pairs to be considered, and does not take into account the amount of work involved for each such pair. A more accurate bound on the running time of the algorithm could be obtained by considering the number of bit operations required by each step. But, due to the complicated nature of the work involved in Step 2 which causes one to be unable, at times, to ascertain accurately the length of the operands involved, this analysis is omitted. However, the corresponding treatment of Step 1 is quite straightforward, as will now be shown.

Bit Operation Count for Step 1

To divide an n-bit integer by one of length r bits (where $r < n$) requires $(n - r)$ subtractions, where each of the operands is r bits long, and so requires $r(n - r)$ bit operations. Therefore, to divide an n-bit integer by all the integers with r bits will require

$$2^{r-1} r (n - r)$$

bit operations.

Now, if we assume that N, the number to be factored, has n bits in its binary pattern, then $[N^{1/3}]$ will have approximately $n/3$ bits. Thus, to divide N by all the integers

$\leqslant [N^{1/3}]$ (except 1) will require at most

$$\sum_{r=2}^{n/3} 2^{r-1} r (n-r)$$

bit operations. However, as has been discussed already, we do not use every integer, but only trial divide with 8 out of every 30 possible numbers. Therefore an approximate bound on the number of bit operations required for Step 1 is

$$\frac{4}{15}\sum_{r=2}^{n/3} 2^{r-1} r (n-r) = \frac{4}{15}\left\{ n \sum_{r=2}^{n/3} r\, 2^{r-1} - \sum_{r=2}^{n/3} r^2 2^{r-1} \right\}.$$

This figure (which we shall denote by S) is only an upper bound since we start by using less than 4/15 of the available integers (when trying the first 4096 primes) and, if a factor is found and removed from N , the number of bits in N will, obviously, be less than n - a factor which is not taken into account in the sum.

Now

$$\sum_{r=2}^{m} a^r = \frac{a^{m-1} - 1}{a - 1}$$

and, by differentiating, one can show that

$$\sum_{r=2}^{m} r a^{r-1} = \frac{(m-1) a^{m-2}}{(a-1)} - \frac{a^{m-1} - 1}{(a-1)^2}$$

and

$$\sum_{r=2}^{m} r(r-1) a^{r-2} = \frac{(m-1)(m-2) a^{m-3}}{(a-1)} - \frac{2(m-1) a^{m-2}}{(a-1)^2} + \frac{2(a^{m-1} - 1)}{(a-1)^3}$$

Using the above identities, we can evaluate the previous sum, thereby deriving an expression for the number of bit operations required for Step 1 .

$$S = \frac{4}{15}\left\{ n \sum_{r=2}^{n/3} r\, 2^{r-1} - 2\sum_{r=2}^{n/3} r(r-1) 2^{r-2} - \sum_{r=2}^{n/3} r\, 2^{r-1} \right\}$$

$$= \frac{4}{15}(n-1)\left\{ \left(\frac{n}{3} - 1\right) 2^{(n/3)-2} - 2^{(n/3)-1} + 1 \right\}$$

$$- 2.\frac{4}{15}\left\{ \left(\frac{n}{3} - 1\right)\left(\frac{n}{3} - 2\right) 2^{(n/3)-3} - 2\left(\frac{n}{3} - 1\right) 2^{(n/3)-2} + 2(2^{(n/3)-1} - 1) \right\}$$

$$= \frac{4}{15}\left\{ \left(n - 1\right) \left(\frac{n}{3} - 1\right) 2^{(n/3)-2} - (n-1)2^{(n/3)-1} + (n-1) \right.$$

$$\left. - 2\left(\frac{n}{3} - 1\right)\left(\frac{n}{3} - 2\right) 2^{(n/3)-3} + 4\left(\frac{n}{3} - 1\right) 2^{(n/3)-2} - 4(2^{(n/3)-1} - 1) \right\}$$

$$= \frac{4}{15} 2^{(n/3)-3} \left\{ \left(n - 1\right)\left(\frac{n}{3} - 1\right) 2 - 4(n-1) - 2\left(\frac{n}{3} - 1\right)\left(\frac{n}{3} - 2\right) \right.$$

$$\left. + 8\left(\frac{n}{3} - 1\right) - 16 \right\} + \frac{4}{15}\{(n-1) - 4\}$$

$$= \frac{4}{15} 2^{(n/3)-3} \left\{ \frac{4}{9} n^2 - 2n - 22 \right\} + \frac{4}{15}(n + 3)$$

$$\approx \frac{16}{135} n^2 2^{(n/3)-3} = \frac{2}{135} n^2 2^{n/3}$$

Chapter 4 The Serial Implementation

In order to assess how much the DAP's extra processing elements improve the running time of the Lehman algorithm, we also implemented the method on a serial machine. Factoring integers can take a long time, and during the running of the program it is important that as few other people use the machine as possible, otherwise the many interrupts caused by their work could affect the accuracy of the execution time given. So, because of the availability (on some nights and week-ends) of surplus cpu hours on the VAX 11/780 in the Computing Science Department, this computer was chosen for the implementation. Also, since it was the language with which the author was most familiar, the code was written in Pascal. How the results so obtained will be used for assessing the DAP's performance, will be discussed later, in Chapter 6.

Because the VAX has a word length of 32 bits, the greatest integer it can handle, (of type "integer" in Pascal), is only $2^{31}-1$. Of course, it can handle much larger real numbers, but, for our work, we must know all the digits of each operand, and hence the inaccuracy associated with floating-point storage is unacceptable. In order to manipulate large integers in a high-level language, they have to be broken up into smaller parts, each of which can be handled by the integer routines supplied. When storing operands in "multiple-precision" form (as it is known) we are effectively changing the base of our number system from 10 to a value of our choice. For this machine, any integer less than $\sqrt{2^{31}-1}$ would be suitable (for then the product of two "components" of multiple-precision numbers will still be less than the upper bound on the type integer), but, because of the obvious advantages in multiplying or dividing by a power of 2 (provided the machine "recognises" it), rather than by a number with more general form, as well as others that can be seen from what follows, we chose the largest such power possible, namely, 2^{15}, to be our base.

In order to work with large integers, routines for performing multiple-precision arithmetic are required. In this case, as can be seen from the code, many of these procedures had to be written. Some of them handle two multiple-precision integers, returning a third, while, if it is known that one of the operands will be strictly less than the new base, (i.e. will have only one digit), then, in the interests of efficiency, a special routine is used which takes this into account (identified by the prefix "SingleDigit"). The algorithms used are based on those given by Knuth [1981].

By far the longest procedure is that for dividing one multiple-precision number by another, for which we used Knuth's "divide-and-correct" technique. Admittedly we could have avoided this by multiplying the dividend by the reciprocal of the divisor; a method which, Knuth claims, can be considerably faster than our choice, for "extremely large numbers". However, whether the numbers we are concerned with are large enough to make the second method preferable, is not certain. Even if they were, the use of this alternative approach would pose certain problems. For one thing, we would now be introducing approximations to real numbers into our calculations, since the required reciprocal would have a (possibly infinite) decimal expansion. These would have to be stored as if they were multiple-precision integers, except that the decimal point would be (conceptually) to the left of the digits (base 2^{15}) rather than to the right. Hence we would also have to write routines to sort out where, in the product of the dividend and the reciprocal of the divisor, the integer part stopped, and the fractional part began. In addition, since the remainder might only be found approximately, this method would not be suitable for the trial division of Step 1 (where a non-zero remainder implies that the divisor concerned is not a factor), or for the *gcd* calculation in Step 2. Thus, we would be forced to have two procedures for long division, which, in the interests of simplicity and brevity, is not desirable. A possible improvement to the routine we have used, that could save some time, will be discussed later in this chapter. Meanwhile, we now describe how the steps of the algorithm were implemented.

Step 1

As was mentioned in the previous chapter, we begin the divisions of Step 1 by trying the first 4096 primes. These are read in, one at a time, and the division performed, along with any necessary repetitions. Because not all these primes are less than 2^{15}, we use a boolean flag to indicate when we must stop using the simpler "single-digit-division" procedure in favour of the more general long-division routine. However, the larger primes are still read in as integers (since the 4096th prime is 38,873, which is less than $2^{31} - 1$), and converted to the multiple-precision format by the program. A slight improvement could be gained by also storing the multiple-precision versions of these divisors, thus saving several thousand "div" and "mod" operations. However, the primes would still need to be held as integers as well, since they have to be compared with $[N^{1/3}]$ to discover whether all the divisors necessary have been tried or not, and the comparing of variables of type integer will be quicker than comparing multiple-precision integers.

If, once all these primes have been tried, further division is necessary, then subsequent divisors are generated by calculating the relevant residues modulo 30, namely

1, 7, 11, 13, 17, 19, 23, 29.

Of course, this should not be done by calculating $30k$, for some k , and then successively forming the sums $30k+1$, $30k+7$, etc.. Instead, all one need do is add 6 to $30k+1$ to get the next divisor, then add 4 to obtain $30k+11$, and so on. Adding 2 to $30k+29$ will (obviously) produce $31k+1$, and so no multiplication is ever required. These 8 increments are held in an array, and a counter (which is reduced to 0 whenever it becomes 8) is used to select which of them is to be added to find the next divisor.

This process continues until either N equals 1, or the next divisor is greater than $[N^{1/3}]$, in which case, the work of Step 1 has been completed. As has already been discussed, whenever a factor is found, the upper limit of $[N^{1/3}]$ is reduced accordingly, using an "integer-only" version of the Newton-Raphson algorithm, which we will discuss later. The factors of N which are identified are stored in a one-way linear list, the nodes of which are variant records so that, depending on their size, the primes can be held as either "normal" integers, or in multiple-precision format.

There are various trivial improvements which could have been made that would result in very slight savings of time. For example, since we have chosen 2^{15} to be our base, when trying the prime 2, initially we need only divide the least significant "digit" of N by it, for if 2 does not divide this last component, it certainly will not divide N . Of course, if the binary pattern of N were available, then by counting the number of zeros (if any) which occurred at the least significant end, we could find the exponent to which 2 divided N without any division. But, as this expansion is not easily accessible from Pascal, we are forced to ignore this trick.

That 3 was not a factor of N could be identified without a division of the several (base 2^{15}) digits of N by dividing instead, the sum of the decimal digits of N , which could be formed while N was read in, digit by digit, at the start of the program. However, if 3 was a factor, then a division of N would be required to remove it (and any repeat occurrences).

At present, $[N^{1/3}]$ is calculated before any division takes place, and so several (if not many) recalculations might be necessary before all the first 4096 primes have been tried. This allows us to stop the division as soon as is possible, even if some of the initial block of primes have not been tried (albeit, an unlikely occurrence for large N). Whereas, as will be seen from the next chapter, in the DAP implementation, all the primes must be used, and hence there, the calculation of the upper bound is not performed until after they have been tried. A small gain could be made in the serial version by delaying the finding of $[N^{1/3}]$ until after the first successful trial division, or after the 4096 primes have been tried, whichever occurs first.

Before the discussion of how the second part of the algorithm has been implemented, it would be appropriate at this point, to describe how the necessary square and cube roots are found. The Newton-Raphson iterative process, because of its convergence of the second order, has long been recognised as one of the most suitable algorithms for finding roots of polynomial functions on a word-based computer with a fast floating-point divide operation. It is normally implemented in real arithmetic. But we have chosen to avoid the inaccuracies of floating-point calculations, preferring to work at all times, with integers (multiple-precision or otherwise). Thus we have to use an "integer only" version of the Newton-Raphson method in which successive estimates to the root are rounded up to the nearest integer before the next iteration takes place. This is the same idea that Morrison and Brillhart [1975] used in their work with the Continued Fraction algorithm. Apart from, at times, slightly slower convergence, the only price to pay in making this simplification is that the answer could be larger than the true value (in the author's experience, the error involved has only been in the region of 1 or 2 units). This does not matter, as has already been discussed, when calculating an upper bound for the trial division of Step 1 or the outer loop for Step 2. However, in Step 2, it is important to have the exact value of $[\sqrt{4kN}]$ for each k , and for that reason, there is a procedure called "TruncSquareRoot" which also uses this Newton-Raphson variation to get an approximate value, (to $\sqrt{X}$ say) and then subtracts 1 from the root and compares its square with the value of X until $[\sqrt{X}]$ is obtained.

Each iteration of the Newton-Raphson algorithm requires the calculation of

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right)$$

where x_{i+1} is the new approximation to the square root of a. Since we only require the nearest integer at each stage, the remainder from the division above is not required. However, since in the *gcd* calculation at the end of Step 2, remainders are required, the long division algorithm that has been written always calculates it. Thus a saving could be made by having a second such procedure which does not return a remainder. But, this saving will only be very slight for the size of numbers we are factorising for two reasons: (i) In the algorithm used, finding a remainder basically only requires a "single-digit" divide, which is fast when compared with all the other work involved; and (ii) whenever the previous estimate is less than 2^{15}, a "single-digit" division procedure is used which does not find a remainder. Of course, if N were so large that the second reason above did not often apply, then the alteration under discussion would be worth making.

Because the number of entries would have been too large, we use the Newton-Raphson algorithm to calculate $[N^{1/3}]$ rather than the look-up table adopted for the DAP version (see Chapter 5). The root is found from scratch each time a factor of N is identified, since, depending on the size of the prime discovered, the old value of $[N^{1/3}]$ could be larger than our initial estimate. In both cases we take as our initial estimate (which is calculated "in line", at the start of the procedure), the smallest integer whose square (or cube) has one more (base 2^{15}) digit than the number concerned, so that we are guaranteed to start the process "to the right" of the root (except for a special square root procedure used to find $[\sqrt{4kN}]$ which will be mentioned below).

Step 2

In order that the inner d loop can start at 1 rather than at 0, we begin the second step of the algorithm by testing to see if N is a perfect square. Since N is known to be odd at this stage, it must be congruent to 1(mod 8), and again because of our choice of base, we need only examine the last "digit" of N. If it passes this test, then we calculate $[\sqrt{N}]$ and test to discover if $N = [\sqrt{N}]^2$. If this is true, then we are finished, and so can stop.

If N is not a square, then we proceed to calculate $[N^{1/3}]$, before beginning to search for a pair of perfect squares which are congruent (mod N). It is true that this value is already known, as it was used in the condition which terminated the first part of the algorithm, (if $N = 1$, which is the other terminating condition, then there is no need to use Step 2!), but since it was decided to use separate programs for each step, in order to tell what proportion of the total time each step takes, unless this upper bound was read in as data, there is no alternative but to recalculate it. However, when compared with the time that the required loop iterations can take, the effect of these few extra instructions is negligible.

As mentioned briefly in the previous chapter, several "short-cuts" have been used which reduce the amount of computation required. For example, the product $4kN$ is not calculated afresh, but obtained by adding $4N$ to the previous value (i.e. $4(k-1)N$), thereby replacing two multiplications by one addition (the increment, $4N$, has to be calculated anyway as it is the value of the product when $k = 1$). Another gain was made, this time in the finding of $[\sqrt{4(k+1)N}]$, by making use of the knowledge of $[\sqrt{4kN}]$, rather than starting "from scratch" each time. As is described in Appendix B, various methods were tried, and the best found to be simply using $[\sqrt{4kN}]$ as the initial estimate to $[\sqrt{4(k+1)N}]$ for the Newton-Raphson algorithm. Hence the inclusion of a second square-root procedure which does not calculate an initial estimate, but receives it as a parameter value.

We have already described in a previous chapter, how our knowledge of $[N^{1/3}]$ is used to calculate the upper bound for each d loop. This upper limit is inversely proportional to

k , and, in fact, once k exceeds $N^{1/3}$ only one inner loop iteration per k value is required. Thus, if the number to be factored was either a prime, or required the performing of a considerable proportion of the outer loop iterations before the factors were found, we would eventually be recalculating a bound which was already known to be 1. However when, in the serial version, a flag was set to indicate when this upper limit had become 1, and hence further calculations were unnecessary, the program actually took longer to run! When the boolean flag was later (after the performing of the long timing runs described in Chapter 6) replaced by an integer flag, the program did run slightly quicker than when the d loop limit was calculated afresh each time (37mins 36secs, compared with 37mins 56secs - whereas the boolean flag version took 42mins 45secs), which implies that the version of Pascal running on this particular VAX is not very efficient when manipulating boolean variables.

For those inner loops which have more than one iteration, we have already described how the successive values of the difference $([\sqrt{4kN}] + d)^2 - 4kN$ can be found by adding $(2[\sqrt{4kN}] + 1)$ to the previous one, thus replacing a "single-digit" addition and a multiple-precision squaring and subtraction, with a quicker multiple-precision addition (following the initial calculation of the increment). A further saving in work was made through a variation of a preliminary test which Fermat used when identifying perfect squares.

Knuth [1981] describes how Fermat examined the last two decimal digits of each number concerned, (since there are only 22 possibilities for this last pair of digits if the number is indeed a square), and only if these were in the required set would he proceed with further testing. Because the values we wish to test are not held as decimal integers, these two digits are not easily obtained. However, since we chose 2^{15} as the base of our calculations, the last "digit" of each number holds its 15 least significant bits. It is easy to show that if an integer is a perfect square, then its binary pattern will have one of the following three possible endings:

$$000,\ 001,\ 100\,.$$

Thus, by using this binary equivalent of Fermat's test, we save (at times) a considerable amount of needless calculation. Each square root calculation has to be performed "from scratch" however, since we have no guarantee that we have any previous knowledge to make use of!

Once a pair of squares has been found, the only work that remains is to calculate the greatest common divisor of N and $(A + B)$ (where A^2 and B^2 are the two squares found), which is performed using the well-known Euclid's algorithm [Knuth 1981]. In some cases, finding $gcd\ (N\ , A-B)$ might be slightly quicker, but, compared with the time required

for the loops of Step 2, any difference at this stage will be negligible.

Chapter 5 The DAP Implementation

We begin this chapter by stating the Lehman algorithm in a form suitable for parallel computation. The work that could be performed in parallel is bounded by the words *par begin* and *par end*. For the sake of convenience, separate algorithms are given for the two parts, and no mention is made of testing N for being a perfect square, since this is a one-off exercise which takes place between Steps 1 and 2, and so could be done in parallel with either (as the last process started in Step 1, or the first one started in Step 2).

Step 1 : *while* trial divisors left *do*
    par begin
    select next divisor and current value of N ;
    perform trial division;
    while successful *do*
        repeat division (keeping count of number of successes);
    if successful *then*
        begin
        replace N by quotient produced;
        reduce upper limit of divisors;
        end;
    par end;

It has been assumed that the values of N and the upper limit for the divisors can only be accessed by one processor at a time. The same is true of the count of (k, d) pairs, along with the values themselves, which are used below.

Step 2 : *while* squares not found *and* values of k , d available *do*

par begin

select next k , d pair;

decrease count of pairs by one;

calculate $\left(\left[\sqrt{4kN}\right] + d\right)^2 - 4kN$;

if this is a square *then*

squares found := true;

par end;

It is obvious that if we had any number of independent processors (in an MIMD design), then all the iterations of both loops could be performed simultaneously, in parallel. However, this is not as good as it appears, for several reasons.

(1) For the smaller k values, there will be many more d values to consider than for larger k , and so processors assigned these former values will have far more work to do than those with larger k , while in addition, all of them have a larger workload than those processors engaged on Step 1. Thus, (depending on how soon a pair of squares were found), most of the processors could spend most of their time doing nothing!

Even if we could assign a pair (k , d) to each of the Step 2 processors, rather than just a value k , these processors would still take much longer over their work than those processors involved in the trial divisions of Step 1. Therefore, unless any necessary repeat divisions could be carried out while the Step 2 calculations were still being performed, we would have a very inefficiently-used system.

(2) Since Step 2 is designed to find a prime factor of an integer known to have at most two such divisors, if any small factors were found by Step 1, then all the work of Step 2 might have to be repeated, since there is no guarantee that the first run of Step 2 would have produced the desired factor.

Thus it would be better to adopt a two-stage process, in which we first performed all the trial divisions of Step 1, at the same time, before going on to carry out the iterations of both loops in Step 2 in parallel. For a given N, this scheme would require "only" $1 \cdot 1028\,[N^{1/3}]$ processors (i.e. equal to the total number of (k , d) pairs required by Step 2).

However, it will be noticed that in Step 2, all the processors working in parallel are performing exactly the same sequence of instructions. Hence, for this part of the algorithm, the flexibility of an MIMD system is not required. If one then reconsiders the work of Step

1, it will be seen that a similar situation exists there except that the tasks will not be identical if factors are found, for then the processors involved have to re-divide N, (each time removing the corresponding divisor), until all the repeat occurrences have been dealt with. Thus, while an SIMD-type machine would be sufficient for Step 2, its use for Step 1 could involve a slight inefficiency. But, since the processors in an MIMD design have to be more complicated than those used in SIMD systems, building a computer of the former type with a large number of processors would be extremely expensive. Hence an SIMD architecture would be the more practical choice, even though it could result in a small amount of inefficiency in Step 1.

But, in adopting this design, we are faced with a further problem - that of initialising the processors so that both loops of Step 2 can be performed simultaneously. It would require a considerable amount of ingenuity to share out the (k, d) pairs efficiently (since the number of d values varies with k) on a DAP-style of architecture, and we could find that the time taken to set up the process would be greater than that required by the actual calculations! Whereas if we adopted the simpler scheme of one k value per processor (and thus, less than $[N^{1/3}]$ processors would suffice for both steps), then we would find that most of the processors spent most of the time waiting for others to finish, since the upper bound for the d loop is very large (depending on N) for small k, but decreases rapidly until, for most of the k values, d need not exceed 1. Although it would certainly run very quickly, it would be hard to justify the construction of a machine with a vast number of processors, most of which were almost always idle!

Of course, it is very unlikely that the building of such a machine would ever be considered. In the real world, one has to settle for more modest facilities, in this case, 4096 processing elements, arranged in a 64×64 array, and a compromise found between running time and the efficient use of resources.

It will be described later how, at times, the DAP will be forced to do more work than is required by the algorithm. However, it will also be shown how, on occasion, this can actually result in considerably reducing the execution time of the algorithm.

To get the best out of a machine with such a definite structure as the DAP, one has to adapt one's programming style to suit the particular details of the computer's design. Thus the number 4096 will appear many times in what follows, but, contrary to what one might have expected, it does not prove to be a "mill stone round the neck" of the programmer. While it is true that, as they say, "you can't win them all" and that sometimes we have to compromise and tolerate certain inefficiencies, at other times, such a rigid bound as this can actually reduce the amount of work required and prompt the design of simpler algorithms.

Step 1

An example of this last point can be seen in connection with Step 1. Since trial division involves repeating the same operation with just a different divisor, and each repetition is independent of the previous one, (e.g. whether or not 7 is a factor of N does not have any bearing on whether 13 divides N), the SIMD architecture of the DAP is suited to this process (except, of course, when repeat divisions have to be performed). Thus we will divide N by 4096 trial divisors simultaneously. Hence, we no longer need to calculate $[N^{1/3}]$. Instead, we only require to find the number of blocks of divisors we will need to use. As will be seen, this can be found in a very simple way on the DAP. But first we consider the formation of the blocks of divisors, and explain why residues modulo 30 and 210 have been chosen.

As has already been justified in Chapter 3, we start by trial dividing with the first 4096 primes. These can be read in by the host program, and transferred to the DAP by means of a named COMMON block. Since the 4096th prime is 38,873, what follows will concern the construction of matrices of divisors in increasing magnitude, where the first element of the first matrix will be the next divisor under the respective "system", after 38,873.

Working modulo 6

The residues (modulo 6) of interest are 1 and 5, and since $38{,}873 \equiv 5 \,(mod\ 6)$, the first matrix of divisors will contain the 4096 elements :

38,874+1 ; 38,874+5 ; 38,880+1 ; 38,880+5 ; 38,886+1 ; 38,886+5 ; $\cdots$ 51,156+1 ; 51,156+5 , (where $51{,}156 = 38{,}868 + (6 \times 2048)$). Such a matrix can be formed simply in DAP-FORTRAN by using a mask to pick out, first, those elements which will contain a trial divisor congruent to 1 (modulo 6), and then (using .NOT. mask) the other elements, which will hold numbers congruent to 5 (modulo 6).

The second matrix will start with the integers:

$$51{,}162+1\ ; \quad 51{,}162+5\ ; \quad 51{,}168+1\ ; \quad 51{,}168+5\ ;$$

and so on. Now,

$$51{,}162 = 38{,}874 + 12{,}288,$$
$$51{,}168 = 38{,}880 + 12{,}288,$$

and so it is easy to see that the second matrix can be obtained from the first by merely adding 12,288 ($= 6 \times 2048$) to every element of it. Indeed, any divisor matrix (except the first, of course) may be so obtained from its predecessor.

Thus, when using these residues, the first matrix of trial divisors can be quickly formed, and subsequent ones easily found. However, as discussed earlier, we have included many numbers which we know to be composite. For example, over 800 elements in each divisor matrix will be divisible by 5. In fact, out of the first 5 blocks, we waste, through multiples of 5, the equivalent of 1 block! So it is wise to investigate the relevant residues to other moduli, in order to find a more efficient compromise for the matrices of divisors we use.

Working modulo 30

Using this modulus will ensure the removal of spurious multiples of 5, leaving us with the eight residues:

$$1, 7, 11, 13, 17, 19, 23, 29\,.$$

Now, since $38{,}873 \equiv 23 \,(mod\ 30)$, we will actually use the residues in the following order:

$$29, 1, 7, 11, 13, 17, 19, 23\,.$$

and the first matrix of trial divisors will be

$$38{,}850 + A$$

where A is the matrix

$$\begin{bmatrix} 0\times 30 + 29 & \cdot & \cdot & \cdot & 504\times 30 + 29 \\ 1\times 30 + 1 & \cdot & \cdot & \cdot & 505\times 30 + 1 \\ 1\times 30 + 7 & \cdot & \cdot & \cdot & 505\times 30 + 7 \\ 1\times 30 + 11 & \cdot & \cdot & \cdot & 505\times 30 + 11 \\ 1\times 30 + 13 & \cdot & \cdot & \cdot & 505\times 30 + 13 \\ 1\times 30 + 17 & \cdot & \cdot & \cdot & 505\times 30 + 17 \\ 1\times 30 + 19 & \cdot & \cdot & \cdot & 505\times 30 + 19 \\ 1\times 30 + 23 & \cdot & \cdot & \cdot & 505\times 30 + 23 \\ 1\times 30 + 29 & \cdot & \cdot & \cdot & 505\times 30 + 29 \\ 2\times 30 + 1 & \cdot & \cdot & \cdot & 506\times 30 + 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ 8\times 30 + 23 & \cdot & \cdot & \cdot & 512\times 30 + 23 \end{bmatrix}$$

Since $4096 = 512 \times 8$, each residue will appear 512 times in this matrix (and in the other ones also) and so the next matrix will have as its first few elements:

$$38{,}850 + (512\times 30) + 29\,; \quad 38{,}850 + (513\times 30) + 1\,; \quad 38{,}850 + (513\times 30) + 7 \ \ etc,$$

and again we have that subsequent divisor matrices can be obtained by adding a constant (in this case 15,360) to each of the previous ones.

While the setting up of the first matrix of trial divisors using residues modulo 30 will take slightly longer than that for the residues modulo 6 case (we now need 8 masked assignments rather than 2) this will be much less than the time saved through the more efficient choice of divisors.

Working to larger moduli

As has already been stated, this selection of divisors tends to a limit, namely, using only primes, and because it is not practical to adopt this limit, we have to compromise, and make do with a system of the form we have just been describing. However, if we wish to get close to the limit, by choosing residues to a larger modulus, we find that the problems of implementation on the DAP could outweigh the advantages.

For example, working modulo 210, there are 48 residues of interest. However, $4096 = 85 \times 48 + 16$ and so, unlike the first two cases, the number of residues does not map "nicely" onto the DAP. We have two possible choices: (i) Start each matrix with a divisor congruent to $29 \ (mod\ 210)$ since $38{,}873 \equiv 23 (mod\ 210)$, the 7th residue of interest, and so be forced to re-use the last sixteen divisors tried at the end of the previous matrix, thereby effectively wasting 16 processing elements; or (ii) start the first matrix with the 7th residue, the second matrix with the 23rd residue, the third matrix with the 39th residue, and then the fourth with the 7th residue, and so on. While the latter certainly makes full use of the processing elements, it does require 3 matrices, each of which will need 48 masked assignments for its setting up, whereas the former requires just one matrix (and hence 48 masked assignments to form it), but at a cost of processor utilisation. How quickly the DAP can perform the assignment of a scalar to selected elements of an integer array will determine how large N needs to be before the time saved by this choice of divisors will be greater than the time taken for the additional initialisations.

The situation is even worse if we decide to remove spurious multiples of 11 also, by using the 480 relevant residues modulo 2310. Unfortunately 480 is not a divisor of 4096, and there are 256 residues "left over" after every block. As before, we either "waste" that number of processing elements, or use 16 different matrices (since $4096 = 16 \times 256$) in order that the recalculation of the divisor matrices can be done in terms of the previous values. Thus we will need either 480 or 7,680 masked assignments to perform the necessary initialisations - requiring N to be even larger to make this extra work worthwhile.

Divisor Generation

Before choosing which method to adopt, let us examine the general question of generating trial divisors, via residues of different moduli, for use on processor arrays like the DAP, of varying sizes. Because of the obvious advantages in efficiency of addressing, we will limit our study to arrays of size p, where p is a power of 2.

As described above, there are two ways in which a processor can fail to do useful work: (i) by using a trial divisor which is composite - in this case we shall say that the processor is "wasted" and (ii) by always having to use a divisor which has been tried

already, composite or otherwise, because the number of relevant residues does not divide exactly the number of processors (where we are choosing to have only one matrix of residues, and not several) - such processors we shall call "idle". Thus we have two quantities to measure (i.e. the number of processors in each of the above categories), and two quantities to examine the effect of varying, namely the size of the processor array, and which modulus we work to, when selecting residues.

We start by considering the 8 relevant residues modulo 30, and a processor array of size $1024 = 32^2$. Now since 8 is a factor of 1024, there are no processors idle. But many are wasted. We have only removed the multiples of the primes 2,3,5 and so $\frac{1}{7}$ of all the divisors will certainly be composite (since 7 is a factor). Similarly, $\frac{1}{11}$ of the divisors will be multiples of 11 . Of course, some of the divisors divisible by 11 will also have 7 as a factor, and so, when calculating the number of composite divisors (and hence wasted P.E.'s) we must be careful not to count some more than once! Thus, the number of processors whose divisor is either a multiple of 7 or 11, or both, is

$$\frac{1}{7}(1024) + \frac{1}{11}(1024) - \frac{1}{11}\left(\frac{1024}{7}\right) = \frac{10}{11}\left(\frac{1024}{7}\right) + \frac{1}{11}(1024)$$

$$= A \text{ , say.}$$

When we take into account multiples of 13, this count increases to

$$A + \frac{1}{13}(1024) - \frac{1}{13}(A) = \frac{12}{13}(A) + \frac{1}{13}(1024) = B \text{ , say.}$$

By counting divisors which are multiples of 17, we find that at least

$$B + \frac{1}{17}(1024) - \frac{1}{17}(B) = \frac{16}{17}(B) + \frac{1}{17}(1024)$$

processors will try composite divisors. Hence, the number of wasted processors will be greater than

$$\frac{16}{17}\left(\frac{12}{13}\left(\frac{10}{11}\left(\frac{1024}{7}\right) + \frac{1}{11}\left(1024\right)\right) + \frac{1}{13}\left(1024\right)\right) + \frac{1}{17}\left(1024\right) > 330 .$$

Performing the same calculation for arrays of different sizes produces the following table.

array size	idle processors	wasted processors*	% not used*
$16^2 = 256$	0	82	32
$32^2 = 1024$	0	330	32
$64^2 = 4096$	0	1323	32
$128^2 = 16384$	0	5292	32

*These figures are actually just lower bounds, since we have only counted multiples of 7,11,13 and 17 and so not included all the divisors which are multiples of 19, 23 ... etc..
The last column is an indication of what proportion of the processors do not contribute anything to the solution, since they are either idle or wasted, and so is calculated using the expression (idle + wasted)/(array size).

If we remove all the multiples of 7 by using the 48 residues of interest (modulo 210), we find that, irrespective of which array size we choose (assuming that it is the square of a power of 2) 16 processors are idle. Thus, to calculate the number of wasted processors, say for an array of size 1024, we use the formula

$$\frac{16}{17}\left[\frac{12}{13}\left[\frac{1008}{11}\right] + \frac{1}{13}\left[1008\right]\right] + \frac{1}{17}\left[1008\right],$$

where 1024 has been replaced by 1008 (= 1024 − 16) since idle processors cannot also be wasted! The figures which apply this time are presented below:

array size	idle processors	wasted processors*	%not used*
256	16	50	26
1024	16	211	22
4096	16	857	21
16384	16	3440	21

When we remove divisors with 11 as a factor, we find that the number of idle processors does not remain constant. We have that

$$\begin{aligned} 1{,}024 &= 2\times480 + 64 \\ 4{,}096 &= 8\times480 + 256 \\ 16{,}384 &= 34\times480 + 64, \end{aligned}$$

and since $64\times4 \equiv 256\ (mod\ 480)$, and $256\times4 \equiv 64\ (mod\ 480)$, the idle processors alternate in number between 64 and 256. To calculate a lower bound for how many

processors are wasted, we now use the expression (say, again, for an array of 1024 processors):

$$\frac{16}{17}\left\lceil\frac{960}{13}\right\rceil + \frac{1}{17}\left\lceil 960\right\rceil$$

(where 960 = 1024 - 64), and so calculate the entries in the following table:

array size	idle processors	wasted processors*	%not used*
1024	64	125	18·5
4096	256	503	18·5
16384	64	2141	13·5

No mention is made of a 16×16 array, since $256 < 480$, and so the use of residues (modulo 2310) would not be possible when using a single matrix of residues.

There are 5,760 residues of interest (modulo 30,030), which makes using this modulus impossible with all but the largest of our suggested arrays. But, since $16{,}384 = (2 \times 5{,}760) + 4{,}864$, we find that 4,864 processors would be idle, and at least $\frac{1}{17}(16{,}384 - 4{,}864) > 677$ processors wasted. Thus, in excess of 5541 ($\approx 33{\cdot}8\%$) of the processors would not be useful, unless many different matrices were used to generate the residues.

If we compare the results for a 64×64 array, we have the following:

modulus	primes removed	idle	wasted	not useful	%
30	2, 3, 5	0	1323	1323	32
210	2,3,5,7	16	857	873	21
2310	2,3,5,7,11	256	503	759	18·5

with the use of residues 30,030 not possible. From this we can see that a large gain is made by choosing residues modulo 210, rather than modulo 30, even without having 3 matrices, (in order to avoid the 16 idle processors). However, the improvement from using the next residue (namely 2310) is not nearly so significant.

We could, in fact, eliminate the overhead of setting up the matrix of residues by forming it beforehand, and then reading it in, along with the matrix of primes, at the start of each factorisation. Thus the drawback with using the moduli 210 and (especially) 2310 can be removed, and so it would appear that using residues modulo 2310 would be best, (that we have not included in our count of non-useful PE's, those with divisors which are multiples of 19, 23 or higher primes, should not affect this conclusion). This is not surprising, as we remarked before about how the process of choosing divisors tends to a limit. What is interesting is that, by deciding to use only one matrix for the residues, as we approach this limit, the best processor utilisation we can have is only 80%, unless we adopt the limit, and hence use all the processors. Unfortunately, as we have already remarked, unless we are prepared to fill vast amounts of backing-store with matrices of primes, this limit is not practical. Thus, the best we can hope for is having 80% of the processors doing useful work.

Rather than adopting the optimal solution, it was felt that a comparison of the moduli 30 and 210 would be more worthwhile, as we could then see whether or not, in practice, there was such a great difference between the two, as the figures above suggested. Thus, both choices were implemented.

Calculation of Limit

Now that methods for forming the blocks have been chosen, we are in a position to discuss how to find the number of blocks required. Since the same process is used for both moduli, we describe the work involved only for the modulo 30 case.

The last elements of the first four, say, blocks of these divisors are, respectively,

$$\begin{aligned} 38{,}850 + (512 \times 30) + 23 &= 54{,}233 \\ 38{,}850 + 2(512 \times 30) + 23 &= 69{,}593 \\ 38{,}850 + 3(512 \times 30) + 23 &= 84{,}953 \\ 38{,}850 + 4(512 \times 30) + 23 &= 100{,}313 \end{aligned}$$

and the 4096th block will end with

$$38{,}850 + 4096(512 \times 30) + 23 = 62{,}953{,}433$$

As will be justified later, it is enough for our purposes, for this implementation to be capable of factoring integers $\leqslant 2^{72}$, and since $62{,}953{,}433 > 2^{24} = 16{,}777{,}216$, the last element of the last block of divisors which we use, will lie between 54,233 and 62,953,433.

Thus, if we were to form a matrix whose i^{th} element (when considered as a long vector) is $38{,}873 + i(15{,}360)$, and set a mask by comparing this matrix with $[N^{1/3}]$, the

number of blocks required should (see later) equal SUM(mask) + 1. However, this still involves the calculation of $[N^{1/3}]$. Also, if we know the value of $[N^{1/3}]$, then a much easier way of calculating the number of blocks required would be to find

$$\frac{([N^{1/3}] - 38{,}879)}{15{,}360},$$

because, rounding up this value to the nearest integer, would give us the required bound.

But, we can avoid calculating $[N^{1/3}]$ by finding instead, the number of blocks, the cube of whose last element, is less than N. This can be done by just cubing the matrix of last elements described above, comparing it to N, and setting those elements of a logical mask which correspond to cubes less than N, to .TRUE.. Again, SUM (mask) + 1 should equal the number of blocks required.

One might be tempted to ask why we should bother with such a scheme, when we could have used the "integer-only" version of the Newton Raphson algorithm. The reason for this choice follows from the fact that the finding of the number of blocks of divisors needed is not a "one-off" calculation, but will need to be repeated every time a factor is found. So, while the initial use of the Newton-Raphson algorithm might take a similar amount of time to this scheme (depending, of course, on how close our initial estimate is), the finding of subsequent bounds would require more arithmetic operations, rather than the comparison and use of SUM which suffices when using this look-up table.

We could have used the ceiling (i.e. rounding up) of

$$\frac{N - (38{,}879)^3}{(15{,}360)^3}$$

to find the number of blocks required, but this too, will eventually involve more calculations than the matrix approach (depending on the number of factors of N which are $\leqslant [N^{1/3}]$). But, this method has a far greater disadvantage (shared with the use of Newton-Raphson) from the point of view of implementation, which will become apparent when we consider the details of program design.

There is a slight disadvantage with this method which was hinted at above, namely, if N lies between the cube of the last element of the kth block, say, and the cube of the first element of the $(k+1)^{th}$ block, then $[N^{1/3}]$ will equal the 4096th element of the kth block, and so only k blocks are required, whereas our method would indicate that $(k+1)$ were needed. This is not as bad as it seems because, once a factor of N is discovered, the bound will be recalculated, and we would be very unfortunate if the same thing happened again. However, such worries can be eliminated by calculating instead, the cubes of the first elements of the blocks. In other words, we could use the matrix whose elements are the

cubes of

$$38{,}850 + (512 \times 30) + 29 = 54{,}239$$
$$38{,}850 + 2(512 \times 30) + 29 = 69{,}599$$
$$\vdots$$
$$38{,}850 + 4096(512 \times 30) + 29 = 62{,}953{,}439.$$

But, we still have the drawback that we must use at least one block of these divisors, even though $[N^{1/3}]$ might be less than the last prime used, or less than the first element of the first matrix generated). This problem can be solved by using the matrix whose i^{th} element is the cube of

$$38{,}850 \ + \ (i-1)(512 \times 30) \ + \ 29 \, .$$

The 4096th element will now be 62,938,079, which is still larger than 2^{24}, as we require. It is, in fact, this latter scheme that we use.

Removal of factors

When a factor is found, we remove it from N and then test for any repetitions by re-dividing with the current block. This raises several questions:

(a) Is it not inefficient to re-use all 4096 divisors, when only a few factors need to be checked for repetitions?

The answer is, unfortunately, "yes", but it would be considerably more inefficient to construct a new matrix, some of whose elements were repeat divisors, and the rest were new.

(b) If more than one element in a given block is a factor, should they all be removed from N before repeating the trial division, or just taken one at a time?

Since the re-use of blocks of divisors is inefficient, it should be kept to a minimum. Thus, only removing one successful divisor before repeating the division would be very wasteful in time, if more than one factor had been found in that block. Hence, all the divisors found, in a given matrix, are removed before repeat occurrences are tested for.

As described in Chapter 2, the first element of the matrix DIVISORS, say, which corresponds to a 1 in the LOGICAL matrix MASK can be selected by using the DAP-FORTRAN statement :

element = DIVISORS(FRST(MASK)).

By repeating this process, each time setting to .FALSE. the bit in MASK corresponding to the element obtained, all the factors found in a given block can be obtained easily. If there is more than one bit set in MASK (which can be discovered by using the DAP-FORTRAN function SUM), a running product is made of the primes selected, so that a scalar division can be used to obtain the new value of N. (Since, as will be discussed later, no multiple-precision long-division routine has been written, once the product is about to exceed 2^{31}, a "single-digit" division is used to remove those factors whose product is less than the base.)

Thus we see that, in order to gain the advantage of performing many trial divisions in parallel, we have to make slight comprises from the standpoint of processor utilisation. However, since we might need to try several thousand blocks of divisors, the inefficiency in having to re-use several is not so significant after all. Indeed, compared to the time that could be required for Step 2, which, in general, is far greater than that for Step 1, questions concerning how many milliseconds we have wasted are not really relevant!

Of more concern, however, is the problem already mentioned in the general points, that if the smaller of the two large factors of N, p say, is very close to $[N^{1/3}]$, then removing it, and using Step 2 to show that what is left of N, q say, is prime, might take longer than to factor pq using Step 2. However, a primality test will be described later, the use of which, before Step 2, will solve this problem.

Multiple-precision arithmetic

It would be appropriate at this point, to describe briefly the way in which we handle the large integers required by the algorithm, on the DAP.

The largest integer which DAP-FORTRAN can manipulate as a variable of type INTEGER is $2^{63}-1$ and so, as with the serial case, we will have to use multiple-precision arithmetic. Like before, the numbers will be stored in arrays, except that this time we will choose our base to be 2^{31} (the largest power of 2 with the property that the square of the previous integer (i.e. $2^{31}-1$) is less than or equal to "maxint"). Apart from the obvious advantage, already mentioned, which is gained from dividing by a power of 2, which requires only shifting, rather than by an integer of a more general form (but only if the compiler recognises it!), such a choice of base means that the binary digits of each multiple-precision integer are easily accessible, since they have simply been divided into groups of 31 bits, each of which can be found in the right hand end of the array elements.

Since, with this choice of base, the multiple-precision digits will be $\leqslant 2^{31}-1$, they can each be stored as INTEGER*4 variables, where only the sign bit is "wasted", thus ensuring efficient use of storage. Further efficiency is obtained by calculating beforehand

how large each array needs to be, assuming that $N \leqslant 2^{72}$. Thus, some arrays have only two elements, some three, and some have four "digits". A count is also stored, as before, of how many elements in each array are actually used. Unfortunately, as has already been mentioned, when we wish to multiply two such digits together, they have both to be converted to 8 byte precision, to avoid integer overflow. However, this cannot be avoided if one wishes to use DAP-FORTRAN.

Of course, since we are using the DAP, most of the variables will have 4096 values to be operated on simultaneously, and so we will be manipulating arrays of matrices, rather than just the one-dimensional arrays of the serial case. Hence, the multiple-precision arithmetic routines which have to be written are more complicated than before.

[Note that a consequence of working in parallel is that our terminology is now in danger of becoming very confusing. It is unfortunate that the word array can refer to two completely different structures : the 64×64 array of PE's in hardware, or an array of variables in software, such as in the languages FORTRAN or Pascal. In order to reduce this ambiguity, we will (for the most part) restrict the use of the term "array" to refer to the way of connecting processing elements together employed in the DAP (e.g. 64×64 array of PE's). The word "matrix" (and "block") will continue to describe a set of numbers to be operated on simultaneously by the processor array, with one number stored per processor. When dealing with multiple-precision integers, we will talk of them as having several "components" or "digits", and whenever we wish to manipulate such numbers in parallel, we will use a "multiple-precision matrix" to hold them, where one matrix will contain all the first components, another matrix will store all the second components, and so on.]

While, in some ways, the multiple-precision arithmetic is more complicated than before, one consolation (which, admittedly, is common to all machines supporting 64-bit integers), is that, since our base is so large, we can divide a multiple-precision integer by decimal integers up to and including $2^{31}-1$ (and for $N \leqslant 2^{72}$, all our trial divisors lie in this range), using the simpler "single-digit-divide" algorithm, rather than the very complicated long-division algorithm which we had to resort to in the serial version. This is the disadvantage with the other methods of finding a value for $[N^{1/3}]$ described earlier. Even if they were to prove slightly quicker than our matrix method, there are other areas of the program where more significant speed gains could be achieved with the same amount of programming effort as would be required to implement a long-division algorithm (although, of course, Knuth's algorithm will probably not be the most suitable for a bit processor). Indeed, implementing such a procedure in DAP-FORTRAN could be considered a waste of time, for reasons that will be given later.

Because we wish to divide N simultaneously by a matrix of trial divisors, we need to have a multiple-precision matrix with identical elements, each of which equals N, that will be re-initialised by broadcasting to each processor the quotient obtained after the removal of all the factors found, in the event of there being any.

The factors, however, can still be stored in a scalar array (of type INTEGER*4), with a second array to hold the corresponding exponents.

Step 2

Here again we have a loop (the k loop), each repetition of which is independent of the others, and hence is suitable for implementation on a SIMD machine. Thus we will perform 4096 outer loop iterations in parallel, using a matrix to hold the counter values. So, as before, the upper bound we need is not the exact value of $[N^{1/3}]$, but rather the number of blocks of k values required. We find this by the same procedure as in Step 1, except that, since k takes successive integer values, starting with $k = 1$, the first elements of the first four, say, blocks are:

$$1, \quad 4096 + 1, \quad (2 \times 4096) + 1, \quad (3 \times 4096) + 1,$$

and so the matrix of first elements this time, has as its ith element

$$((4096 \times i) + 1)^3,$$

where $1 \leqslant i \leqslant 4096$. The first element of the first block of values is not included in this matrix because N will certainly be greater than 1!

Using just this one matrix, we can find the k loop bound for all integers $N < (4096 \times 4096 + 1)^3 = (2^{24} + 1)^3$. Now, because the purpose of this exercise is to compare the performance of a serial machine with the DAP, we are restricted in our choice of input data by the time the serial implementation would take to complete its task. Integers with, in the region of, 20 decimal digits can take up to many hours of cpu time to factor with this algorithm on a serial machine, which suggests that using numbers any larger than this would not be practical. Since 2^{72} has 22 decimal digits, and is within the range mentioned above, we have restricted the input data for both programs to integers which are $\leqslant 2^{72}$. (We could, in fact, have used integers which are $< (2^{24} + 1)^3$, but this bound is not so convenient as a power of 2.) How the DAP implementation can be extended to deal with larger integers will be mentioned later.

In the general points discussed in Chapter 3, it was remarked that the upper limit for each d loop would be found in terms of $[N^{1/3}]$, rather than $N^{1/6}$, since we had already calculated the former value. But, with the above scheme, we have succeeded in avoiding doing this! However, the exact value of $[N^{1/3}]$ can be found by repeating the above, after replacing the matrix of first elements with the last matrix of values with the property that

N is greater than or equal to the cube of its first element. (Because N is known, at this stage, to be the product of at most 2 primes, it cannot be equal to the cube of an integer, and so "strictly greater than the cube of its first element" will always be the case.) If a mask is set, as before, to indicate those elements (cubes) which are strictly less than N (again noting that N cannot be a perfect cube), then

$$[N^{1/3}] = \{(number\ of\ \ k\ blocks\ required) - 1\} \times 4096 + SUM(MASK)\,.$$

The above is just a "one-off" calculation, and so we do not require the advantage of rapid recalculation of upper bounds which favoured this approach for Step 1. However, when using this method (unlike the "integer-only" version of the Newton-Raphson algorithm, which would require further multiplications to test its accuracy), we are guaranteed the return of the exact value of $[N^{1/3}]$. So, while the Newton-Raphson method might be quicker here, the simplicity and accuracy of this matrix algorithm is very appealing. Of course, saving (or wasting) a few milliseconds over a calculation performed only once is not of the utmost importance. So, since the code for the above was already written for Step 1, and because it constitutes, to a certain extent, "better parallel programming", it was decided to use this matrix technique again.

When, after finding the value of $[N^{1/3}]$, we proceed to calculate the upper limit for the d loops ($= [\,(\,[N^{1/3}]\,/\ 16k\,)^{1/2}]\,)$, we encounter another problem (which has already been mentioned) : the number of iterations in each d loop is inversely proportional to k. Hence, for a given block of k values, the corresponding d limits could vary, with those at the start of the block being larger than those at the end. Thus we might contemplate calculating a matrix of d limits, and using a mask to indicate which processors (and hence which iterations of the k loop) still had d values to use. In practice (depending on the size of N), it is only the early blocks (probably only the first block) of k values which have a wide spread of corresponding d limits, and so, for the sake of perhaps only the first block, we would have introduced a large amount of extra logical testing (unnecessary for most of the k loop iterations), and have made the code more complicated to read. While the time required for all the masked assignments is negligible on the DAP, this approach could still, in fact, have an extremely detrimental effect on execution time.

As has already been remarked, processors masked out in an assignment still perform any calculations involved, and are only excluded from the storing of the new value. Thus, using a masked assignment does not save any work or time. In fact, it takes slightly longer, though this extra time is negligible. While in many situations, masking can prove a very powerful tool in selecting values of interest, its use here to ignore what some (and eventually many) processors are doing, is pointless - indeed, inefficient. Even if what the

processors were doing was of no interest, allowing them to continue working for d values larger than the relevant limit, would not do any harm, since in this case, d loops for different k's are completely independent. It would, in fact, make the program much simpler to read and understand, (and cause a (probably) slight improvement in execution time). However, a much more spectacular gain can result in letting all the d loops run for as long as the smallest k in the block requires. We will show later that, if the prime factors of N, p and q, where $q > p$, are such that $\frac{q}{p}$ is very close to a simple rational number (i.e. a fraction with small numerator and denominator), then by searching for d's in a range larger than that given above, a pair (k, d) and hence a pair of squares congruent modulo N, can sometimes be found in less time than Voorhoeve's version of the Lehman algorithm requires.

Thus, by choosing to perform the k loops in parallel, and hence the d loops serially, what could have been considered inefficient, namely, doing far more work for certain k values than is necessary (and which, admittedly, will be an (unavoidable) waste of time for many integers), will, for some numbers, prove very advantageous indeed, as will be demonstrated later.

Another, though less significant, gain which results from this is that the matrix of d loop limits (as well as the matrix arithmetic operations needed to form it each time) is no longer necessary, and a scalar, initialised to the bound corresponding to the first element of the k matrix, is all that is required to control the loop. As has already been mentioned, scalar arithmetic is approximately ten times quicker than the equivalent matrix calculations to the same precision.

Much more important than calculating bounds efficiently, is performing each of the possibly many loop iterations in an optimal way. As has already been noted, adding 4096 to each of the values in the current block will produce the next block of k values. Hence, the next product matrix can be obtained by adding $4 \times 4096 \times N$ to each element of the current one (and calculating this increment requires no extra work since it will already have been found as the last element of the first product matrix).

Similarly, once in the d loops, subsequent values of $([\sqrt{4kN}] + d)$, which are stored in the (software) array of matrices called AM_DIGITS, say, can be obtained by just adding 1 to each of the elements of the previous matrix. The other "short-cut" used in the serial case, namely, finding $([\sqrt{4kN}] + d + 1)^2 - 4kN$ by adding $2([\sqrt{4kN}] + d) + 1$ to $([\sqrt{4kN}] + d)^2 - 4kN$ can also be used here except that the scalar increment is replaced by the matrix (or matrices, if multiple-precision working is required) equal to 2(AM_DIGITS) + 1. The effectiveness of such "tricks" as these, which replace multiplications with additions and subtractions, is directly proportional to the difference in

time between multiplication and addition on the computer concerned. Thus, on a machine like the DAP where this difference is so large, these modifications are very worthwhile; more so than on a word-based machine like the VAX.

However, of much more significance will be the way we choose to find the required square roots, and identify perfect squares. As will be justified later, the success of this parallel implementation of the Lehman algorithm is probably partly due to the method we use, and so it will now be described in detail.

Finding Square Roots

Gostick [1979] describes the algorithm used for the DAP-FORTRAN function SQRT as follows, where N is the number whose square root we wish to find. The aim is to find x such that $x^2 = N$, to the precision we have decided to work. The root is constructed, bit by bit, in the following way.

Suppose, after n steps, we have found the first n bits of x, giving us an n-bit number x_n which approximates x to this number of bits, and satisfies $x_n^2 < N$. The next approximation, x_{n+1}, will have the same first n bits, and the $(n+1)^{th}$ bit will be chosen such that $x_{n+1}^2 \leqslant N$. (If equality is the case, then we are obviously finished.) To see how this is done, let $x_n = 0 \cdot a_1 a_2 \cdots a_n$, $b_{n+1} = 0 \cdot 0 0 \cdots 0 \beta_{n+1}$, where $\beta_{n+1} = 0$ or 1, and is preceded by n zeros after the binary point, and let r_n denote the error at the nth stage. Then

$$r_n = N - x_n^2$$

Now

$$x_{n+1} = x_n + b_{n+1} = 0 \cdot a_1 a_2 \cdots a_n \beta_{n+1}$$

and so

$$\begin{aligned} r_{n+1} &= N - {x_{n+1}}^2 \\ &= N - (x_n + b_{n+1})^2 \\ &= r_n - (2x_n + b_{n+1}) b_{n+1} . \end{aligned}$$

Compute $(2x_n + b_{n+1})b_{n+1}$ with $\beta_{n+1} = 1$. If this expression equals r_n (which implies that $r_{n+1} = 0$) we may finish. Otherwise, if taking $\beta_{n+1} = 1$ makes $r_{n+1} > 0$, then put $x_{n+1} = 0 \cdot a_1 a_2 \cdots a_n 1$, while, if r_{n+1} would be < 0 with $\beta_{n+1} = 1$, we should make $x_{n+1} = 0 \cdot a_1 a_2 \cdots a_n 0$. Thus, starting with a first approximation $x_0 = 0$, we form the root digit by digit. If, at any stage, we find that $r_n = 0$, then the corresponding x_n is the exact root, and we terminate there.

As Gostick points out: "No arithmetic is needed in forming the error terms. $2x_n$ is x_n shifted one place to the left, producing a 0 at the right; addition of b_{n+1} is then done by putting a 1 at the right of this 0 and the final multiplication by b_{n+1} is a shift of $n+1$ places to the right." He then goes on to claim that: "when the program is worked in detail, it is found that the number of operations required is about half that for a division to the same number of bits." This explains the unusual fact mentioned earlier, that finding the 4096 square roots of a real matrix takes less time than multiplication (or division) for real matrices.

Unfortunately, this algorithm is only designed to handle numbers in floating-point format (and thus, to find the square root of an integer in DAP-FORTRAN, it must first be converted to type REAL), whereas all our operands are integers, often with values spread over several integer variables. So, at first sight, this algorithm is not applicable to our problem.

However, as we have remarked before, the DAP is very effective when operating on LOGICAL variables. Also, because LOGICALS are stored as single bits, we can equivalence an INTEGER variable to a LOGICAL array, and so immediately gain access to the binary pattern of the integer concerned. (In this section, contrary to our previous convention, the word "array" *will* be used to refer to the software structure, since one-dimensional arrays of LOGICAL scalars, and later, one-dimensional arrays of LOGICAL matrices, will be required in the implementation of this algorithm. However, no ambiguity should arise, as we will not be referring to the structure of the DAP for some time.)

If we could obtain the binary pattern (stored as a LOGICAL array) of the multiple-precision integer whose square root we required, then we could imagine that there was a binary point immediately to the left of the bits, and so have the number in a suitable form to use the above algorithm. This can be done by equivalencing each INTEGER*4 "digit" of the number, to a LOGICAL vector and then copying bits 34 → 64 into the required positions in a LOGICAL array. An array with 32 elements cannot be used in the EQUIVALENCE statement, because of the way scalars are stored in the DAP : an integer scalar is stored in one row (aligned to the right), whereas arrays (even LOGICAL ones) are stored with only one element per row. Of course, this restriction does not apply when equivalencing INTEGER matrices and arrays of LOGICAL matrices, since then, both variables are stored "vertically".

Our first approximation equals 0, and so the first error term will be $r_0 = N - x_0^2 = N$. Thus, the binary pattern of N should be copied directly into the LOGICAL array (called REM) which holds the value of r_n . In addition, we need two similar arrays to hold the value of $(2x_n + b_{n+1})b_{n+1}$ and the difference between it and r_n

(called NEWR and DIFF respectively); as well as a smaller array, by the name of X, to hold the binary pattern of the root, as it gets built up. Since, at times, we require to shift patterns one place to the left, we adopt the convention for all the arrays, that the binary point lies between positions 1 and 2. Thus the most significant digit of N will be copied into the second element of REM, and so on.

One problem we have not yet dealt with, is how to detect when to terminate the algorithm. If we were to continue the process until $r_n = 0$ for some n, then, unless N were a perfect square, we would never decide to stop. In fact, we would not need to make such a decision, since the system would make it for us, by terminating the whole program, when we tried to exceed the bounds of one of the arrays! Even if we could continue indefinitely, all we would be finding would be more and more of the digits after the binary point in the expansion of the root, (and, since the square root of a non-square integer is irrational, we would never have $r_n = 0$). But, all we need is the integer part of the square root, and so the algorithm should be terminated before it starts to calculate the root's fractional part. This can be done by counting, for if N has $2n$ digits, for some n, then

$$2^{2n-2} < 2^{2n-1} \leqslant N < 2^{2n}$$

which implies that

$$2^{n-1} \leqslant [\sqrt{N}] < 2^n .$$

In other words, $[\sqrt{N}]$ has exactly n digits. Similarly, if N has $2n-1$ digits, then

$$2^{2n-2} \leqslant N < 2^{2n-1} < 2^{2n}$$

which implies that

$$2^{n-1} \leqslant [\sqrt{N}] < 2^n,$$

and so, again $[\sqrt{N}]$ has exactly n digits. Thus, if N has d binary digits, say, then after $[(d+1)/2]$ ($= l$, say) steps, the integer part of the root will have been found, and will lie in locations $2 \rightarrow (l+1)$ of the array X. If the remainder term, r_l, is zero at this point, then it follows that N is a perfect square, and so the exact root has, in fact, been calculated.

The above is, of course, just a binary version of the method taught in many schools, for finding the square root of a decimal number (integer or floating-point), in which, by a process not unlike long-division when written down, the decimal digits of the number are taken two at a time, and the root subsequently produced. However, when the operand has an odd number of digits, then the first digit has to be handled on its own, while the rest are used in pairs. In order to avoid making a special case of such numbers (in an implementation), a zero could be actually inserted to the left of the first (i.e. most

significant) digit of such operands. Then we would always be dealing with numbers which had an even number of digits, and so the digits could always be taken "two by two".

Since this is also true for the binary version, whenever the integer N has an odd number of binary digits, we store a zero (or .FALSE.) in position 2 of the LOGICAL array, and let the binary pattern of N begin in the third location.

To show how all of this works in practice, we now give two examples.

Example 1

Let $N = 33 = 100001_2 = 0 \cdot 100001 \times 2^6$

Put $x_0 = 0$

$r_0 = 0 \cdot 100001$

Try $\beta_1 = 1$: Shift x_0 1 place to the left : $00\cdot$

Addition of b_1 : $00 \cdot 1$

Now shift 1 place to the right : $0 \cdot 01$

$r_1 = r_0 - 0 \cdot 01 = 0 \cdot 100001 - 0 \cdot 01 = 0 \cdot 010001 > 0$,

therefore keep $\beta_1 = 1$, and so $x_1 = 0 \cdot 1$.

Try $\beta_2 = 1$: Shift x_1 1 place to the left : $1 \cdot 0$

Addition of b_2 : $1 \cdot 01$

Now shift 2 places to the right : $0 \cdot 0101$

$r_2 = r_1 - 0 \cdot 0101 = 0 \cdot 010001 - 0 \cdot 0101 < 0$,

therefore put $\beta_2 = 0$, and so $x_2 = 0 \cdot 10$, and $r_2 = r_1$.

Try $\beta_3 = 1$: Shift x_2 1 place to the left : $1 \cdot 00$

Addition of b_3 : $1 \cdot 001$

Now shift 3 places to the right : $0 \cdot 001001$

$r_3 = r_2 - 0 \cdot 001001 = 0 \cdot 010001 - 0 \cdot 001001 = 0 \cdot 001000 > 0$,

therefore keep $\beta_3 = 1$, and so $x_3 = 0 \cdot 101$.

Now, $[7/\ 2] = 3$, and so we should stop at this point. Since $r_3 \neq 0$, the algorithm indicates, correctly, that 33 is not a perfect square. Also

$$[\sqrt{33}] = 0 \cdot 101 \times 2^3 = 101_2 = 5$$

as required.

Example 2

Let $N = 25 = 11001_2 = 0 \cdot 011001 \times 2^6$

Put $x_0 = 0$

$r_0 = 0 \cdot 011001$

Try $\beta_1 = 1$: Shift x_0 1 place to the left : $00 \cdot$

Addition of b_1 : $00 \cdot 1$

Now shift 1 place to the right : $0 \cdot 01$

$r_1 = r_0 - 0 \cdot 01 = 0 \cdot 011001 - 0 \cdot 01 = 0 \cdot 001001 > 0$,

therefore keep $\beta_1 = 1$, and so $x_1 = 0 \cdot 1$.

Try $\beta_2 = 1$: Shift x_1 1 place to the left : $1 \cdot 0$

Addition of b_2 : $1 \cdot 01$

Now shift 2 places to the right : $0 \cdot 0101$

$r_2 = r_1 - 0 \cdot 0101 = 0 \cdot 001001 - 0 \cdot 0101 < 0$,

therefore put $\beta_2 = 0$, and so $x_2 = 0 \cdot 10$, and $r_2 = r_1$.

Try $\beta_3 = 1$: Shift x_2 1 place to the left : $1 \cdot 00$

Addition of b_3 : $1 \cdot 001$

Now shift 3 places to the right : $0 \cdot 001001$

$r_3 = r_2 - 0 \cdot 001001 = 0 \cdot 001001 - 0 \cdot 001001 = 0$,

therefore keep $\beta_3 = 1$, and so $x_3 = 0 \cdot 101$.

Now, after three steps, we have correctly identified 25 as being a perfect square (since $r_3 = 0$), and

$$\sqrt{25} = 0 \cdot 101 \times 2^3 = 101_3 = 5$$

as required.

Thus, summarizing the above, to find the integer part of the square root of a multiple-precision integer (e.g. the number to be factored, N) we perform the following:

(a) Calculate the number of digits (denoted by d, say) in the binary pattern of N. (This can be done either by searching for the first 1 (or .TRUE.) in the most significant (base 2^{31}) digit of N, or by expanding this digit to 8-byte precision, equivalencing it to a LOGICAL vector, and then using the DAP-FORTRAN functions FRST and ELN to

produce the index of the first .TRUE..

(b) If d is even, then copy N into the LOGICAL array REM, starting with the most significant bit in position 2, while if d is odd, replace d with $d+1$, store .FALSE. in REM(2), and start the binary pattern of N in the third location of the array.

(c) Perform the above β loop $\frac{d}{2}$ times.

(d) Copy elements $2 \rightarrow \{(d/2)+1\}$ of the LOGICAL array X into the right-hand end of an INTEGER variable (since $[\sqrt{N}] \leqslant 2^{36}$).

Notes

(1) If all of locations $2 \rightarrow (d+1)$ of REM contain zeros (i.e. are .FALSE.), then N is a perfect square. Thus, this method is suitable for use at the start of Step 2 to discover if N is a perfect square.

(2) Testing the contents of REM every time to check if N is a perfect square is not advisable. It is far more probable that N is not a perfect square, and so the checking of REM, element by element (until a .TRUE. is found), on each step, would be a waste of time. Indeed, its effect could be significant, even though the process only involves (quick) logical tests, because the rest of the algorithm also consists of logical manipulations. Thus, we chose to ignore the possibility of finishing early, and instead, perform the β loop $\frac{d}{2}$ times before testing for a zero remainder.

(3) Consider the following table which shows the number of digits in the binary patterns (that is, after the binary point, as described above) of the following numbers:

i	x_i	$(2x_i + b_{i+1})b_{i+1}$	x_{i+1}
0	0	2	1
1	1	4	2
2	2	6	3
3	3	8	4

Thus, after n steps, $(2x_{n-1} + b_n)b_n$, (note that $i = 0$ on the first step), has $2n$ bits after the binary point. Hence, on the jth step (when $i = j-1$), only $2j$ binary digits need to be

subtracted from (the more significant end of) REM. This was taken into account when writing the subroutine for subtracting two binary patterns stored as LOGICAL arrays, in order to eliminate unnecessary subtracting by zeros. Similarly, if $\beta_{i+1} = 1$, rather than recalculating r_{i+1}, all we require to do is to replace the contents of locations $2 \rightarrow (2j+1)$ of REM, with the corresponding elements from DIFF. The other case, (i.e. $\beta_{i+1} = 0$), can be easily detected, for then the difference will be negative, and so after the subtraction, the borrow will still be 1 (i.e. .TRUE.).

Since these subtractions are the only arithmetic involved in the algorithm, it is easy to see that, if shifts are ignored, the number of bit operations required to find the integral part of the square root of a $2n$-bit integer is equal to

$$\begin{aligned} & 2+4+6+\cdots+2n \\ = \; & 2\sum_{i=1}^{n} i \\ = \; & 2.\frac{n}{2}(n+1) \\ = \; & n^2+n \end{aligned}$$

which is, indeed, less than half the work required to form the quotient of two such numbers, as Gostick stated.

(4) It would appear from the examples that we need to have two places to the left of the binary point, in order that we may shift $x_0 = 0\cdot$ one place to the left, making it equal to $00\cdot$. This is, of course, unnecessary. Rather, on the first step, we do not shift x_0 at all, while on subsequent steps, we only shift locations to the right of (and including) the second.

The only drawback with the above is that it finds just one square root each time, whereas, after the initial test for N being a perfect square, we will require 4096 roots to be calculated simultaneously (for example, the values of $[\sqrt{4kN}]$ for a block of k values. As we have already remarked, since each root-taking operation is independent of the other ones in the same k block, and involves using the same algorithm on just different data, such a task is suitable for an SIMD machine. Thus we require a version of the above method which manipulates arrays of LOGICAL matrices rather than scalars. (For this section, we will continue to go against our convention, and use the term array to describe the software structure.)

It is at this point that we encounter a problem. Since the k's vary in size throughout each block, so, obviously, will the products $4kN$. Thus we could find that some of the binary patterns to be manipulated are much longer than others, and so finding some roots

will require more steps than others. We faced a similar situation earlier, with the d loops, except that then, the extra work carried out by some processors, rather than being a hindrance to us, could, in some cases, prove very beneficial. However, the same is not true here.

Once the integer part of a square root is found, further steps will only supply the beginning of the binary expansion of the fractional part of the root, and since we only require the integer part, these extra bits will never be of use. Indeed, they could complicate our working, because now we would need to know where in each LOGICAL array the integer part stopped, and the fractional part began! There is certainly no way to avoid performing all the steps required to find the square root of the largest entry in the matrix concerned, even if the smallest root could be found after only half this number; this is a consequence of the particular architecture we are using. But, unless care is taken in setting up the matrix of LOGICAL arrays (actually declared as an array of LOGICAL matrices), we could complicate matters and "make life much harder" for ourselves.

For example, if we had copied the binary patterns of the 4096 numbers of interest into the start of each array, then, before beginning the algorithm, we would have to calculate how many digits of each root array would be of interest, and store these values in another matrix. Once all the steps had been completed, the required number of bits from each array could be copied into the less significant end of an INTEGER variable - or could they? The problem here is that if some roots had more digits than others then, for example, their least significant digits would lie in different store planes in the DAP. It is not possible, in one instruction, to obtain a LOGICAL matrix (or a matrix of any other variable type), some of whose elements come from one store plane, and some from another (since, as was remarked earlier, the DAP can only process one store plane at a time). Masked assignments would be the only way of doing this. Thus, to obtain the least significant digits of the 4096 roots could require rather complicated logical masking. Of course, this problem is not unique to the least significant digits, and so we might have to use a considerable amount of masking to obtain the integers we require. That such a state of affairs is bound to happen, at least in the early steps, can be seen from the following.

We need to find the value of $[\sqrt{4kN}\,]$ for 4096 values of k simultaneously. The first element of the first product matrix is $4N$, while the last element of the same block is $4\times4096\times N$ which equals $2^{12}\times$ (the first element) . Thus, in the first step, the binary patterns of the products will differ in length by up to twelve digits. The first and last elements of the second block are, respectively, $4\times4097\times N$ and $4\times8192\times N$ (almost twice the first), and so some of the binary patterns will almost certainly (depending on N) have one more bit than others. In the third matrix, the last product is nearly $\frac{3}{2}$ times the first

one, while in the fourth block, the factor is almost $\frac{4}{3}$ and so in both cases, all the binary patterns need not have the same length. In general, for $n > 1$, the last element of the nth product matrix is almost $\frac{n}{n-1}$ times the first element, which implies that, while the probability of it happening decreases as N increases, in any block we could be faced with this problem of the roots being of different lengths.

Of course, in the cases where the binary patterns of the products differ in length by at most one (i.e. all blocks except the first one), if the smaller patterns have an odd length, say $(2d - 1)$, then all the roots will be the same length, namely d. In this case too, assigning the LOGICAL arrays for the initial product is very simple. In location 2 we either store a 0 (i.e. .FALSE.) or the most significant bit (i.e. .TRUE.), depending on whether the product has length $(2d - 1)$ or $2d$ bits respectively. The next location will either hold the most significant digit for the smaller numbers, or the second most significant bit for the larger. In both cases, the digit stored is the $(2d - 1)^{th}$ bit (counting the least significant bit as first), and so they all will lie in the same store plane to start with. The same is obviously true for all subsequent array elements, and indeed, for the first element as well, because, whenever the binary pattern of an integer does not fill the number of bytes allocated to it, the sign bit is duplicated to fill up the remaining bits, and in this case, since the products (along with every number we deal with) are positive, the sign bit is a 0 as we require. It could be the case that the $(2d)^{th}$ bit (of the larger numbers) occupies the least significant position of another multiple-precision digit. But, since multiple-precision numbers are "padded out" with zeros, where necessary, again the elements of this store plane corresponding to the smaller products, will already contain the required zero. Thus, the assignment of the matrix of LOGICAL arrays is simply a matter of transferring whole planes of store.

Such a method is clearly not limited to the situation when the patterns differ in length by only one bit, but could also be used for the first block, in which case up to twelve 0's might be stored at the front of some of the binary patterns. Again, this will happen "automatically" for the reasons given above, and so we do not need to know how long each binary pattern is. Instead, all we require is the length of the longest one, which can be found by searching for the first plane (starting at the most significant end of the most significant digit), in the relevant matrix of multiple-precision numbers, which does not contain all zeros. (It does not matter, for our purposes, how many 1's there are in this plane.) Depending on whether the length of the longest binary pattern(s) is odd or even, we either store a plane of zeros in location 2, and start copying at position 3, or begin initialising in position 2. We could use an EQUIVALENCE statement and then perform the necessary shifting to remove the first planes of each INTEGER*4 digit, since these are sign bits that we

will not require, as well as any "surplus" zeros at the front of the binary patterns due to the corresponding multiple-precision integers being smaller than $(2^{124} - 1)$. However, as the latter would involve changing the values of the INTEGER matrix concerned, we chose the former method.

Under this scheme, once the steps have been performed, all the integer parts of the roots will lie "right-aligned" in the LOGICAL matrix array, and so can be shifted, plane by plane, into an INTEGER matrix of the required precision. That some roots might be extended to the left with zeros does not matter, since these bits in the corresponding INTEGER variables would hold zeros anyway (duplicated sign bits).

Thus we have a situation where, if time has to be "wasted" with some processors doing needless tasks, it is much better that this happens at the start of the calculations, rather than at the end.

Another interesting point is demonstrated here. Many serial algorithms contain loops of the form "do until done". This square root algorithm, as originally described, is such an example. If the error, r_n becomes zero before we have finished the required number of steps, then we can stop, because the integer is a perfect square, and so the remaining bits in its binary pattern will be zero. (We chose to ignore the possibility of finishing early, due to the overhead of checking for zero (or .FALSE.) all the components of the error array each time.) However, in the parallel case, we can only stop early if all the tasks are done, and to detect this could require a considerable amount of logical testing. Thus, with an algorithm like ours, since it is very unlikely that all 4096 matrix elements will be perfect squares, it is probably better to calculate beforehand the maximum number of steps we will require, given the data, and then perform them all; in other words, adopt a "for loop" rather than the "do until" construction.

Of course, in the case of the products $4kN$, we have already shown that none of these will be perfect squares (since we only need to calculate them if N is not itself a perfect square), and so we know beforehand that there is no point in looking at the r_n ! However, the same is not true when searching for a pair of squares congruent modulo N. We have already described how this algorithm can be used to identify perfect squares, and because this method, with its logical manipulations, is so suited to the DAP's bit-processor design, it was decided to use it here, for that purpose. But since, for the majority of cases, the numbers tested will not be perfect squares, it is better to wait until the end of all the necessary steps, before testing for an r_n which equals zero, thus saving many needless (and, in the context, expensive) logical tests.

As has already been mentioned, processors masked out still perform the same calculations as the active ones, and only avoid the final assignment. Thus, when trying to stop early, no work will be saved until each of the 4096 jobs is done. So, unless it is likely that all the processors could be finished "early", and that the cost of testing their status is small compared to the work involved in each step (not the case here), it is advisable to replace the "do until" condition with a "for loop". Indeed, it could be the case, as here, that the program becomes much simpler by adopting a "for loop", and rearranging the data accordingly.

In the serial version, we only tested for a perfect square if the last three binary digits of the number concerned, were of a certain form. The relevant condition here (at the start) is that, unless the binary pattern ends with 001, the number concerned cannot possibly be a perfect square. We use this as a preliminary test on N at the start of Step 2 (and since N is known to be odd at this stage, we need only examine the second and third least significant bits) to discover if the binary square-root routine is required. However, such a "short-cut" is not suitable when 4096 data streams are being processed simultaneously, for no time would be saved unless some of the entries in the relevant matrix were of the required forms (i.e. 000, 001, 100). Even if just a few of the elements were worth further testing, the entire matrix of LOGICAL arrays would need to be manipulated, since the cost of selecting the relevant arrays for processing, either 64 at a time (using vectors), or one after the other (if there were only a small number of them), would almost certainly outweigh any time saved. Thus, we have another example of a serial improvement which would only slow down a parallel implementation.

Further Points

(1) The limit on the size of numbers which can be handled by this implementation is not, so to speak, "absolute", but was self-imposed, for two reasons.

Firstly, choosing 2^{31} to be the base for our multiple-precision arithmetic enables us to divide multiple-precision numbers by integers up to (but excluding) this value using the comparatively simple "single-digit-division" algorithm, described earlier. To divide by larger numbers would require a more complicated one. In the serial case, we used the long-division algorithm which Knuth gives. Of course, since the DAP is more suited to working at the bit level, this previous choice would not be as suitable here. Instead, we would be better to copy the operands concerned into LOGICAL arrays, and use the well-known binary algorithm, which performs division by repeated shifting and subtraction. Once we had been "forced into" implementing this algorithm to perform 4096 divisions in parallel, we could use it to replace the "single-digit-division" algorithm, with all its INTEGER divisions and

multiplications, which, in the interest of simplicity, was adopted earlier. However, even this would be a mistake, as will be explained in (2) below.

The method we chose to find $[N^{1/3}]$ was the other limiting constraint. In Step 2, with the matrix of first elements, we had that N must be strictly less than $(2^{24}+1)^3$ while, in Step 1, the corresponding bound was $62{,}953{,}439^3 < 2^{78}$. To cater for larger numbers, we could continue to use this look-up table approach by using several matrices of first elements. However, in Step 2, using the second such matrix would only extend the range to $N < (4096 \times 8192 + 1)^3$. A larger bound could be achieved by first comparing N to a matrix whose elements were the first entries of the first 4096 matrices of first elements (although, as before, there is no need to include the first matrix of first elements, and so the ith entry in this initial matrix should be the first element of the $(i+1)^{th}$ matrix of first elements) to find which "normal" matrix of first elements would be the relevant one to form. For Step 2, this modification would only enable us to handle integers $N < ((2^{24} \times 4096) + 1)^3 = (2^{36}+1)^3$. To increase our range beyond this would probably require changing the method used to find cube roots. Since by this stage, we would have implemented a division algorithm to enable us to perform the required trial divisions (for divisors $\geqslant 2^{31}$) we could make use of the Newton-Raphson algorithm here, and so completely remove the upper bound on N. It is also conceivable that we could write a binary algorithm analogous to that used for square-root finding. However, this time, the calculation of the error terms, would not be so simple, for if $r_n = N - x_n{}^3$, then, in the notation used before,

$$\begin{aligned} r_{n+1} &= N - x_{n+1}{}^3 \\ &= N - (x_n + b_{n+1})^3 \\ &= N - x_n^3 - (3x_n^2 + 3x_n b_{n+1} + b_{n+1}^2) b_{n+1} \\ &= r_n - 3x_n b_{n+1}(x_n + b_{n+1}) - b_{n+1}{}^3 . \end{aligned}$$

But, multiplication by 3 is equivalent to adding the operand to twice itself, and so a shift and an addition is all that is required. As before, the multiplication by b_{n+1}, and the forming of b_{n+1}^3, only require shifting. It would appear, though, that a multiplication (i.e. many shifts and additions) is required to form the product of $x_n b_{n+1}$ and $(x_n + b_{n+1})$. But, $x_n = x_{n-1} + b_n$ and so

$$\begin{aligned} &3x_n b_{n+1}(x_n + b_{n+1}) \\ &= 3(x_{n-1} + b_n) b_{n+1}(x_{n-1} + b_n + b_{n+1}) \\ &= (3x_{n-1}^2 + 3x_{n-1} b_n + 3x_{n-1} b_{n+1}) b_{n+1} + \textit{terms involving } b_n, b_{n+1}, \end{aligned}$$

where the first two terms in the brackets are already formed (in the previous step), the third term can be calculated by just a shift and an addition, and the terms involving b_n (which could be zero, thus saving some of the work) and b_{n+1} require only shifts. Similarly, to find the value of $b_n\ b_{n+1} x_{n-1}$ will require only shifts and one addition. Thus, by making use of previous knowledge, the calculation of each successive r_n can be reduced to only several shifts and additions. Hence, this second method should prove ideal for the DAP, as its counterpart did before.

These suggestions have not been implemented since, in studying the algorithm, the aim was to compare a parallel implementation with a serial one, and so we chose to limit our input data, to values which the serial version could process in a reasonable amount of time (i.e. in up to several hours of cpu time). Thus, having $N < 2^{72}$ was quite satisfactory. But, as we have now shown, it is quite possible to remove this restriction by making certain changes, (though, of course, since the multiple-precision numbers could have more components than before, we will also need to alter the assignment of the matrices which hold a count of how many digits each number has). Of course, a binary division algorithm would probably have been slightly quicker than the method we chose, but why it was not implemented will now be discussed.

(2) It was mentioned in Chapter 2 how shortcuts which make use of binary patterns can be implemented in DAP-FORTRAN through LOGICAL arrays. One example of this was the square root routine described above. Another such "trick" is the result of our choice of base. Many of the multiple-precision arithmetic routines which have had to be written involved multiplication and/or division by BASE. But, since it is known beforehand that the value involved is a power of 2, these operations only require the shifting of the operand 31 places, rather than the use of the system INTEGER multiplication or division routines. However, when such savings were attempted, the reduction in running was only slight, due mainly to the overhead of all the indexing that was required. The same would have been the case if the implementation of a long-division algorithm using LOGICAL arrays had been attempted.

Thus, it would be better for such tricks to be implemented in APAL. Certainly, trying a DAP-FORTRAN implementation through LOGICAL arrays, of a low-level algorithm can be useful in showing the feasibility of the idea in question. But, it should not be considered a final version, since much of the effectiveness of the new method would be lost because of the indexing overhead involved.

Two points could be argued from this.

(a) It should not be left up to the users to write any multiple-precision arithmetic routines (including those for square and cube root-finding) that they may require. Instead, such routines should be written, once for all, and made available to all DAP users. In fact, such a provision could serve to attract new users, especially Number Theorists.

(b) What is really required on the DAP is a high-level assembly language, similar to the language C. While still retaining some of the advantages of a high-level language, such a facility would enable much of the potential flexibility of the DAP to be exploited without having to resort to assembly code programming.

When discussing the serial implementation in Chapter 4, it was mentioned how the greatest common divisor calculation needed in Step 2 was performed using Euclid's algorithm. However, because of the size of the numbers involved, this method could require the use of a long-division algorithm, rather than just a "single-digit" one. This was not a hindrance to the version on the VAX, but, as has been discussed above, it does pose a problem for the DAP program, since no such procedure is available. Thus, this calculation is not performed on the DAP, and the program terminates once squares are found and their values printed (or all the (k, d) pairs have been considered). Since, in general, it is the searching for such squares that is the time-consuming part of Step 2 (with the time required for Euclid's algorithm insignificant in comparison), this omission will not affect the conclusions drawn in the next chapter.

(3) While, above, we have referred to the array of processors, and hence matrices of operands, it is not crucial to the implementation that the P.E.'s be arranged in such a way. Indeed, the interconnection network is really irrelevant to us. We certainly do require the ability to broadcast a single value to all the processors (e.g. constants for the assignment of the matrices which store the size of each multiple-precision number, etc.) but being able to shift values from one processor to another, is not necessary for our application. In both Steps 1 and 2 we do make use of this latter feature "subconsciously", in calling the DAP subroutine X05_LONG_INDEX in order to form a matrix whose i^{th} element is, say, i. However, this could be avoided by just reading in the required matrix from backing store.

Chapter 6 Assessment of Results

The problem of assessing the performance of parallel computers is an extremely complicated one with the issues involved not yet fully resolved. Even when one is trying to compare two conventional serial machines, there are many factors which have to be considered, for example :-

- number of registers
- amount of memory, and speed of access
- processor cycle time
- amount of internal parallelism
- language and compiler .

All these can affect the relative performance of two different computers. Indeed, when one is trying to compare two machines, what one is really doing is comparing two systems which comprise both hardware and software, with the latter consisting of not just the benchmark program written by the user, but also any system software used (e.g. compiler). When one considers parallel processors, other factors, including

- number of processors
- power of the processors
- organization of the processors
- interconnection network

must be added to the list.

Since, as the above illustrates, assessing the relative performance of different computers is so complicated, it is worth considering why such a task is ever attempted. There are two reasons :

(i) to assess the advantages, within a given architecture, of modifications such as pipelining, a memory cache, or extra buffers ; and

(ii) to highlight the relative merits of two different machine architectures.

Comparing the different models of a given computer series is very worthwhile - indeed, a practical necessity. Otherwise, potential customers would be unable to make a wise choice concerning which machine was appropriate to their needs, and manufacturers would have no data on the relative worth of various improvements, to guide them in the design of

more powerful computers. In this kind of study, by keeping constant most of the factors mentioned above, while varying a limited number, one is able to produce reliable statistics on the effect of the changes made. In this context, the two performance measures traditionally used for assessing parallel processors, namely :

$$SPEED\text{-}UP = \frac{T_1}{T_p} = \frac{\textit{Time using 1 processor}}{\textit{Time using p processors}}$$

and

$$EFFICIENCY = \frac{SPEED\text{-}UP \textit{ for p processors}}{p},$$

will be accurate guides to the success of the system.

For example, the improvement which is gained over a single CRAY-1 computer by building a multi-processor system comprising, say, p CRAY-1's would be indicated reliably by calculating the ratio

$$\frac{\textit{Time using 1 CRAY-1}}{\textit{Time using p CRAY-1's}}$$

for a wide range of application programs.

Similarly, if one wished to assess the effect of having only 1024 PE's in a DAP rather than 4096, then the ratio

$$\frac{\textit{Time with 1024 processors}}{\textit{Time with 4096 processors}}$$

would provide an accurate guide to the Speed-up obtained, while the quantity

$$\frac{\textit{Speed-up}}{4}$$

would be a fair reflection on the efficiency of the larger system, for a given program. If one wished to know how much quicker, in general, the larger array was, then, as with any general comparison, many different application programs would have to be used in the assessment.

Thus, these parameters are meaningful if one is comparing a single machine with a system made up of several (or many) of the same (or very similar). The measures are also worthwhile when one is assessing the effect of increasing the number of processors in either an SIMD or MIMD system. However, it is unfortunate that these parameters are widely used to compare serial word-based machines with arrays of single bit processors, since they are not at all suitable for this task.

Parkinson [1980] presents two possible reasons for the failure of the Efficiency parameter:

(1) the lack of a true meaning of T_1 for bit-organised systems ; and

(2) the failure of the formula to include any component arising from the overall system organisation.

Certainly, to compare $\frac{T_1}{T_{4096}}$ with $\frac{T_1}{T_{1024}}$ will give a reliable indication of the effect of increasing the number of PE's in the array. However, to substitute for T_1 the time taken by one's favourite 32-bit minicomputer or 64-bit mainframe, would not be fair to the DAP in terms of the resulting Efficiency figure. Yet, Buzbee [1984] has suggested that the Speed-up parameter should be altered to equal

$$\frac{\textit{time for best possible serial implementation}}{\textit{time for this particular parallel processor}} .$$

But, what is the "best possible serial implementation"? Unless one searched for a serial machine which used no form of parallelism or pipelining in its internal processing, one would be inclined to use a computer such as the CRAY-1, which would only compound the inaccuracies of these parameters.

One factor that should be taken into account when assessing SIMD machines is that, just because a processor is idle, (or, as with the DAP, busy, but "masked out"), it does not follow that it is not contributing information to the solution of the problem. An example of this can be seen in how the maximum element of a matrix of type INTEGER is found on the DAP. Since the elements are stored "vertically" all the corresponding bits of each number will lie in the same bit plane. Thus, to find the largest element, all that is required is to consider each bit plane in turn, starting with the most significant, and for each, set to .FALSE. the activity bit of those processors which have not been previously masked out and which, for this plane, contain a zero, (unless, of course, no active processor holds a 1, in which case no action is taken). If, at any stage, only one processor is left active, then it contains the maximum element. Similarly, if more than one processor is still active once all the planes have been considered, then there must have been several occurrences of the largest number in the matrix. Thus it can be seen that those processors which become inactive during this algorithm still play a part in the finding of the required value, since they indicate which numbers are smaller than other elements of the matrix. Yet, the Efficiency parameter, having a value of 3%, does not reflect this.

A more surprising fact that must also be considered is that, to a certain extent, the number of processors, p , in the DAP does not remain constant throughout a program! Certainly, when matrices are being manipulated, then the DAP behaves as a 64 $\times$ 64 array of bit processors. However, since vectors are stored across columns, with all 64 entries in the same bit plane, all the bits of every element in a given vector can be operated on

simultaneously. Thus the DAP performs as if it were an array of 64 processors, each of which could handle 64 bit numbers though not as efficiently as a machine with a 64-bit word since, for example, in the former, carries have to be propagated by software. Finally, when scalar operations are being carried out, the DAP could be considered to be a single-processor system, the power of which is greater than either of the processors in the above two configurations. Thus, if one were to attempt to use the traditional performance parameters, what value should p have? Should it be 4096 or 64, or perhaps an average figure, depending on how many matrix instructions there were in the program, compared with vector ones? Similarly, which "processor" should be used for the calculation of T_1.

From the above discussion, it can be seen how complicated the whole area of performance assessment is, especially for SIMD single bit processor arrays. In addition, serial machines, even with the same word size, vary widely in performance, and so it is not possible to put a single value on how much quicker the DAP is than a conventional computer. Comparisons, for them to be meaningful, can only be made for particular programs and particular machines. (An example of this can be seen in the way that prospective users run benchmark programs as a factor to be taken into account in the purchase of a new machine.) Parkinson [1980] summarised the situation well when he said: "The ratio of the parallel machine to any given machine is therefore a function of the size of the problem with the parallel machine giving its best performance when the number of processors matches the size of the problem. There is therefore no magic single figure which expresses the speed of DAP relative to a given serial computer. The performance is highly application dependent and those worthy gentlemen who spend much time trying to measure performance in terms of arbitrary measures such as GAMM, POWU, MIPS, MEGAFLOPS, etc., are wasting their time."

As was mentioned above, when the execution time of a program on the DAP is compared with the corresponding serial version, what are really being investigated are two systems. Naturally, some systems will favour the parallel machine and others, a serial computer. Therefore, in order to set some standards against which the parallel implementation of the Lehman algorithm could be compared, three other systems were considered, two of which suited the DAP, while the other was designed with a serial computer in mind. Of course, the only part of each system that is within easy reach of the programmer is the user program itself. So it was important to chose these so that for them all, the other factors in the system were approximately the same. For example, both steps of the Lehman algorithm involve, almost entirely, integer arithmetic (the only place floating-point numbers are involved is for the calculation of the upper limit for the d loops - a calculation which is not required for every value of k, since a flag is set to indicate when

the bound equals 1). Thus it would not be valid to compare the relative performance of these programs on both machines, with benchmarks which consisted, mainly, of real arithmetic.

In Chapter 9 a simple prime-generating program (written in Pascal) for the VAX is described and it is mentioned how, when the DAP was treated as an SISD machine, the latter took three times as long as the VAX to perform the same algorithm. Admittedly some floating-point arithmetic was involved in the calculation of the square roots required by the algorithm. However, compared with the amount of integer arithmetic (especially division) needed, the effect of the former should not be significant. It is worth noting that the DAP version could have been made even slower by manipulating matrices, all of whose elements were identical, rather than scalars. But even without this gross inefficiency, it is clear that if one ignores the potential in the DAP's architecture then it is possible to find that a program will take longer to run on this "supercomputer" than on a 32-bit minicomputer like the VAX 11/780.

On the other hand, a task that would clearly suit the DAP would be repeating the same arithmetic operation on a large number of different operands. The parallel execution of 4096 such calculations is achieved by means of matrices in DAP-FORTRAN. Thus a program was written (in DAP-FORTRAN) to compare the time to add two INTEGER*4 matrices together on the DAP with the time taken for a Pascal program on the VAX to perform 4096 additions serially. Both programs involved a loop so that the relevant calculation(s) could be repeated a certain number of times, (as well as some necessary initialisations). Similar programs were written with the addition operation replaced by multiplication. The results obtained for various limits on the outer loop are given below. The times given are in seconds. It should be noted that both computers have the same cycle time (200 nanoseconds).

ADDITION (seconds)			
repetitions	VAX 11/780	DAP	VAX/DAP ratio
500	11·3	0·038	297 : 1
1000	22·6	0·076	297 : 1
10000	225	0·768	293 : 1

MULTIPLICATION (seconds)			
repetitions	VAX 11/780	DAP	VAX/DAP ratio
500	13·2	0·203	65 : 1
1000	26·5	0·420	63 : 1
10000	277·3	4·642	60 : 1

These figures illustrate clearly the much larger difference in time between performing addition and multiplication on the DAP, compared with that required by the VAX. This is what one would have expected, since the former performs these operations bit-serially, and it emphasises how appropriate, for the DAP especially, was the inclusion of some of the shortcuts mentioned earlier (e.g. adding $4N$ to $4kN$).

What would not have been expected is the size of the VAX/DAP ratios. The DAP can simultaneously operate on 128 times the number of bits that the VAX can and so, for addition of 32-bit integers, one would only have expected a speed-up factor in the region of 128 and not 300. However, the numbers used in the above program, while being stored as 32-bit integers, were much smaller (since we were repeatedly forming the sum of the first 4096 positive integers), and this can help to explain the larger speed-ups, for, as the author has been assured by staff at the DAP Support Unit at QMC, the DAP only does as much work as it has to and, therefore, does not need to manipulate all 32 bit planes of each operand. A similar comment applies to the case of multiplication where, because the number of bit operations is proportional to the product of the length of the operands, a speed-up of as little as four-fold could have been expected.

The above results also imply that the speed-up from using the DAP, rather than the VAX, will vary according to the type of work involved, with the VAX/DAP ratio varying in value between 60 and 300. Thus, for a program containing a mixture of instructions, one would hope to achieve an increase in speed of between 150- and 200-fold.

Of course, it must be remembered that these ratios only apply to the case when both machines are performing the same algorithm and so the increase in speed is due simply to the parallelism made possible by the DAP's many processing elements. However, greater speed-ups can be obtained at times, for two reasons:

(1) the algorithm used on the DAP involves less work than the method used on the serial machine would have required; and

(2) less iterations were actually needed to accomplish the required task on the DAP and so, irrespective of whether the algorithm used was more efficient than performing the

serial algorithm in parallel, the processor array finished sooner than would have been expected.

Thus, not surprisingly, the algorithm chosen for a particular task can affect greatly the execution time of the program concerned.

At times it will not be possible to achieve an improvement factor of 200 with the DAP, simply by performing the serial algorithm 4096 times simultaneously, since these extra repetitions might not be necessary. Therefore a different method has to be (developed and) used just so that most, if not all, of the processing elements can be used effectively in the solution of the problem. This is not the case with the Lehman algorithm, however. The loops of both Steps 1 and 2 are suited (except for the first block of k values in Step 2 and the occasional re-division in Step 1) to the SIMD approach, and so performing the serial method for 4096 different data streams in parallel should result in running times over 200 times faster than the VAX. But, as will be seen in connection with Step 2, for various reasons, greater improvements were obtained.

Because, as will be discussed in the next chapter, the time required for Step 2 can be very small, depending on the prime factors of the integer involved, it was decided to compare maximum running times on both machines and so input data was restricted to only primes. Of course this means that the time required to remove a prime factor, p say, from N (i.e. replace N by N / p), and store it in an array, is not included. But as this event will be comparatively rare in general (N has, on average, only $loglogN$ prime factors and, for example, $loglog10^{70} \approx 5$), this omission will not be significant. Similarly, in Step 2, not performing the one-off gcd calculation will not put in doubt the validity of any conclusions drawn.

In order to see how the maximum running time of the algorithm depends on the length of the number whose factorisation is being attempted, primes were chosen whose binary patterns differed in length by a constant amount. The numbers used were the first primes after 2^i, for $i = 35, 40, 45, \cdots 70$, and thus were the values

i	Prime
35	34,359,738,421
40	1,099,511,627,791
45	35,184,372,088,891
50	1,125,899,906,842,679
55	36,028,797,018,963,971
60	1,152,921,504,606,847,009
65	36,893,488,147,419,103,363
70	1,180,591,620,717,411,303,449

The results obtained (in seconds) when the relevant residues modulo 30 were used to generate the divisors, once the first 4096 primes had been tried, are given below. Since the times obtained with these primes will be representative of the worst case for i-bit integers ($i = 35, 40, \cdots, 70$) rather than (i+1)-bit numbers, in the tables we consider the primes to be respectively $35, 40, \cdots, 70$ bits long, instead of $36, 41, \cdots, 71$.

	VAX 11/780		DAP	
length of N	$[N^{1/3}]$	time	blocks*	DAP-time
35	3251	0·4	0	0·019
40	10,322	1·0	0	0·020
45	32,769	2·9	0	0·020
50	104,032	76·6	5	0·056
55	330,281	311·1	19	0·159
60	1,048,577	1148·1	66	0·527
65	3,329,022	3855·0	215	2·354
70	10,568,984	13762·7	686	7·963

*This figure does not include the first matrix of primes.

If the ratio (DAP-time/$N^{1/3}$) were to be calculated, then one would see that it is not constant. There are two reasons for this. (1) The use of only primes, rather than 8/30 of all the integers, for the first block, has a distorting effect upon the results for the smaller test cases. (2) As mentioned earlier, the length of time an operation takes depends on the length

of the operands. Thus, the times increase, not only because more divisors have to be used, but because the dividend is longer than the previous case, and so all the earlier blocks will take proportionally longer also.

The analysis at the end of Chapter 3 indicated that the number of bit operations required for Step 1 was given by a function exponential in the length of the integer to be factored. This is borne out by the shape of the graphs when the above results were plotted (see Plate 1, at the end of the chapter). It is also worth pointing out that the graph of the DAP-time is a similar shape to that for the VAX except that, due to performing 4096 divisions in parallel on the former machine, the exponential growth is delayed somewhat.

When one calculates the ratio of the time the VAX required, compared with the DAP-time needed, the following values are obtained.

length of prime (bits)	VAX/DAP ratio
35	21
40	50
45	145
50	1368
55	1957
60	2178
65	1638
70	1728

There are two points worth noting from these figures. First, how the ratios are very small at the start, and second, how they increase to values much larger than might have been expected, in the light of the above discussion. It was mentioned in Chapter 4 how the DAP must use all of the first 4096 primes, whereas the VAX version stops whenever the necessary trial divisions have been completed, even if only a few of the primes are all that have been used. This was the case with the first three test cases, since they are all less than the cube of 38,879, the first element of the first block of calculated divisors. The ratios increased because, obviously, as the size of the test cases increased, a larger proportion of the primes were required for the trial division, and so the advantage of the VAX version decreased.

The very large ratios obtained for the other test cases stem from the different values of maxint for the two machines. Because $2^{31}-1$ is the largest value an integer variable can have on the VAX, 2^{15} was chosen as the base for the multiple-precision arithmetic rather than the value 2^{31} which could be used on the DAP. Thus, towards the end of the initial block of primes, the serial version has to stop using the single-digit-divide algorithm and call the more complicated long-division procedure, while on the DAP, the simpler version can be used all the time. As can be seen from the program listings, the long-division procedure is much longer, and requires considerably more work than the single-digit version, and so here we have an example of Point (1) above - the DAP does not need to do as much work as the VAX to accomplish the same task - and also an explanation for the high ratio values. The reason they fluctuate so much can be seen from the following table:

j	Cube of first element of jth block	Ratio
5	1,009,600,560,761,759	
Prime	1,125,899,906,842,679	1368:1
6	1,547,973,697,191,839	
19	31,362,862,163,813,279	
Prime	36,028,797,018,963,971	1957:1
20	36,172,409,463,084,959	
66	1,116,057,979,038,068,639	
Prime	1,152,921,504,606,847,009	2178:1
67	1,166,375,448,489,993,119	

The nearer the integer being divided is to the cube of the first element of the next block of divisors, then the fewer are the unnecessary divisions performed by the DAP, and hence a higher ratio when compared with the VAX which always does no more divisions than are necessary.

Of course, these impressive gains are, therefore, not due to the extra processors of the DAP, but would apply, in part, to a comparison of the VAX with any computer which could operate on 64-bit integers. Thus, let us consider again the case of the 46 bit test case. Now, $2^{15} = 32{,}768$ which is $< 32{,}771$, the $3{,}513^{th}$ prime. In other words, the VAX only used $\frac{439}{512}$ of the first 4096 primes. Scaling up the time taken accordingly, it is reasonable to suppose that the serial version would have taken about $3 \cdot 4$ seconds to perform 4096 single-

digit-divisions, giving a VAX/DAP ratio of around 170. In the above estimate we also include a proportion of the time required to find the cube root of the 46-bit number, but since the latter is so close to 2^{45}, it will have taken very little time for this value to be calculated. Since in general, an integer for which the trial-division would require 4096 primes, will not be so near a cube, more work could be involved in finding its cube root. Thus, it is probably the case that a ratio slightly in excess of 170 is to be expected.

However, it would not be wise to draw firm conclusions on the basis of such short programs, for several reasons.

(i) Any inaccuracy present in execution times will be magnified when such small numbers are involved.

(ii) The effect of the initialisation of the factor array on execution time (especially for the DAP), will be greater than if more divisors had been used.

(iii) The look-up table method used on the DAP will probably not be as quick as the Newton-Raphson algorithm, (since the former involves the manipulation of much larger integers than the latter, for a number with only 46 bits) and this difference could have a distorting effect with so comparatively few other instructions used.

Nevertheless, these results would suggest that the DAP performs well when used for trial division, comparing especially favourably with machines like the VAX, which cannot operate on 64-bit integers.

Effect of reducing the Array Size

In order to simulate the effect of having only 1024 processors in the array, the DAP version was run again, but with only one quarter of each matrix used. Since, after the division by all the primes, it is known that 2 is not a factor of the number under consideration, the last 3072 locations of each divisor matrix were initialised to 2 rather than use the previous division matrices and then continually mask out these elements. To avoid writing a new routine to find the number of smaller blocks required, four times the number of larger blocks needed was used instead. Admittedly, with the smaller spread of values in each matrix, this value could be up to three too large, but the effect of this will be negligible since hundreds of blocks have to be tried anyway. The results obtained are given below, again in seconds.

bits	Time for 4096 DAP	Time for 1024 DAP	(1024 DAP)/(4096 DAP) ratio
45	0·020	0·039	1·95
50	0·056	0·179	3·20
55	0·159	0·592	3·91
60	0·527	2·061	3·98
65	2·354	9·368	3·98
70	7·963	31·807	3·99

From the table it can be seen that, after the initial irregularity, the smaller DAP was taking approximately four times as long as the larger one, as we would have expected. The smaller ratios at the top of the table reflect the fact that arithmetic operations are a function of the precision being used, with the DAP only doing as much work as is necessary. Since a SIMD machine has to work at the speed of the slowest processor, when there is a wide range in the size of numbers being operated on, as in the first few blocks, processing them in smaller groups is less wasteful.

Using Residues (Mod 210)

In the previous chapter it was claimed theoretically that generating divisors from the relevant modulo 210 should be significantly faster than using residues modulo 30. Since in the former case, 873 PE's, out of 4296, are not useful, rather than 1323 with the former case, an improvement of just over 10% is to be expected. But, while such a gain was obtained in practice, as can be seen from the table below, it varies slightly according to what proportion of the last block of divisors is unnecessary in each case. Once again, the times given are in seconds.

bits in prime	Residues (mod 30)	Residues (mod 210)	(mod 210 time)/(mod 30 time)
45	0·020	0·020	-
50	0·056	0·052	0·93
55	0·159	0·148	0·93
60	0·527	0·461	0·87
65	2·354	2·032	0·86
70	7·963	6·855	0·86

What has not been investigated in practice is the effect that finding a factor, and hence having to redivide by a block, will have on the results. However it is clear that this will have a more adverse effect on the DAP execution times than on the VAX's, since a repeat division on the former is approximately equivalent to requiring to use an extra block. But since factors are comparatively rare (an integer, N, has on average only $\log\log N$ factors, and $\log\log 10^{70} \approx 5$), re-division will not be required very often, and when it is, the inefficiency of the DAP is one of the consequences of this type of architecture, and so has to be tolerated.

Step 2

The same test cases were used for Step 2, and the results are given below, again in seconds. (A similar comment on the ratio (DAP-time/$N^{1/3}$) to that made for Step 1 applies here.)

bits in N	$[N^{1/3}]$	VAX time	blocks	DAP-time	VAX/DAP ratio
35	3,251	56·4	1	0·649	87
40	10,322	166·3	3	1·509	110
45	32,769	1,107·2	8	3·924	282
50	104,032	4,505·7	26	10·264	439
55	330,281	13,911·3	81	29·660	469
60	1,048,577	44,917·5	256	92·818	484
65	3,329,022	127,704·7	813	312·849	408
70	10,568,984	> 100hrs	2581	1,074·965	

These results, for both machines, have been plotted on Plate 2. While, as was stated in Chapter 3, a bit operation analysis was not possible for Step 2, these graphs would indicate that a fast-growing function, exponential in the length of the number to be factored, would have been the result (as was obtained for Step 1).

These figures are much harder to interpret than the previous ones because there are so many factors involved.

The DAP does less work than the VAX for two reasons:

(i) As noted above, multiple-precision integers of the same size have more "digits" on the VAX than on the DAP, and so the former has to manipulate more array elements than

the latter to perform the same arithmetic operation.

(ii) The method for finding square roots used on the DAP is so much simpler than the Newton-Raphson algorithm.

On the other hand, the DAP has to do more work than the VAX for three reasons:

(a) Not every value of $([\sqrt{4kN}] + d)^2 - 4kN$ need be completely tested for being a perfect square on the VAX, since Fermat's method is used as a fast preliminary check.

(b) More k values will probably be processed than necessary on the DAP, since they have to be taken in blocks of 4096.

(c) As was discussed in the previous chapter, 4096 d loops are performed in parallel, with all of them running for the length of the longest in that block, and this will involve considerably more work than need be done on the VAX.

An indication of how much extra work has to be performed by the DAP can be obtained from Plate 3, where the number of d loop iterations performed by the VAX has been plotted against those of the DAP version for the 46-bit prime. If one were to count the number of d loop iterations performed on each machine (whether in parallel or serially) for the 51-bit prime, the following results would be obtained:

DAP	VAX	DAP/VAX ratio
438,272	114,613	3·82

Thus we have that, despite doing four times as many iterations, the DAP was still, at times, over 400 times quicker than the VAX. Both programs involved the performing of multiple-precision arithmetic, and while the former would still have an advantage over the VAX because of the different choice of base, one would not have expected as large an improvement as was obtained in connection with Step 1, when single-digit operations were being compared with multiple-precision work. Yet, if the DAP had only performed the same number of iterations as the VAX, the Speed-up factor would, again, have been in the region of 1,600 or more. This suggests that the use of the simpler square root routine on the DAP has contributed positively to its performance. But, in order to quantify the gain, one would need to move the serial implementation to a machine which could handle 64-bit integers.

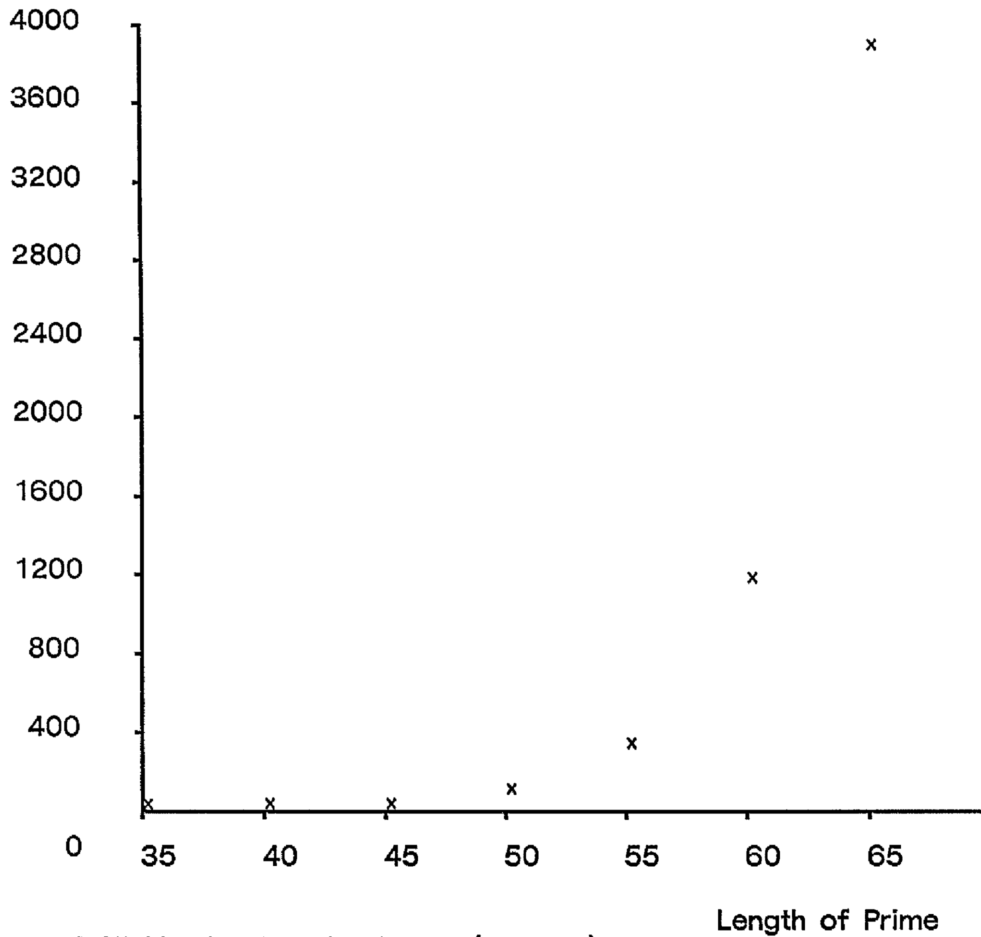
Comparison with only trial division

When it is noted how much longer Step 2 took, when compared with Step 1, for the above test cases, one might wonder if using the Lehman algorithm was a mistake, and that we would have been better to use only trial division up to $N^{1/2}$. However, if one takes as an approximate time for trial dividing a 40-bit integer up to its square root, the time that Step 1 took for the 60-bit prime (in fact, as already mentioned, the latter will take slightly longer), one finds that the use of only trial division would have taken the VAX over 1,000 seconds, rather than the $167 \cdot 3$ seconds required for both parts of the Lehman algorithm. The equivalent figures for the DAP are $0 \cdot 527$ and $1 \cdot 529$ seconds respectively, with the Lehman algorithm taking longer because, with such small integers, the effect of all the extra d loop iterations is magnified. To perform such a comparison on the DAP for much larger primes, to discover how large N needs to be before the maximum running time of the Lehman algorithm is less than that for only trial division, would require a multiple-precision long-division routine, and so has had to be omitted. But, as we will discuss later, in the next chapter, one can compromise and trial divide beyond $N^{1/3}$ in Step 1, thereby reducing the time needed for Step 2.

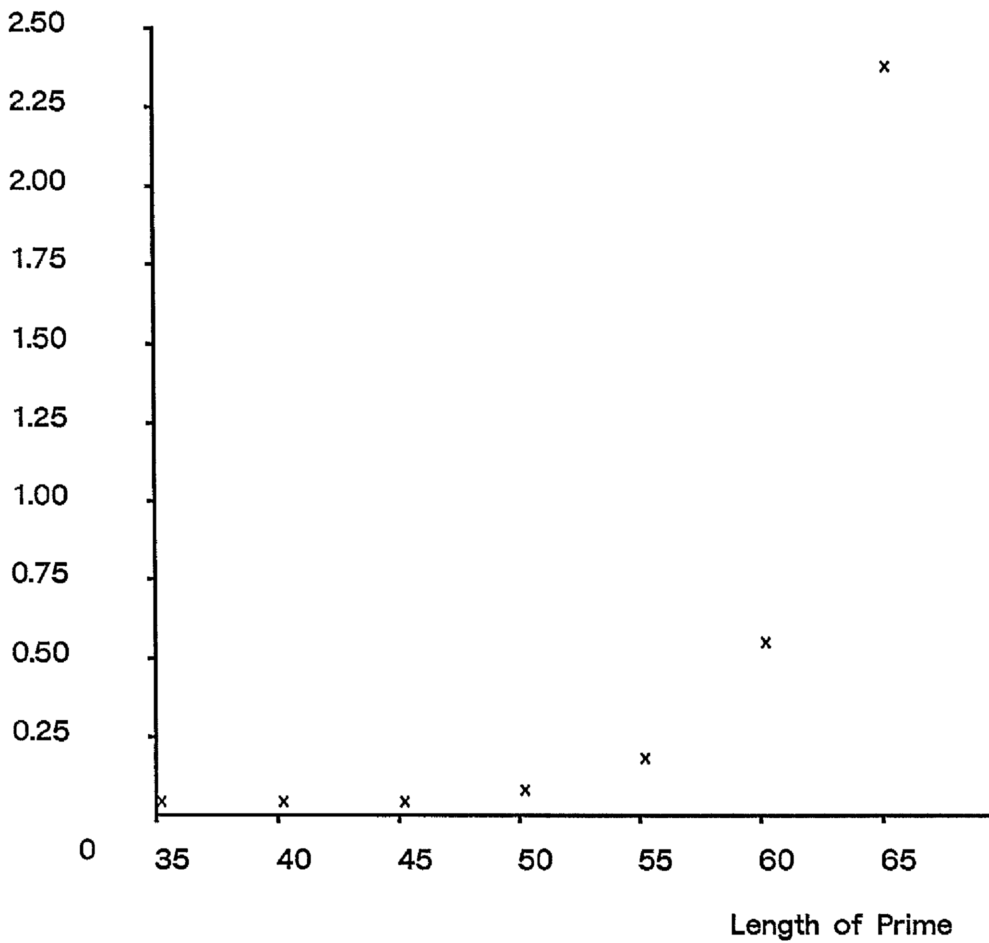
However, it must be remembered that the times given above were for prime test cases - numbers for which neither algorithm would normally be used (a primality test would be all that is required). It is certainly true that for some numbers, ordinary trial division would be the quicker method (e.g. those with only small prime factors (in which case Step 2 would not be required, and so both algorithms are equivalent), or those integers whose second largest prime factor is not much larger than its cube root), but it would be very surprising if one found, after comparing the times that both algorithms took to factor thousands of integers chosen at random, that in general, the $O(N^{1/2})$ algorithm was faster.

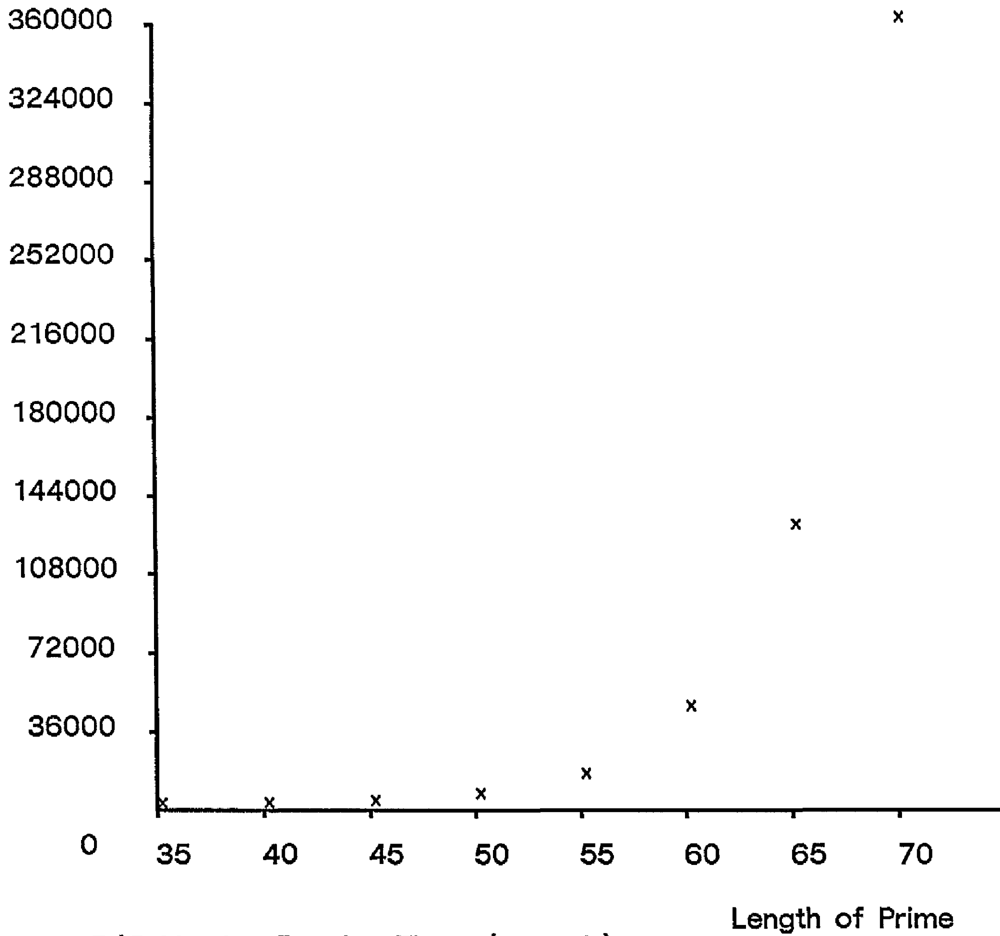
VAX Version Running Time (seconds)

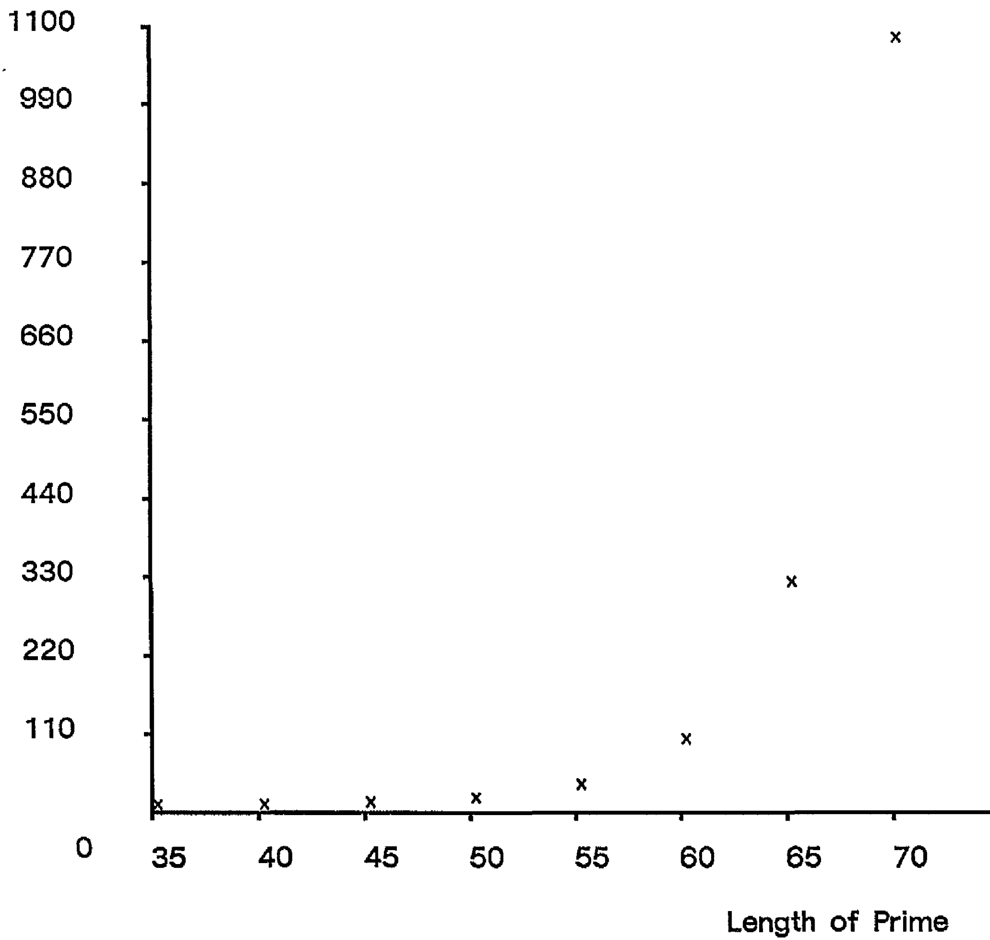
PLATE 1 : Step 1



DAP Version Running Time (seconds)




VAX Version Running Time (seconds)
360000
324000
288000
252000
216000
180000
144000
108000
72000
36000
0
35
40
45
50
55
60
65
70
Length of Prime


DAP Version Running Time (seconds)
1100
990
880
770
660
550
440
330
220
110
0
35
40
45
50
55
60
65
70
Length of Prime

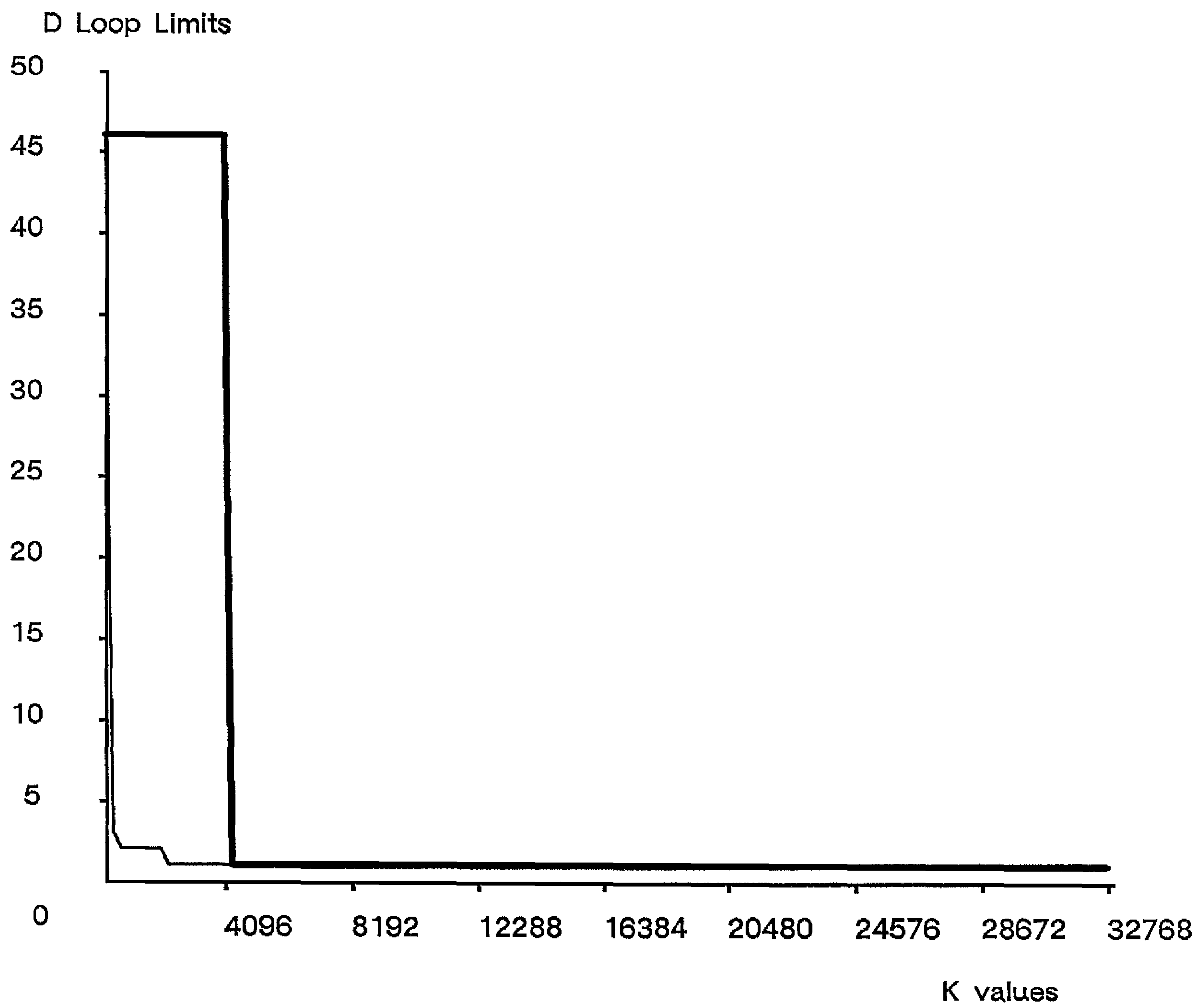


PLATE 3 : VAX – DAP comparison for 46-bit prime

Chapter 7 The Generalised Lehman Algorithm

From the above discussions, it should be clear that, while using an array of 4096 bit processors has many points in its favour, there are disadvantages associated with such an architecture. In Chapter 5, when describing the parallel implementation of the Lehman algorithm, we noted how, for each block of k values, either all the inner d loops would have to run for the length of the longest, or some (and eventually many) processors be masked out, once "their" d loop, so to speak, had completed the required number of iterations. Because using masked assignments does not actually save any time, since all the processors still have to obey the instructions concerned, it was decided to adopt the former scheme. It was also claimed that this apparent inefficiency could cause remarkable savings in time. In this chapter, we justify that claim, giving an example of a factorisation which took less than one tenth of the time it would normally have required.

The number in question is 11,111,111,111,111,111 which is the product of the primes 2,071,723 and 5,363,222,357, and since it has no small factors, Step 1 will not contribute to the factorisation process (except to indicate that N, say, is of the required form for Step 2). Under normal circumstances, a pair of squares is not found until k takes the value 41,420, for then, with d equal to 1, we have that

$$([\sqrt{4kN}] + 1)^2 - 4kN = 197{,}763^2 = B^2, \text{say},$$

where

$$([\sqrt{4kN}] + 1) = 42{,}905{,}581{,}093 = A, \text{say}.$$

Then,

$$A - B = 42{,}905{,}383{,}330 = 20{,}710 \times 2{,}071{,}723$$

$$A + B = 42{,}905{,}778{,}856 = 8 \times 5{,}363{,}222{,}357$$

and hence we can obtain the prime factors we seek, since

$$11{,}111{,}111{,}111{,}111{,}111 = 2{,}071{,}723 \times 5{,}363{,}222{,}357.$$

It took just under 38 minutes (2276 seconds to be precise) for the serial version of Step 2 on the VAX to reach the required (k, d) pair. For the parallel version on the DAP to find this same pair of squares, it would require just over 12 seconds (12,214 milliseconds). To ensure that this was the pair of squares found it was necessary to refrain from checking if any squares had been found until the eleventh block of k values was processed (the block in which 41,420 lies), since otherwise the program would stop after only 1043 millisecs, having discovered that

$$(10{,}726{,}913{,}204)^2 - 4kN = 468{,}490^2 .$$

Now,

$$10{,}726{,}913{,}204 - 468{,}490 = 2 \times 5{,}363{,}222{,}357$$

and

$$10{,}726{,}913{,}204 + 468{,}490 = 2 \times 2589 \times 2{,}071{,}723$$

and accordingly, these squares were found when k had the value 2589 $(A^2 - B^2 = 4kN)$, with a corresponding d value of 11. Using the serial version of the algorithm, this inner loop would only have had 3 iterations, but because the inner loop for a k value of 1 (the first value in the block) requires 119 iterations, this value of d was reached on the DAP, and the squares found.

It is worth asking the question: "Did this happen by chance, or is there some mathematical reason for it?" We will now show how this result can be explained, thus giving rise to a generalised version of the Lehman algorithm, which will be described towards the end of the chapter.

The proof of the algorithm depended on the existence of a pair of natural numbers, r and s, such that

$$rs < N^{1/3} \quad \text{and} \quad | pr - qs | < N^{1/3}$$

where $pq = N$. This is proved in Appendix A where, by considering the continued fraction expansion of $\frac{q}{p}$, we show that, if $\frac{r}{s}$ is the last convergent of $\frac{q}{p}$ such that $rs < N^{1/3}$, then the second inequality above holds. Since we put $k = rs$, it is clear where the upper bound for the outer loop in the algorithm comes from. However, it is worth asking why $N^{1/3}$ had to be the value which was used in the proof.

We require an upper bound for the product rs, where $\frac{r}{s}$ is one of the convergents of the continued fraction expansion of $\frac{q}{p}$. Since $\frac{q}{p}$ is rational, the last convergent will be $\frac{q}{p}$

itself, thus giving the trivial bound

$$rs \leqslant N$$

for all the convergents $\frac{r}{s}$. But, using this value would produce an algorithm with a maximum running time of the order $O(N)$, which would not be very useful ! However, if we denote the nth convergent by $\frac{r_n}{s_n}$, then we have that $r_1 = \left[\frac{q}{p}\right] < \frac{N^{2/3}}{N^{1/3}} = N^{1/3}$, $s_1 = 1$ and hence $r_1 s_1 < N^{1/3}$. We also know from the theory of continued fractions, that $r_n s_n$ is an increasing sequence, and have shown that the product of the numerator and denominator of the last convergent is N. Thus, it must be possible to find an m such that

(1) $r_m s_m < N^{1/3}$

(2) $r_{m+1} s_{m+1} > N^{1/3}$

(since N has at most two prime factors, $N^{1/3}$ is not an integer, and so equality cannot hold in either of the above). Since $N^{1/3}$ is the best such bound known at this stage (the other bound being N), it is adopted for the product rs, and hence for k.

However, since $N^{1/3} < p \leqslant q < N^{2/3}$, taking $N^{1/3}$ as an upper bound for $\frac{q}{p}$ is very much a "worst case choice". If we had known beforehand that, say, p was nearer $N^{1/2}$ than $N^{1/3}$, then the value of $\frac{q}{p}$ would be much less than $N^{1/3}$, and so we could lower the bound on the outer loop. What effect this would have can be seen from the following.

Suppose that $\frac{q}{p} \leqslant T < N^{1/3}$, and let $\frac{r_m}{s_m}$ be the last convergent of $\frac{q}{p}$ such that

(1) $r_m s_m \leqslant T$

(2) $r_{m+1} s_{m+1} > T$.

Then, as shown in Appendix A, putting $r_m = r$ and $s_m = s$, we have that

$$\left| \frac{r}{s} - \frac{q}{p} \right| \leqslant \left| \frac{r}{s} - \frac{r_{m+1}}{s_{m+1}} \right| = \frac{1}{s\, s_{m+1}}$$

which implies that

$$| pr - qs | \leqslant \frac{p}{s_{m+1}}.$$

As before, by considering the convergents to $\frac{p}{q}$ we have that

$$| pr - qs | \leqslant \frac{q}{r_{m+1}} .$$

Hence,

$$| pr - qs |^2 \leqslant \frac{pq}{r_{m+1}s_{m+1}} .$$

In other words,

$$| pr - qs | \leqslant \left(\frac{pq}{r_{m+1}s_{m+1}} \right)^{1/2} < \left(\frac{N}{T} \right)^{1/2} , \tag{7.1}$$

and since $T < N^{1/3}$,

$$\left(\frac{N}{T} \right)^{1/2} > \sqrt{N^{2/3}} = N^{1/3}.$$

Hence we have that $rs < T$, but that

$$| pr - qs | \quad \text{could be} > N^{1/3} .$$

Put $k = rs$, and consider

$$(pr + qs)^2 - 4kN = (pr - qs)^2 < \frac{N}{T} .$$

Then, if d is defined by $pr + qs = [\sqrt{4kN}] + d$, where $d \geqslant 1$, we have

$$\frac{N}{T} > (pr - qs)^2 = (pr + qs - \sqrt{4kN})(pr + qs + \sqrt{4kN})$$

$$> (d - 1)(2\sqrt{4kN}) .$$

Hence ,

$$d < \frac{N}{4T\sqrt{kN}} + 1 \tag{7.2}$$

$$= \frac{N^{1/2}}{4T\sqrt{k}} + 1$$

and since $T < N^{1/3}$, this last bound is strictly greater than

$$\frac{N^{1/6}}{4\sqrt{k}} + 1 .$$

Now, if q and p denote the prime factors of 11,111,111,111,111,111 , where $q > p$, then the continued fraction expansion for $\frac{q}{p}$ is

$$[2588;1,3,2,2,1,2,2,4,1,5,1,1,1,2,5,1,5].$$

Therefore, the first few convergents are

$$\frac{r_1}{s_1} = \frac{2588}{1}, \quad \frac{r_2}{s_2} = \frac{2589}{1}, \quad \frac{r_3}{s_3} = \frac{10355}{4},$$

and so it can be seen that, since

$$r_2 s_2 \leqslant 2589 \quad \text{and} \quad r_3 s_3 > 2589,$$

the number 2589 could be used as an upper limit for the k loops. The counter for the corresponding inner loops would then need to take values in the range

$$1 \leqslant d < \frac{N^{1/2}}{2589.4.\sqrt{k}} + 1.$$

For a k value of 2589, the limit on d would be 201, and thus the (k, d) pair found was well within the permitted range. This bound of 201 is very large when compared with the value of 3 required by the normal form of the algorithm. The reason for this is that the use of $\left(\frac{N}{T}\right)^{1/2}$ as a bound for $\frac{pq}{s_{m+1}r_{m+1}}$ was again a "worst case choice", but one has to be made, since we have no knowledge about the size of r_{m+1} or s_{m+1} except their product is $>T$. In this case,

$$r_{m+1}s_{m+1} = r_3 s_3 = 10{,}355 \times 4 = 41{,}420$$

(the reason why this number is familiar will be discussed later) which is much greater than $T = 2589$, with the result that more iterations are performed in each d loop, than are really necessary. For example, the value of $\frac{p^2}{4s^2_{m+1}\sqrt{kN}}$ for N, p, s_{m+1} as above and $k = 2589$ is 12. This is greater than the bound for the "normal" algorithm since, in the latter, a different s_{m+1} is used (see later).

If one were to plot the function $\frac{N^{1/2}}{2589.4.\sqrt{k}} + 1$ for k between 1 and 2589, then the graph of Plate 4 (see end of chapter) would be obtained. Now, the maximum number of d loop iterations for the Lehman algorithm with $T = 2589$ equals

$$\sum_{k=1}^{2589}\left(\frac{N^{1/2}}{2589.4.\sqrt{k}} + 1\right) \leqslant \frac{N^{1/2}}{2589.4}\int_0^{2589}\frac{dk}{\sqrt{k}} + 2589$$

$$= \frac{105{,}409{,}255}{10{,}356}\left[2\sqrt{k}\right]_0^{2589} + 2589$$

$$= \frac{105{,}409{,}255}{10{,}356}.2(\sqrt{2589}) + 2589$$

$$\approx 1{,}038{,}400\,.$$

But, the corresponding bound for the $[N^{1/3}]$ version has already been shown to be $\frac{3}{2}N^{1/3} \approx 334{,}716$. (In the present analysis, we have chosen to use the first method of obtaining a bound on the number of $(k\,,d\,)$ pairs given in Chapter 3, rather than the second, more complicated, one. However, as we use the same estimate in all the following comparisons, this choice will not invalidate the conclusions drawn.) Thus, reducing the limit on the outer loop could result in more work being required - a fact that can also be seen from the comparison of the graphs of the d loop bounds with $T = 2589$ and $T = [N^{1/3}]$ in Plate 5, where the long "tail" of the $T = [N^{1/3}]$ graph has been omitted in the interests of clarity.

In fact, for this particular number, 11,111,111,111,111,111, to have performed the modified algorithm with $T = 2589$ on a serial machine would have resulted in a longer execution time than if the normal algorithm had been used, since, for this value of T, the total number of d loop iterations required is:

$$\sum_{k=1}^{2588}\left(\frac{N^{1/2}}{2589\,.\,4\sqrt{k}} + 1\right) + 11 \leqslant \frac{105{,}409{,}255}{10{,}356}\int_0^{2588}\frac{dk}{\sqrt{k}} + 2588 + 11$$

$$\approx 1{,}038{,}207$$

(where the sum on the left-hand side only goes up to 2588 since only 11 of the d values for $k = 2589$ need be used), whereas for $T = [N^{1/3}]$ this value equals

$$\sum_{k=1}^{41420}\left(\frac{N^{1/6}}{4\sqrt{k}} + 1\right) \leqslant \frac{N^{1/6}}{4}.2(\sqrt{41420}) + 41420 \approx 89{,}489\,.$$

(Actually, the former bound should have been

$$\sum_{k=1}^{2587}\left[\frac{N^{1/2}}{2589\,.\,4\sqrt{k}} + 1\right] + 120 \leqslant 1{,}038{,}111$$

since, as will be described below, the pair (2588,120) also produces squares congruent modulo N, and the d value of 120 is within the range corresponding to $k = 2588$ and $T = 2589$.)

Thus, the drawback with reducing the number of iterations in the k loop is that the inner loop bound has to be increased. In addition to making the maximum running time larger than the $T = [N^{1/3}]$ case, this modification could result in the actual searching time being more than before, since one has no knowledge about how soon a $(k\,,d\,)$ pair will be found. Indeed, there might not be a $(k\,,d\,)$ pair to find! From the theory above it can be

seen that reducing the k loop bound to T, say, is only possible if there exists a convergent $\frac{r}{s}$ to $\frac{q}{p}$ with $rs < T$. This is implied by $\frac{q}{p} < T$ But, if one "took a chance" so to speak, and chose T to be some convenient value (e.g. 4096 or 8192), without having known that it was a valid reduction, then there could be no (k, d) in the range being searched. Certainly, for any k value there will always be a d value such that

$$([\sqrt{4kN}] + d)^2 - 4kN$$

is a perfect square. For,

$$4kN = (k + N)^2 - (k - N)^2$$

and since,

$$(k + N)^2 \geqslant 4kN ,$$

we must have that

$$k + N = [\sqrt{4kN}] + d$$

for some $d \geqslant 0$. However, for a given k and N, d could be huge. What the Lehman algorithm guarantees is that if N is the product of two primes, both greater than $[N^{1/3}]$ then, for at least one k value ($\leqslant [N^{1/3}]$), a suitable d value will lie within a certain range.

It has already been shown that, in addition to $T = [N^{1/3}]$ for 11,111,111,111,111,111, $T = 2589$ is also a valid choice for the other loop limit. If, on the same diagram as the graphs of the d loop limits corresponding to these choices of T, there was to be plotted the graph representing the work performed by the DAP version for the former value of T, then Plate 6 would be obtained (where the long "tails" have again been omitted). The position of the pair (2589, 11) has also been indicated, and from this it can be seen that the graph for the parallel program includes enough of the area under the $T = 2589$ graph, for this pair to be "within range". Since, in this implementation, the k values are processed in consecutive blocks of 4096 values, starting at $k = 1$, the (k, d) pair with $k = 2589$ is found very quickly.

Such an occurrence cannot be guaranteed in general, since there might not be a (k, d) pair lying above the $[N^{1/3}]$ version graph, but within the graph of the work done by the DAP. Nevertheless, the way this algorithm has been implemented on the DAP allows us to take advantage of this happening (without, obviously, knowing beforehand), and so gain an additional speed-up of possibly, as in this example, more than 10-fold.

Possible values for k

From the proof of the normal algorithm, it can be seen that the k value (corresponding to a d value within a certain range) whose existence is proved is, in fact, equal to the product of the numerator and denominator of a convergent to $\frac{q}{p}$. Thus, if there exists a convergent, $\frac{r_n}{s_n}$ say, with $r_n s_n$ much less than $[N^{1/3}]$, then $\frac{r_{n+1}}{s_{n+1}}$ might also be such that $r_{n+1}s_{n+1} \leqslant [N^{1/3}]$. This was the case here, since

$$r_2 s_2 = 2{,}589 \times 1 = 2{,}589$$

and

$$r_3 s_3 = 10{,}3544 \times 4 = 41{,}420\,.$$

But, we also have that

$$r_1 s_1 = 2{,}588 \times 1 = 2{,}588 < [N^{1/3}],$$

and so there should be a d value corresponding to $k = 2588$. This is indeed the case, namely $d = 120$. However, this value is well beyond the bound used in the serial version (= 3, for $k = 2588$), and is also outside the area searched by the DAP in this case (which is $1 \leqslant d \leqslant 119$). It is only when k equals the last product (of the numerator and denominator of a convergent to $\frac{q}{p}$) which is $< N^{1/3}$, that the proof guaranteed that the corresponding d value is within the range specified.

Another observation that can be made is that if $\frac{r}{s}$ is a convergent to $\frac{q}{p}$ such that rs is considerably smaller than $[N^{1/3}]$, then if there exists an integer t, say, such that $rst^2 \leqslant [N^{1/3}]$, then this value for k will also correspond to a relatively small d value for the following reason. Suppose that

$$4t^2 rsN = (prt + qst)^2 - (prt - qst)^2\,.$$

Then, if

$$(pr + qs) = [\sqrt{4kN}] + d$$

it follows that

$$\begin{aligned}(prt + qst) &= t[\sqrt{4kN}] + td \\ &\leqslant [t\sqrt{4kN}] + td \\ &= [\sqrt{4t^2kN}] + td\,,\end{aligned}$$

and so the d value for tk will be less than or equal to t times the d value that

corresponded to k. But, again, there is no guarantee that this value of d will be inside either the limit used for the serial version, or within that on the DAP. An example of this can be seen with 11,111,111,111,111,111, where the pair $k = 10{,}356 = 4 \times 2589$ and $d = 21$ will result in squares congruent modulo N being found, though in the performing of the algorithm, these values will not be tested, since the d limit in both cases, for $k = 10{,}356$ is only 2. How the existence of these pairs is known will be explained later in the chapter.

However, while points like the above are interesting and worth mentioning, they are only of limited value unless they can be used to reduce the time required to factor a given integer. We now state some facts which, if a certain amount of extra knowledge were known, could significantly reduce the running time of the algorithm.

The Lower Bound for the Outer Loop

It has already been pointed out how the k value whose existence is proved in Chapter 3, is the product of the numerator and denominator of one of the convergents to $\frac{q}{p}$. As noted at the start of this chapter, it is this fact that gives rise to the upper bound on the k loop. Now the lower bound for this loop was taken to be 1 since the first convergent $\frac{r_1}{s_1}$ say, has $s_1 = 1$ and $r_1 = [q/p] \geqslant 1$, since $q \geqslant p$. However, if it was known that $\frac{q}{p} > S$, where S is an integer, then the loop iterations for k in the range $1 \rightarrow (S-1)$ would be unnecessary and so could be omitted. Even if S were not much larger than 1, this could result in a considerable reduction in running time, since it is the inner loops corresponding to these initial k values which involve the most iterations. It would especially benefit the DAP version since it is for the low values of k that the inefficiency from performing more d loop iterations than necessary, is greatest. Of course, we have just shown above how this extra work can sometimes bring rapid results. But, if one knew that $\frac{q}{p} > S$, then it would probably be better to make use of this fact, rather than ignore it and "take a chance" on a (k, d) pair being found quickly. A possible compromise would be to take as the upper bound for the d loops corresponding to the k values $S \leqslant k \leqslant S + 4095$, the limit on the d loop for a k value of $[S / 4096] \times 4096$.

In general, if presented with an integer N to factor, one would have no way of knowing of a bound for $\frac{q}{p}$. But, if it was known that N had been formed for use in a public key encryption system where it had been specified that one prime be chosen very close to 10^{60} and the other very near 10^{80}, then a lower bound for $\frac{q}{p}$ could, almost

certainly, be taken to be 10^{19}, thus saving a considerable amount of work.

When $d = 1$ will suffice

Suppose that there exist natural numbers a, b with

$$|qa - bp| < (2(4abN)^{1/2} + 1)^{1/2} . \tag{7.3}$$

Then, if ab were known, we could proceed as follows:

$$(qa + bp)^2 = 4abN + (qa - bp)^2$$

and, by (7.3), this implies that

$$4abN \leqslant (qa + bp)^2 < 4abN + 2(4abN)^{1/2} + 1$$

i.e.

$$4abN \leqslant (qa + bp)^2 < ((4abN)^{1/2} + 1)^2$$

and so

$$(4abN)^{1/2} \leqslant qa + bp < (4abN)^{1/2} + 1 ,$$

which implies that

$$qa + bp = [(4abN)^{1/2}] + 1$$

if $p \neq q$ and $ab < pq = N$. In other words, the pair $k = ab$, $d = 1$ will produce squares, as required.

Thus, if we knew a pair a, b such that (7.3) held then starting the k loop at $S = ab$ would result in squares being found immediately. This compares very favourably with the number of d loop iterations that the Voorhoeve version of the algorithm would require which equals*

$$\sum_{k=1}^{ab} \left(\left[\frac{N^{1/6}}{4\sqrt{k}} \right] + 1 \right) \leqslant \frac{N^{1/6}}{4} \left(\sum_{k=1}^{ab} \frac{1}{\sqrt{k}} \right) + ab$$

$$\leqslant \frac{N^{1/6}}{4} \int_0^{ab} \frac{dk}{\sqrt{k}} + ab$$

$$= \frac{N^{1/6}}{4} (2(ab)^{1/2}) + ab$$

$$= \tfrac{1}{2} N^{1/6} (ab)^{1/2} + ab . \tag{7.4}$$

*[Note that here we have assumed that $ab > \frac{N^{1/3}}{16}$, so that $\left(\left[\frac{N^{1/6}}{4\sqrt{k}}\right] + 1\right)$ would be 1 when $k = ab$, otherwise the number of iterations needed to reach the (k, d) pair $(ab, 1)$ will be slightly (i.e. $[N^{1/6}/(4\sqrt{ab})]$) less than the sum on the left hand side.]

Even if we only knew that (7.3) was true for some a, b with $ab < L$, (but not the actual values of a and b), then this bound would still apply, and while its exact value would not be known, we would still have that it was

$$\leqslant \frac{N^{1/6}}{2} L^{1/2} + L .$$

However, we have shown above that the d value corresponding to $k = ab$ will equal 1. Thus, there is no need to perform any more than just the first iteration of each inner loop. If this alteration is made then, the maximum number of (k, d) pairs that need to be considered is $\leqslant L$, rather than the much larger bound above.

The above has important consequences for public key encryption systems, and explains the following comment made by Knuth [1981, p.388]

> "in fact, we don't want the ratio $\frac{p}{q}$ to be near a simple fraction, otherwise Lehman's generalisation of Algorithm C would find them"

as will now be shown. Suppose that two primes p, q have been chosen with $N = pq$, and that there exist natural numbers a, b such that (7.3) holds. Then

$$| qa - bp | < (2(4abN)^{1/2} + 1)^{1/2}$$

if and only if

$$pa \left| \frac{q}{p} - \frac{b}{a} \right| < (2(4abN)^{1/2} + 1)^{1/2}$$

if and only if

$$\left| \frac{q}{p} - \frac{b}{a} \right| < \frac{1}{pa} (2(4abN)^{1/2} + 1)^{1/2} .$$

Therefore, if

$$\left| \frac{q}{p} - \frac{b}{a} \right| < \frac{1}{[N^{1/3}]a} (2(4abN)^{1/2} + 1)^{1/2} \tag{7.5}$$

then, since $p > [N^{1/3}]$, we would have that (7.3) holds, and so the Lehman algorithm would, indeed, find a pair of squares, and hence the factors of N, with the number of iterations required bounded above by (7.4). If $\frac{b}{a}$ is a simple fraction, then ab (i.e. L) is

small, and so Step 2 would then be an $O(N^{1/6})$ algorithm. Of course, if one knew to expect that (7.5) would hold for some a , b , then N could be factored quickly by only performing the first iteration of each d loop, (with the running time being especially small if $\frac{b}{a}$ is a simple fraction).

A special case of the above is when $\frac{q}{p}$ is close to 1, for then,

$$|q - p| < (4N^{1/2} + 1)^{1/2}$$

and so, by the above, the (k, d) pair $(1,1)$ will give rise to a pair of squares congruent modulo N, and hence to the required factors. For example, the integer 15,241,578,503,276,943 can be shown to be the product of 123,456,789 and 123,456,787 in only 0.1 seconds of cpu time on the VAX (compared with nearly 40 minutes which 11,111,111,111,111,111 required). This example also illustrates another interesting point : p and q do not have to be prime (e.g. 3 is a factor of 123,456,789), for Step 2 of the algorithm to find them.

In the proof of the theorem, the primality of p and q was never used. All that was required was that $N^{1/3} < p \leqslant q < N^{2/3}$. The Voorhoeve version of the algorithm guaranteed this by removing any small factors of N so that what was left was either prime or the product of two primes in the required range. However, if N had two factors (prime or composite) in the range above, then the trial division of Step 1 would be unnecessary. Instead only Step 2 (preceded by checking if N were a perfect square) would be needed to produce one (and hence both) of them (which would then require factoring themselves, if they were not prime). Of course in general, a composite N need not have two such factors. For example, the integer 95,704,051,146,663,247 which equals the product of 112,103 and 853,715,343,449 will be claimed to be prime by Step 2, if Step 1 is omitted!

The Generalised Lehman Algorithm

The points discussed above motivate a generalisation of the algorithm. Let N be a natural number. Then, for any real numbers S, T with $1 \leqslant S \leqslant T \leqslant N^{1/2}$, the (S, T) Generalised Lehman Algorithm may be described as follows.

For each integer k, $S \leqslant k < T$ and each integer d with $1 \leqslant d \leqslant \left[\frac{N^{1/2}}{4T\sqrt{k}}\right] + 1$, form

$$u = ([\sqrt{4kN}] + d)^2 - 4kN.$$

If u is a perfect square then (no more values of u need be considered, and) write $B = u^{1/2}$

and $A = [(4kN)^{1/2}] + d$. Then $A > B$ and

$$A \leqslant (4kN)^{1/2} + \frac{N^{1/2}}{4k^{1/2}T} + 1 < (4N^{3/2})^{1/2} + \frac{N^{1/2}}{4} + 1$$

$$= 2N^{3/4} + \frac{N^{1/2}}{4} + 1$$

$$< \frac{N}{2}$$

if $N \geqslant N_0$ (a small positive integer - see below).

Thus we have that

$$4kN = A^2 - B^2 = (A + B)(A - B)$$

with

$$A + B < 2A < N$$

and hence

$$A - B < N$$

for $N > N_0$. Therefore, $gcd(A + B, N)$ will be a non-trivial divisor of N, as will $gcd(A - B, N)$ (by a similar method to that used in Chapter 3 one can show that $A - B > 1$).

By a straightforward extension to the working already given above, and in Chapter 3, it can be proved that this algorithm will find a factor, p , of a composite N which lies in the range

$$\left(\frac{N}{T}\right)^{1/2} < p \leqslant \left(\frac{N}{S}\right)^{1/2}$$

(where the above inequality is implied by $S \leqslant \frac{q}{p} < T$). That $N_0 = 1{,}024$ will suffice in the above can also be shown, by noting that $2N^{3/4} + N^{1/2}/4 + 1 - \frac{N}{2}$ is a function of N which is negative and decreasing for $N \geqslant 1{,}024$.

The algorithm which Voorhoeve gives, and we implemented, is the $(1, N^{1/3})$ version, which will find a factor p with

$$N^{1/3} < p \leqslant N^{1/2}.$$

As has already been pointed out, the trial division of Step 1 is required to ensure that such a factor exists (unless what is left of N is prime). It is clear from the above that the amount of trial division required will depend on the choice of T. Putting $T = [N^{1/2}]$ will imply the need for trial division of N up to $[N^{1/4}]$. However, T could, in fact, take values up to N

itself. While one would not normally choose such a value, since it would result in an algorithm with a maximum running time of $O(N)$, it is interesting to note that, by putting $T = N$, we obtain the algorithm that Fermat himself actually used to factor integers.

It is also interesting to note that one can, in addition, generalise (7.3) to obtain a pair of inequalities which show how $|pr - qs|$ depends on the values of k, d, and N.

Let $N = pq$ (where we have already checked that $p \neq q$), and let (rs, d) be a (k, d) pair giving rise to a pair of squares congruent modulo N. Then

$$4rsN = (qs + pr)^2 - (qs - pr)^2$$

and

$$(qs + pr)^2 = ([\sqrt{4rsN}] + d)^2.$$

Therefore

$$\begin{aligned} (qs - pr)^2 &= ([\sqrt{4rsN}] + d)^2 - 4rsN \\ &\leqslant (\sqrt{4rsN} + d)^2 - 4rsN \\ &= 2d\sqrt{4rsN} + d^2. \end{aligned}$$

But, we also have that

$$\begin{aligned} (qs - pr)^2 &= ([\sqrt{4rsN}] + d)^2 - 4rsN \\ &> (\sqrt{4rsN} - 1 + d)^2 - 4rsN \\ &= 2(d-1)\sqrt{4rsN} + (d-1)^2. \end{aligned}$$

Hence, if the pair (rs, d) give rise to squares congruent modulo N, then

$$2(d-1)(4rsN)^{1/2} + (d-1)^2 < (qs - pr)^2 \leqslant 2d(4rsN)^{1/2} + d^2.$$

It is easy to see that when the value $d = 1$ is put into the right hand side of the above, the inequality (7.3) is obtained. (The left-hand inequality was not included in (7.3) since we were allowing for the possibility that p and q were equal.)

Searching for k's in general

We have shown above how, in particular situations, or if certain extra knowledge is available, the running time of the second step in the Lehman algorithm can be very small indeed. But, suppose such extra information is not known, or that all we can tell is that $\frac{q}{p}$

(and hence k) lies between some numbers S and T. Is our only option then to search the values consecutively, or are some possibilities for k more likely than others?

The extra (k,d) pairs that were mentioned in connection with N = 11,111,111,111,111,111 were found when a search was made of all the (k,d) pairs in the range $1 \leqslant k \leqslant 61{,}440$ and $1 \leqslant d \leqslant 120$, to discover which gave rise to squares congruent modulo N. The results are shown in Plate 7 where the range that the DAP would consider has also been plotted. As can be seen, there are very few "successful" pairs within easy reach. What would be instructive to know, but certainly non-trivial to discover, is whether it is possible to predict beforehand where suitable (k,d) pairs are likely to be. Certainly, this algorithm guarantees that at least one such pair will lie in a specified range, and we have shown, above, how this range can be reduced if certain conditions are met. However, it would be interesting to know, in general, the probability distribution of those (k,d) pairs which give rise to squares congruent modulo N, as this could suggest a more effective searching strategy.

For example, given that k lies between S and T, is it more likely to be nearer one end of the range than the other? Indeed, is our best plan just to consider each possible value consecutively, starting with $k = S$? Lehman [1974] makes an interesting point on this subject:

"In going through the integers k from 1 to r [his notation for the upper limit], there is an advantage in going through them in a prescribed order. Let $d(k)$ be the number of positive divisors of k. If a/b is closest to the ratio of the divisors of n [the number to be factored], which $k = ab$ should we try first?

He gives, as an example, $k = 23{,}220 = 2^2.3^3.5.43$ and continues:

"Thus, there are $d(k) = 3.4.2.2 = 48$ different representations a/b that we look at simultaneously. Clearly, it is better to first choose k with $d(k)$ large. For that reason, we chose to look at multiples of

$$30 = 2.3.5,\quad 24 = 2^3.3,\quad 12 = 2^2.3,\quad 18 = 2.3^2,\quad 6 = 2.3,\quad 2,\quad 15,\quad 43.$$

The program is designed to go through these sequences."

While Lehman gives examples of rapid factorisations through using this "trick", it must be noted that, at times, adopting this searching strategy would have a detrimental effect on the running time. For example, if p and q were close to each other, then, as described above, a k value equal to 1 would produce the required squares, but under Lehman's scheme, this would not be the first k value to be considered (indeed, far from it!). Also, considering say $k = t^2rs$, where rs is the k value whose d value will be inside the limit, could be a waste of time since, as has been shown above, the d corresponding to this

larger value of k need not lie within the prescribed range. Thus, while particular factorisations do benefit from this approach, it is not certain that this method will be quicker in general (i.e. for more than 50% of all the numbers of the form suitable for Step 2).

However, even if Lehman's method could be proved to be better (for a serial implementation) than taking the k's consecutively, it would not necessarily suit a parallel approach. As has already been commented on, the d loops for a given block of k values must all run for the length of that inner loop corresponding to the smallest k in the block. Hence, there is an obvious advantage in considering all the very small k values first (i.e. performing, say, the first 4096 inner loops in parallel), so that the performing of many extra d loop iterations is kept to a minimum. Although thereafter, it would be quite possible to consider the remaining k values in the order Lehman suggests (where each sequence would be taken 4096 members at a time).

Deciding on an optimal strategy

It has been shown above how the time required for Step 2 can be reduced by lowering the bound on the outer loop. However, unless one knew in advance some extra information about the factors of N, one would need to perform more trial divisions than in the Voorhoeve version to ensure that N had a prime divisor in the range being searched in the second step. From the results in Chapter 6 it can be seen that, for our test cases, Step 1 took only a fraction of the time that Step 2 required. What would be interesting to know is if this was the optimal strategy, or if further trial division would have resulted in a smaller execution time for Step 2, and a quicker running time for the whole algorithm.

So, suppose that we trial divide up to R, where $N^{1/3} \leqslant R \leqslant N^{1/2}$. Then, what is left of N (which we shall still denote by N) is either prime or the product of two primes p, q with $R < p \leqslant q < \frac{N}{R}$ (or $N = 1$, in which case we need not continue). Suppose also that we have checked that N is not a perfect square. Then, if N is composite, we have that

$$1 < \frac{q}{p} < \frac{N}{R^2}$$

and so we need only consider k values in the range $1 \leqslant k < \frac{N}{R^2}$ (i.e. put $S = 1$ and $T = \frac{N}{R^2}$, to use the earlier notation).

By considering the continued fraction expansions of $\frac{q}{p}$ and $\frac{p}{q}$ one can show, as before, that there exist natural numbers r, s with

$$rs < \frac{N}{R^2} \quad \text{and} \quad | pr - qs | < R .$$

Again as before, putting $k = rs$ and $A = pr + qs = [\sqrt{4kN}] + d$, one can show that

$$4kN = (pr + qs)^2 - (pr - qs)^2$$

implies that

$$A \leqslant [\sqrt{4kN}] + \left[\frac{R^2}{4\sqrt{kN}}\right] + 1 ,$$

and one can again show that $d \geqslant 1$.

Thus, in this case, the number of (k, d) pairs which need to be considered for Step 2 becomes

$$\sum_{k=1}^{\left[\frac{N}{R^2}\right]} \sum_{d=1}^{\left[\frac{R^2}{4\sqrt{(kN)}}\right]+1} 1 \leqslant \sum_{k=1}^{\frac{N}{R^2}} \sum_{d=1}^{\frac{R^2}{4\sqrt{(kN)}}+1} 1$$

$$= \sum_{d=1}^{\frac{R^2}{4\sqrt{N}}+1} \sum_{k=1}^{\min\left(\frac{N}{R^2}, \frac{R^4}{16(d-1)^2 N}\right)} 1$$

$$= \frac{N}{R^2} + \frac{R^4}{16N} \sum_{\delta=1}^{\frac{R^2}{4\sqrt{N}}} \frac{1}{\delta^2}$$

$$\leqslant \frac{N}{R^2} + \frac{R^4}{16N} \cdot \frac{\pi^2}{6}$$

$$= K(R),$$

say.

Now, let us find the minimum value of $K(R)$.

$$\frac{dK(R)}{dR} = -\frac{2N}{R^3} + \frac{R^3\pi^2}{24N}$$

which equals zero when

$$\frac{2N}{R^3} = \frac{R^3\pi^2}{24N}$$

i.e. when

$$R^6 = 48\frac{N^2}{\pi^2}$$

i.e. when

$$R = \left(48\frac{N^2}{\pi^2}\right)^{1/6} \approx (1\cdot 30)N^{1/3}.$$

It is easy to see that this value of R represents a minimum value for $K(R)$, which is equal to

$$\frac{96N^2 + ((48N^2)/\pi^2)\times\pi^2}{96N \times (48/\pi^2)^{1/3}N^{2/3}}$$

$$= \frac{144N^2}{96N^{5/3}}\left(\frac{\pi^2}{48}\right)^{1/3}$$

$$= \frac{3}{2}\left(\frac{\pi^2}{48}\right)^{1/3} N^{1/3} \approx 0\cdot 885N^{1/3}.$$

However, as was shown earlier, the amount of trial division required in Step 1 is inversely proportional to the upper limit for k in Step 2. We have proved above that to minimise the maximum time required for Step 2 will require trial division up to $\left(\frac{48}{\pi^2}\right)^{1/6} N^{1/3}$, which is more than that for the Voorhoeve version. Therefore, this choice for R does not necessarily correspond to the smallest maximum running time of the whole algorithm. To find this value we would need to minimise the function

$$\gamma R + \frac{N}{R^2} + \frac{R^4\pi^2}{96N}$$

where γR is the amount of time Step 1 takes, and so will depend on which sequence of trial divisors is used. In assigning a value to the constant γ, one will also need to take into account the difference in time required for the operations of Step 1 (i.e. a division, after an addition to generate the divisor when necessary) compared with the various multiplications, additions, subtractions and perfect square tests involved in Step 2. Thus, to a certain extent, this value, and hence the optimal strategy, will depend on the machine being used.

Indeed, it may be that the optimal strategy will not be to perform all of Step 1 before the work of Step 2. It is clear that trial division finds small factors very quickly, and we have shown above how Step 2 can yield factors rapidly in certain circumstances (e.g. if $N = pq$, where p is very close to q). Thus, the optimal strategy might consist of alternating between the two parts of the algorithm. However, to ascertain if this is the case would involve factoring a large number of integers, chosen at random, and consequently require a vast amount of machine time. Hence, it was not feasible to include this analysis in this research project.

Possible values for d

The above discussion concerning the selection of (k, d) pairs has centered on which k values are worth considering, and we have tacitly assumed that all the corresponding d values need to be considered. This is, in fact, not the case. We will show below how, as Lehman points out, not every d value needs to be examined, and then use this result to achieve a lower bound for the number of (k, d) pairs that need to be processed in Step 2. The exclusion, until now, of this consideration does not invalidate the conclusions already drawn. Instead, as can be seen from below, it has served to make the working involved in the running time estimates simpler and easier to follow. The (k, d) pair bound obtained at the end of this discussion is for the Voorhoeve ($T = N^{1/3}$) version. A similar analysis could be done for the optimal strategy estimate above.

Putting, as before, $A = pr + qs$ and $B = | pr - qs |$ (where $k = rs$ and $N = pq$) we have that

$$A^2 + B^2 = 4kN .$$

Now, suppose that A is even. Then either both pr and qs are even, or they are both odd. The former condition implies that both r and s are even (since p and q are odd) which is a contradiction because $(r, s) = 1$ (this follows from the fact that $\frac{r}{s}$ is a convergent to $\frac{q}{p}$). Thus the latter must apply. This implies that both r and s, and hence k, are odd. In other words, A even implies that k is odd.

Similarly, if A is odd, then either pr is even and qs is odd, or vice versa. The former implies that r is even and s is odd (since both p and q are odd), while the latter implies the opposite. From both possibilities, therefore, it follows that $k = rs$ is even, i.e. A odd implies that k is even.

Thus, in both cases, we have that

$$A \equiv k + 1 (mod\ 2)$$

$$\text{i.e. } d \equiv k + 1 - [\sqrt{4kN}] (mod\ 2)$$

and so we need only look at approximately half of all the possible (k, d) pairs.

In fact, we can do better than this for, suppose that k is odd. Then both r and s are odd. Now

$$(p - s)(q - r) = pq + rs - pr - qs = N + k - A ,$$

and because $(p - s)$ and $(q - r)$ are both even (and > 0 since, for example, $s \leqslant k < p$),

we have that

$$A \equiv N + k \,(mod\ 4)$$

$$\text{i.e. } d \equiv N + k - [\sqrt{4kN}\,](mod\ 4)\,.$$

Hence, if k is odd, we need only consider one out of every four of the possible values for d .

Of course, for much of the time spent on Step 2 (depending on how soon a pair of squares is found), the limits on the d loops will be less than 4 (and possibly less than 2 also) and so making use of this feature will not, in general, result in only looking at $\frac{3}{8}$ of all the possible (k, d) pairs. However, as will now be shown, when this pre-selection of d values is included in the count of operations (i.e. count of (k, d) pairs) given towards the end of Chapter 3, one is able, as was mentioned, to reduce the constant associated with $N^{1/3}$.

The count of (k, d) pairs can be divided into two parts. When k is even, we need only include those d's which satisfy $d \equiv k + 1 - [\sqrt{4kN}\,](mod\ 2) \equiv \alpha_k\ (mod\ 2)$, say, and when k is odd, we need only count those d values for which $d \equiv \beta_k\ (mod\ 4)$, where $\beta_k \equiv N + k - [\sqrt{4kN}\,](mod\ 4)$. Thus, for a given N, in both cases the selection of d values depends only on k. From this one can see that the number of (k, d) pairs to be considered becomes

$$S = \sum_{k=1,\,k\ even}^{[N^{1/3}]} \ \sum_{d=1,\,d\equiv\alpha_k (mod\ 2)}^{\left[\frac{N^{1/6}}{4\sqrt{k}}\right]+1} 1 \ + \sum_{k=1,\,k\ odd}^{[N^{1/3}]} \ \sum_{d=1,\,d\equiv\beta_k (mod\ 4)}^{\left[\frac{N^{1/6}}{4\sqrt{k}}\right]+1} 1$$

$$= S_1 + S_2$$

say. We deal first with S_1.

If $\left[\frac{N^{1/6}}{4\sqrt{k}}\right]$ is odd, then the inner sum in S_1 is just $\frac{1}{2}\left(\left[\frac{N^{1/6}}{4\sqrt{k}}\right] + 1\right)$, while if $\left[\frac{N^{1/6}}{4\sqrt{k}}\right]$ is even, the inner sum is $\leqslant \frac{1}{2}\left[\frac{N^{1/6}}{4\sqrt{k}}\right] + 1$. Thus, in both cases, the sum over d is $\leqslant \frac{1}{2}\left[\frac{N^{1/6}}{4\sqrt{k}}\right] + 1$, and so

$$S \leqslant \sum_{k=1,\,k\ even}^{[N^{1/3}]} \left(\frac{1}{2}\left[\frac{N^{1/6}}{4\sqrt{k}}\right] + 1\right)$$

$$= \sum_{k=1,\, k \text{ even}}^{[N^{1/3}]} 1 + \sum_{k=1,\, k \text{ even}}^{[N^{1/3}]} \frac{1}{2} \left[\frac{N^{1/6}}{4\sqrt{k}} \right]$$

$$\leqslant \sum_{k=1,\, k \text{ even}}^{N^{1/3}} 1 + \sum_{k=1,\, k \text{ even}}^{N^{1/3}} \sum_{d=1}^{\frac{N^{1/6}}{8\sqrt{k}}} 1$$

$$= \sum_{k=1,\, k \text{ even}}^{N^{1/3}} 1 + \sum_{d=1}^{\frac{N^{1/6}}{8\sqrt{2}}} \sum_{k=1,\, k \text{ even}}^{\frac{N^{1/3}}{64d^2}} 1$$

$$\leqslant \sum_{k=1,\, k \text{ even}}^{N^{1/3}} 1 + \sum_{d=1}^{\frac{N^{1/6}}{8\sqrt{2}}} \frac{N^{1/3}}{128d^2}$$

(If $\left[\frac{N^{1/3}}{64d^2}\right]$ is even, the inner sum of the second term equals $\frac{1}{2}\left[\frac{N^{1/3}}{64d^2}\right] \leqslant \frac{N^{1/3}}{128d^2}$, whereas if not, the inner sum equals $\left[\frac{1}{2}\left[\frac{N^{1/3}}{64d^2}\right]\right] \leqslant \frac{N^{1/3}}{128d^2}$ - hence the inequality above.)

$$= \sum_{k=1,\, k \text{ even}}^{N^{1/3}} 1 + \frac{N^{1/3}}{128} \sum_{d=1}^{\frac{N^{1/6}}{8\sqrt{2}}} \frac{1}{d^2}$$

$$\leqslant \sum_{k=1,\, k \text{ even}}^{N^{1/3}} 1 + \frac{N^{1/3}}{128} \sum_{d=1}^{\infty} \frac{1}{d^2}$$

$$= \sum_{k=1,\, k \text{ even}}^{N^{1/3}} 1 + \frac{N^{1/3}\pi^2}{6 \times 128}. \tag{7.6}$$

Similarly, for S_2 we have that

$$S_2 \leqslant \sum_{k=1,\, k \text{ odd}}^{[N^{1/3}]} \left(\frac{1}{4} \left[\frac{N^{1/6}}{4\sqrt{k}} \right] + 1 \right)$$

$$= \sum_{k=1,\, k \text{ odd}}^{[N^{1/3}]} 1 + \sum_{k=1,\, k \text{ odd}}^{[N^{1/3}]} \frac{1}{4} \left[\frac{N^{1/6}}{4\sqrt{k}} \right]$$

$$\leqslant \sum_{k=1,\, k \text{ odd}}^{N^{1/3}} 1 + \sum_{k=1,\, k \text{ odd}}^{N^{1/3}} \sum_{d=1}^{\frac{N^{1/6}}{16\sqrt{k}}} 1$$

$$= \sum_{k=1,\, k \text{ odd}}^{N^{1/3}} 1 + \sum_{d=1}^{\frac{N^{1/6}}{16}} \sum_{k=1,\, k \text{ odd}}^{\frac{N^{1/3}}{256d^2}} 1 .$$

Now, if $\left[\frac{N^{1/3}}{256d^2}\right]$ is even, the inner sum in the second term equals $\frac{1}{2}\left[\frac{N^{1/3}}{256d^2}\right]$, while if not, this sum equals $\left[\frac{1}{2}\left[\frac{N^{1/3}}{256d^2}\right]\right] + 1$. In both cases we have that the inner sum is $\leqslant \frac{N^{1/3}}{512d^2} + 1$ and so we have that S_2 is

$$\leqslant \sum_{k=1,\, k \text{ odd}}^{N^{1/3}} 1 + \frac{N^{1/3}}{512}\sum_{d=1}^{\infty}\frac{1}{d^2} + \frac{N^{1/6}}{16}$$

$$= \sum_{k=1,\, k \text{ odd}}^{N^{1/3}} 1 + \frac{N^{1/3}}{512}\cdot\frac{\pi^2}{6} + \frac{N^{1/6}}{16}. \tag{7.7}$$

Combining (7.6) and (7.7) we can see that

$$\begin{aligned} S &= S_1 + S_2 \\ &\leqslant \sum_{k=1}^{N^{1/3}} 1 + \frac{\pi^2 N^{1/3}}{6}\left(\frac{1}{128} + \frac{1}{512}\right) + \frac{N^{1/6}}{16} \\ &= \left(1 + \frac{5\pi^2}{3072}\right)N^{1/3} + \frac{N^{1/6}}{16}, \end{aligned}$$

which is not in the form we would have wished. However, one can eliminate the term in $N^{1/6}$ by improving the bound for S_1.

From two lines before (7.6) we have that

$$\begin{aligned} S_1 &\leqslant \sum_{k=1,\, k \text{ even}}^{N^{1/3}} 1 + \frac{N^{1/3}}{128}\sum_{d=1}^{\frac{N^{1/6}}{8\sqrt{2}}}\frac{1}{d^2} \\ &= \sum_{k=1,\, k \text{ even}}^{N^{1/3}} 1 + \frac{N^{1/3}}{128}\sum_{d=1}^{\infty}\frac{1}{d^2} - \frac{N^{1/3}}{128}\sum_{d=\left[\frac{N^{1/6}}{8\sqrt{2}}\right]+1}^{\infty}\frac{1}{d^2}. \end{aligned}$$

Now, let X and Y be positive integers. Then

$$\sum_{d=X}^{Y}\frac{1}{d^2} > \sum_{d=X}^{Y}\frac{1}{d(d+1)} = \sum_{d=X}^{Y}\left(\frac{1}{d} - \frac{1}{d+1}\right) = \frac{1}{X} - \frac{1}{Y}.$$

Letting $Y \to \infty$ we see that

$$\sum_{d=X}^{\infty}\frac{1}{d^2} \geqslant \frac{1}{X}$$

where X is an integer. Hence

$$S_1 \leqslant \sum_{k=1,\, k\ even}^{N^{1/3}} 1 + \frac{N^{1/3}}{128} \cdot \frac{\pi^6}{6} - \frac{N^{1/3}}{128} \frac{1}{\left[\frac{N^{1/6}}{8\sqrt{2}}\right] + 1}$$

and we will have

$$S \leqslant \left(1 + \frac{5\pi^2}{3072}\right) \approx (1\cdot016) N^{1/3},$$

provided that

$$\frac{N^{1/6}}{16} \leqslant \frac{N^{1/3}}{128} \cdot \frac{1}{\left[\frac{N^{1/6}}{8\sqrt{2}}\right] + 1} \leqslant \frac{N^{1/3}}{128} \cdot \frac{1}{\frac{N^{1/6}}{8\sqrt{2}} + 1} = \frac{N^{1/3}}{16} \cdot \frac{\sqrt{2}}{(N^{1/6} + 8\sqrt{2})}$$

i.e. provided that

$$N^{1/6}(N^{1/6} + 8\sqrt{2}) \leqslant \sqrt{2} N^{1/3}$$

i.e. provided that

$$(\sqrt{2} - 1) N^{1/3} \geqslant 8\sqrt{2} N^{1/6}$$

i.e. provided that

$$N \geqslant \left(\frac{8\sqrt{2}}{\sqrt{2} - 1}\right)^6 \approx 4\cdot 152 \times 10^8 .$$

Thus, provided that N is greater than this bound, the number of (k, d) pairs to be considered is approximately $(1\cdot016) N^{1/3}$ which is about 10% less than the best bound obtained in Chapter 3 (which was $1\cdot1028\, N^{1/3}$).

Therefore, by including this feature, the times obtained on the VAX for the test cases in Chapter 6 could be reduced by 10%. Of course, the improvement could be much greater in factorisations where squares are found comparatively quickly (i.e. before the d loop limit becomes 1), for then reductions of up to (and sometimes exceeding) 50% are possible.

Unfortunately, though, this modification is not suitable for inclusion in the DAP implementation in which, as already discussed, 4096 k values are handled simultaneously. Even if we took the even values first and then the odd, there is no guarantee that, say, for every member of a given block of 4096 even k values, the expression $k + 1 - [\sqrt{4kN}]$ will give rise to the same residue modulo 2, and thus some of the corresponding inner loops would require d to take the values $1, 3, 5, \cdots$ while others involved the processing of $d = 2, 4, 6, \cdots$. This is a drawback with using an SIMD machine that we have already mentioned (of course, such a structure has very real advantages in other ways), and provides another example of the "best possible serial version" being different to that for the parallel

machine.

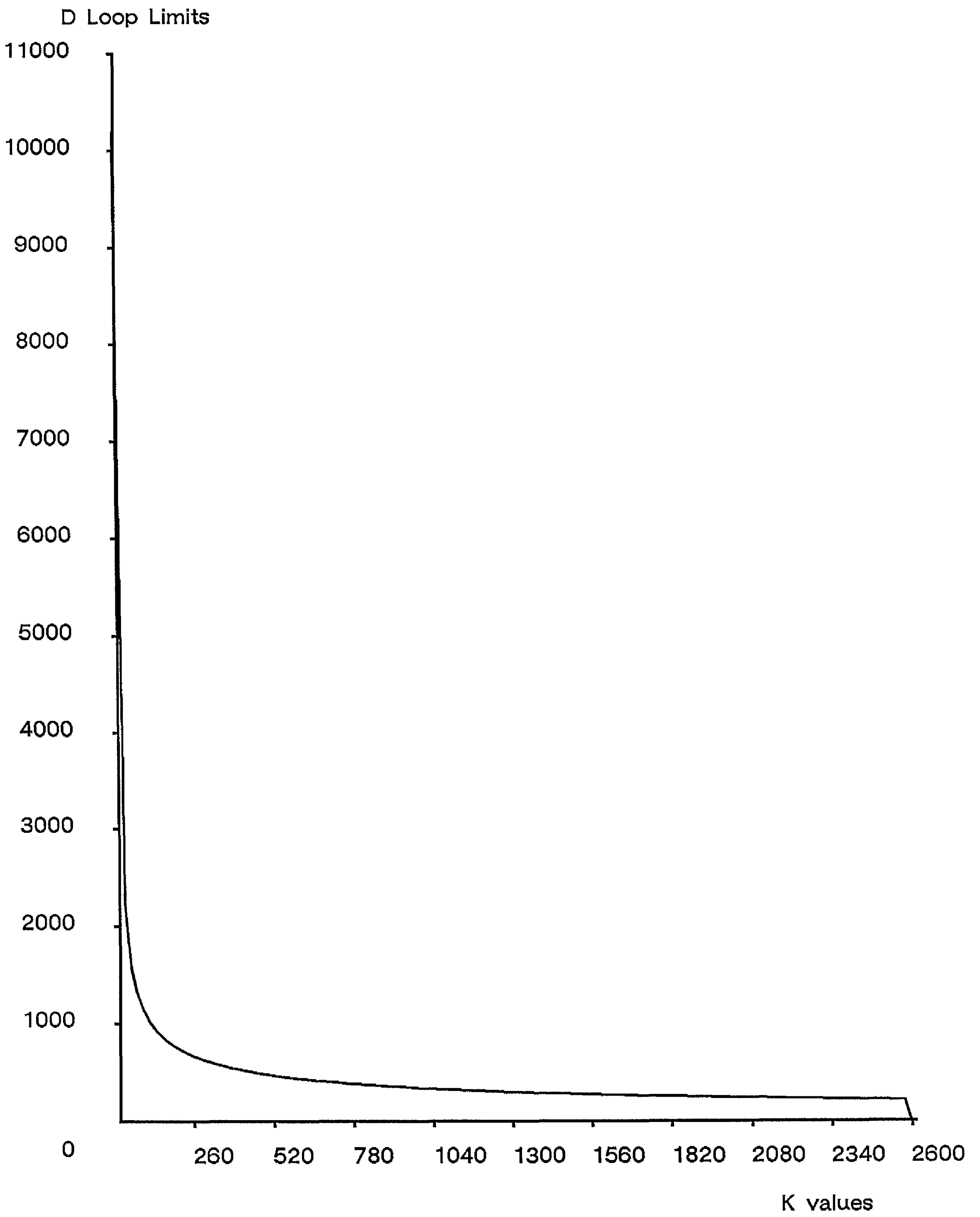


PLATE 4 : T = 2589 with 11,111,111,111,111,111

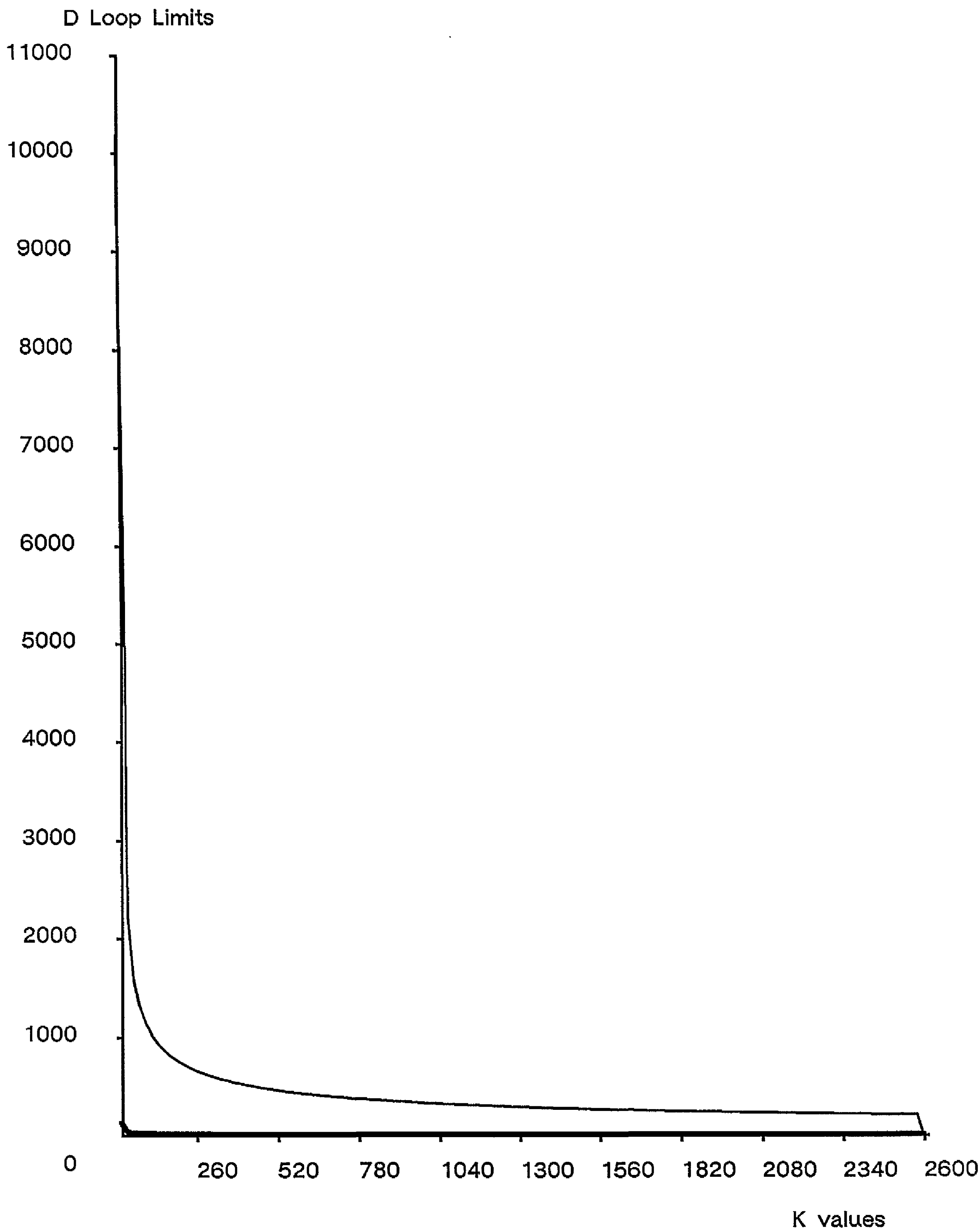


PLATE 5 : comparison for 11,111,111,111,111,111

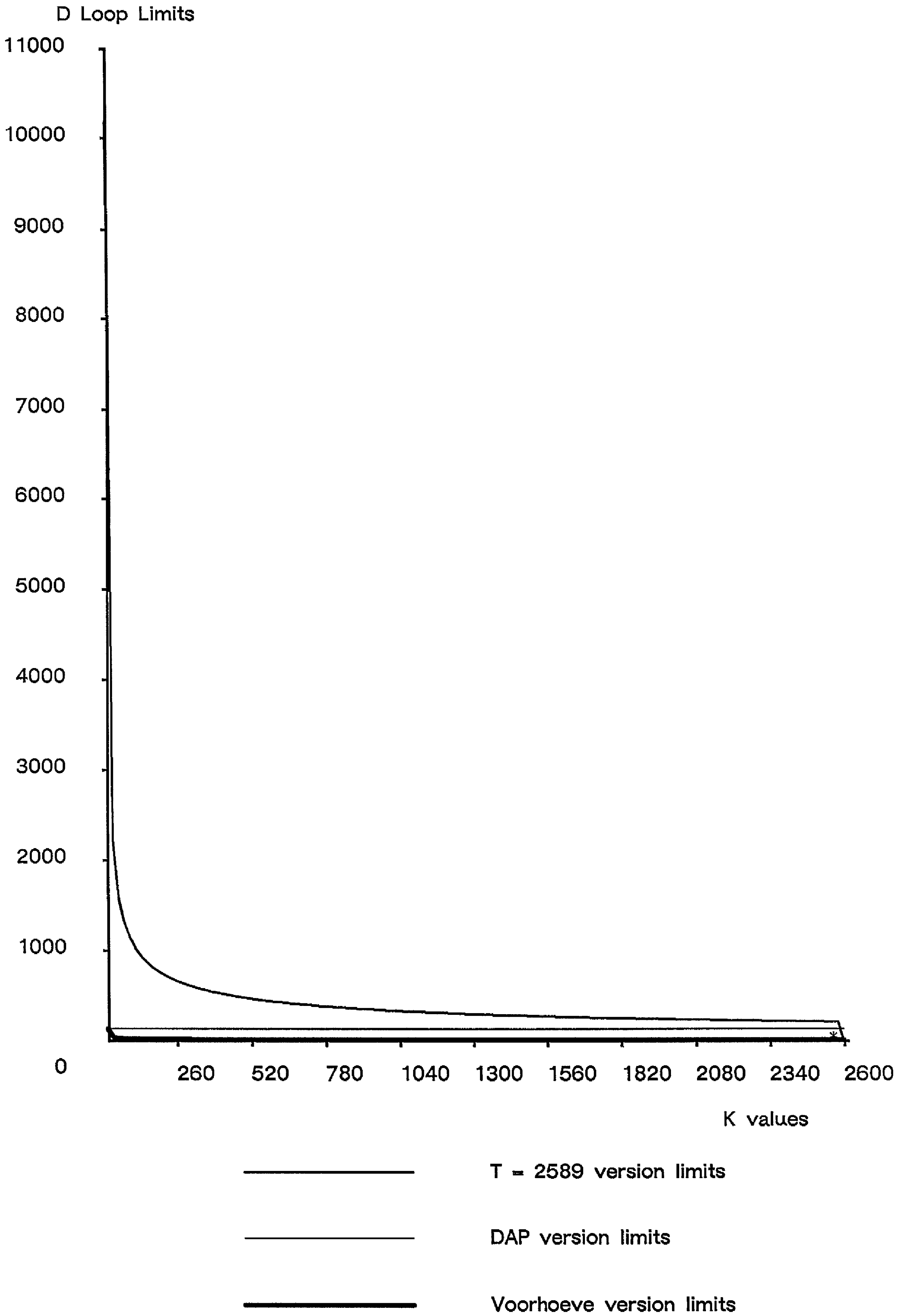


PLATE 6 : further comparison for 11,111,111,111,111,111

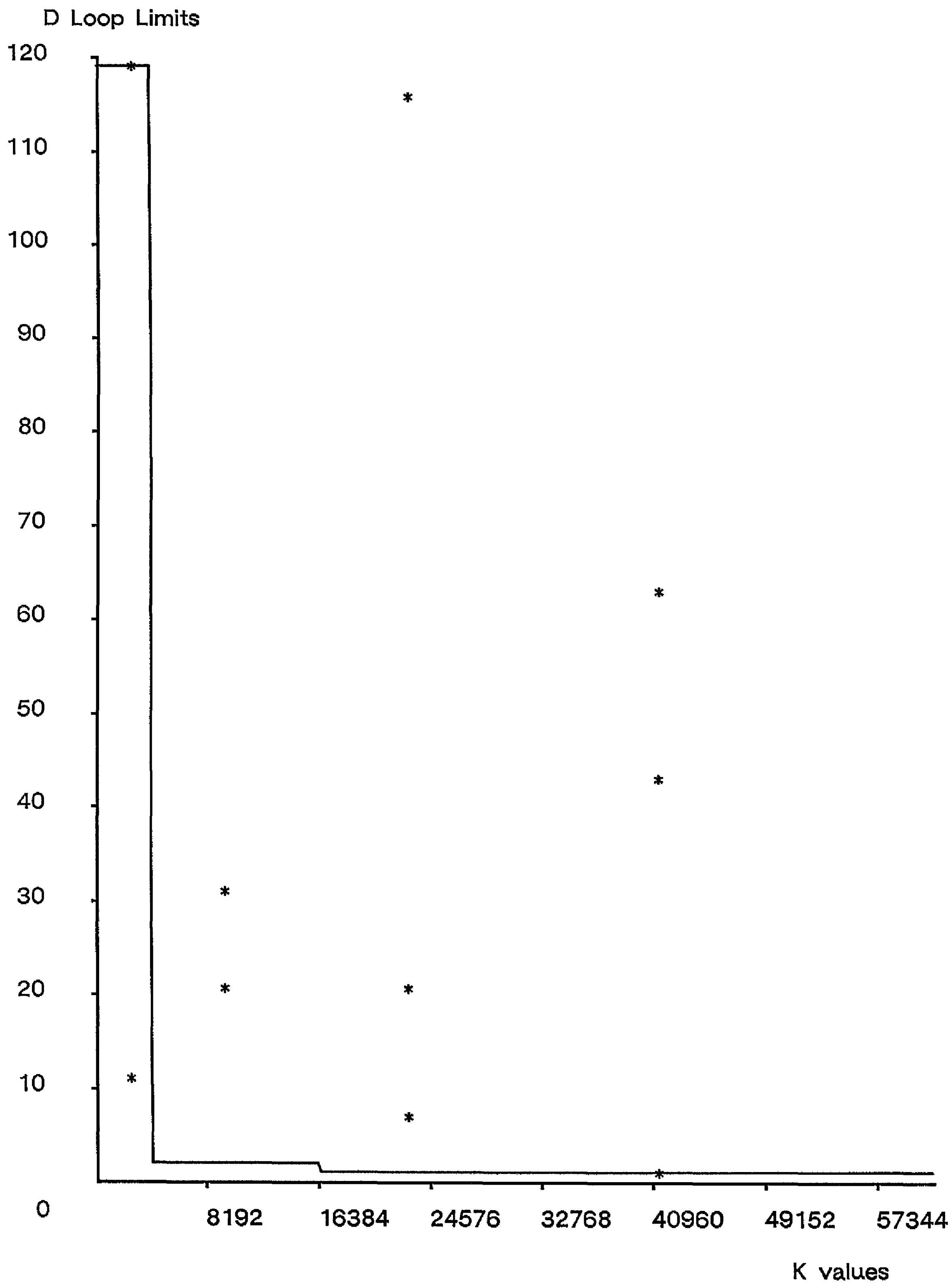


PLATE 7 : (k,d) pairs within "easy reach"

Chapter 8 An Alternative Technique for Detecting Divisors

It is well known that a decimal integer is divisible by 3 if, and only if, the sum of its digits is also divisible by 3. Mathematically, what we are saying is this.

Suppose the integer in question, N , say, equals $(d_n \; d_{n-1} \cdots d_1 d_0) = \sum_{i=0}^{n} d_i 10^i$. Then

$$\begin{aligned}
N \; mod \; 3 &= \left(\sum_{i=0}^{n} d_i 10^i \right) mod \; 3 \\
&= \left(\sum_{i=0}^{n} (d_i 10^i \; mod \; 3) \right) mod \; 3 \\
&= \left(\sum_{i=0}^{n} (d_i mod \; 3).(10^i mod \; 3) \right) mod \; 3 \\
&= \left(\sum_{i=0}^{n} (d_i mod \; 3).1 \right) mod \; 3 \\
&= \left(\sum_{i=0}^{n} d_i \; mod \; 3 \right) mod \; 3 \\
&= \left(\sum_{i=0}^{n} d_i \right) mod \; 3 .
\end{aligned}$$

Thus,

$$N \; mod \; 3 = 0 \quad if \; and \; only \; if \quad \left(\sum_{i=0}^{n} d_i \right) mod \; 3 = 0 .$$

In other words,

$$\text{N is divisible by 3} \quad if \; and \; only \; if \quad \left(\sum_{i=0}^{n} d_i \right) mod \; 3 = 0 .$$

This method is easy to use because $10^i \; mod \; 3 = 1$ for all $i \geqslant 0$, and so, apart from the final division by 3, the only work required is to calculate a sum. However, it could easily be generalised to other number bases, and to other divisors - the only cost being that the

arithmetic involved might not be so simple. In particular, a variation of this technique could be used to identify divisors of an integer, given its binary expansion.

Suppose that we consider the binary representation of N, which we shall denote by $b_n\, b_{n-1} \cdots b_1 b_0$ (i.e. $N = \sum_{i=0}^{n} b_i 2^i$), and wish to know if a prime p divides N. Then,

$$\begin{aligned} N \bmod p &= \left(\sum_{i=0}^{n} b_i 2^i \right) \bmod p \\ &= \left(\sum_{i=0}^{n} (b_i \bmod p)(2^i \bmod p) \right) \bmod p \\ &= \left(\sum_{i=0}^{n} b_i (2^i \bmod p) \right) \bmod p , \quad \text{since } b_i = 0 \text{ or } 1 \\ &= \left(\sum_{i=0,\, b_i=1}^{n} 2^i \bmod p \right) \bmod p . \end{aligned}$$

$$\text{i.e. } N \bmod p = 0 \text{ if and } \textit{only} \text{ if } \left(\sum_{i=0,\, b_i=1}^{n} 2^i \bmod p \right) \bmod p = 0 .$$

Thus, to see if p divides N, all we need to do is to sum those $2^i \bmod p$ for which the corresponding b_i is non-zero. If this sum, modulo p, is equal to 0, then p divides N.

This method would apply even if p was not a prime. However, as our aim is to find the prime factors of a given integer, we will only be interested in primes (and perhaps some powers of primes).

To use this idea to factorise a number N, say, into primes, we would need to find all the primes (and powers of primes) p, such that $\left(\sum_{i=0,\, b_i=1}^{n} 2^i \bmod p \right) \bmod p = 0$, where $N = \sum_{i=0}^{n} b_i 2^i$, as before. This involves, for each prime p tried, the calculation of $2^i \bmod p$ for possibly many values of i and accordingly, a considerable amount of work for just one division. However, if we had a fixed set of primes (called a factor base), and we wished to factor many integers over this base, then the initial calculation of all the residues of the powers of 2 could be offset by the speed of each individual factorisation.

As Professor D. Parkinson suggested to the author, this method is ideally suited to an array processor, for each processor could hold a different prime p (or if necessary, a power of p), as well as the residues $2^i \bmod p$ for i in the range 0 to the word length being used, which could all be calculated at the start. Then, as we considered each binary digit of N in turn, whenever a 1 was encountered, all the sums (one per processor) could be updated simultaneously with the respective residue. Thus, for the DAP, we could simultaneously

divide N by up to 4096 primes. The primes which did divide N could easily be found at the end, by picking out all the processors (and hence primes), whose sum, modulo its particular prime, was equal to 0.

A version of the above which makes use of a more efficient binary representation of the number concerned has been implemented on the DAP by the author, and a discussion of it now follows.

In the theory above, it was assumed that the number to be factored would be represented in normal binary form as:

$$N = \sum_{i=0}^{[log_2 N]} b_i 2^i , \quad b_i = 0,1 .$$

However, an alternative binary representation is:

$$N = \sum_{i=0}^{[log_2 N]+1} c_i 2^i , \quad c_i = -1,0,1 ,$$

where a block of 1's is replaced by 0's except for a -1 in the location of the rightmost 1, and the 0 immediately to the left of the block is replaced by a 1. This is a valid transformation since $\sum_{i=j}^{k} 2^i = 2^{k+1} - 2^j$. (The sum goes up to $[log_2 N] + 1$ to cater for when the leftmost binary digits of N form a block of 1's, in which case a 1 is inserted to the left of the block, in the $([\log_2 N] + 1)^{th}$ position. Otherwise, this position will (be considered to) contain a 0.)

The time taken for this algorithm is proportional to the number of additions and subtractions (if any) which we perform. With the first representation, this is $\sum_{i=0}^{[log_2 N]} b_i$, whereas the second method only requires $\sum_{i=0}^{[\log_2 N]+1} |c_i|$ and, as can be seen from the following example, this second sum could be much less than the first.

Suppose we are considering the number whose normal binary representation is

1111111001.

Dividing this by a given set of trial divisors would require 8 additions using the method already given. However, if we considered the number to be

1000000(−1)001

then only 3 "additions" would be needed, (where we assume that a subtraction takes approximately the same time as an addition).

Thus, this second version would appear to be the better method. It is instructive to note that this representation is analagous to the "short-cutting" method of multiplication used in hand calculators [Comrie and Hartley 1939]. However, before this algorithm can be implemented, one has to decide on a way of obtaining the alternative binary representation described above. Examining the normal binary pattern of the number concerned beforehand, through LOGICAL vectors and converting it where necessary is not possible, since we cannot store the digits 0,1, and -1 in terms of the values .TRUE. and .FALSE.! But, in fact, it is not necessary to physically change the bits of N : the process of conversion can be carried out "conceptually" as the algorithm is performed. We now describe two ways in which this can be done.

First suggestion

The first scheme uses a LOGICAL flag (initialised to .FALSE.) to identify the start of a block of (possibly only one) 1's (or .TRUE.'s). Working from right to left along the binary pattern for N , the method consists of:

- Whenever a 1 is encountered (and flag is .FALSE., implying that this could be the start of a block), perform a subtraction, and set the flag to .TRUE..
- If any more 1's are encountered while the flag is .TRUE., do nothing.
- At the first 0 while the flag is .TRUE., perform an addition, and set the flag to .FALSE., and repeat the whole process if there are any more binary digits to consider. (Whenever the bit of N in question is 0, but the flag is .FALSE., do nothing.)
- If, after all the binary digits of N have been considered, the flag is still .TRUE. then perform an addition corresponding to the next power of 2, and set flag to .FALSE. (i.e. consider N to have an extra binary digit which equals 0 on the left).

There are four logical possibilities to consider here:

Flag	Bit under consideration	Action
T	1	do nothing
F	0	do nothing
T	0	add ; flag $\leftarrow$ F
F	1	subtract ; flag $\leftarrow$ T

The only "snag" with this is that a single 1 in the binary pattern of N is treated in the same way as a block of 1's, and so, instead of just an addition, one has to perform a subtraction, and then add on the next iteration of the loop. However, the method can be extended to take this point into account, as we shall now show.

Second suggestion

Again we use a LOGICAL flag, initialised to .FALSE., to identify blocks of 1's. However, this time, as we work from right to left along the binary pattern of N , we do the following. Whenever a 1 is encountered:

- if flag is .FALSE. and the next digit is 0 (i.e. a single 1), then add, but leave flag unchanged;
- if flag is .TRUE. and the next digit is 0 (i.e. a 1 at the left hand end of a block of 1's), do nothing;
- if flag is .FALSE. and the next digit is 1 (i.e. a 1 at the start of a block of 1's), then subtract and set flag to .TRUE.;
- if flag is .TRUE. and the next digit is 1 (i.e. a 1 in the middle of a block of 1's), do nothing.

Whenever a 0 is encountered:

- if flag is .TRUE. (i.e. the first 0 after a block of more than one 1's), add, and set flag to .FALSE.;
- if flag is .FALSE. (i.e. a 0 after a single 1 or after another 0), do nothing.

The logical possibilities here are:

Flag	Next bit	Bit under consideration	Action
F	0	1	add ; flag unchanged
T	0	1	do nothing
F	1	1	subtract ; flag $\leftarrow$ T
T	1	1	do nothing
F	irrelevant	0	do nothing
T	irrelevant	0	add ; flag $\leftarrow$ F

Assuming the declarations:

```
LOGICAL     FLAG , BITS (73)
INTEGER*8   TOTALS ( , )
INTEGER*4   MODPOWERS ( , ,73) , PRECISION
```

where the LOGICAL array BITS holds the normal binary representation of N (with the least significant bit stored in the first location), the array of INTEGER matrices MODPOWERS will already have been assigned the required residues, and the INTEGER scalar PRECISION holds the length of the binary pattern of N (where N is assumed to be $\leqslant 2^{72}$), the following piece of DAP-FORTRAN will perform this algorithm:

```
   FLAG = .FALSE.
   TOTALS = 0
   BITS (PRECISION+1) = .FALSE.
   DO 50 I = 1 , PRECISION
   IF ((.NOT.(FLAG .OR. BITS (I+1))) .AND. BITS (I)) GO TO 40
   IF ((.NOT.FLAG) .AND. (BITS(I) .AND. BITS (I+1))) GO TO 20
   IF (FLAG .AND. (.NOT BITS (I))) GO TO 10
   GO TO 50
10 TOTALS = TOTALS + MODPOWERS( , ,  I)
   GO TO 30
20 TOTALS = TOTALS - MODPOWERS ( , ,  I)
30 FLAG = .NOT.FLAG
   GO TO 50
40 TOTALS = TOTALS + MODPOWERS ( , ,  I)
50 CONTINUE
```

It is this second method that has been implemented. As we have already remarked, this idea is suitable for use when trying to factor many integers (one at a time) over a fixed set of divisors. This is what is required at the start of Step 1 of the Lehman algorithm. We have described in Chapter 4 how N , the integer to be factored, is divided by a matrix containing the first 4096 primes, which is read in at the start of the program. If instead, the 73 matrices of residues required by the above algorithm were read in, then this method could be used rather than normal division, to identify which, if any, of the first 4096 primes were factors of N . It could be used for all the other blocks too, provided we had sufficient memory to store the tens of thousands of matrices that would be required, and of course, we don't! But, even though it is relatively expensive in storage, this scheme is certainly worth it for, as we will show, it constitutes a much more efficient way of identifying divisors of an

integer.

As has already been discussed, arithmetic operations on matrices have to be performed one bit plane at a time by software routines. When one of the operands is a scalar, this allows us the possibility of taking advantage of the form of its binary pattern. For instance, to multiply a matrix by a scalar will involve repeated shifts and additions of the elements of the matrix. However, if a given bit of the scalar is zero, there is no need to shift the matrix elements. Instead one could wait until a 1 was encountered, and then shift the elements the appropriate number of places.

Such a trick, unfortunately, is not possible with long-division of a scalar by a matrix. In this case all the divisors are shifted until their most significant bits correspond to that of N, and then shifted to the right one bit at a time, with subtractions from N being performed where possible, until all the least significant bits are aligned. Thus, the number of shifts and subtractions is proportional to the length of the binary pattern of N. With the algorithm given above, however, the number of additions and subtractions is proportional to the number of 1's in the binary expansion of N (and no shifting is required). In fact, the number of additions and subtractions equals the sum of the number of single 1's, plus twice the number of blocks which contain two or more 1's in the binary pattern. So we have a method which, like the trick for multiplication mentioned above, allows us to do only as much work as is necessary to identify divisors of N. Admittedly, the above does not find the quotient after dividing N by the product of the factors, but this can be found easily by scalar division. We have already remarked how scalar operations are approximately ten times quicker than the matrix equivalents, and since this method should take less than 90% of the time required for conventional division, (an assertion which will be justified below), it will still be quicker. Indeed, when more than 1 factor is found by using the latter method, say p, q, r, then a scalar division routine still has to be used to find $\frac{N}{pqr}$ since we will only know $\frac{N}{p}$, $\frac{N}{q}$, $\frac{N}{r}$, and we have already discussed, in Chapter 5, how it is quicker to remove all the factors found at once, rather than one at a time.

When this method was used to factor the integer 2^{70}, a number for which this technique should have been very suitable, it took 545 millisecs of DAP-time, whereas the normal division routine took only 405. One reason for this was that because of space limitations, the matrix of powers had to be computed at the start of the former program. When corresponding statements were added to the latter program, it took 503 millisecs. Apart from the obvious reason, namely the amount of indexing required in using a high-level language, another possibility is described below.

In the high-level code given above, it was assumed that the binary pattern of N was stored in an array, rather than having the bits accessed directly from the INTEGER*4 locations across which they are stored. This is because of a small problem with equivalencing LOGICAL vectors to INTEGER scalars: the elements of a vector are indexed from left to right, whereas the binary pattern of a scalar is "written" from right to left. Thus, because the counter in a FORTRAN DO loop cannot take decreasing values, a second indexing variable has to be used, whose value is decreased by 1 as part of each loop iteration. It was felt that the above code would be easier to follow if the bits of N had been copied into a LOGICAL array (with the least significant bit in location 1), so that a single indexing variable would suffice, and in fact, this was the version that was run initially. When the program was changed to process the binary expansion of N without copying it into an array, the program took slightly longer to run, (577 millisecs) because the former version included a search for the most significant bit of N , and so as each bit of N was processed in turn, it was known how many there were to consider. In the second method no such search was made. Instead, for simplicity, all 31 bits of each multiple-precision digit of N were considered. Thus, a reduction in the running could be made, by including an examination of N beforehand, to identify its most significant bit. But, of course, what would be more worthwhile would be to implement this algorithm in APAL, thus allowing it to be compared fairly with the normal division software routine.

In a previous paragraph it was supposed that this alternative technique should require at most 90% of the time taken by the normal division routine. We will now show how, in theory, this assertion can be justified.

Suppose, for example, that the number to be factored, N has 70 bits. Since the first and 4096th primes have respectively 2 and 16 bits, to divide N by the normal method will require 69 shifts and subtractions, most of which (55, to be exact) will involve operating on 16 bits. Thus, ignoring shifts, the number of bit operations will be equal to

$$(55 \times 16) + \sum_{i=1}^{15} i = 880 + 120 = 1000 .$$

This figure is constant for all N of this length, since it is independent of the number of bits set in the binary pattern.

On the other hand, one would imagine that the alternative method should be quicker for those N whose binary patterns contain more 0's than 1's, with perhaps less of an improvement in the case where more than half the bits in N are set. Since this suggests a natural division of possibilities, we will consider each case separately.

Suppose, first, that less than (or equal to) half of the bits in the binary pattern of N, are 1's. Hence there are at most 35 out of the 70 bits set. The best we could have is for all the 35 1's to be in consecutive locations, and thus only one addition and one subtraction will be required by the algorithm. Since the largest residue can have, at most, 16 bits, the number of bit operations required to form the necessary sums will be $\leqslant 32$. The worst case will be when one addition will be required for each bit set, and this will correspond to the 1's alternating with 0's, or appearing in pairs, with a zero on either side. Hence, the number of bit operations needed will be at most

$$35 \times 16 \ = \ 560 .$$

Of course, once the sums have been formed, they need to be reduced modulo the corresponding prime. Now, as will be shown below, for integers with 70 bits, the most number of additions required will be 47, and so the largest sum will be $\leqslant 47 \times 38{,}872 = 1{,}826{,}984$. and so have at most 21 bits. Thus, the reduction will require, by an argument similar to the first estimate above, at most

$$(6 \times 16) + \sum_{i=1}^{15} i \ = \ 96 + 120 \ = \ 216$$

bit operations. From this we can see that if less than 49 additions or subtractions are required for the alternative algorithm, then it will be quicker than normal division, and as has been shown, this is the case for at least half of the numbers with 70 bits. In fact, this is the case for every integer $< 2^{70}$.

The number of bit operations required in the worst case rises to a maximum when 47 ($= [70/3]$) bits are set. If these bits occurred in pairs (except for the "odd one") separated by zeros,

$$47 \times 16 \ = \ 752$$

bit operations would be required. When 48 bits are set, the worst case consists of 22 pairs and a block of four 1's, (or 21 pairs and two blocks of three 1's etc.), thus requiring 46 additions and subtractions. If 49 bits are set, the most work is required, among other possibilities, when the 1's occur in 21 pairs and one block of 7, in which case 44 additions and subtractions are needed. The worst case when 50 bits are set corresponds to 20 pairs and a block of ten 1's (as well as other arrangements); a situation which would require 42 additions and subtractions, and so on.

In the above we have not included the time needed for the division to remove any factors found. In the case when more than one factor has been found, both factoring methods will require the use of a scalar division procedure. Admittedly, to remove the

primes p, q say, when using the alternative method, the value of $\frac{N}{pq}$ would need to be calculated, whereas the knowledge of $\frac{N}{q}$ could be used to slightly reduce the amount of work involved, if the normal method had been employed. However, depending on the size and number of the factors found, the difference should only be slight, and thus it is when only a single prime divides N that there could be a disadvantage in using the alternative technique. But, the worst case would be the removal of 2 for which (unless the binary pattern of N were shifted, in which case dividing N by 3 would involve the most work) $2 \times 69 = 138$ bit operations would be required. Allowing for the fact that scalar operations are about 10 times quicker than the equivalent matrix ones, this extra task should only take the equivalent of 14 of the operations which were counted above. This is approximately the same as the amount of work involved in dealing with a 1 in the binary pattern of N, and so including this figure will reduce the "cut-off" point from 49 to 48 additions or subtractions, which is still strictly greater than the most possible, namely 47.

The time required to form the product of all the divisors found, before removing them from N, has not been included either, in the above, since, whenever more than two factors have been found, both techniques will require this operation, and, in the case when only two factors have been found, the time required for the multiplication will be small, and so not affect the above conclusions.

The above analysis can be generalised as follows. To divide (using the normal method) an n-bit integer by the first 4096 primes will require

$$((n + 1) - 16) \times 16 + \sum_{i=1}^{15} i = (n - 15) \times 16 + 120$$

bit operations, whereas the amount of work involved in using the alternative method is

$$16 \times (\text{number of "additions"}) + \text{work to reduce the sums modulo } p$$

which equals

$$16 \times (\text{number of "additions"}) + (y - 15) \times 16 + 120$$

bit operations, where y satisfies

$$2^{(y-1)} \leqslant (\text{maximum number of "additions"}) \times 38{,}872 < 2^{y} .$$

Therefore, if the number of additions or subtractions required is $\leqslant n - y$ (or $< n - y$ to allow for the extra scalar division that could be necessary), then to use normal division would be slower. As was described above, the number of additions reaches a maximum when two-thirds of the bits in the dividend are set. Thus, if $\frac{2}{3}n < n - y$, i.e. if $y < \frac{n}{3}$, then, irrespective of how many bits are set, this alternative technique will be quicker. In other words, if

$$\text{(maximum number of "additions")} \times 38{,}872 < 2^{n/3}$$

i.e. if

$$\frac{2n}{3} \times 38{,}872 < 2^{n/3},$$

then this alternative technique will be quicker. It is easy to show that, for $n \geqslant 63$, the above inequality is satisfied. Of course, if numbers less than 2^{63} were to be considered, then one would find that, as n became smaller, some numbers would suit the normal division method rather than this alternative idea. But, to find the "cut-off" point, so to speak, for the size of n, below which more than 50% of the numbers would require more time if this alternative were used, is not trivial. It would require complicated combinatorial mathematics to decide, for numbers whose binary pattern was of a certain length, with a given number of bits set, how many suited long-division and how many, this method. Thus, this analysis is omitted. However, there are further points that can be made.

If only one bit were set in the binary pattern of the number in question, then a flag could be put to .TRUE., to indicate that the sums obtained did not need to be reduced modulo the relevant prime. Similarly, if there were only two 1's in the binary expansion under consideration, another flag could be set to show that the reduction of the sums could be done with only one subtraction (where needed), and that the longer division routine would be unnecessary. Two points follow from this.

(1) There will always be integers, no matter how small, and irrespective of how many bits are set in their binary patterns, for which this alternative method will be quicker. As was demonstrated above, when considering numbers between 2^{69} and 2^{70}, this improvement can be quite considerable - at times, in excess of 10-fold.

(2) Waiting until all the additions were finished before reducing the sums modulo the corresponding prime will, for many integers, prove slower than reducing the sums, where necessary, after each addition. Of course, if an extra subtraction was needed every time a 1 was encountered in the binary pattern of the number being factored,

then this reduction of the running sums could prove more time-consuming than a final division. It would be interesting to know, but again require complicated mathematics to find out, how likely this occurrence is. It may be that it would always be quicker to reduce the sums where necessary after each addition, thus making the latter even more favourable.

Two factors were not taken into account in the above analysis, namely, the time required to decide whether to add or subtract or do nothing for each bit of the operand, and the amount of storage required. Since the former are only scalar logical tests, their contribution to the running time will be slight, when compared with the other operations involved in the test. The second factor is more significant, however. Since the largest prime involved is 38,873, ($> 2^{15}$), at most 16 bits of storage will be required to hold each matrix element. For a DAP-FORTRAN implementation, this necessitates the use of INTEGER*3 matrices, but an APAL version could be more efficient. Even so, to store all these residues will require a large amount of memory. But this is the price that has to be paid for the increased speed. One could certainly re-compute the residues as they were required, and hence only store one of the matrices at any given moment. However, if this technique was used in conjunction with the Continued Fraction or Quadratic Sieve algorithms, then there would probably be millions of integers which had to be factored over the factor base, and so the time required for re-computation would be considerable, (As can be seen from above, it took 98 millisecs of DAP-time to generate the required matrices of the residues of the powers of 2 up to and including 2^{73}.) and possibly, depending on the length of the binary patterns involved, outweigh the advantage in using this method. However, it is very unlikely that there would not be enough memory available in the 16K DAP at QMC to store all the matrices. The amount of source code for the rest of one's program would need to be very large for there not to be enough space for these residues. For example, all the routines for Step 1 of the Lehman algorithm along with 73 matrices of residues could be accommodated in only a 4K DAP (since at times, the DAP at QMC processes two streams of jobs - one for a 12K (or 6 Mbyte) DAP, and the other with programs for a 4K (or 2 Mbyte) machine). It must also be noted that the storage requirements do not grow rapidly (unlike that required for storing all the primes $\leqslant N$, which grows as the size of N), but increase with the length of the numbers to be factored.

Thus we have here a technique which seems very promising, as it provides the possibility of speeding up trial division by more than 10-fold. While it could be used in connection with the block of primes in Step 1 of the Lehman algorithm, it should prove even more useful in conjunction with the Continued Fraction and Quadratic Sieve algorithms, in both of which, up to millions of numbers need to be factored over a known

factor base of primes. Therefore it is certainly worth further consideration and, in particular, an implementation in the DAP's assembly language, APAL.

Chapter 9 Primes and Primality Testing

In Chapter 3, during the general discussion of the Lehman algorithm, it was described how, since small factors are more common than large ones, it would be wise to "make the most of" the first matrix of divisors, and use all primes. Hence the serial version of the algorithm also began by trying the first 4096 primes before generating subsequent trial divisors via residues modulo 30. However, it was not stated where the primes would come from, and so we start this chapter by briefly describing how they were generated.

One of the best known methods for generating a list of consecutive primes on a serial machine, is the sieve of Eratosthenes, which Wunderlich [1967] describes as follows:

"We let $A^{(1)}$ be the sequence of positive integers greater than 1 and store a finite portion of this sequence in the main memory of the machine. $A^{(2)}$ is obtained by eliminating all integral multiples of 2, the first element of $A^{(1)}$. $A^{(3)}$ is obtained by eliminating all integral multiples of 3, the second element of $A^{(2)}$. In general, $A^{(k+1)}$ is obtained from $A^{(k)}$ by eliminating all integral multiples of p_k, which is the kth element of $A^{(k)}$. The eliminations are executed within two nested DO loops by storing zero in the appropriate words. When the kth element of $A^{(k)}$ exceeds $\sqrt{x}$, the surviving sequence is the set of all primes less than x."

Hence, no division is required to pick out the composite members of the sequence. Instead, these are identified by their position in the list. Thus, it is not necessary to store the sequences $A^{(i)}$ at all. Rather, we could use a logical array of the same length with each element initialised to .TRUE.. A representation of $A^{(2)}$ could be obtained by setting every second location to .FALSE.. Then setting every third location to .FALSE. (except element 3) will produce $A^{(3)}$, and so on. The required primes are then just the indices of the locations in the array which still hold the value .TRUE..

As Wunderlich points out, the advantage of using such a sieving procedure (which is normally reducing the computing time by a factor of 100) has to be set against the main disadvantage, namely, the large amount of storage that would normally be required. Of course, representing the sequence of integers as a "packed array" of type "boolean" in Pascal would overcome this problem.

However, since we only require to generate this sequence once, it is not vital that the most efficient algorithm is used, and so a more naive method was employed, as we will now

show.

We start with, say, the first 4 primes, and then "guess" what the next one will be. If the number picked is composite, then it will be divisible by at least one of the primes already found (unless it is so large that it is the product of primes not already found - a possibility that is, needless to say, avoided by a "sensible" choice for the next "possible prime"). Thus, all one has to do, to check if the number is prime, is to divide it by all the primes so far obtained. Of course, this would be more work than is necessary, since the prime factors of a composite integer are all less than or equal to that integer's square root. Hence, one need only check for divisors amongst this smaller set of primes.

We have already discussed how to generate a sequence containing all the primes, and as few composite numbers as possible. But, since for our purposes, the generation of this list of primes is a "one off" exercise, it does not matter if our program is not the most efficient that could have been written. Therefore, because of its simplicity, the sequence formed by the terms: $6k + 1$ and $6k - 1$ ($k \geqslant 1$) was used as a source for the next "guess", (subsequent terms of the sequence are obtained by adding alternately 2 or 4 to the previous one). It took the VAX $15 \cdot 5$ seconds of cpu time to produce the required list. When exactly the same algorithm was written in DAP-FORTRAN, and run on the DAP, (thereby ignoring its potential for parallelism, and treating the DAP as an SISD machine), the time taken was $49 \cdot 7$ seconds! This suggests that the master control unit, which performs scalar operations in addition to decoding instructions and broadcasting data, is not as quick a machine (at least for a program which involves mainly fixed point arithmetic) as a VAX 11/780.

Another factor which contributed to the mcu's poor performance was that the primes, as they were generated, were not held in a conventional (software) one-dimensional array, but were stored as the elements of a matrix, with one prime located in each processor. Thus, to divide the next "guess" by a prime already found, involved picking the latter out from the matrix and transferring it to the mcu. A natural improvement to the method was to replace the repeated transferring of primes to the mcu with transferring each "guess" to the processor array, and performing all the divisions simultaneously. As the next prime was found, it was stored in the next available location in the matrix of primes (considered for this purpose, as a long vector). Matrix elements which had not yet been assigned a prime were initialised to the value 2.

Following the division of the next member of the sequence, a LOGICAL mask was used to indicate the processors in which no remainder had been produced. Thus it was important for those processors which had not yet been allocated a prime, to be unable to affect the outcome of a division by claiming not to have a remainder when the dividend had, in fact, been a prime. Rather than using a second LOGICAL matrix to mark out those processors

which were to be ignored (an element of which would then have to be changed whenever a prime was found), these PE's were assigned the value 2, since all the members of the sequence $6k + 1, 6k - 1$ $(k \geqslant 1)$ are odd.

The time the DAP required for this method was just over $11 \cdot 5$ seconds. This was certainly an improvement on the previous time, but not much quicker than the serial machine. In an effort to discover why the DAP had not been even quicker, instructions were added to the program to count how many times a mask was tested, how many scalar instructions there were, and how many times the processor array was used for calculations, the assigning of a mask, or the assigning of an integer matrix. The results were as follows:

Tests of masks	12,955
Scalar instructions	21,140
4096 PE's used	38,866
1 PE used	4095
	77,056

The fourth row refers to the storing of each successive prime. The number recorded is 4095 and not 4096, since the first prime, 2, was broadcast to all the processors, and so this instruction is counted in the previous row of the table. Thus one can see that only just over half of the instructions made use of the SIMD capability of the DAP. The other instructions, for the most part, were either scalar operations or the logical combining of matrix elements. When one considers that out of the 38,866 matrix operations performed, many of the processors were doing a needless task (i.e. dividing by 2), it is clear why the DAP's time was so poor in comparison with the VAX.

The sad fact is, though, that apart from minor modifications (e.g. using a vector to store and divide by the first 64 primes), there does not seem to be a more suitable method for generating consecutive primes on the DAP. The Sieve of Eratosthenes would not suit the DAP's architecture for, unlike the CRAY-1 which has a stride facility enabling the "picking out" of those locations of a vector which are a certain distance apart (e.g. every third one), accessing matrix elements which are not a power of 2 apart is not easily possible. The mask that one would have to form to mark every fifth, say, element in the first matrix of possible primes would not suit the second matrix, but would need to be shifted one place to the left (considered as a long vector). Certainly, this would not take very long, but the situation would not be so simple in the case of the prime 647 $(4096 = 6 \times 647 + 214)$ which would

require shifting 214 places left as a long vector, or 3 cyclic shifts west of the matrix and then 22 shifts left as a long vector, and would get even worse when considering a prime, like 4231, which is larger than the number of elements in one matrix.

Thus it would appear that, until a more ingenious method can be developed (if possible) that suits its architecture, if one requires the generation of tables of consecutive primes very quickly, then the DAP is not the machine to use.

However, as will now be shown, the DAP is very suited to a related problem which is more complicated than the above, namely, identifying whether a given integer is prime or composite.

When generating primes, the next one was recognised by proving that it had no factors (> 1) less than or equal to its square root, and this method is adequate for the small numbers being dealt with. However, to test a 50-digit decimal integer for primality by simple trial division would not be wise simply because of the time it would take! Indeed, using any factoring algorithm to identify large primes would be extremely wasteful, for it would involve doing far more work than is actually necessary. As Lenstra [1982] remarks:

> "To the uninitiated reader it may seem surprising that it is possible to prove that a number is composite, without the proof yielding a factorisation."

but if it were not true, then the advances that have been made with primality testing, would not have been possible.

The main result, to date, in this field is that of Adleman, Pomerance and Rumely [1983], a version of whose test has been implemented by H. Cohen on the CDC-Cyber 170-750 computer of the SARA Computer Centre in Amsterdam. Lenstra reports how, at the time of his writing, it was the only primality test in existence that could routinely handle numbers of up to 100 decimal digits; doing so within approximately 45 seconds! (According to Dixon [1984], the "most recent report is that this program can routinely handle proofs of primality for primes up to 200 digits in less than 10 minutes".) However, the algorithm involved is extremely complicated, and an investigation of its potential for a parallel implementation is a non-trivial problem. In view of the difficulties associated with putting a new algorithm on a machine like the DAP, mentioned earlier, it was considered more suitable for the present piece of research, to investigate other, simpler, algorithms. The test that has been implemented is capable of proving that a given integer is composite, but can only indicate that a number is probably prime. However, the algorithm is such that, as we will now describe, the probability of drawing the wrong conclusion is so small that, if it did happen, then this would be of much more interest to statisticians and number theorists than being wrong would be disappointing.

The idea which underlies the test follows from Fermat's Theorem, namely, that if p is a prime, and x is not a multiple of p, then

$$x^{p-1} \equiv 1 \ (mod\ p).$$

Knuth [1982] describes how this can be used to show the non-primality of a given integer as follows.

> "When n is not a prime, it is always possible to find a value of $x < n$ such that $x^{n-1}\ mod\ n \neq 1$; experience shows that, in fact, such a value can almost always be found very quickly."

However, there are certain composite integers, called Carmichael numbers, for which $x^{n-1} \equiv 1 \ (mod\ n)$ for every $x \in Z$ coprime to n. But, as Knuth points out, it can be shown that if n is a Carmichael number, then it has a factor less than $n^{1/3}$.

Thus, this test could be of use between Steps 1 and 2 of the Lehman algorithm, because by that time it is known that what is left (N, say) of the number to be factored, is either prime or the product of only two primes. If we were to pick at random several x's and find for each of them, that $x^{N-1} \equiv 1 \ (mod\ N)$, then this would suggest that N were prime, and hence the iterations of Step 2 would be unnecessary.

But, how sure could we be that we were right? If we had chosen an x such that $x^{N-1} \not\equiv 1 \ (mod\ N)$, then we could be certain that N was composite. But, just because all the x's tried did satisfy the congruence does not imply that N must be prime. We could have been "unlucky" in those numbers picked, or had not chosen enough.

The following variation is better, because one can place a numerical bound on the probability of N behaving like a prime when it is not. Knuth states the algorithm as follows:

Let $n = 1 + 2^k q$, where q is odd.

P1 [Generate x.] Let x be a random integer in the range $1 < x < n$.

P2 [Exponentiate.] Set $j \leftarrow 0$ and $y \leftarrow x^q\ mod\ n$.

P3 [Done?] (Now $y = x^{2^j q}\ mod\ n$.) If $j = 0$ and $y = 1$, or if $y = n - 1$, terminate the algorithm and say "n is probably prime". If $j > 0$ and $y = 1$, go to step P5.

P4 [Increase j.] Increase j by 1. If $j < k$, set $y \leftarrow y^2\ mod\ n$ and return to step P3.

P5 [Not prime.] Terminate the algorithm and say that "n is definitely not prime."

The motivation for this test is that, if $n = 1 + 2^k q$ is prime then, by Fermat's Theorem, $x^{n-1} = x^{2^k q}$ will be congruent to 1 modulo n. Now, having

$$y^2 = 1 \ (mod\ p)$$

is equivalent to saying that

$$p \mid (y-1)(y+1)$$

which implies, if p is a prime, that

$$y \mid (p-1) \quad \text{or} \quad y \mid (p+1)$$

In other words, y must be congruent to 1 or $(p-1)$ modulo p (since $p-1 \equiv -1 \,(mod\ p)$). So, if p is a prime, the sequence

$$x^{q}\ mod\ p\ ,\quad x^{2q}\ mod\ p\ ,\quad x^{4q}\ mod\ p\ ,\quad \cdots\ ,\quad x^{2^k q}\ mod\ p$$

will end with 1, and as we have just proved above, the previous member of the sequence will be either 1 or $p-1$. Hence the tests of the algorithm.

However, just because $x^{2^i q}\ mod\ n$ equals 1 for some i does not mean that n must be prime. If there had been no i with $1 \leqslant i \leqslant k$, for which the above equality held, then we could be certain that n was composite. But, as with the previous test, the primality of the number concerned cannot be rigorously proved. The advantage with this test over the other is that one can put a bound on the probability of being wrong. Knuth shows how to obtain the value $\frac{1}{4}$; a useful result since this value is independent of the size of n. He goes on to describe how this algorithm (which he calls Algorithm P) can be used as the basis of a primality test which is almost completely reliable.

"Suppose we invoke Algorithm P repeatedly, choosing x independently and at random whenever we get to step P1. If the algorithm ever reports that n is nonprime, we can say that n definitely is not prime. But if the algorithm reports 25 times in a row that n is "probably prime", we can say that n is "almost surely prime." For the probability is less than $(1/4)^{25}$ that such a 25-times-in-a-row procedure gives the wrong information about n. This is less than one chance in a quadrillion; even if we certified a billion different primes with such a procedure, the expected number of mistakes would be less than $\frac{1}{1000000}$. It is much more likely that our computer has dropped a bit in its calculations, due to hardware malfunctions or cosmic radiations, than the Algorithm P has repeatedly guessed wrong!"

This algorithm is very suitable for implementation on the DAP for several reasons:

(1) The work associated with each of the values is independent of that for all the other random numbers. Thus, a vector could be used to perform the test simultaneously for 64 random numbers. The probability of being wrong would then be $(1/4)^{64}$. If this were not small enough, then matrices could be used instead to give a probability of making a wrong judgment, equal to $(1/4)^{4096}$!

(2) Much time and effort has already been spent developing quick and reliable random number generators for the DAP and so there would be no difficulty in obtaining a vector or matrix of x values.

But, the main disadvantage in using the DAP is that one is forced to do far more work than is really necessary. For a serial implementation, 25 repetitions of the algorithm with different values, would give an acceptable probability of error. One would certainly never think of performing the algorithm 4096 times! However, this is just one of the drawbacks in using a large array of bit processors.

Certainly one could say that an event with a probability of happening equal to $\frac{1}{4^{25}}$ was more likely than an event with a probability of $\frac{1}{4^{4096}}$ associated with it, but in practice, neither would be expected. But, if the latter did happen, then the disappointment in deciding, wrongly, than an integer was prime would be considerably outweighed by the interest raised by the occurrence of such an unlikely event.

This algorithm has been implemented in DAP-FORTRAN, but because no multiple-precision long-division procedure has been written (which would be required to reduce an integer modulo n, if n were large), the program just manipulates integers $< 2^{64}$, and so can only test integers less than $< 2^{32}$. However, this test is certainly worth implementing for multiple-precision integers, once a long-division procedure is available, since, as can be seen from below, the algorithm is "good" in the sense that its running time is bounded by a polynomial in the length of the integer to be tested.

It is well known that by a process of repeated squaring (modulo n), the exponential $x^q \ (mod\ n)$ can be calculated in $O(\log q)$ steps. What one does is to produce the sequence

$$x\,,\quad x^2(mod\ n),\quad x^4(mod\ n),\quad x^8(mod\ n)\ \ etc.$$

and form the product of those $x^{2^i}\ (mod\ n)$ for which $\alpha_i = 1$ in the binary representation $\sum_{i=0}^{[log_2 q]} \alpha_i 2^i$ of q.

Since at every stage, the value to be squared will be $< n$, at most $(log_2 n)^2$ bit operations will be required for each multiplication. The result will be $< n^2$, and so have $\leqslant 2\log_2 n$ bits. Hence, the greatest number of bit operations required for the reduction modulo n will again be $(log_2 n)^2$. Thus, the squaring involved in the initial exponential calculation will require at most

$$2\,(log_2 n)^2 . \log_2 q$$

bit operations.

By a similar argument, each multiplication needed to form the final product will require at most $2(log_2 n)^2$ bit operations. Since the number of 1's in the binary pattern of q cannot exceed $\log_2 q + 1$, at most $\log_2 q$ multiplications will be required (no multiplication is needed to take account of the least significant bit of q being set), and the number of bit operations required to calculate the exponential will be bounded above by

$$4\,(log_2 n)^2 . \log_2 q \ .$$

At this point it may be possible to terminate the algorithm. If not, then the process of squaring and reduction modulo n must begin, each iteration of which will require, as shown above, $2(log_2 n)^2$ bit operations. The worst case will be when $x^{2^k q}\ (mod\ n)$ must be formed, thus involving a total of $2k\,(log_2 n)^2$ bit operations for the second stage.

Hence, the whole algorithm requires at most

$$2\,(log_2 n)^2 (\,2\,(log_2 q) + k\,)$$

bit operations. But, since $n - 1 = 2^k q$, we have that $k + \log_2 q \leqslant \log_2 n$, and so the upper bound on the work involved in the test is

$$2\,(log_2 n)^3 + 2\,(log_2 n)^2 . \log_2 q \quad \leqslant \quad 4\,(log_2 n)^3$$

bit operations.

Thus, the running time of the algorithm is indeed bounded by a polynomial in the length of the integer in question. This could make this test theoretically quicker even for a serial machine than the one implemented by Cohen mentioned above. (It depends on the constant implied in saying that Cohen's test is $O((log\ n)^{c \log\log\log n})$, where c is another constant.) However, the price we have chosen to pay is the possibility of deciding that a composite number is prime. But, when one takes into account that the probability of this happening is $\leqslant (1/\ 4)^{4096}$, then it would appear not to have been such a high price to pay.

Chapter 10 Conclusions

The Lehman Algorithm

We have shown how this algorithm is indeed suitable for a parallel implementation. However, while the flexibility of an MIMD system is not, for the most part, required, the SIMD structure of the DAP forced us to accept some inefficiences. For example, wishing to re-divide with one divisor in a given block necessitates re-division by the whole block, since to construct a new matrix of divisors would be time-consuming. Another example was seen in connection with performing the d loops. Because the inner loop limits vary inversely with the outer loop counter, we had to do far more work than was necessary in the d loops corresponding to very small values of k.

On the mathematical side, in addition to refining the running time bound for Step 2, it has been demonstrated how, for numbers of certain forms, the Lehman algorithm can find the factors quickly, sometimes "instantly". It has also been shown that extra knowledge about the relative size of the factors of N can, in some cases, be used to reduce the running time. We have noted too, how the rigidity of the DAP can sometimes allow us to take advantage of the form of a number without knowing about it!

However, there are certain points worth further investigation.

(1) If it is possible to know, in general, the probability distribution of the (k, d) pairs (i.e. which k values or (k, d) pairs are more likely), this could suggest better ways of performing the search of Step 2.

(2) In the algorithm, several pieces of information are ignored. For instance, the primality of p and q, the factors of N, is never used (all we use is that p and q lie within a certain range, and that after a perfect square test we can say that $p \neq q$). Also, the search for (k, d) pairs seems to be on the basis of "pot luck". The fact that we know values of k and d which did not work, is not made use of when selecting the next pair to try. If a way could be found to make use of this knowledge, then it might be possible to develop a much more efficient deterministic factoring algorithm.

The DAP

When one reads the historical survey of integer factorization in [Brillhart et al 1983], it becomes clear that there are two facilities which make a computer suitable for number theoretic calculations :

(i) the returning of a double-length result from a multiplication; and

(ii) the provision of a multiple-precision arithmetic package.

To a certain extent, the DAP does provide the first. However, a 64-bit result can only be obtained in DAP-FORTRAN if the operands are also of that length. Unfortunately, when one multiplies together two 32-bit integer variables, and the answer is larger than $2^{31} - 1$, INTEGER overflow occurs, even if the variable on the left-hand side , which is to store the result, is of 64-bit precision. What would be a useful alternative is for the precision of the result of a calculation to be decided by the length of the variable on the left-hand side of the assignment statement, rather than being governed by the precision of the operands on the right.

It is unfortunate that the second requirement is not provided, and that the writing of any multiple-precision routines required is left to the user. It is also disappointing that much of the advantage that should be gained from the DAP's flexibility in handling large integers is lost when DAP-FORTRAN is used. It is unlikely that many mathematicians, or DAP users in general, will want to learn APAL, the DAP's assembly language, in order to avoid these problems. Thus, in addition to (or perhaps instead of) the suggestion above, two things are required.

(a) a multiple-precision arithmetic package, which could be even more useful if it included routines for finding roots of polynomials, (especially square and cube roots), and for calculating the quadratic residue (necessary for solving quadratic congruences - a task required in the Quadratic Sieve algorithm) and Jacobi symbols ; and

(b) a high-level assembly language, like C , so that users could have access to the flexibility of the bit-manipulating capability of the machine, without the overhead of array indexing (as at present, when using LOGICAL arrays in DAP-FORTRAN).

The suitabilty of the DAP for variable-precision arithmetic is due to it being an array of single bit processors. Of course, the disadvantage with this design is that floating-point operations are relatively time-consuming. But, since number theory calculations (especially primality testing and integer factorization) typically involve only integer arithmetic, this factor does not affect such work. However, improvements could be made concerning the time that multiplication takes (see times in Chapter 6), and it would seem to have been a wise decision to make, to (presumably) use more powerful bit processors in the new version

of the DAP (which has a cycle time of 150 nanoseconds, rather than 200).

Perhaps the one drawback with the new machine is that the array size has been reduced to 1024 processors. As we showed in Chapter 6, this will, as would have been expected, result in it taking four times as long as the 4096 DAP to perform the same task. But, the advantage in having a smaller array, from the point of view of the Lehman algorithm, is that the inefficiencies mentioned at the start, will not be so great in relation to the total amount of work required. Although, it is a tribute to the power of the original machine that, despite these inefficiencies, some very high speed-up ratios were obtained. Of course, it must be added that the new, smaller DAP, at one fiftieth of the price, but less than a quarter of the power, is much better value than the old one!

A further piece of work, whose outcome would be interesting, is the re-implementation of the programs described in this thesis, in APAL, so that, as well as using the parallelism of the DAP (as has been done), the bit-processing flexibility of the machine can be exploited.

Appendix A Proof of the Assertion in Chapter 3

In the proof we asserted that there exist natural numbers r , s such that

$$rs < N^{1/3} \quad \text{and} \quad |pr - qs| < N^{1/3}.$$

This will now be proved.

Let $\frac{r_n}{s_n}$ denote the nth convergent of the continued fraction expansion of $\frac{q}{p}$. Now, $p > N^{1/3}$ and $q < N^{2/3}$, so that $\frac{q}{p} < N^{-1/3}.N^{2/3}$. Thus $r_1 = \left[\frac{q}{p}\right] < N^{1/3}$. Also, since $q \geqslant p$, we have that $\frac{q}{p} \geqslant 1$ and so $r_1 = \left[\frac{q}{p}\right] \geqslant 1$. Hence since $s_1 = 1$ we have that $0 < r_1 s_1 < N^{1/3}$. Now, $\{r_n\}, \{s_n\}$ are increasing sequences, and therefore so is $\{r_n s_n\}$. We have shown that $r_1 s_1 < N^{1/3}$. Since $\frac{q}{p}$ is rational, and $q \neq p$, the last convergent will be $\frac{q}{p}$, and $pq = N > N^{1/3}$, and so, we can find an m such that

(1) $r_m s_m < N^{1/3}$

(2) $r_{m+1} s_{m+1} > N^{1/3}$.

(Both inequalities are strict because $N = pq$ is not a cube.) Take $r = r_m$, $s = s_m$. Then, by (1), $rs < N^{1/3}$. So, it remains only to show that $|pr - qs| < N^{1/3}$. Now, for this choice of r and s , we have that

$$\left|\frac{r}{s} - \frac{q}{p}\right| \leqslant \left|\frac{r}{s} - \frac{r_{m+1}}{s_{m+1}}\right|$$

(since $\frac{r}{s}$ and $\frac{r_{m+1}}{s_{m+1}}$ are consecutive convergents to $\frac{q}{p}$)

$$= \left|\frac{r s_{m+1} - r_{m+1} s}{s s_{m+1}}\right|$$

$$= \frac{|r s_{m+1} - r_{m+1} s|}{s s_{m+1}}$$

$$= \frac{1}{s s_{m+1}}$$

(from the theory of continued fractions, if $\frac{r_m}{s_m}$ and $\frac{r_{m+1}}{s_{m+1}}$ are consecutive convergents to a

given real number, then $|r_m s_{m+1} - r_{m+1}s_m| = 1$), which implies that

$$|pr - qs| \leqslant \frac{ps}{ss_{m+1}} = \frac{p}{s_{m+1}}.$$

By considering the convergents of the continued fractions expansion of $\frac{p}{q}$, it can be shown in a similar manner that

$$|pr - qs| \leqslant \frac{q}{r_{m+1}}.$$

Hence

$$(pr - qs)^2 \leqslant \frac{pq}{r_{m+1}s_{m+1}} < \frac{N}{N^{1/3}} = N^{2/3}.$$

i.e.

$$|pr - qs| < N^{1/3},$$

as required.

Appendix B Using previous knowledge in Step 2

Once the value of $4(k+1)N$ had been found, the next task to be done was to calculate $[\sqrt{4(k+1)N}]$. This could have been done "from scratch" each time. However, since we already knew $[\sqrt{4kN}]$, it was natural to try and use this knowledge to save some time. We tried various schemes to do this, but, as will now be described, only one met with any success.

(i) The gap between k^2 and $(k+1)^2$ is equal to $2k+1$ (assuming $k > 0$); the gap between $(k+1)^2$ and $(k+2)^2$ is $2k+3$, and so on. Now, since $[\sqrt{4(k+1)N}]$ is equal to the number of perfect squares which are less than or equal to $4(k+1)N$, if we knew one of these perfect squares, say X^2, then we could repeatedly subtract $2X+1$, $2X+3$, ... etc., from $(4(k+1)N - X^2)$ = Gap, say, until Gap were $\leqslant 0$. Then, if Gap equals 0, $[\sqrt{4(k+1)N}]$ would equal the sum of X and the number of $(2X+i)$'s subtracted, whereas, if Gap < 0 the value of $[\sqrt{4(k+1)N}]$ would be one less than the previous sum.

Obviously we would want the X^2 chosen to be as large as possible, and this is where we tried to make use of our previous knowledge. $[\sqrt{4kN}]^2$ is a perfect square, whose square root is known, which is less than $4(k+1)N$, and as this is the largest such number known at this stage of the program, it could be used to "play the part of" X. The following algorithm could then perform the process:

```
Gap := 4(k+1)N - [√4kN]²;
[√4(k+1)N] := [√4kN];
SmallerGap := 2* [√4kN] + 1;
while Gap >= SmallerGap do
  begin
  Gap := Gap - SmallerGap;
  SmallerGap := SmallerGap + 2;
  [√4(k+1)N] := [√4(k+1)N] + 1
  end;
```

This method is certainly very simple, using for the most part, only addition and subtraction - operations which are quicker than the multiplications required by the Newton-Raphson algorithm, and it proved very satisfactory for small N. However, when it was tried with larger N, the gaps between successive values of $4kN$ were just too large. Subtracting the "Smaller Gap's" two or four at a time only provided a temporary improvement, and it was found that, long before N was as large as 2^{70}, this scheme became impractical.

(ii) Consider the following

$$\begin{aligned}\sqrt{4(k+1)N} &= \sqrt{4kN + 4N} \\ &= \sqrt{4kN}\left(1 + \frac{1}{k}\right)^{1/2} \\ &= \sqrt{4kN}\left(1 + \frac{1}{2k} - \frac{1}{8k^2} + \frac{1}{16k^3} - \frac{5}{128k^4} + \frac{7}{256k^5} - \cdots\right)\end{aligned}$$

by the Binomial Theorem. Now, since the series:

$$\left(1 + \frac{1}{k}\right)^{1/2} = 1 + \frac{1}{2k} - \frac{1}{8k^2} + \frac{1}{16k^3} - \frac{5}{128k^4} + \frac{7}{256k^5} - \cdots$$

is absolutely convergent for $k \geqslant 1$, we can rearrange the series in an infinite manner as shown :

$$\begin{aligned}\left(1 + \frac{1}{k}\right)^{1/2} &= 1 + \frac{1}{2k} - \left(\frac{1}{8k^2} - \frac{1}{16k^3}\right) - \left(\frac{5}{128k^4} - \frac{7}{256k^5}\right) - \cdots \\ &< 1 + \frac{1}{2k}\end{aligned}$$

since each of the brackets on the right hand side, contains a positive term. Hence,

$$\begin{aligned}\sqrt{4(k+1)N} &= \sqrt{4kN}\left(1 + \frac{1}{k}\right)^{1/2} \\ &< \sqrt{4kN}\left(1 + \frac{1}{2k}\right) \\ &= \sqrt{4kN} + \frac{N^{1/2}}{k^{1/2}} \\ &\leqslant \sqrt{4kN} + \frac{N^{1/2}}{[k^{1/2}]} \\ &< [\sqrt{4kN}] + 1 + \left[\frac{N^{1/2}}{[k^{1/2}]}\right] + 1 .\end{aligned}$$

Now, it can be shown that

$$\left[\frac{N^{1/2}}{[k^{1/2}]}\right] \leqslant \left[\frac{[N^{1/2}]}{[k^{1/2}]}\right] + 1$$

and so we have that

$$[\sqrt{4(k+1)N}\,] \leqslant \sqrt{4(k+1)N}$$

$$< [\sqrt{4kN}\,] + \left[\frac{[N^{1/2}]}{[k^{1/2}]}\right] + 3$$

Thus, if we know the values of $[N^{1/2}]$ and $[\sqrt{k}\,]$, we can use our knowledge of $[\sqrt{4kN}\,]$ to calculate an upper bound for $[\sqrt{4(k+1)N}\,]$. Obviously, we have to find the first value, $[\sqrt{4N}\,]$, from scratch, and this can be used to obtain $[\sqrt{N}\,]$, since

$$[\sqrt{N}\,] = [\sqrt{4N}\,]\ div\ 2\,.$$

Since the iteration variable, k, takes as its value each successive integer (up to some limit) starting at 1, in ascending order, a simple counting procedure can be used to determine when $[\sqrt{k}\,]$ should be increased by one (because, as mentioned before, if $X \geqslant 0$, then $(X+1)^2 - X^2 = 2X + 1$).

As with the previous idea, this method worked well for small N, but for larger N proved unsatisfactory. The problem is that, since we only used two terms of the above series in our working, the upper bound produced, for even relatively large N, is much greater than the actual value, and so more time is spent reducing our estimate than it would have taken to find $[\sqrt{4(k+1)N}\,]$ from scratch. Taking more of the terms in the series for $\left(1 + \frac{1}{k}\right)^{1/2}$ into account would not necessarily solve the problem, since the forming of these would require multiplications and divisions, and because, for very large N, so many terms would be needed before the bound was sufficiently close to the actual value, the work involved would be greater than that for the Newton-Raphson method.

(iii) The third way to take advantage of the knowledge of $[\sqrt{4kN}\,]$ which was tried, was to use it as the initial estimate for $[\sqrt{4(k+1)N}\,]$ when using the Newton-Raphson algorithm. While it may not be the most imaginative plan, it certainly proved to be effective.

Computing Science References

Anderson J P, Hoffman J A, Shifman J and Williams R J 1962

"D825 - A Multiple-Computer System for Command and Control" AFIPS Conference Proceedings, 1962, Vol. 22, pp. 86-96.

Anderson D W, Sparacio F J and Tomasulo R M 1967

"The IBM System/360 Model 91: machine philosophy and instruction handling" IBM J. Res. Dev., Vol. 11, pp. 8-24.

Barnes G H, Brown R M, Kato M, Kuck D J, Slotnick D L and Stokes R A 1968

The ILLIAC IV computer", IEEE Transactions on Computers, Vol. C-17, pp. 746-57.

Batcher K E 1974

"STARAN parallel processor system hardware", Proc. of the National Computer Conference, Vol. 43, pp. 405-410.

Batcher K E 1980

"Design of a Massively Parallel Processor", IEEE Transactions on Computers, Vol. C-29, No. 9, September 1980, pp. 1-9.

CACM 1984

"ACM Commemorates IEEE Centennial", news report in Communications of the ACM, August 1984, p. 851.

Clementi E, Corongiu G, Detrich J H and Khanmohammedbaiji H 1984

"Parallelism in computational chemistry: applications in quantum and statistical mechanics", paper given at Vector and Parallel Processors in Computational Science II, Oxford, 28-31 August 1984 (Proceedings to appear).

Comrie L J and Hartley H O 1939

"Modern machine calculation", translated and revised from original by H Sabielney, (Scientific Computing Service Ltd., London).

Dekel E and Sahni S 1981

"Binary Trees and Parallel Scheduling Algorithms", The University of Texas at Dallas Technical Report #95.

Dennis J B 1979

"The Varieties of Dataflow Computers", Proc. of the First International Conference on

Distributed Computing Systems, October 1979, pp. 430-439.

Enslow P H Jr 1974

"Multiprocessors and Parallel Processing", Comtre Corporation (John Wiley and Sons, New York).

Falk H 1976

"Reaching for the gigaflop", IEEE Spectrum, Vol. 13 (10), pp. 65-70.

Flanders P M, Hunt D J, Reddaway S F and Parkinson D 1977

"Efficient high speed computing with the distributed array processor", High Speed Computer and Algorithm Organization, ed. D J Kuck, D H Lawrie and A H Sameh, (Academic Press, Inc. (London) Ltd.), pp. 113-28.

Flynn M J 1966

"Very High Speed Computing Systems", Proceedings of the IEEE Vol. 54 pp. 1901-9.

Flynn M J 1972

"Some computer organisations and their effectiveness", IEEE Transactions on Computers, Vol. C-21, pp. 948-60.

Gostick R W 1979

"Software and algorithms for the Distributed-Array Processors", ICL Technical Journal, Vol. 1, Issue 2, May 1979, pp. 116-135.

Gregory J and McReynolds R 1963

"The SOLOMON computer", IEEE Transactions on Electronic Computers, Vol. EC-12, pp. 774-81.

Gurd J K 1984

"The Manchester dataflow machine", paper given at Vector and Parallel Processors in Computational Science II, Oxford, 28-31 August 1984 (Proceedings to appear).

Hockney R W and Jesshope C R 1981

"Parallel Computers", (Adam Hilger Ltd., Bristol).

Jones A K and Schwarz P 1980

"Experience Using Multiprocessor Systems - A Status Report", ACM Computing Surveys, Vol. 12, No 2, pp. 121-65.

Knuth D E 1981

"Seminumerical Algorithms", The Art of Computer Programming (Second Edition), Vol. 2, (Addison-Wesley Publishing Company, Reading, Mass.).

Kuck D J 1977

"A survey of parallel machine organisation and programming", ACM Computing

Surveys, Vol. 9, pp. 29-59.

Kuck D J 1980

"High-Speed Machines and Their Compilers", paper from CREST Conference on Parallel Processing, contained in "Parallel Processing Systems - An Advanced Course", ed. D J Evans, (CUP).

Kung H T

"Why Systolic Architectures?", IEEE Computer, January 1982, pp. 37-46.

Lundstrom S F and Barnes G H 1980

"A Controllable MIMD Architecture", Proceedings of the 1980 International Conference on Parallel Processing, pp. 19-27.

von Neumann J 1946

"Preliminary discussion of an electronic computing instrument", Collected Works of von Neumann, Vol. 5, pp. 34-79, (actually co-authored by A W Burks, H H Goldstine and J von Neumann).

von Neumann J 1949

"The Future of High-speed Computing" Collected Works of von Neumann, Vol. 5, p236.

von Neumann J 1954

"The NORC and Problems in High-speed Computing", Collected Works of von Neumann, Vol. 5, pp. 238-247.

Ohlsson L and Svensson B 1983

"Matrix Multiplication on LUCAS", Proceedings of the Sixth Symposium on Computer Arithmetic (IEEE Computer Society 1983), pp. 116-22.

Parkinson D 1980

"Practical Parallel Processors And Their Uses", paper from CREST Conference on Parallel Processing, contained in "Parallel Processing Systems - An Advanced Course", ed. D J Evans, (CUP).

Parkinson D and Wunderlich M 1984

"A compact algorithm for Gaussian elimination over GF(2) implemented on highly parallel computers", Parallel Computing, Vol. 1, Number 1, August 1984, pp. 65-73.

Robinson A L 1979

"Array Processors: Maxi Number Crunching for a Mini Price", SCIENCE, Vol. 203, 12 January 1979, pp. 156-160.

Russell R M 1978

"The CRAY-1 computer system", Communications of the ACM, Vol. 21, No. 1, pp. 63-72.

Siegel H J 1979

"Interconnection networks for SIMD machines", IEEE Computer, Vol. 12 (6), pp. 57-65.

Slotnick D L, Borck W C, and McReynolds R C 1962

"The SOLOMON computer", AFIPS Conf. Proc., Vol. 22, pp. 97-107.

Slotnick D L 1967

"Unconventional systems", AFIPS Conf. Proc., Vol. 30, pp. 477-81.

Snelling D F 1984

"Applications of the HEP", paper given at Vector and Parallel Processors in Computational Science II, Oxford, 28-31 August 1984 (Proceedings to appear).

Stone H S 1975

1975 "Introduction to Computer Architecture", ed. H S Stone (Science Research Associates, Inc.), Chapter 8, pp. 318-74.

Stokes R A 1977

"Burroughs Scientific Processor", High Speed Computer and Algorithm Organization, ed D J Kuck, D H Lawrie and A H Sameh, (Academic Press, Inc. (London) Ltd.), pp. 85-9.

Swan R J, Fuller S H and Siewiorek D P 1977

"Cm* - A Modular Multi-Microprocessor", Proceedings of the National Computer Conference, 1977, pp. 39-46.

Swan R J, Bechtolsheim A, Lai K-W and Ousterhout J K 1977

"The Implementation of the Cm* Multi-Microprocessor", Proceedings of the National Computer Conference, 1977, pp. 645-55.

Thornton J E 1964

"Parallel operation in the control data 6600", AFIPS Conf. Proc., Vol. 26 (part II), pp. 33-40.

Thurber K J and Wald L D 1975

"Associative and parallel processors", ACM Computing Surveys, Vol. 7, No. 4, pp. 215-55.

Unger S H 1958

"A computer oriented towards spatial problems", Proc. Inst. Radio Eng. (USA), Vol. 46, pp. 1744-50.

Watson W J 1972

"The TI ASC : A Highly Modular and Flexible Super Computer Architecture", AFIPS Proceedings FJCC, pp. 221-228, reproduced on pp. 753-762 of "Computer Structures : Principles and Examples", by D P Siewiorek, C G Bell and A Newell (McGraw-Hill Book Company, New York, 1982).

Widdoes L C Jr and Correl S 1979

"The S-1 Project: Developing High-Performance Digital Computers", Energy and Technology Review, September 1979.

Wilkinson J H 1953

"The pilot ACE", Computer Structures: Readings and Examples Chapter 11, ed. C G Bell and A Newell (New York: McGraw-Hill) pp. 193-9.

Wulf W A and Bell C G 1972

"C.mmp - A Multi-mini-microprocessor", AFIPS Conference Proceedings vol.14 part II, FJCC 1972, pp. 765-77.

Yau S S and Fung H S 1977

"Associative processor architecture - a survey", ACM Computing Surveys, Vol. 9, No. 1, pp. 3-27.

Mathematics References

Adleman L M, Pomerance C and Rumely R S 1983

"On distinguishing prime numbers from composite numbers", Annals of Mathematics, 117 (1983), pp. 173-206.

Brillhart J et al 1983

"Factorizations of $b^n \pm 1$", Contemporary Mathematics, Vol. 22, (American Mathematical Society, Providence, Rhode Island).

Dixon J D 1984

"factorization and Primality Tests", The American Mathematical Monthly, Vol. 91, No. 6, June-July 1984, pp. 333-352.

Gerver J L 1983

"Factoring Large Numbers With a Quadratic Sieve", Mathematics of Computation, Vol. 41, No. 163, July 1983, pp. 287-294.

Lehman R S 1974

"Factoring Large Integers", Mathematics of Computation, Vol. 28, pp. 329-336.

Lehmer D H 1933

"A Photo-Electric Number Sieve", American Math. Monthly, Vol. 40, pp. 401-406.

Kolata G 1983

"Factoring Gets Easier", SCIENCE, Vol. 222, 2 December 1983, pp. 999-1001.

Lenstra H W Jr 1982

"Primality Testing" pp. 55-77 of "Computational Methods in Number Theory", Part 1, ed. H W Lenstra Jr and R Tijdeman, Mathematical Centre Tracts, No. 154, (Amsterdam).

Morrison M A and Brillhart J 1975

"A Method of Factoring and the Factorization of F_7" Mathematics of Computation, Vol. 29, No. 129, January 1975, pp. 183-205.

Pomerance C 1982

"Analysis and Comparison of Some Integer Factoring Algorithms", pp. 89-139 of "Computational Methods in Number Theory", Part 1, ed. H W Lenstra Jr and R Tigdeman, Mathematical Centre Tracts, No. 154, (Amsterdam).

Rivest R L, Shamir A and Adleman L 1978

"A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM, Vol. 21, No. 2, pp. 120-6.

Voorhoeve M 1982

"Factorization Algorithms of Exponential Order" pp. 79-87 of "Computational Methods in Number Theory", Part 1, ed. H W Lenstra Jr and R Tigdeman, Mathematical Centre Tracts, No. 154, (Amsterdam).

Wunderlich M C 1967

"Sieving Procedures on a Digital Computer", Journal of the ACM, Vol. 14, No. 1, January 1967, pp. 10-19.