



The Australian National University

**TR-CS-95-01**

**Factorisation of Large Integers on  
some Vector and Parallel Computers**

**Craig Eldershaw and Richard P. Brent**

**January 1995**

Joint Computer Science Technical Report Series

Department of Computer Science  
Faculty of Engineering and Information Technology

Computer Sciences Laboratory  
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports  
Department of Computer Science  
Faculty of Engineering and Information Technology  
The Australian National University  
Canberra ACT 0200  
Australia

or send email to:

`techreports@cs.anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

`file://dcssoft.anu.edu.au/pub/www/dcs/techreports/`

**Recent reports in this series:**

- TR-CS-94-10 John N. Zigman and Peiyi Tang. *Implementing global address space in distributed local memories*. October 1994.
- TR-CS-94-09 Nianshu Gao and Peiyi Tang. *Vectorization using reversible data dependencies*. October 1994.
- TR-CS-94-08 David Sitsky. *Implementation of MPI on the Fujitsu AP1000: Technical details release 1.1*. September 1994.
- TR-CS-94-07 Markus Hegland. *Real and complex fast fourier transformations on the Fujitsu VPP500*. June 1994.
- TR-CS-94-06 Richard P. Brent. *Uses of randomness in computation*. June 1994.
- TR-CS-94-05 Trevor Vickers and Paul Gardner. *A language of refinements*. May 1994.

# Factorization of Large Integers on some Vector and Parallel Computers\*

Craig Eldershaw  
Mathematics Department  
University of Queensland  
St Lucia, Queensland 4072  
cs324391@student.uq.edu.au

Richard P. Brent  
Computer Sciences Laboratory  
Australian National University  
Canberra, ACT 0200  
rpb@cslab.anu.edu.au

Report TR-CS-95-01  
31 January 1995

## Abstract

We compare implementations of two integer factorization algorithms, the elliptic curve method (ECM) and a variant of the Pollard “rho” method, on three machines with parallel and/or vector architectures. ECM is scalable and well suited for both vector and parallel architectures. The “rho” method is simpler than ECM but is not scalable.

## 1 Introduction

The factorization of large integers is a significant mathematical problem with practical applications to public-key cryptography [16]. Although the theoretical complexity of factorization is unknown, it is a computationally expensive task with the best known algorithms. The development of new algorithms and faster machines has made the factorization of “general” integers with 100–120 digits feasible.

Several authors have considered vector and parallel implementations of the MPQS and NFS algorithms [5, 8, 10, 14, 15]. These algorithms have the property that the run-time depends mainly on the size of the number  $N$  to be factored. For another class of algorithms the run-time depends mainly on the size of the factor found. This class includes Lenstra’s “elliptic curve method” (ECM) [11] and Pollard’s “rho” method [13], which are considered in this paper.

We have implemented variants of ECM and Pollard “rho” on three computers with different architectures –

- The Fujitsu AP1000, which is a parallel machine with up to 1024 processors [9]. Each processor is a 25MHz RISC microprocessor with 16MB of memory. The processors are connected by a torus with wormhole routing. Each processor has a floating-point unit with a peak speed of 5.6 Mflop in double-precision. Our machine has 128 processors, so its peak speed is about 0.7 Gflop.
- The Fujitsu VP2200/10, which is a vector processor with a peak speed of 1.25 Gflop [17].
- The Fujitsu VPP500, which is a parallel machine with up to 224 vector processors connected by a crossbar network [6]. Each processor is similar to the VP2200/10 and has a peak speed of 1.6 Gflop. The machine available to us had 4 processors and peak speed 6.4 Gflop.

---

\*Copyright © 1995, the authors. To appear in *Proceedings of Neural, Parallel and Scientific Computations* (edited by M. Sambandham), Vol. 1, 1995. rpb156tr typeset using L<sup>A</sup>T<sub>E</sub>X

In the following Sections we describe the implementation of ECM and Pollard “rho” on the AP1000, VP2200 and VPP500, and compare their performance. Many examples of successful factorizations may be found in [4, 5, 7], so here we concentrate on vectorization and parallelization aspects of the implementations.

## 2 The Elliptic Curve Method

The elliptic curve method (usually abbreviated ECM) was proposed by Lenstra [11]. Practical improvements, such as the addition of a second phase, were suggested by Brent [3], Montgomery [12] and others. Each “trial” of the algorithm depends on a random seed and has a positive (but generally small) probability of finding a factor  $f$ . Because many independent trials can be performed in parallel, ECM is obviously amenable to a parallel implementation. The speedup is expected to be proportional to the number of processors provided  $f$  is not too small.

A vectorized implementation of ECM on the Fujitsu VP100 was written in 1988. The language used was a dialect of Fortran (close to Fortran 77 with directives for vectorization). An improved version was implemented in 1991 on the Fujitsu VP2200. Because the Fujitsu vector processors are designed for fast floating-point arithmetic, the inner loop uses 64-bit floating-point multiply, add, INT and DFLOAT operations (for details see [5]). The base  $\beta = 2^{26}$  of the multiple-precision number representation is chosen so that integers up to  $\beta^2$  can be represented exactly in floating-point format. Operations which are not critical to performance, such as input and output, are done with the MP package [1], which is convenient but slow because it was not written with vectorization in mind.

In 1994 Eldershaw modified Brent’s VP2200 implementation of ECM to obtain AP1000 and VPP500 implementations. The modifications were along the lines suggested in [4]. On the AP1000 each processor performs one or more independent trials (without vectorization) and reports back to the host processor if a factor is found. On the VPP500 it is important for each processor to perform several trials, since the vector length of inner-loop operations is proportional to the number of trials per processor.

In more detail:  $P$  blocks of  $R$  trials (phase 1) are carried out simultaneously on  $P$  processors. The trials within each block have consecutive seed values, and the first seed value for each block is  $R$  larger than that for the previous block. In effect  $R \times P$  trials are being carried out with consecutive seeds. At the end of the block of  $R$  trials (phase 1), each processor checks if a factor has been found. If not, each processor carries out phase 2 of the algorithm for each of its  $R$  trials using the corresponding first phase results.

In [4, 5] Brent gave the theoretical expected run-time  $T_P$  for a machine with  $P$  processors:

$$T_P = T_1/P + O(T_1^{1/2+\epsilon}) \quad (1)$$

Tests were run on the AP1000 with varying numbers of processors (powers of 2 from  $2^0$  to  $2^7$ ). The results confirmed (1). Typical results are shown in Figure 1. Linear regression shows that there is less than 3% error in the gradient of a linear fit to the data points. We can say that ECM is scalable, meaning that the speedup for sufficiently large problems on a parallel machine with  $P$  processors is proportional to  $P$ .

The VP2200 and VPP500 programs were very effectively vectorized – in a typical run, at least 90% of the overall time was spent using the vector unit. The AP1000 ran only in scalar mode, but the multiple processors reduced the time by two orders of magnitude (as predicted in (1)) in comparison to a single processor run.

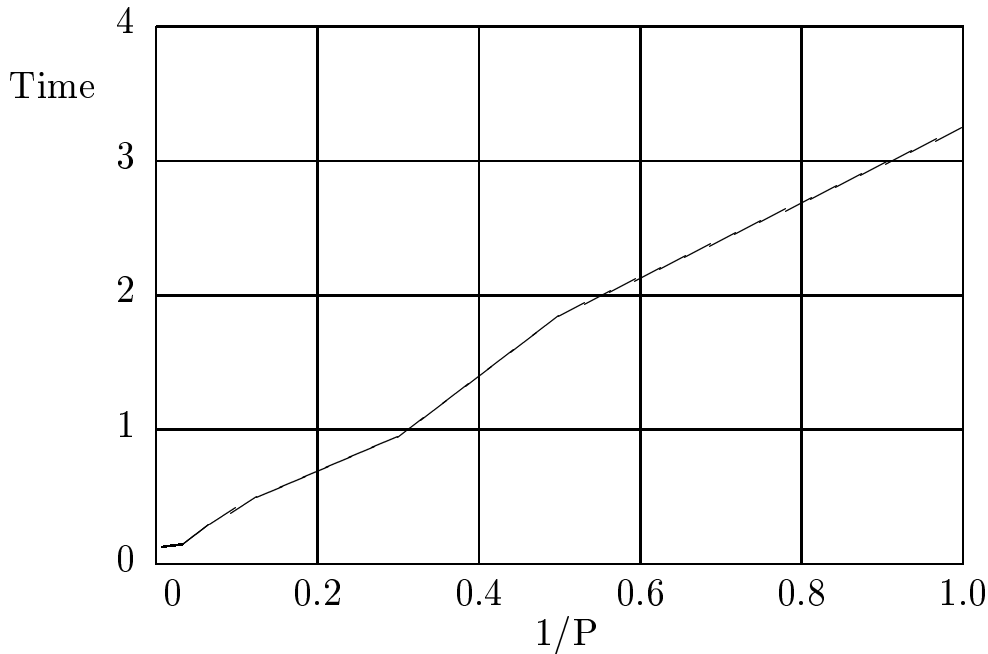


Figure 1: Time  $T_P$  (arbitrary units) versus  $1/P$  for ECM on AP1000

To give one example of the performance of the programs: a test run found a 32-digit factor

12567880628356583361572166052961

of a 79-digit number ( $98^{85} + 1$  divided by known factors) in 1555 seconds on the first attempt on the AP1000, using the first phase of ECM with limit 100000 and 256 trials. The same computation could be performed on the VP2200 in 915 seconds.

The Fortran compilers on the VP2200 and VPP500 can achieve close to peak speed for well-vectorized loops. Considering their peak speeds, the VPP500 should have performed about 5 times faster than the VP2200. However, various overheads due to the parallelisation reduced this ratio for short runs. Times typical of small runs on the three machines are: on the VP2200, 30 seconds; on the AP1000, 68 seconds; and on the VPP500, 23 seconds (all to perform the same amount of work). However for longer runs, the VPP500 performed better. For example, on one run the VP2200 took 292 seconds and the VPP500 took only 50 seconds. The ratio (5.84) is greater than the ratio of peak speeds (5.12). This may be because the VPP500 has a better memory bandwidth per flop, so it is easier for the compiler to achieve close to peak performance in vectorized loops. If INT and DFLOAT are counted as floating-point operations (which is reasonable, since they use the vector pipelines) then our programs achieve greater than 55% of peak performance on the VP2200 and about 64% on the VPP500.

### 3 Brent-Pollard “rho”

The Brent-Pollard “rho” programs, written by Eldershaw, were based on algorithm  $P_2''$  in Brent’s paper [2] which improved the efficiency of Pollard’s original “rho” method [13]. On the AP1000 the calculations are performed using MP [1]. Parallelization is carried out as suggested in [4]. That is, each processor independently repeats the same procedure using a different function  $F$  (the difference depending upon a single parameter) to generate a pseudo-random sequence.

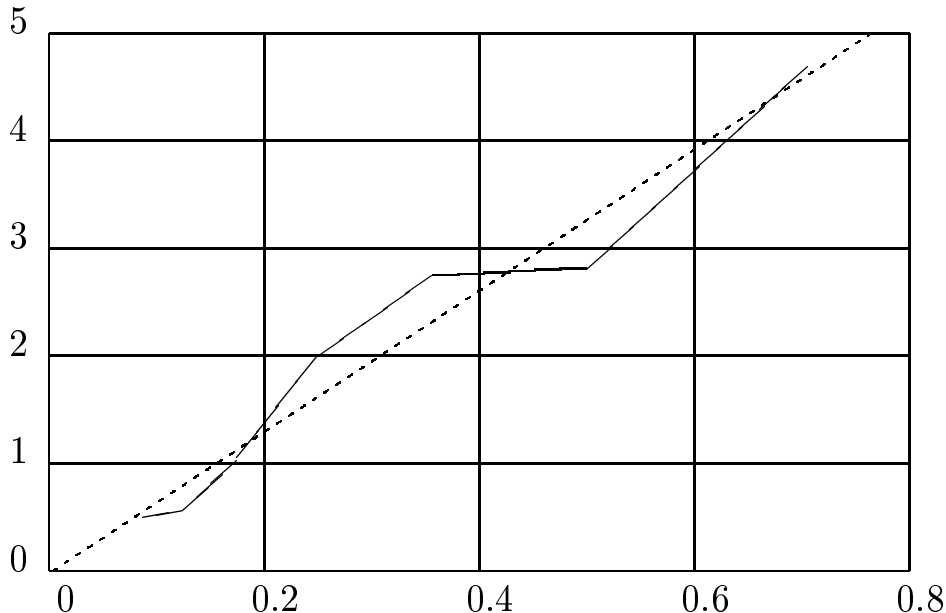


Figure 2: Time  $T_P$  (arbitrary units) versus  $1/\sqrt{P}$  for Brent-Pollard Rho

As pointed out in [4], a speedup of order  $\sqrt{P}$  is all that can be expected when using  $P$  processors. This gives an *expected* run time of:

$$T_P \sim T_1/\sqrt{P} \quad (2)$$

assuming that the computation stops as soon as one processor finds a factor.

Tests were run on the AP1000 with the number of processors varying in powers of two (from  $2^0$  to  $2^7$ ). The results were reasonably consistent with the prediction (2). Linear regression of  $T_P$  vs  $1/\sqrt{P}$  shows an error of less than 10% in the gradient of a linear fit to the data points. Experimental data points along with the fitted line (dotted) are shown in Figure 2. Parallelization of the “rho” algorithm is not nearly as effective as for the ECM, i.e. “rho” is not scalable.

Consider implementing the “rho” algorithm on a vector machine with vector lengths  $v$ . Because of vector startup times and parts of the code which run in scalar mode, we can expect the time for  $v$  independent function evaluations to be proportional to  $v + v_{1/2}$ , where  $v_{1/2}$  is a constant. Since the expected number of function evaluations to find a factor is proportional to  $\sqrt{v}$ , the expected run time  $T_v$  is proportional to

$$\frac{v + v_{1/2}}{\sqrt{v}} \quad (3)$$

The function in (3) has a minimum of  $2\sqrt{v_{1/2}}$  at  $v = v_{1/2}$ . Thus, the maximum speedup is approximately  $\sqrt{v_{1/2}}/2$ .

A vectorized version was implemented on the VP2200. The effect of varying the vector length  $v$  is shown in Figure 3. Note that the units of time are arbitrary and  $T_1 = 19.4$  is well off the page. The results are roughly as predicted by (3). For the optimal value of  $v$  (a few hundred) the speedup over  $v = 1$  is about 12.

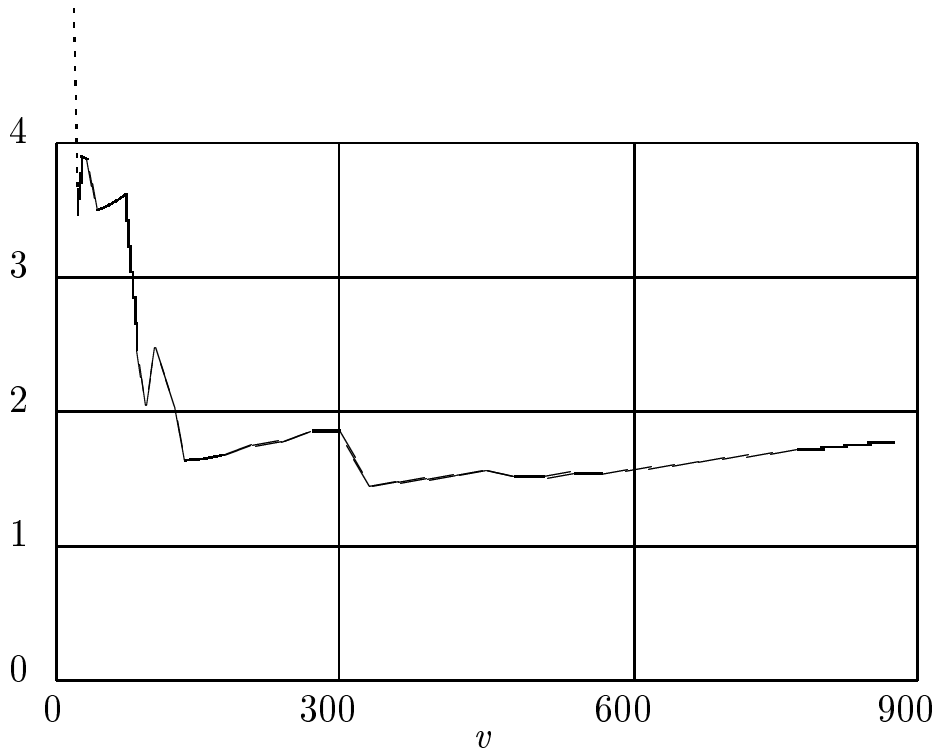


Figure 3: Time versus vector length  $v$  for Brent-Pollard Rho

## 4 Conclusion

The effect of parallelization was as predicted – ECM obtains close to linear speedup, but “rho” only obtains a speedup of order  $\sqrt{P}$  on a machine with  $P$  processors. Nevertheless, “rho” is much simpler than ECM and should be faster than ECM for small factors and a small number of processors, provided both implementations are equally well vectorized.

A good factorization strategy is to use trial division to remove very small factors, possibly followed by Brent-Pollard “rho” and/or Pollard “ $p \pm 1$ ” [12], and then ECM. If ECM can not complete the factorization in a reasonable time, MPQS (or NFS) is needed. Even in this case, the time spent on ECM is not wasted, because the time required by MPQS is greatly reduced for each factor found by ECM. Most of the nontrivial factorizations listed in [7] were found using a combination of methods, and could not have been found in a reasonable time with a single method.

## Acknowledgements

The work of the first author was performed during a visit to the Research School of Information Sciences and Engineering at the Australian National University as a summer scholar. The ANU Supercomputer Facility, the ANU-Fujitsu CAP Project, and Fujitsu Ltd. kindly provided access to the VP2200, AP1000 and VPP500 respectively. We thank Dr M. Hegland, Dr I. Macleod, Mr D. Sitsky and Dr B. Zhou for their assistance.

## References

- [1] R. P. Brent, “Algorithm 524: MP, a Fortran multiple-precision arithmetic package [A1], *ACM Trans. on Mathematical Software* 4 (1978), 71–81.
- [2] R. P. Brent, “An improved Monte Carlo factorization algorithm,” *BIT* 20 (1980), 176–184.
- [3] R. P. Brent, “Some integer factorization algorithms using elliptic curves,” *Australian Computer Science Communications* 8 (1986), 149–163.
- [4] R. P. Brent, “Parallel algorithms for integer factorisation”, in *Number Theory and Cryptography* (edited by J. H. Loxton), Cambridge University Press, 1990.
- [5] R. P. Brent, “Vector and parallel algorithms for integer factorisation”, *Proc. Third Australian Supercomputer Conference*, Melbourne, 1990.
- [6] R. P. Brent, A. J. Cleary, M. Hegland, J. H. Jenkinson, Z. Leyk, M. Nakanishi, M. R. Osborne, P. J. Price, S. Roberts and D. B. Singleton, “Implementation and performance of scalable scientific library subroutines on Fujitsu’s VPP500 parallel-vector supercomputer”, *Proc. Scalable High Performance Computing Conference*, (Knoxville, Tennessee, 23-25 May, 1994), IEEE Computer Society Press, Los Alamitos, California, 1994, 526–533.
- [7] R. P. Brent and H. J. J. te Riele, “Factorizations of  $a^n \pm 1$ ,  $13 \leq a < 100$ ”, Report NM-R9212, Centrum voor Wiskunde en Informatica, Amsterdam, June 1992, v+363 pp. ISSN 0169-0388.
- [8] T. R. Caron and R. D. Silverman, “Parallel implementation of the quadratic sieve”, *J. Supercomputing* 1 (1988), 273–290.
- [9] H. Ishihata, T. Horie and T. Shimizu, “Architecture for the AP1000 highly parallel computer”, *Fujitsu Sci. Tech. J.* 29 (1993), 6–14.
- [10] A. K. Lenstra and H. W. Lenstra (editors), *The Development of the Number Field Sieve*, Lecture Notes in Mathematics 1554, Springer-Verlag, Berlin, 1993.
- [11] H. W. Lenstra, Jr., “Factoring integers with elliptic curves”, *Annals of Math.* (2) 126 (1987), 649–673.
- [12] P. L. Montgomery, “Speeding the Pollard and elliptic curve methods of factorisation”, *Mathematics of Computation* 48 (1987), 243–264.
- [13] J. M. Pollard, “A Monte Carlo method for factorisation”, *BIT* 15 (1975), 331–334.
- [14] C. Pomerance, J. W. Smith and R. Tuler, “A pipeline architecture for factoring large integers with the quadratic sieve algorithm”, *SIAM J. on Computing* 17 (1988), 387–403.
- [15] H. J. J. te Riele, W. Lioen and D. Winter, “Factoring with the quadratic sieve on large vector computers”, *Belgian J. Comp. Appl. Math.* 27(1989), 267–278.
- [16] R. L. Rivest, A. Shamir and L. Adelman, “A method for obtaining digital signatures and public-key cryptosystems”, *Comm. ACM* 21 (1978), 120–126.
- [17] N. Uchida, “Fujitsu VP 2000 series supercomputers”, *Int. J. High Speed Computing* 3 (1991), 169–185.