# Using the Parallel Karatsuba Algorithm for Long Integer Multiplication and Division

Tudor Jebelean

RISC-Linz, A-4040 Linz, Austria
Jebelean@RISC.Uni-Linz.ac.at

**Abstract.** We experiment with sequential and parallel versions of the Karatsuba multiplication algorithm implemented under the `paclib` computer algebra system on a Sequent Symmetry shared-memory architecture. In comparison with the classical multiplication algorithm, the sequential version gives a speed-up of 2 at 50 words, up to 5 at 500 words. On 9 processors, the parallel Karatsuba algorithm exhibits a combined speed-up of 10 (50 words) up to 40 (500 words).
Moreover, we use the Karatsuba algorithm within long integer division with remainder, using a recent divide-and-conquer technique which delays part of the dividend updates until they can be performed by multiplication between large operands. The sequential algorithm is about two times slower than Karatsuba multiplication and shows a speed-up of 2 at 200 words and of 3 at 500 words, when compared to the classical division method. Using parallel multiplication on 9 processors leads to a combined speed-up of almost 3 at 100 words and more than 10 at 500 words.

## Introduction

Fast methods for operations over long integers are very important for applications like cryptography and arbitrary precision arithmetic. The asymptotically fast algorithms (e.g. FFT - see [13]) do not yield a speed-up for lengths which are mostly used in practice (see e.g. [15, 11]).

The only exception is Karatsuba divide-and-conquer scheme for *multiplication* [8], whose break-even point may be between 4 and 100 computer-words, depending on the implementation. Moreover, this algorithm is easy to parallelize – for detailed experiments on a shared-memory architecture see [11]. We obtain similar results using the **`paclib`** [3] computer algebra system (a shared-memory parallelization of **`saclib`** [1]): the Karatsuba threshold is at 6 words (of 29 bits), at 50 words the speed-up over the classical algorithm is 2, at 100 words is 2.5. On Sequent Symmetry shared-memory architecture, the speed-up of the parallel Karatsuba algorithm over the sequential one ranges from 2 (at 30 words) to almost 3 (at 100 words) on 3 processors, while on 9 processors it ranges from 2 (20 words) to 6.5 (100 words) to 8 (500 words). The combined speed-up of the Karatsuba parallel algorithm on 3 processors over the sequential classical algorithm is 7 (at 100 words), while on 9 processors is 16 (at 100 words) and 40 (at 500 words).

The situation is different for long integer *division*. Although this operation has the same theoretical complexity as multiplication (see e.g. [9], p. 275), performing division by Karatsuba multiplications involves a loss of about 30 times in speed (see an analysis in [10]), which leads to a break-even point of about 1000 words. If one cleverly uses squaring (half as expensive) instead of multiplication, then one can hope to decrease this point to 250 words. Also, parallelization of integer division is difficult, because each iteration of the main loop depends essentially on the results of the previous one. Theoretical parallel algorithms have been designed (see [12] for a survey), but practical implementations are realized mostly for VLSI design [14] and on systolic architectures [6, 4].

In the present paper we experiment with a novel technique developed in [5], which allows to use Karatsuba multiplication for division with a slow-down of only a factor of two. Under the **paclib** computer algebra system the algorithm starts to be faster than the classical method at 15 words, at 200 words the speed-up is 2, and it becomes 3 at 500 words. (We use here the length of the divisor and of the quotient).

Embedding the parallel Karatsuba algorithm into division yields a combined speed-up of 2 (50 words) up to 10 (500 words) on 9 processors, and of 2 (60 words) up to 7 (500 words) on 3 processors.

## 1 Multiplication

The Karatsuba multiplication algorithm [8], [9], p. 258, consists in splitting the operands $A, B$ into halves: $A = A_1\alpha + A_0$, $B = B_1\alpha + B_0$, where $\alpha$ is a suitable power of the radix, and computing the product by:

$$AB = (A_1B_1)\alpha^2 + ((A_0 + A_1)(B_0 + B_1) - A_0A_1 - B_0B_1)\alpha + A_0B_0. \qquad (1)$$

Each of the three products is computed again by the same strategy. The number of digit products of this scheme is $n^{\log 3/\log 2}$ for length-$n$ operands, as opposed to $n^2$ in the classical scheme (see e.g. [9], p. 233). In practice one stops the recursion and applies the classical algorithm when the lengths of the operands are smaller than an experimentally determined threshold. Depending on the implementation, the threshold varies from 4 words (e.g. **gnu mp** package [2]) to 100 words (e.g. the first version of **saclib**, which was operating on lists – see [7]). In our implementation under **paclib**, this threshold is 6 words of 29 bits.

Table 1 shows the timings of the classical algorithm (column 2), the Karatsuba algorithm (column 3) as well as the speed-up (column 4). The timings are given in milliseconds and they are averaged over 100 runs. The absolute values should be regarded with the additional fact that the Sequent computer we used has i386 processors running at 25 MHz.

Parallelization of the Karatsuba scheme is quite straight-forward: one just performs in parallel the three multiplications occurring in (1). In our implementation we perform the operations $(A_0 + A_1)(B_0 + B_1)$ within the current task, while $A_0B_0$ and $A_1B_1$ are performed by two parallel tasks. Since the base-case

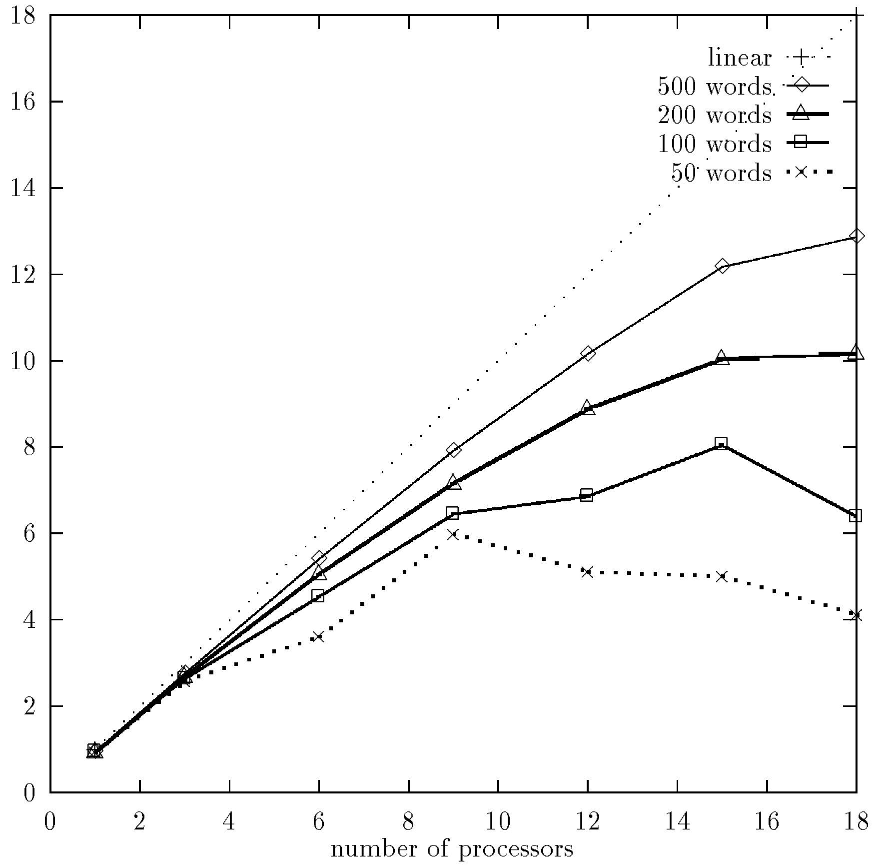
**Table 1.** Timing of multiplication in milliseconds, lengths are in words of 29 bits.

| length | classic | seq. Karatsuba | | parallel Karatsuba | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | ($\mathcal{C}$) | ($\mathcal{K}$) | | on 3 processors | | | on 9 processors | | |
| | | abs. | vs.$\mathcal{C}$ | abs. | vs.$\mathcal{C}$ | vs.$\mathcal{K}$ | abs. | vs.$\mathcal{C}$ | vs.$\mathcal{K}$ |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 10 | 4 | 4 | *1.00* | 4 | *1.00* | *1.00* | 4 | *1.00* | *1.00* |
| 15 | 10 | 8 | *1.25* | 5 | *2.00* | *1.60* | 6 | *1.67* | *1.33* |
| 20 | 17 | 12 | *1.42* | 6 | *2.83* | *2.00* | 6 | *2.83* | *2.00* |
| 30 | 37 | 23 | *1.61* | 12 | *3.08* | *1.92* | 7 | *5.29* | *3.29* |
| 40 | 65 | 37 | *1.76* | 15 | *4.33* | *2.47* | 8 | *8.13* | *4.63* |
| 60 | 144 | 69 | *2.09* | 28 | *5.14* | *2.46* | 13 | *11.08* | *5.31* |
| 80 | 254 | 110 | *2.31* | 42 | *6.05* | *2.62* | 19 | *13.37* | *5.79* |
| 100 | 398 | 158 | *2.52* | 57 | *6.98* | *2.77* | 25 | *15.92* | *6.32* |
| 150 | 892 | 297 | *3.00* | 105 | *8.50* | *2.83* | 43 | *20.74* | *6.91* |
| 200 | 1,585 | 480 | *3.30* | 168 | *9.43* | *2.86* | 66 | *24.02* | *7.27* |
| 300 | 3,552 | 898 | *3.96* | 323 | *11.00* | *2.78* | 120 | *29.60* | *7.48* |
| 400 | 6,317 | 1,441 | *4.38* | 492 | *12.84* | *2.93* | 183 | *34.52* | *7.87* |
| 500 | 9,862 | 2,034 | *4.85* | 704 | *14.01* | *2.89* | 255 | *38.67* | *7.98* |

Karatsuba tasks are rather fine-grained, creating a parallel task for each of them introduces a significant parallelization overhead (we measured 17% to 40% for lengths between 10 and 100 words). The overhead is minimal when we limit the parallelization depth: to 1 for using 3 processors, and to 2 for using 9 processors. Table 1 lists the experimental timings of the sequential and parallel Karatsuba algorithm. Columns 5 and 8 show the absolute time on 3 and 9 processors respectively, columns 7 and 10 show the speed-up over the sequential Karatsuba algorithm, and columns 6, 9 show the combined speed-up over the sequential classical algorithm. The efficiency surpasses 75% on 3 processors already at 40 words, and on 9 processors at 150 words. At 500 words the efficiency is around 90%. The combined speed-up on 3 processors ranges from 3 (at 30 words), to 7 (100 words) to 14 (500 words), and on 9 processors from 5 (at 30 words), to 16 (100 words) to 39 (500 words).

In order to obtain a **scalable algorithm** one has to give up the control over the recursion depth and to increase instead the parallelization threshold to 4 times the Karatsuba threshold – i.e. to 24 words. Timing the program on one processor (see Fig. 1) we noticed that this decreases the overhead to an acceptable level (less than 10%). However the number of tasks created is too low for lengths under 100 words. The algorithm scales well until 18 processors only for the length of 500 words - see Fig. 1. For this length one obtains a speed-up of 13 (efficiency 72%) over the sequential Karatsuba algorithm and a combined speed-up of 61 over the sequential classical algorithm.

**Fig. 1.** Speed-up of the parallel Karatsuba multiplication.




## 2 Division

The classical division algorithm (see [9] p. 237) consists of a series of successive updates of the dividend $A$ by subtracting the divisor $B$ multiplied by the current digit of the quotient $Q$. (This quotient digit is computed before each update using the 3 most significant digits of the current $A$ and the 2 most significant digits of $B$.) Each update is right–shifted one position w.r.t. the previous one. This process is represented pictorially by the parallelogram in Fig. 2. The final value of $A$ will be the remainder $R$.

The scheme introduced in [5] starts by splitting the quotient $Q$ into its high-order part $Q_H$ of length $q_H$ and its low-order part $Q_L$ of length $q_L$ such that $q_L \leq q_H \leq q_L + 1$, and also the divisor $B$ into its high order part $B_H$ of length $b_H$ and its low-order part $B_L$ such that $q_H + 3 \leq b_H$.

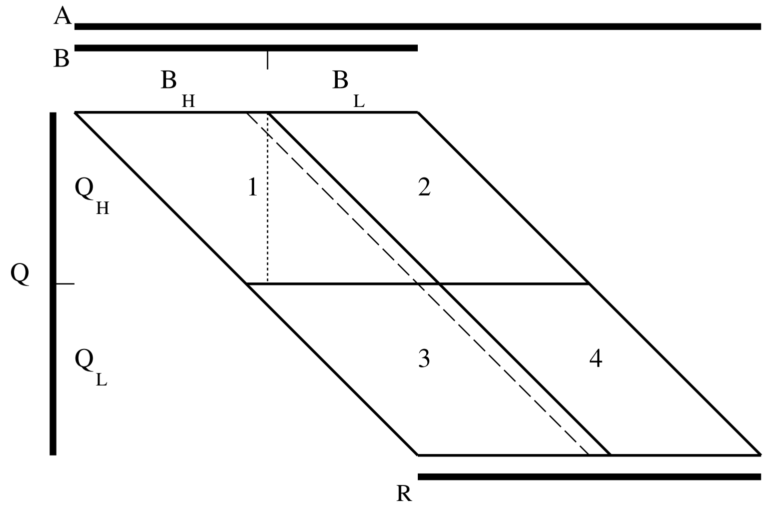



**Fig. 2.** Organization of the dividend updates.

Computing the high-order part $Q_H$ of the quotient would normally require to perform the updates in the upper half of the parallelogram of Fig. 2 (areas 1 and 2). However, Krandick [10] proves that in most cases it is enough to update only 3 words below the lowest digit of $A$ needed for the lowest quotient digit to be computed (i.e. down to the vertical line crossing area 1). Doing so, the probability that a quotient digit will be wrong is less than $q_H/2^w$, where $w$ is the bit-length of the word. Moreover, such a failure is easy to detect by inspecting the most significant updated value of $A$ at each step.

This allows to split the updates of the dividend into 4 parts as shown by 1, 2, 3, 4 in Fig.2, and to perform them in the order of the numbering: parts 2 and 4 by Karatsuba multiplication, and parts 1 and 3 by the same recursive technique. When the recursion is applied to operands shorter than a threshold (15 words in our implementation), then the updates of type 1, 3 are performed by classical division. This divide-and-conquer algorithm has Karatsuba-like complexity, moreover the number of digit-products performed is only 2 times higher than the number of digit products of Karatsuba multiplication (for details see [5]).

The timing of our sequential implementation under `paclib` is shown in Table 2. Column 1 lists the word-length of the dividend and of the divisor, columns 2 and 3 list the time of the classical algorithm and the new algorithm in milliseconds, and column 4 shows the speedup of the new algorithm. The speed-up is visible at 15 words, it becomes 1.6 at 100 words and 3 at 500 words.

The speed-up can be increased significantly by using the parallel version of the Karatsuba multiplication algorithm presented in the previous section. Table 2 shows the results of the experiments using the "controlled depth" algorithms on 3 and 9 processors. Columns 5 and 8 list the absolute time, columns 7 and 10 list the speed-up over the sequential algorithm, and columns 6 and 9 list

**Table 2.** Timing of division in milliseconds, lengths are in words of 29 bits.

| dividend-len./divisor-len. | classic ($\mathcal{C}$) | seq. Karatsuba ($\mathcal{K}$) | | parallel Karatsuba | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | on 3 processors | | | on 9 processors | | |
| | | abs. | vs.$\mathcal{C}$ | abs. | vs.$\mathcal{C}$ | vs.$\mathcal{K}$ | abs. | vs.$\mathcal{C}$ | vs.$\mathcal{K}$ |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 16/10 | 4 | 4 | *1.00* | 4 | *1.00* | *1.00* | 4 | *1.00* | *1.00* |
| 26/15 | 10 | 9 | *1.11* | 9 | *1.11* | *1.00* | 9 | *1.11* | *1.00* |
| 36/20 | 17 | 15 | *1.13* | 15 | *1.13* | *1.00* | 16 | *1.06* | *0.94* |
| 56/30 | 39 | 34 | *1.15* | 29 | *1.34* | *1.17* | 31 | *1.26* | *1.10* |
| 76/40 | 70 | 55 | *1.27* | 45 | *1.56* | *1.22* | 47 | *1.49* | *1.17* |
| 116/60 | 159 | 113 | *1.41* | 75 | *2.12* | *1.51* | 73 | *2.18* | *1.55* |
| 156/80 | 285 | 186 | *1.53* | 121 | *2.36* | *1.54* | 114 | *2.50* | *1.63* |
| 196/100 | 447 | 278 | *1.61* | 175 | *2.55* | *1.59* | 161 | *2.78* | *1.73* |
| 236/120 | 644 | 365 | *1.76* | 205 | *3.14* | *1.78* | 175 | *3.68* | *2.09* |
| 296/150 | 1,008 | 527 | *1.91* | 288 | *3.50* | *1.83* | 239 | *4.22* | *2.21* |
| 396/200 | 1,790 | 881 | *2.03* | 454 | *3.94* | *1.94* | 365 | *4.90* | *2.41* |
| 596/300 | 4,040 | 1,652 | *2.45* | 794 | *5.09* | *2.08* | 573 | *7.05* | *2.88* |
| 796/400 | 7,188 | 2,702 | *2.66* | 1,231 | *5.84* | *2.19* | 866 | *8.30* | *3.12* |
| 996/500 | 11,255 | 3,825 | *2.94* | 1,659 | *6.78* | *2.31* | 1,111 | *10.13* | *3.44* |

the combined speed-up over the sequential classical algorithm. As expected, the efficiency of the parallel division is much lower than the one of multiplication, because the Karatsuba multiplication is called many times, and mostly with short-length operands. However, the combined speed-up is quite significant: on 3 processors it ranges from 2 times at 50 words, to 2.5 at 100 words, to almost 7 at 500 words, and on 9 processors from 2 times at 50 words, to almost 3 at 100 words, to more than 10 times at 500 words.

## Conclusions and Further Work

Combining Karatsuba multiplication with a divide-and-conquer scheme for long integer division gives a significant speed-up both in a sequential and in a parallel implementation.

As usually with long-integer algorithms, both the sequential and the parallel devices presented in the paper are applicable to polynomial arithmetic.

## References


1. B. Buchberger, G. E. Collins, M. J. Encarnacion, H. Hong, J. R. Johnson, W. Krandick, R. Loos, A. M. Mandache, A. Neubacher, and H. Vielhaber. SACLIB 1.1 User's Guide. Technical Report 93–19, RISC–Linz, 1993.
2. T. Granlund. GNU MP: The GNU multiple precision arithmetic library, 1991.

3. Hoon Hong, Wolfgang Schreiner, Andreas Neubacher, Kurt Siegl, Hans-Wolfgang Loidl, Tudor Jebelean, and Peter Zettler. PACLIB User Manual. Technical Report 92-32, RISC-Linz, 1992.
4. T. Jebelean. Integer and rational arithmetic on MasPar. In *DISCO'96*, pages 162–173, Karlsruhe, Germany, September 1996. Springer Verlag LNCS 1128.
5. T. Jebelean. Integer Division with Karatsuba Complexity. Technical Report 96-29, RISC-Linz, 1996. Sumitted to ISSAC 97.
6. T. Jebelean. Systolic algorithms for exact division. *Mitteilungen–Gesellschaft für Informatik e. V., Parallel Algorithmen und Rechnerstrukturen*, (12):40–50, July 1993.
7. Tudor Jebelean. Systolic multiprecision arithmetic. Technical Report 94-37, RISC-Linz, 1994. PhD Thesis.
8. A. Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. *Sov. Phys. Dokl.*, 7:595–596, 1962.
9. D. E. Knuth. *The art of computer programming*, volume 2. Addison-Wesley, 2 edition, 1981.
10. W. Krandick and T. Jebelean. Bidirectional exact integer division. *Journal of Symbolic Computation*, 21:441–455, 1996.
11. W. Kuechlin, D. Lutz, and N. Nevin. Integer multiplication on PARSAC-2 on stock microprocessors. In H. F. Mattson, T. Mora, and T. R. N. Rao, editors, *AAECC-9*, pages 216–217, New Orleans, 1991. Springer Verlag. LNCS 539.
12. S. Lakshmivarahan and S. K. Dhall. *Analysis and design of parallel algorithms: Arithmetic and matrix problems*. McGraw-Hill, 1990.
13. J. D. Lipson. *Elements of Algebra and Algebraic Computing*. Benjamin/Cummings, 1981.
14. E. E. Swartzlander, editor. *Computer Arithmetic*, volume 2. IEEE Computer Society Press, 1990.
15. D. Zuras. More on squaring and multiplying large integers. *IEEE Trans. on Computers*, 43(8):899–908, 1994.