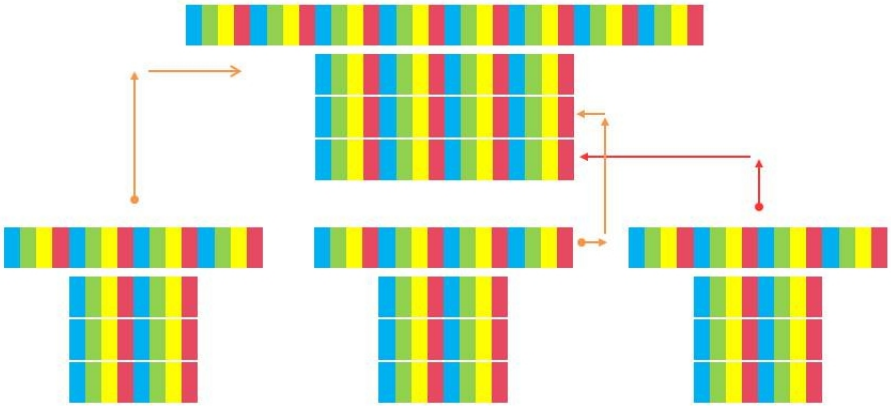
"The Karatsuba Algorithm for Integer Multiplication:

a Parallel Implementation Using BSP"



Laurens Willenborg
Student registration number 3020274

Thesis for the Master Program
Science Education and Communication
Utrecht University

Supervisors: prof. dr. R.H. Bisseling and prof. dr. F. Beukers

April 25, 2014

Abstract

In the last decades of the 20th century, algorithms for multiplying large integers have been improved considerably. Today's state-of-the-art algorithms are based on the Fermat Number Transform and the Fast Fourier Transform, originally developed by Schönhage-Strassen in the 1970's. These convolution-based algorithms are of order $\mathcal{O}(n \log n \log \log n)$.

Another algorithm for multiplying integers is the Karatsuba method, an algorithm of order $\mathcal{O}(n^{\log_2 3})$. Although less efficient, it still outperforms the transform-based algorithms for numbers of size $n < 100,000$ decimal digits.

In this thesis, a BSP-based parallel implementation of the Karatsuba algorithm is presented. In the literature there were some doubts about the possibility to design an efficient parallel implementation, due to high communication costs. This implementation therefore focused on reducing communication as much as possible.

The parallel BSP-program was run at the Cartesius parallel computer at the SurfSARA computer center in Amsterdam. Numbers up to a size of $n = 1 \times 10^9$ words ($\approx 10^{10}$ decimal digits) were multiplied, with nice results for parallel speedup and parallel efficiency ($E_p > 0.8$).

By combining this parallel implementation of the Karatsuba algorithm with a sequential convolution-based algorithm for integer multiplication, a parallel implementation for convolution-based algorithms can be achieved, which makes multiplying ultra long integers in parallel mode possible. Unfortunately satisfactory parallel speedups in this case can theoretically only be achieved for a small number of processors $p \leq 27$.

Although the Karatsuba algorithm is not fit for the multiplication of ultra long integers, a cost-efficient parallel implementation has been designed successfully and can be useful for the multiplication of mid-range sized integers $10^4 \leq n_w \leq 10^7$ words, $\approx 10^5 \leq n_d \leq 10^8$ decimal digits.

Contents

1 Introduction **4**
- 1.1 History and literature . . . 4
- 1.2 Theoretical background . . . 5
  - 1.2.1 Description of the algorithm . . . 5
  - 1.2.2 Complexity of the algorithm . . . 6
- 1.3 Sequential implementation of the algorithm . . . 7

2 Parallel implementation **10**
- 2.1 Introductory remarks . . . 10
- 2.2 Data distributions . . . 11
- 2.3 Description of the parallel implementation . . . 12
- 2.4 Communication costs . . . 24

3 Results **26**
- 3.1 Verification tests . . . 26
- 3.2 Tuning parameters . . . 26
- 3.3 Test results . . . 28
  - 3.3.1 Run times . . . 28
  - 3.3.2 Parallel speedup . . . 28
  - 3.3.3 Parallel efficiency . . . 30
  - 3.3.4 Comparison with results from the literature . . . 30
- 3.4 Analysis of the stages in the algorithm . . . 31
  - 3.4.1 Parallel stages . . . 31
  - 3.4.2 Parallel accumulated versus sequential . . . 33
- 3.5 Analysis of the communication costs . . . 35
- 3.6 Test Ultra Long Integers . . . 36
- 3.7 Comparison thin and fat nodes . . . 38

4 Discussion and Conclusion **40**
- 4.1 Performance of the implementation . . . 40
- 4.2 Parallelizing convolution based algorithms . . . 40
- 4.3 Suggestions for improvement . . . 40
- 4.4 Practical applicability . . . 41

5 References **42**

1 Introduction

1.1 History and literature

In 1962 A. Karatsuba and Y. Ofman described a method[9] to multiply large integers more efficiently when compared to methods typically learned in elementary school (often referred to as classical multiplication). The Karatsuba algorithm was the first attempt to find faster algorithms for multiplying large integers, and multiplication methods have improved further in the last couple of decades of the 20th century.

Today's state-of-the-art algorithms for multiplying ultra long integers, the Schönhage-Strassen methods[13], are based on the Fourier Transform, the Number Theoretic Transform and the Fermat Number Transform. They all make use of the Convolution Theorem, known from Fourier Theory, and are therefore also called convolution-based methods; detailed descriptions of these methods can be found in Nussbaumer[12]. They are so efficient thanks to the development of the Fast Fourier Transform, an algorithm that makes computing a Fourier Transform possible in time $\mathcal{O}(n \log n)$.

Although the Karatsuba algorithm cannot compete with these state-of-the-art algorithms when it comes to multiplying ultra long integers, it still is an efficient algorithm for numbers of considerable size. In articles published in 1993 and 1994, Zuras claimed the Karatsuba algorithm and similar methods to be more efficient compared to the Schönhage-Strassen methods for numbers up to at least 3 million bits (1993)[14] or even 37 million bits (1994)[15], the integers stored as arrays of unsigned 32-bit words. Thresholds used for convolution-based multiplication today are in the order of 10.000 32-bit words[7], about 100.000 decimal digits, depending on the configuration used.

This makes it interesting to develop an efficient and scalable parallel implementation of the Karatsuba algorithm. Some have done so in the past, e.g. Cesari & Maeder[3] and Jebelean[8], others claim the Karatsuba algorithm to be unsuitable for parallelization, for different reasons. For instance, Chen and Schaumont[4], when developing a parallel implementation of the Montgomery algorithm, do not wish to consider parallelizing the Karatsuba method. They argue, that Karatsuba optimization 'leads to a subquadratic increase of the amount of multiplications' but 'also has a superquadratic increase in the number of accumulations' and conclude 'In parallel software, where the cost of an addition and a multiplication is similar, Karatsuba optimization therefore does not lead to an obvious advantage.'. Fagin[6] considers the Karatsuba algorithm unsuitable for parallel implementation due to high communication costs, caused by the 'divide and conquer approach, applying the same technique to smaller and smaller pieces of the input and then reassembling the results' and concludes 'This reassembly requires significant interprocessor communication time on a multiprocessor and suggests an examination of other algorithms.'.

Jebelean's approach[8] is process-driven; parts of the algorithm are processed sequentially before distributing tasks among other processors. The implementation is tested with relatively small numbers ($n \leq 500$, with n the number of 29 bit-words), for p a multiple of 3, $p \leq 18$.

Cesari & Maeder[3] have a similar approach, but they designed different algorithms for p a power of 2 and for p a power of 3. For the algorithm with p a power of 2, parallel speedups are poor, and the parallel efficiency drops quickly:

theoretically, for $p \geq 9$, $E_{par} < 0.4$ (see section 3.3 for definitions of parallel speedup and parallel efficiency). For algorithms designed for p a power of 3, results are reported for $p = 9$, $p = 27$, and $p = 81$, with $2^{10} \leq n \leq 2^{20}$, n a power of 2 (the word length is not mentioned). In section 3.3.4 these results will be referenced for comparison.

The goal in this research is to develop a fully scalable and efficient parallel implementation of the Karatsuba algorithm. The focus will be on reducing communication costs, in order to obtain good parallel speedups, also with a large number of processors.

1.2 Theoretical background

1.2.1 Description of the algorithm

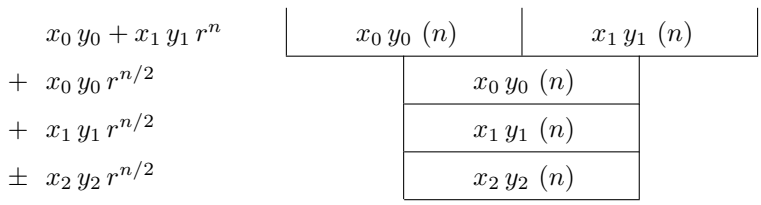
Let x and y be n-digit integers in base (radix) r: $x = \sum_{i=0}^{n-1} a_i r^i$, $y = \sum_{j=0}^{n-1} b_j r^j$ $(0 \leq a_i, b_j < r)$. Split x and y into two parts: $x = x_0 + x_1 r^m$, $y = y_0 + y_1 r^m$ $(m < n)$. Using the Karatsuba algorithm for multiplying x and y, the product $x\,y$ is written as

$$\begin{aligned}
x\,y &= (x_0 + x_1\,r^m)(y_0 + y_1\,r^m) \\
&= x_0\,y_0 + (x_0\,y_1 + x_1\,y_0)\,r^m + x_1\,y_1\,r^{2m} \\
&= x_0\,y_0 + \big((x_0 + x_1)(y_0 + y_1) - x_0\,y_0 - x_1\,y_1\big)\,r^m + x_1\,y_1\,r^{2m} \qquad (1) \\
&= x_0\,y_0 - \big((x_0 - x_1)(y_0 - y_1) - x_0\,y_0 - x_1\,y_1\big)\,r^m + x_1\,y_1\,r^{2m} \qquad (2)
\end{aligned}$$

In both alternatives the number of multiplications is reduced from 4 to 3, at the cost of 3 extra additions/subtractions. This procedure then is repeatedly applied (in each step taking $m_{i+1} \approx \frac{m_i}{2}$), until at some point the multiplication is computed using the classical algorithm. A numerical example will be given in section 1.3.

In the second alternative, $x_0 - x_1$ and $y_0 - y_1$ are computed, opposed to $x_0 + x_1$ and $y_0 + y_1$ in the first. For reasons discussed later, the second alternative will be used for implementation of the sequential, recursive algorithm (applied in the local computation of products), while the first alternative is used for implementation of the parallel part of the algorithm.

Using (2) and assuming n is even and x and y are split into parts of size $n/2$ each, recombining the intermediate results $\big(x_2\,y_2 \equiv |x_0 - x_1||y_0 - y_1|\big)$ into the final outcome $x\,y$ can be schematically represented in array format as



where the first entries of the 3 lower n-sized arrays coincide with the $(\frac{2n}{4})^{th}$ entry of the first $2n$-sized array, i.e. addition/subtraction of the 3 bottom arrays starts at entry $\frac{n}{2}$ of the top array.

1.2.2 Complexity of the algorithm

Consider two numbers x and y of size $n = 2^d$ digits. To compute the product $x\,y$, both numbers x and y are split into two equally sized parts x_0, x_1 and y_0, y_1 respectively. Applying the just described split in the Karatsuba algorithm once, one needs 3 (classical) multiplications of size $n/2$ and 6 additions/subtractions of size n in order to compute the product $x\,y$. Actually there are 3 additions of size n and 2 subtractions, $x_0 - x_1$ and $y_0 - y_1$, of size $n/2$, equivalent to 4 additions/subtractions of size n in total, but a few extra additions/subtractions are needed to handle carries and borrows (on average 1.5 in total); to compensate for some overhead, e.g. the generation of the partial output $x_0\,y_0 + x_1\,y_1 r^n$, the total is rounded up to 6 additions/subtractions of size n (note that the +-sign in $x_0\,y_0 + x_1\,y_1 r^n$ does not really result in an addition; the expression merely represents the filling of a $2n$-sized array).

Applying the split recursively, each multiplication results in 3 new multiplications and 6 extra additions/subtractions, the size being half the size of the previous split. In split 1, there are 6 additions/subtractions of size n, so the total number of operations to process these additions/subtractions is equal to $O_{Kar}(n,1) = 3^0 * 6 * (n/2^0) = 6n$. In split 2, $O_{Kar}(n,2) = 3^1 * 6 * (n/2^1)$, and in split k, $O_{Kar}(n,k) = 3^{k-1} * 6 * (n/2^{k-1}) = (3^k/2^{k-2}) * n = 4 * (3/2)^k * n$.

Assuming after split k the multiplication proceeds using the classical algorithm, there are 3^k multiplications of size $n/2^k$ to compute. Since one single multiplication of size m takes $3m^2$ operations (m^2 multiplications and $2m^2$ additions, the handling of carries included) using the classical algorithm, it takes in total $O_{Class}(n,k) = 3^k * 3(n/2^k)^2$ operations to compute all multiplications that remain after split k, using the classical multiplication algorithm throughout after split k.

To estimate the total number of operations (all multiplications, additions and subtractions) $O(n,k)$ in order to process the (up to split k recursively applied) Karatsuba algorithm, one totals (i) all operations to compute the additions and subtractions up to split k and (ii) all operations to compute the classical multiplications after split k:

$$
\begin{aligned}
O(n,k) &= \sum_{i=1}^{k} \Big(O_{Kar}(n,i) \Big) + O_{Class}(n,k) \\
&= \sum_{i=1}^{k} \left(4 * \left(\frac{3}{2}\right)^i * n \right) + 3^k * 3\left(\frac{n}{2^k}\right)^2 \\
&= 4\frac{\frac{3}{2} - (\frac{3}{2})^{k+1}}{1 - \frac{3}{2}} n + \frac{3^{k+1}}{2^{2k}} n^2 \\
&= 8\left(\left(\frac{3}{2}\right)^{k+1} - \frac{3}{2} \right) n + 3\left(\frac{3}{4}\right)^k n^2 \qquad (3)
\end{aligned}
$$

Now assume the recursive calls are continued until multiplications of size 1 are left: $k = \log_2 n$. The number of multiplications to compute will be equal to $3^k = 3^{\log_2 n} = (2^{\log_2 3})^{\log_2 n} = (2^{\log_2 n})^{\log_2 3} = n^{\log_2 3} \approx n^{1.58}$

Using $(\frac{3}{2})^{\log_2 n} = n^{\log_2 3 - 1} = n^{\log_2 3} * n^{-1}$ and similarly $(\frac{3}{4})^{\log_2 n} = n^{\log_2 3} * n^{-2}$ one obtains the total number of operations $O(n, \log_2 n) = 15n^{\log_2 3} - 12n$. This makes the recursively applied Karatsuba algorithm an $\mathcal{O}(n^{\log_2 3})$ algorithm.

The complexity can also be derived from the recurrence equation: let $T(n)$ be the time needed to compute the product $x\,y$, both x and y of size n, then $T(n) = 3T(\frac{n}{2}) + 6n$; using the Master Theorem for recursive algorithms[5] this implies an $\mathcal{O}(n^{\log_2 3})$ algorithm, since clearly $n^{\log_2 3}$ dominates $6n$.

However, considering the total number of operations during the recursive Karatsuba algorithm, it is possibly more efficient to cut off the recursive calls at an earlier stage: stop the recursive calls at split k, and compute the remaining multiplications using the classical multiplication algorithm.

By solving the equation $6m + 3 * 3(\frac{m}{2})^2 > 3m^2$ (for which m the number of operations splitting up x and y is larger compared to the number of operations using the classical multiplication algorithm), one obtains the point, where it is more efficient to continue the computation using the classical algorithm. This yields $6m + (\frac{9}{4} - 3)m^2 > 0 \Rightarrow m(6 - \frac{3}{4}m) > 0 \Rightarrow m < 8$.

The total number of operations stopping the recursive calls as soon as $n \leq 8$ will be equal to $O(n, \log_2 n - 3) = 8((\frac{3}{2})^{\log_2 n - 2} - \frac{3}{2})n + 3((\frac{3}{4})^{\log_2 n - 3})n^2$. Using $(\frac{3}{2})^{\log_2 n} = n^{\log_2 3 - 1} = n^{\log_2 3} * n^{-1}$ and similarly $(\frac{3}{4})^{\log_2 n} = n^{\log_2 3} * n^{-2}$ one obtains $O(n, \log_2 n - 3) = 8(n^{\log_2 3} n^{-1}(\frac{2}{3})^2 - \frac{3}{2})n + 3(n^{\log_2 3} n^{-2}(\frac{4}{3})^3)n^2$. This yields $O(n, \log_2 n - 3) = (8(\frac{2}{3})^2 + 3(\frac{4}{3})^3)n^{\log_2 3} - 12n = \frac{32}{3}n^{\log_2 3} - 12n$, slightly less when compared to $O(n, \log_2 n) = 15n^{\log_2 3} - 12n$ above (when the recursive calls are continued all the way to the end).

The threshold used in the algorithm will be determined experimentally and can deviate from the theoretical value due to overhead caused by memory management and system resources needed while executing the recursive calls.

1.3 Sequential implementation of the algorithm

The algorithm is implemented assuming n (the number of digits of x and y) contains enough powers of 2 to split the numbers up to the size n_Cs at which computation proceeds using the classical multiplication algorithm. This makes subsequent splits most efficient (i.e. the gain compared to the classical algorithm is maximized). If x and/or y do not contain the required powers of 2, trailing zeroes are added (numbers are represented polynomially: more significant digits are at the right, in the higher entries of the arrays).

As mentioned in section 1.2.1 on page 5, the second alternative for computing $x\,y$ is used for the sequential implementation of the Karatsuba algorithm. An example of the algorithm (for $r = 10$, $n = 4$), applying the split once, is given on page 9, showing also the difference between both alternatives. To avoid confusion, numbers are given in the usual, ordinary format (most significant digits on the left) as well as in array format (most significant digits on the right, between brackets).

As becomes clear, computing $x_0 + x_1$ and $y_0 + y_1$ has a major disadvantage: either intermediate results become larger than the radix, or extra table entries have to be reserved to store a possible overflow-carry. Moreover, this problem gets worse after consecutive recursive calls. Computing $x_0 - x_1$ and $y_0 - y_1$ in the Karatsuba algorithm instead, the sign has to be stored.

Computations are in general more efficient if a radix as large as possible is used for the representation of the numbers: (i) less memory is needed, which leads to a more efficient use of cache, (ii) on many systems, computing the product of 1-bit sized integers is just as costly as computing the product of 1 word-sized integers, at least on the machine this implementation is tested on (in

fact, the optimal choice is half the maximum size of the product $x\,y$ the machine can compute and store using the built-in operations), (iii) if conversion from or to another base is desired, this is only an $\mathcal{O}(n)$ algorithm with n the size of the number to be converted[11].

In the Karatsuba algorithm, computing $x_0 - x_1$ and $y_0 - y_1$ makes it possible to fully exploit the advantages of a radix as large as possible, the main reason to choose this alternative above the one in which $x_0 + x_1$ and $y_0 + y_1$ are computed. Determining and storing the sign of $x_0 - x_1$ and $y_0 - y_1$ hardly takes any resources and is easy to implement as well.

An outline of the implementation of the algorithm is presented below.

The sequential recursive Karatsuba algorithm

```
//  start sequential Karatsuba algorithm
*******************************************************************
input:  vector x[n], n=2^s
        vector y[n], n=2^s
output: vector z[2n]=x[n]*y[n]
function-call: Karatsuba(x,n,y,n,z,2n,n_Cs,szr)
               szr: determines the radix r=2^szr
               n_Cs: the size for switching to the classical algorithm
*******************************************************************
    if n<=n_Cs then Multiply(x,n,y,n,z,2n,szr); return z[];
    allocate vectors x0[n/2],x1[n/2],x2[n/2];
    for i:=0 to n/2-1 step 1 do
        x0[i]:=x[i];
        x1[i]:=x[i+n/2];
    for i:=0 to n/2-1 step 1 do
//      processing of borrows not included here
        x2[i]:=|x0[i]-x1[i]|;
    allocate vectors y0[n/2],y1[n/2],y2[n/2];
    for i:=0 to n/2-1 step 1 do
        y0[i]:=y[i];
        y1[i]:=y[i+n/2];
    for i:=0 to n/2-1 step 1 do
//      processing of borrows not included here
        y2[i]:=|y0[i]-y1[i]|;
    allocate vectors x0y0[n],x1y1[n],x2y2[n];
//  Recursive calls
    Karatsuba(x0,n/2,y0,n/2,x0y0,n,n_Cs,szr);
    Karatsuba(x1,n/2,y1,n/2,x1y1,n,n_Cs,szr);
    Karatsuba(x2,n/2,y2,n/2,x2y2,n,n_Cs,szr);
//  Recombine the results
    for i:=0 to n-1 step 1 do
//      z:=xoyo+x1y1*r^n
        z[i]:=x0y0[i];
        z[i+n]:=x1y1[i];
    if ``(x0-x1) and (y0-y1) have opposite signs''
     then
        for i:=0 to n-1 step 1 do
//          z:=xoyo+(x0y0+x1y1+x2y2)*r^(n/2)+x1y1*r^n
//          processing of carries not included here
            z[i+n/2]:=z[i+n/2]+x0y0[i]+x1y1[i]+x2y2[i];
     else
        for i:=0 to n-1 step 1 do
//          z:=xoyo+(x0y0+x1y1-x2y2)*r^(n/2)+x1y1*r^n
//          processing of carries/borrows not included here
            z[i+n/2]:=z[i+n/2]+x0y0[i]+x1y1[i]-x2y2[i];
    return z[];
//  end sequential Karatsuba algorithm
```

Worked example of the Karatsuba algorithm A full example of the Karatsuba algorithm, also showing the difference between the 2 alternatives (1) and (2) presented on page 5.

$x = \{2,4,6,8\} = 2 * 10^0 + 4 * 10^1 + 6 * 10^2 + 8 * 10^3 = 8642$
$y = \{9,7,5,3\} = 9 * 10^0 + 7 * 10^1 + 5 * 10^2 + 3 * 10^3 = 3579$
$x\,y = 8642 * 3579 = 30929718 = \{8,1,7,9,2,9,0,3\}$
$x_0 = \{2,4\} = 42 \quad x_1 = \{6,8\} = 86$
$y_0 = \{9,7\} = 79 \quad y_1 = \{5,3\} = 35$
$x_0\,y_0 = 42 * 79 = 3318 = \{8,1,3,3\}$
$x_1\,y_1 = 86 * 35 = 3010 = \{0,1,0,3\}$

Using the first alternative (1) on page 5, $x_0 + x_1$ and $y_0 + y_1$ are computed

$x_2 = x_0 + x_1 = 128 = \{8,12\} = \{8,2,1\} \quad y_2 = y_0 + y_1 = 114 = \{4,11\} = \{4,1,1\}$
$x_2\,y_2 = 128 * 114 = 14592 = \{2,9,5,14\} = \{2,9,5,4,1\}$

and recombining the above results into the final outcome $x\,y = x_0\,y_0 + (-x_0\,y_0 - x_1\,y_1 + x_2\,y_2)\,10^2 + x_1\,y_1\,10^4$ gives

8	1	3	3	0	1	0	3
-		8	1	3	3		
-		0	1	0	3		
+		2	9	5	14		I
+		2	9	5	4	1	II
8	1	7	9	2	9	0	3

Choosing I here implies a radix r sufficiently small to store the digit (in this example 14) larger than r in a variable of type integer. Choosing II implies an overflow digit (in this example 1) will occur, making the problem asymmetric.

Using the second alternative (2) on page 5, $|x_0 - x_1|$ and $|y_0 - y_1|$ are computed

$x_2 = |x_0 - x_1| = 44 = \{4,4\} \quad y_2 = |y_0 - y_1| = 44 = \{4,4\}$
$x_2\,y_2 = 44 * 44 = 1936 = \{6,3,9,1\}$

Since $x_0 - x_1 < 0$ and $y_0 - y_1 > 0$ have opposite signs, $x_2\,y_2$ must be added, not subtracted, when recombining the above results into the final outcome: $x\,y = x_0\,y_0 + (x_0\,y_0 + x_1\,y_1 + x_2\,y_2)\,10^2 + x_1\,y_1\,10^4$ (note that carries should be processed from left to right)

8	1	3	3	0	1	0	3
+		8	1	3	3		
+		0	1	0	3		
+		6	3	9	1		
8	1	7	9	2	9	0	3

2 Parallel implementation

2.1 Introductory remarks

In this parallel implementation, five steps are distinguished: (i) generate the lower level multiplications of reduced size, (ii) distribute the generated multiplications, (iii) compute these multiplications locally using the sequential recursive algorithm, (iv) distribute the results from the previous step, and (v) assemble the higher-level products until the final product is obtained.

Two observations can be made when considering how to parallelize the Karatsuba algorithm: (i) at each split the arrays that are to be multiplied are split into two parts and (ii) each split generates 3 new problems half the size of the original problem.

From a process point of view, it seems reasonable to start assuming the number of processors is a power of 3: $p = 3^m$. This gives an optimal computational load balance for the lower level multiplications to be computed, of which there are a power of 3 as well. However, if one wishes the parallel implementation to be fully scalable and efficient for a large number of processors, it becomes more and more important, that the computational load balance in the first step (the execution of all additions/subtractions) is optimal, and the communication needed for the process minimal. This asks for a data distribution that makes this possible. As will become clear, this implies a correlation between the number of processors and the size of the numbers x and y that are to be multiplied: it might be necessary to adjust the size of the input by adding trailing zeroes (powers of 2 play a dominant role here). Although this increases the total computation time of the algorithm, the overall negative effect is expected to be rather small for larger numbers.

It is also assumed x and y have equal size; if not, their size is made equal by adding trailing zeroes to the smallest number. The goal is to show a scalable and efficient parallel implementation is possible, and if it is for equally large numbers, it can be made efficient for numbers of unequal size as well. Besides, if one of the numbers is small in size, there is no point in using the Karatsuba algorithm (only a few recursive steps, i.e. a very limited number of split-ups, is possible); one might just as well use the classical multiplication instead.

Another important issue is, which alternative of the Karatsuba algorithm to use. As pointed out in section 1 on page 5, there are two alternatives: computing the sum of the lower and upper part (first alternative) or computing their absolute difference (second alternative). For reasons explained before, the second alternative is used in the implementation of the sequential algorithm. For implementation of the parallel part however, the first alternative is used. The rationale for this choice is, that it turns out to be possible to compute all additions while postponing the processing of carries. Provided a suitable data distribution is chosen, the first step of the algorithm can be fully executed with only one (on average) communication step at the end, in order to perform (the slightly adapted) processing of carries. A similar approach is not possible when adopting the second alternative, because the largest of two intermediate results has to be determined before computing the absolute differences, a process that cannot be delayed and also requires the immediate processing of borrows. Applying the second alternative therefore implies two (on average) communication steps at each level. On the other hand, the additions might result in an overflow

carry, increasing the size of the numbers to be multiplied by one digit, making it odd. This causes the subsequent sequential Karatsuba's to be less efficient, although the effect for large numbers is on average rather small.

If the number of processors is not a power of 3, a solution must be found to prevent a possibly unbalanced computational load in the second step: at the lowest level $L = l$ (i.e. after splitting the data l times), there are 3^l multiplications to be computed using the sequential, recursive Karatsuba algorithm, and p processors are available to do the job. A solution for this will be provided.

Notational issues: the product to be computed is $z = x\,y$. This is the top level of the tree: level $L = 0$. At level $L = 1$, both x and y are split into a lower and an upper part; using superscripts these will be denoted by x^0, y^0 for the lower half (i.e. the lower entries of the array, the least significant digits) and x^1, y^1 for the upper half of x and y respectively. The sums are defined as $x^2 = x^0 + x^1$ and $y^2 = y^0 + y^1$. After the next split, a second index is used to denote the data at level $L = 2$: x^{00}, x^{01} and x^{02} for the lower and upper half of x^0 and their sum; x^1 splits into x^{10}, x^{11} and x^{12}; x^2 into x^{20}, x^{21} and x^{22}. The same structure is used to describe the y-variables at this level. Finally, at level $L = l$, both x and y are split into 3^l parts and their sums, denoted as $x^{i_1 i_2 \cdots i_l}$ and $y^{i_1 i_2 \cdots i_l}$ respectively, $i_j = 0, 1, 2$. Referring to the array-representation on page 5, the product z is split into 3 parts: the lower part z_l contains the first $\frac{1}{4}$ array entries of the product z, the middle part z_m contains the middle $\frac{1}{2}$ array entries, and the upper part z_u the last $\frac{1}{4}$ entries. Using the same superscripts to refer to the level of these products, one gets $z^{i_1 i_2 \cdots}$ for the product $x^{i_1 i_2 \cdots} * y^{i_1 i_2 \cdots}$ and $z_l^{i_1 i_2 \cdots}$, $z_m^{i_1 i_2 \cdots}$, and $z_u^{i_1 i_2 \cdots}$ for the lower, middle, and upper parts of the product $z^{i_1 i_2 \cdots}$ at the specified level.

2.2 Data distributions

Common data distributions applied in parallel software are the cyclic, the block and the block-cyclic distribution[2]. The processor numbering in the definitions below is arbitrary, i.e. the numbering can be randomly permuted without affecting the performance on the assumed hardware.

Cyclic distribution: let x be a vector of size n and p the number of processors available in the parallel process. The cyclic distribution assigns the vector element x_i to processor $i \bmod p$ $(0 \leq i < n)$.

Block distribution: let x be a vector of size n and p the number of available processors. Define the block-size $\alpha = \lceil \frac{n}{p} \rceil$. The block distribution assigns the vector element x_i to processor $i \,\mathtt{div}\, \alpha$ $(0 \leq i < n)$.

1D Block-cyclic distribution: the 1-dimensional block-cyclic distribution assigns blocks instead of individual vector elements to a processor. Let x be a vector of size n and p the number of available processors. The 1-dimensional block-cyclic distribution with block-size α assigns the vector element x_i to processor $(i \,\mathtt{div}\, \alpha) \bmod p$ $(0 \leq i < n)$.

2D Block-cyclic distribution: commonly used in matrix calculations, this distribution assigns rectangular submatrices instead of individual matrix elements to a processor. Let A be an $n \times n$-matrix and let $p = M \times N$ be the number of available processors. The 2-dimensional $M \times N$ block-cyclic distribution with block-size $\alpha_0 \times \alpha_1$ assigns the matrix element A_{ij} to processor $P(s, t)$, where $s = (i \,\mathtt{div}\, \alpha_0) \bmod M$, $t = j \,\mathtt{div}\, \alpha_1 \bmod N$ $(0 \leq i, j < n)$. Here the pair (s, t)

merely serves as a set of processor identifiers and stems from the idea of numbering the processors in matrix calculations by a row-identifier s and a column identifier t; if two index pairs (i_1, j_1) and (i_2, j_2) are mapped to the same pair of identifiers (s, t), the corresponding matrix elements $A_{i_1 j_1}$ and $A_{i_2 j_2}$ are assigned to the same processor $P(s, t)$. The 2D $1 \times N$ block-cyclic distribution for a $1 \times N$ matrix is equivalent with the 1D block-cyclic distribution for a row vector of length n.

Choice of data distribution The numbers x, y and $z = x * y$ in this implementation are stored in 1-dimensional arrays. The exact structure of these arrays will be described later; the focus now is on the best choice for distributing the input x and y over the available processors.

It turns out that the 1D block-cyclic distribution makes it possible to perform all additions in the first step of the parallel implementation without communication (apart from the processing of carries).

x^0						x^1					
x^{00}			x^{01}			x^{10}			x^{11}		
0	1	2	0	1	2	0	1	2	0	1	2

block-cyclic distribution

In this figure, blocks of size α are cyclicly assigned to 3 processors numbered 0, 1 and 2. Denote the 12 blocks in this picture $\alpha_0, \alpha_1, \ldots, \alpha_{11}$. To accomplish the first level addition $x^0 + x^1$, $\alpha_i + \alpha_{i+6}$ $(0 \leq i < 6)$ must be computed locally. As is clear from the picture, these additions can be completed without communication; each processor owns the terms that must be added. At the second level, $x^{00} + x^{01}$ and $x^{10} + x^{11}$ must be computed, this means adding $\alpha_i + \alpha_{i+3}$ and $\alpha_{i+6} + \alpha_{i+9}$ $(0 \leq i < 3)$ locally. Again all processors own the terms needed for the calculations.

2.3 Description of the parallel implementation

Step 1.1: determination of parameters
To determine the block size α, the number of times the data is split (or the lowest level) must be determined first. The lowest level $L = l$ is related to the number of available processors (p), the number of multiplications to be computed at this level (3^l) and the required efficiency for computing the sequential Karatsuba's.

Computational load balancing: let p be the number of available processors and l the level at which the sequential recursive Karatsuba's are (locally) computed. Assume in each computation cycle p processors are used simultaneously. The number of cycles to complete 3^l Karatsuba's is $C_l = \lceil \frac{3^l}{p} \rceil$, since 3^l is the total number of Karatsuba's to be computed. The efficiency is defined as $E_l = \frac{3^l}{C_l\, p}$. The best possible efficiency $E_l = 1$ is reached with p a power of 3, or else as l goes to infinity. This definition thus reflects how well (indeed how efficient) the available processor capacity is used.

The lowest level $L = l$ is computed using this efficiency definition. Input for this calculation are two (stored) parameters: $E_{min} \leq 1$, the desired minimal

efficiency, and l_e, used to determine the lowest level allowed, i.e. the maximum number of splits: $l_{low} = \lfloor \log_3 p \rfloor + l_e$ (the parameter l_e prevents splitting up the data to a very low level, only to gain a small fraction in efficiency). A few more conditions are used in the actual calculation, amongst others a limit that prevents splitting up the data to a level where the multiplicands become smaller than the size n_Cs, the limit size for applying the Classical sequential multiplication algorithm.

The efficiency E_i is computed for $i = 1, 2, \ldots$, until $E_i \geq E_{min}$ or $i > l_{low}$. If $E_i \geq E_{min}$, $l = i$, else l is the smallest i corresponding to the highest calculated efficiency (the latter implies the lowest level allowed is insufficient to reach the desired efficiency). Note that l always takes the minimum value to meet the required efficiency; if $E_{l+1} = E_l$, the extra level would increase the required memory for the additions in step 1.3 as well as communication costs in step 2.1 by a factor $\frac{3}{2}$, without gaining computational efficiency in step 2.2.

Take as an example $p = 64$. Then $E_3 = \frac{27}{1*64} \approx 0.422$, $E_4 = \frac{81}{2*64} \approx 0.633$, $E_5 = \frac{243}{4*64} \approx 0.949$, $E_6 = E_5 = \frac{729}{12*64} \approx 0.949$, $E_7 = \frac{2187}{35*64} \approx 0.976$. Had we chosen $E_{min} = 0.95$, then $L = 7$, provided at least 4 extra levels are allowed ($l_e \geq 4 \Rightarrow l_{low} \geq \lfloor \log_3 p \rfloor + l_e = 3 + 4 = 7$). Had we chosen $E_{min} = 0.95$ and only 3 extra levels are allowed,($l_e = 3 \Rightarrow l_{low} = \lfloor \log_3 p \rfloor + l_e = 3 + 3 = 6$), then $L = 5$, the level corresponding to the maximum efficiency possible with the given restrictions (although $E_5 = E_6$, the obvious choice here is $L = 5$, since there is nothing to gain splitting the data one level more).

Block size α of the block cyclic distribution: let p be the number of available processors and l the level up to which the data is split. Let n be the size of the numbers x and y that are to be multiplied. The block size applied is equal to $\alpha = \lceil \frac{n}{2^l p} \rceil$. If n is not a multiple of $2^l p$, trailing zeroes are added to make x and y of size $n' = \lceil \frac{n}{2^l p} \rceil * 2^l p$. This n' is denoted the size n of the input x and y in the remainder of this section.

Note that this block size α is the maximum possible size for which no communication is needed in the first step of the algorithm. Smaller block sizes are possible, but lead to more carry communication and, more importantly, to larger computational overhead.

Step 1.2: data distribution
Blocks of size α are cyclicly assigned to processors $0, \ldots, p-1$.

Conversion between global and local indices: the digits of the numbers x and y are indexed $x_0, x_1, \ldots x_{n-1}$ and $y_0, y_1, \ldots y_{n-1}$ respectively. Denote the global index by j and the local index by i. The digit globally indexed by j is assigned to processor $s = (j \,\mathtt{div}\, \alpha) \bmod p$. The conversion formulas are (i) global $\rightarrow$ local: $i = (j \,\mathtt{div}\, \alpha p)\alpha + j \bmod \alpha$ and (ii) local $\rightarrow$ global: $j = (i \,\mathtt{div}\, \alpha)\alpha p + \alpha s + i \bmod \alpha$. Similarly there are conversion formulas for blocks: the blocks of the numbers x and y are indexed $x_0, \ldots x_{(n/\alpha)-1}$ and $y_0, \ldots y_{(n/\alpha)-1}$. Substituting $\alpha = 1$ in the former formulas gives the equivalent block conversion formulas for the block-cyclic distribution: the block globally indexed by j is assigned to processor $s = j \bmod p$. The conversion formulas are then (iii) global $\rightarrow$ local: $i = j \,\mathtt{div}\, p$ and (iv) local $\rightarrow$ global $j = ip + s$. After distributing the data, each processor owns $\frac{n}{\alpha p}$ blocks of size α of both x and y.

Data storage: the numbers x and y are stored in two local arrays Xl and Yl, containing only the lowest level blocks of size α, in the same order as when reading the lowest level in the tree from left to right: $x^{00\ldots0}, x^{00\ldots1}, x^{00\ldots2}, x^{00\ldots10}, \ldots, x^{22\ldots0}, x^{22\ldots1}, x^{22\ldots2}$ (the upper indices in $x^{i_0 i_1 \ldots i_l}$ run from 0 to 2 cyclicly starting with i_l (the index referring to the lowest level l, i.e. the right most index in the above notation), and adding one to the previous index at the completion of the right neighbor cycle).

The carries generated when computing the additions, are stored in arrays CX and CY having a similar structure. For each block in Xl and Yl these arrays have one digit reserved to store the carry that has to be processed by the right next processor.

The arrays Xl and Yl contain all information needed to do the computations, and in fact is the minimal memory needed to perform these. The required space for storing the results of the additions is reserved beforehand in Xl and Yl (in fact all elements containing one or more 2's in the upper index). For the sake of simplicity, CX and CY have space reserved to store carries for all blocks, although carries can only refer to blocks where the result of an addition is stored (i.e. blocks with a 2 in the upper index); this way there is a direct link between the carry and the block it applies to. For larger numbers it only implies a small, negligible redundance in storing capacity.

To determine the blocks that make up a number at a certain level, a set of Master Indices MI is created. This array reflects the local data structure and is used for distributing x and y and in the algorithm to compute the additions.

Generation of Master Indices MI: start with the number 0, and compute the first offset $d = 3^0$. Then copy what you already have, adding this offset d: 0, 1. Compute the next offset $d = 3^1$ and again copy what you already have, now adding the newly computed offset: 0, 1, 3, 4. Proceed until the last offset 3^{l-1}. The size of this array is 2^l. If $l = 4$, the array MI contains the numbers: 0, 1, 3, 4, 9, 10, 12, 13, 27, 28, 30, 31, 36, 37, 39, 40. These numbers refer to the array-blocks in which the input numbers x and y are stored; the 'holes' between the numbers are the exact spots to store the results of the additions in Xl and Yl and possible carries in CX and CY (and refer to all blocks containing one or more 2's in their upper index).

The algorithm for generating the array MI is given below.

Input

l: lowest level, i.e. the number of times the input x and y is split

$f = 1$: factor to determine the length of the starting block

Output

$MI[]$: array of Master Indices; determines the local data structure

Algorithm

```
For i = 0 to f − 1 do
    MI[i] := i; (first level: k = 0)
For k = 1 to l do
    i := f ∗ 2^(k−1); (first free entry in MI)
    d := f ∗ 3^(k−1); (offset for level k)
    For j = 0 to i − 1 do (copy from previous indices)
        MI[i + j] := MI[j] + d;
```

Step 1.3: local computation of all level additions
To compute the level-1 primary addition $x^2 = x^0 + x^1$, a number of level-l block-additions must be executed, because only the lowest level l is stored in array Xl. For instance, if the lowest level $l = 3$, computing x^2 requires a total of 4 level-3 block-additions: $x^{200} = x^{000} + x^{100}$, $x^{201} = x^{001} + x^{101}$, $x^{210} = x^{010} + x^{110}$, and $x^{211} = x^{011} + x^{111}$. At level 2, the 3 primary additions to compute are $x^{02} = x^{00} + x^{01}$, $x^{12} = x^{10} + x^{11}$, and $x^{22} = x^{20} + x^{21}$. Each of them requires 2 level-3 block-additions: computing x^{02} implies $x^{020} = x^{000} + x^{010}$ and $x^{021} = x^{001} + x^{011}$; x^{12} is obtained by $x^{120} = x^{100} + x^{110}$ and $x^{121} = x^{101} + x^{111}$; x^{22} requires the block-additions $x^{220} = x^{200} + x^{210}$ and $x^{221} = x^{201} + x^{211}$. Finally, level 3 has 9 primary additions, each of them requiring 1 block-addition: $x^{002} = x^{000} + x^{001}$; ...; $x^{222} = x^{220} + x^{221}$.

The primary additions are processed level by level, starting with the level-1 additions, up to the lowest level l. Referring to the above example $l = 3$, the relation between the primary additions for a specific level and the required block-additions is shown: for each sum the level of computation is given.

x^0			x^0					
x^{00}			x^{01}			x^{02}		
x^{000}	x^{001}	x^{002}	x^{010}	x^{011}	x^{012}	x^{020}	x^{021}	x^{022}
input	input	$l = 3$	input	input	$l = 3$	$l = 2$	$l = 2$	$l = 3$

x^1			x^1					
x^{10}			x^{11}			x^{12}		
x^{100}	x^{101}	x^{102}	x^{110}	x^{111}	x^{112}	x^{120}	x^{121}	x^{122}
input	input	$l = 3$	input	input	$l = 3$	$l = 2$	$l = 2$	$l = 3$

x^2			x^2					
x^{20}			x^{21}			x^{22}		
x^{200}	x^{201}	x^{202}	x^{210}	x^{211}	x^{212}	x^{220}	x^{221}	x^{222}
$l = 1$	$l = 1$	$l = 3$	$l = 1$	$l = 1$	$l = 3$	$l = 2$	$l = 2$	$l = 3$

addition process for $l = 3$

In general, the number of primary additions at level k is 3^{k-1}, each primary addition requiring 2^{l-k} block-additions. The following algorithm executes the level-1 to level-l additions according to the description above.

Input
$Xl[]$ and $Yl[]$=arrays containing the lowest level numbers (of size α)
$CX[]$ and $CY[]$=arrays containing the carries; initially set to zero
$MI[]$=array of Master Indices
Output
$Xl[]$ and $Yl[]$=updated array of lowest level numbers
$CX[]$ and $CY[]$=updated array of carries
Algorithm

```
01  For k = 1 to l do
02      npa := 3^(k-1); (number of primary additions at level k)
03      nba := 2^(l-k); (number of block additions per primary at level k)
04      For i = 0 to npa - 1 do
05          d := i * 3^(l-k+1);
06          For j = 0 to nba - 1 do
07              i0 = MI[j] + d;
08              i1 = MI[j] + 3^(l-k) + d;
09              i2 = MI[j] + 2 * 3^(l-k) + d;
10              Xl[i2] := Xl[i0] ⊕ Xl[i1];
11              CX[i2] := CX[i0] + CX[i1] + carry;
12              Yl[i2] := Yl[i0] ⊕ Yl[i1];
13              CY[i2] := CY[i0] + CY[i1] + carry;
```

Note 1: the indices $i0$, $i1$ and $i2$ are block-indices: they refer to blocks
Note 2: the $\oplus$ in lines 10 and 12 is a block-addition
Note 3: the carry in lines 11 and 13 results from the previous block-addition

Step 1.4: carry handling
As mentioned before, processing of carries is delayed until the last, lowest-level additions are executed. This asks for a careful handling of these carries, especially because only the lowest level l is stored in Xl and Yl. It turns out to be possible to store the carries in arrays with a similar structure, and have these carries refer to only the lowest level additions.

When computing the first level addition $x^2 = x^0 + x^1$, the (lowest-level) block-additions generate carries that need to be stored, since processing of all carries is postponed. These carries must be taken into account, when computing the second level addition $x^{22} = x^{20} + x^{21}$, where the lower (x^{20}) and upper (x^{21}) part of x^2 are added: carries generated at the higher level, accumulate to lower levels. The following example illustrates this accumulation of carries.

x^{000}	x^{001}	x^{002}	x^{010}	x^{011}	x^{012}	x^{020}	x^{021}	x^{022}
input	input	c_{10}	input	input	c_{11}	c_4	c_5	$c_{4,5,12}$

x^{100}	x^{101}	x^{102}	x^{110}	x^{111}	x^{112}	x^{120}	x^{121}	x^{122}
input	input	c_{13}	input	input	c_{14}	c_6	c_7	$c_{6,7,15}$

x^{200}	x^{201}	x^{202}	x^{210}	x^{211}	x^{212}	x^{220}	x^{221}	x^{222}
c_0	c_1	$c_{0,1,16}$	c_2	c_3	$c_{2,3,17}$	$c_{0,2,8}$	$c_{1,3,9}$	$c_{0,1,2,3,8,9,18}$

accumulation of carries; compare with addition process on page 15

It is assumed the order in which the additions is executed is the same, and at level $l = k$ the number of (possible) carries generated is equal to $3^{k-1} * 2^{l-k}$. The carries are numbered $c_0, \ldots, c_{18}$: $1 * 4 = 4$ carries $(c_0, \ldots, c_3)$ are generated during execution of the primary level-1 addition $x^2 = x^0 + x^1$, $3 * 2 = 6$ carries $(c_4, \ldots, c_9)$ emerge from the 3 level-2 primary additions, and $9 * 1 = 9$ carries $(c_{10} \ldots, c_{18})$ result from the 9 level-3 primary additions. Multiple indices are used to denote accumulation of carries: $c_{4,5,12}$ denotes accumulation of carries c_4, c_5, and c_{12}.

The size of a carry is equal to 1 digit, but its value can be larger than 1 due to this accumulation of carries; its maximum value is equal to $\sum_{i=1}^{l} 2^{i-1} = 2^l - 1$. For practical purposes, 1 digit is sufficient to contain this maximum, provided the chosen radix is large enough ($\geq 2^{16}$ is more than sufficient).

Once these carries are stored properly, the processing of these carries is rather straightforward: processor s sends the local carry-arrays CX and CY to processor $t = (s + 1) \mod p$. Most of the carries in CX and CY are likely to have a value unequal 0, so there is not much, if anything at all, to gain by communicating only the non-zero entries, since this needs communicating the indices as well. The carries are processed by processor t in a first cycle C_0, possibly (but unlikely) resulting in new carries that need processing in a next cycle of communication and computation. Since new carries are rare, only the newly generated carries are communicated, instead of arrays CX and CY. This process repeats until there are no more carries to communicate and process; this is controlled by means of an array, that indicates if any new carries are generated by any processor (note that it is possible that a processor t has no carries to process in some cycle C_i, but receives carries in a next cycle C_j.

If the last processor $s = p - 1$ generates a carry, this is an overflow carry that will increase the size of the global block by 1 digit. All carries generated by the last processor are sent to processor $t = 0$, whose only role it is to save and accumulate the received carries (since processor 0 cannot receive carries that need further processing, it serves as the keeper of these overflow carries, thus saving memory).

One important issue remains to be solved: in a sequential implementation, carries are (naturally) propagated all the way to the most significant digit at all levels, whereas in this parallel implementation carries are only propagated to the most significant digit of the level-l global blocks (of size αp). The question is: is the partial propagation of carries in this parallel implementation sufficient to make it a valid one? The answer is yes, simplifying things noticeably.

Let $x = x_0 + x_1 r^m$ and $y = y_0 + y_1 r^m$ be two global variables at level k with $m < 2^{l-k} \alpha p$; underscores are used here to denote the lower and upper parts of x and y, the first k indices are left out (e.g. for $k = 1$ two of these variables are actually $x_2 = x_{20} + x_{21} r^m$ and $y_2 = y_{20} + y_{21} r^m$). Their product $z = x\,y$ is obtained using $z_0 = x_0\,y_0$, $z_1 = x_1\,y_1$ and $z_2 = x_2\,y_2 = (x_0 + x_1)(y_0 + y_1)$, the products at level $k + 1$ (e.g. for $k = 1$ the product $z_2 = x_2\,y_2$ is obtained using the level-2 products $z_{20} = x_{20}\,y_{20}$, $z_{21} = x_{21}\,y_{21}$ and $z_{22} = x_{22}\,y_{22}$). Assume full propagation of carries is applied for these level-k variables when executing the additions at this level (e.g. for $k = 1$, these additions are $x_2 = x_0 + x_1$ and $y_2 = y_0 + y_1$). Let $x' = x'_0 + x'_1 r^m$ and $y' = y'_0 + y'_1 r^m$ be the same variables at level k and assume partial propagation of carries is applied for the primed level-k variables when executing the additions at this level. The lower and upper parts x_0, x_1, y_0, y_1 and their primed counterparts are the global variables at

level $k+1$, possible overflow carries included.

Now write $x'_0 = x_0 + a\,r^m$, $x'_1 = x_1 - a$ and $y'_0 = y_0 + b\,r^m$, $y'_1 = y_1 - b$, carries a and b not having been transferred from x'_0 and y'_0 to their neighboring blocks x'_1 and y'_1 respectively due to the partial carry propagation for the primed variables. Applying the Karatsuba algorithm (see section 1.2 on page 5) to combine the products at level $k+1$ into the product $x'\,y'$ at level k now yields

$$\begin{aligned}
x'\,y' &= x'_0\,y'_0 + \big((x'_0 + x'_1)(y'_0 + y'_1) - x'_0\,y'_0 - x'_1\,y'_1\big)\,r^m + x'_1\,y'_1\,r^{2m} \\
&= (x_0 + a\,r^m)(y_0 + b\,r^m) \\
&\quad + \quad (x_0 + a\,r^m + x_1 - a)(y_0 + b\,r^m + y_1 - b)\,r^m \\
&\quad - \quad (x_0 + a\,r^m)(y_0 + b\,r^m)\,r^m \\
&\quad - \quad (x_1 - a)(y_1 - b)\,r^m \\
&\quad + \quad (x_1 - a)(y_1 - b)\,r^{2m} \\
&= x_0\,y_0 + (x_0\,y_1 + x_1\,y_0)\,r^m + x_1\,y_1\,r^{2m} = x\,y
\end{aligned}$$

the last equation following in a straightforward way, since writing out the multiplications reveals that all other terms cancel.

The above shows, that assembling the product $x'\,y'$ at level k from the products at level $k+1$ gives the correct result $x\,y$ if partial propagation of carries is applied for variables at level k. Note however, that the lower level products $z'_0 = x'_0\,y'_0$, $z'_1 = x'_1\,y'_1$ and $z'_2 = x'_2\,y'_2 = (x'_0 + x'_1)(y'_0 + y'_1)$ may, and often will, have different values when compared to their unprimed counterparts (for which full propagation of carries is applied).

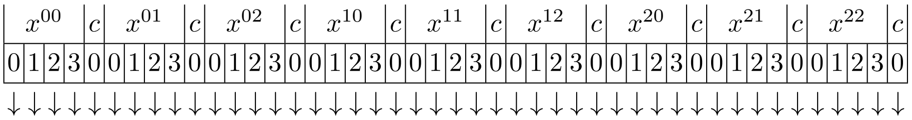
This proves partial propagation makes the implementation as described valid. The final result, the product at the highest level $L = 0$, is correct, irrespective of the method of carry propagation, full or partial: although the assembled products at levels $L = i$, $1 < i \leq l$, are likely to have different values when comparing full and partial propagation of carries, the final result $z = x\,y$ at the top level $L = 0$ is always the same.

Step 2: redistribution for the local sequential Karatsuba's

After all level-l blocks are generated through the above addition process, a redistribution of data is needed to have processors compute all level-l products $z^{i_1 i_2 \ldots i_l} = x^{i_1 i_2 \ldots i_l} * y^{i_1 i_2 \ldots i_l}$ applying sequential, recursive Karatsuba's. This implies constructing global blocks of size $\alpha p + 1$ (an extra digit for a possible overflow carry included), destined for a single processor, from the local blocks of size α, each of them owned by a different processor (the overflow carry being positioned on processor 0).

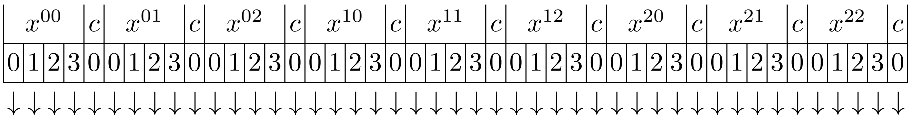
There are several options for this redistribution. One option is to distribute the (global) blocks cyclicly over the available processors. The picture on the next page illustrates this for the redistribution of 9 level-2 blocks over 4 processors. A "c" denotes a possible carry, only 1 digit in size, usually much smaller than the neighboring blocks in this picture, αp digits in size. Applying the cyclic distribution is straightforward and maximal computational load balance is guaranteed.

A second option for this redistribution is to apply a block-size of $3(\alpha p + 1)$, i.e. 3 global blocks. Although processor 3 is idle during the execution of the local sequential Karatsuba's, the computational efficiency is the same for both distributions: 3 cycles are needed to complete the 9 multiplications.



x^{00}	x^{01}	x^{02}	x^{10}	x^{11}	x^{12}	x^{20}	x^{21}	x^{22}
0	1	2	3	0	1	2	3	0

cyclic distribution



x^{00}	x^{01}	x^{02}	x^{10}	x^{11}	x^{12}	x^{20}	x^{21}	x^{22}
0	0	0	1	1	1	2	2	2

distribution in blocks

After completing the local sequential Karatsuba's, the lower level products must be combined into higher level products. Since assembling a product at level $k-1$ needs 3 level-k neighboring products, applying the cyclic distribution requires an immediate redistribution of data after the execution of the local sequential Karatsuba's, while applying the block distribution or the block cyclic distribution with a proper block-size makes it possible to postpone this redistribution: the assembling of the products can partly be performed locally.

Delaying the redistribution makes it possible to apply a larger block size β in the final step of the parallel algorithm, assembling the higher level products by parallel execution of additions and subtractions (see page 20). But applying carry-handling in the last step in a similar way as described before, a larger block size is hardly more efficient, if at all: on average only 1 (accumulated) carry per block is processed. Of more importance is the amount of data to be communicated and consequently the memory needed for the final step, if part of the product assembling can be performed locally. For each level that the redistribution of data can be postponed, this amount reduces by a factor of $\frac{3}{2}$, since instead of distributing 3 level-k products of size n_k, only 1 product at level $k-1$ of size $n_{k-1} = 2n_k$ needs to be distributed.

It seems to make sense to try and find the optimal distribution here. But for determining the lowest level (l), and consequently the number of multiplications at this stage (3^l), the parameter E_{min} (the minimal required computational efficiency for the local sequential Karatsuba's) is used, as described on page 12. The algorithm used is such, that l is always the lowest value to meet the required efficiency. This means, that distributing the multiplications in blocks of size 3^k always leads to a lower computational load balance in the next step. If one wishes to allow that, the parameter E_{min} can simply be set to a lower value, thus reducing the memory needed in the previous step and communication costs in the current step by a factor $(\frac{3}{2})^k$, as well as reducing the number of multiplications by a factor of 3^k (which makes local assembling of products no longer relevant, since there are not enough products to distribute). There is no need to search for another than the cyclic distribution, and the parameter E_{min} is all that is needed to exchange computational efficiency in the next step for

memory efficiency in the previous and communication costs in the current step, while at the same time optimizing memory and communication costs in the next step becomes irrelevant, the number of products to distribute being too small.

Referring to the example given: the computational efficiency for computing the sequential Karatsuba's in the next step is equal to $E_2 = \frac{9}{3*4} = 0.75$. But if E_2 meets the required efficiency E_{min}, then so would $E_1 = \frac{3}{1*4} = 0.75$. In other words, the lowest level l would have been 1, not 2, the number of multiplications to distribute 3 instead of 9: the given example would not occur and a distribution in blocks is no longer relevant.

As a consequence, the cyclic distribution will always be applied.

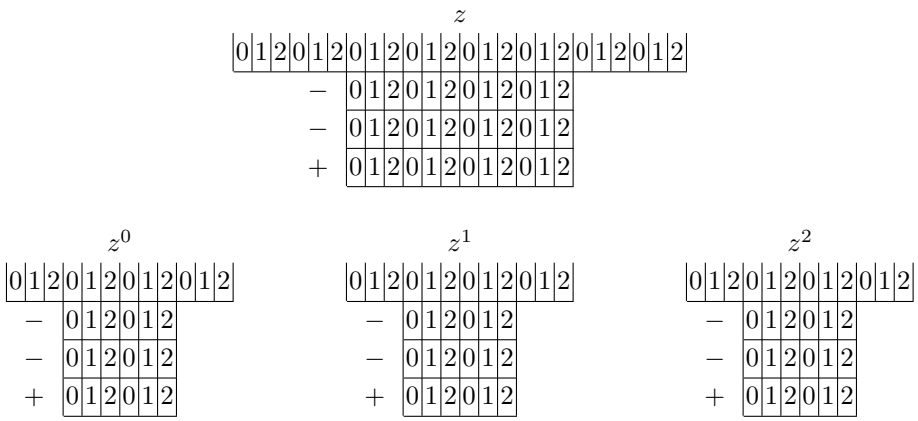
Step 3: execution of the local sequential Karatsuba's

As discussed in section 1, a parameter n_Cs is used to determine when to switch from the sequential Karatsuba to the sequential classical algorithm. To make sure the global blocks that serve as input for the local sequential Karatsuba's can be repeatedly split into half until this limit size n_Cs is reached, the size of these blocks is adjusted to make them contain the required powers of 2. Let p be the number of available processors, α the block-size, $c = 1$ if overflow carries occurred, $c = 0$ if not, and n_Cs the limit size for switching to the classical algorithm. Then the adjusted size of the global blocks is equal to $n\_Ks = \lceil \frac{\alpha p + c}{2^h} \rceil * 2^h$ with $h = \lceil \log_2 \frac{\alpha p + c}{n\_Cs} \rceil$ the required powers of 2.

The level-l products $z^{i_1 i_2 \ldots i_l} = x^{i_1 i_2 \ldots i_l} * y^{i_1 i_2 \ldots i_l}$ are computed using the sequential recursive Karatsuba algorithm as described in section 1.3 on page 7. The size of each of the products is equal to $2(\alpha p + 1)$, filled up with zeroes up to size $2n\_Ks$.

Step 4: redistribution for the parallel assembling

After the execution of the local sequential Karatsuba's, 3^l products $z^{i_1 i_2 \ldots i_l}$ are waiting to be redistributed for the assembling of the higher level products by parallel execution of additions and subtractions.



distribution with $p = 3$

The structure of the assembling process for $l = 2$ is illustrated above, again applying the block-cyclic distribution in order to avoid communication in be-

tween the levels (see also section 2.2 on page 12). The numbers denote the processors, here $p = 3$.

As is clear from this example, all processors are owners of the correct blocks, and the assembling can be done up to the top level $L = 0$ without communication. To make the above structure possible and to guarantee a correct execution of additions and subtractions in the next step, one must realize that the problem has become asymmetric due to the overflow carries that have emerged in the first step of the algorithms (the additions 'downwards'). The products computed in the previous step have gained two extra digits due to these overflow carries, and in order to guarantee the correct multiplication factors r^m and r^{2m} (see section 1.2.1) are applied, the following procedure holds for determining the block size β and the content of each of the blocks.

1 Split all level-l products (of size $2\alpha p + 2$) into a lower part of size αp and an upper part of size $\alpha p + 2$
2 Make the size of one global block a multiple of p: $n\_Kp = \lceil \frac{\alpha p+2}{p} \rceil * p$
3 Compute the block size: $\beta = \frac{n\_Kp}{p}$
4 Fill the odd global blocks (of size βp) with the first αp digits (digit 0 through $\alpha p - 1$) of the products; add zeroes to fill up
5 Fill the even global blocks (of size βp) with digits αp through $2\alpha p + 1$ of the products; add zeroes to fill up

The blocks are redistributed applying the block-cyclic distribution with block size β. The last 2 digits (and possible trailing zeroes) of the even global blocks are denoted overflow-blocks and are processed in the final stage of the algorithm: generating the output from the local arrays (see the last step 5.2 of the algorithm).

The used data structure for storing the products in the last step is similar to the one applied in the first step: a local array Zl is used, and an array CZ to store the carries. Again a set of Master Indices (MI) is used to determine which blocks make up a product at a certain level. Because each number (product) is now made up of twice as many blocks, this array MI must be generated again, using an algorithm analogous to the one given before (in the first step of the algorithm), only the input factor $f = 2$ here.

Generation of Master Indices MI: start with the number 0 and 1, and compute the first offset $d = 2 * 3^0$. Then copy what you already have, adding this offset d: 0, 1, 2, 3. Compute the next offset $d = 2 * 3^1$ and again copy what you already have, now adding the newly computed offset: 0, 1, 3, 4, 6, 7, 8, 9. Proceed until the last offset $2 * 3^{l-1}$. The size of this array is 2^{l+1}. If $l = 4$, the array MI contains the numbers: 0, 1, 3, 4, 6, 7, 8, 9, 18, 19, 20, 21, 24, 25, 26, 27, 54, 55, 56, 57, 60, 61, 62, 63, 72, 73, 74, 75, 78, 79, 80, 81. They refer to the blocks that make up a product generated at level k. The 'holes' in this array are the exact spots of the products that become irrelevant in the process (and refer to all blocks containing one or more 2's in their upper index).

Step 5.1: parallel assembling of the higher level products
To explain the procedure, take as example $l = 3$. Denote the lower, middle, and upper part of a product z (of size $4n_z$) z_l (size n_z), z_m (size $2n_z$), and z_u (size n_z) respectively.

To assemble the product z^{00} at level 2, $z_m^{00} = z_m^{00} - z^{000} - z^{001} + z^{002}$ must be computed, each of these terms containing two blocks; beforehand z^{000} is equivalent with z_l^{00} and z^{001} with z_u^{00}; after this computation the original products z^{000} and z^{001} are destroyed, whereas the product z^{002} has become irrelevant for further calculations. The product z^{00} is now made up of the blocks z_l^{00}, z_m^{00}, and z_u^{00}. In a similar way the products $z^{01}, \ldots, z^{22}$ are assembled, 9 products in total at level 2, so there are 9 primary additions/subtractions to be calculated, each of them consisting of 2 block-additions.

To compute the level-1 product z^0, $z_m^0 = z_m^0 - z^{00} - z^{01} + z^{02}$ must be computed; the product z^0 is then made up of the blocks z_l^0, z_m^0, and z_u^0. The products z^1 and z^2 are produced likewise, 3 products in total for this level, or 3 primary additions/subtractions each consisting of 4 block-additions.

The final result is obtained by $z_m = z_m - z^0 - z^1 + z^2$: 1 primary addition/-subtraction consisting of 8 block-additions.

The addition/subtraction process is illustrated here. The logical view (for $l = 2$) is shown first, the data structure of array Zl is given on the next page (for $l = 3$).

z

z_l^0	z_u^0	z_l^1	z_u^1
$-$	z_l^0	z_u^0	
$-$	z_l^1	z_u^1	
$+$	z_l^2	z_u^2	

z^0

z_l^{00}	z_u^{00}	z_l^{01}	z_u^{01}
$-$	z_l^{00}	z_u^{00}	
$-$	z_l^{01}	z_u^{01}	
$+$	z_l^{02}	z_u^{02}	

z^1

z_l^{10}	z_u^{10}	z_l^{11}	z_u^{11}
$-$	z_l^{10}	z_u^{10}	
$-$	z_l^{11}	z_u^{11}	
$+$	z_l^{12}	z_u^{12}	

z^2

z_l^{20}	z_u^{20}	z_l^{21}	z_u^{21}
$-$	z_l^{20}	z_u^{20}	
$-$	z_l^{21}	z_u^{21}	
$+$	z_l^{22}	z_u^{22}	

product generation for $l = 2$

In general, there are 3^k primary additions/subtractions for assembling products at level k, each of them requiring 2^{l-k} block additions. Each level up, the first $\frac{1}{4}$ of the final product that is generated, becomes a factor 2 larger. This means, the addition/subtractions start at block 2^{l-k-1} plus some offset d, to account for the products containing a 2 in their index, that become obsolete in the procedure of assembling higher level products.

Processing of carries is postponed until the final product is produced. The procedure is the same as described before in the first step of the algorithm on page 16, the only difference being that a (accumulated) carry can be negative here (in fact a borrow).

The following algorithm executes the assembling of products, starting at level $L = l - 1$ up to the top level $L = 0$.

z_l						z_l											
z_l^0						z_u^0											
z_l^{00}		z_u^{00}				z_l^{01}		z_u^{01}				z_l^{02}		z_u^{02}			
z_l^{000}	z_u^{000}	z_l^{001}	z_u^{001}	z_l^{002}	z_u^{002}	z_l^{010}	z_u^{010}	z_l^{011}	z_u^{011}	z_l^{012}	z_u^{012}	z_l^{020}	z_u^{020}	z_l^{021}	z_u^{021}	z_l^{022}	z_u^{022}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

z_u						z_u											
z_l^1						z_u^1											
z_l^{10}		z_u^{10}				z_l^{11}		z_u^{11}				z_l^{12}		z_u^{12}			
z_l^{100}	z_u^{100}	z_l^{101}	z_u^{101}	z_l^{102}	z_u^{102}	z_l^{110}	z_u^{110}	z_l^{111}	z_u^{111}	z_l^{112}	z_u^{112}	z_l^{120}	z_u^{120}	z_l^{121}	z_u^{121}	z_l^{122}	z_u^{122}
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

z_l^2						z_u^2											
z_l^{20}		z_u^{20}				z_l^{21}		z_u^{21}				z_l^{22}		z_u^{22}			
z_l^{200}	z_u^{200}	z_l^{201}	z_u^{201}	z_l^{202}	z_u^{202}	z_l^{210}	z_u^{210}	z_l^{211}	z_u^{211}	z_l^{212}	z_u^{212}	z_l^{220}	z_u^{220}	z_l^{221}	z_u^{221}	z_l^{222}	z_u^{222}
36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53

array Zl for $l = 3$

Input
$Zl[]$=array containing the lowest level product blocks (of size β)
$CZ[]$=array containing the carries; initially set to zero, can be positive and negative
$MI[]$=array of Master Indices
Output
$Zl[]$=updated array of products
$CZ[]$=updated array of carries
Algorithm

```
01  For k = l − 1 to 0 do
02      Zl_sav[·] = Zl[·]; (make a copy of array Zl level k)
03      CZ_sav[·] = CZ[·]; (make a copy of array CZ level k)
04      npa := 3^k; (number of primary additions at level k)
05      nba := 2^(l−k); (number of block additions per primary at level k)
06      For i = 0 to npa − 1 do
07          d := i * 2 * 3^(l−k);
08          For j = 0 to nba − 1 do
09              i0 = MI[2^(l−k−1) + j] + d;
10              i1 = MI[j] + d;
11              i2 = MI[j] + 2 * 3^(l−k−1) + d;
12              i3 = MI[j] + 2 * 2 * 3^(l−k−1) + d;
13              Zl[i0] := Zl_sav[i0] ⊖ Zl_sav[i1] ⊖ Zl_sav[i2] ⊕ Zl_sav[i3];
14              CZ[i0] := CZ_sav[i0] − CZ_sav[i1] − CZ_sav[i2] + CZ_sav[i3] + carry;
```

Note 1: in the process, Zl and CZ are destroyed, hence the copy in lines 02 and 03
Note 2: the indices $i0$, $i1$, $i2$ and $i3$ are block-indices: they refer to blocks
Note 3: the $\ominus$ and $\oplus$ in line 13 refer to a block
Note 4: the carry in line 14 results from the block-operations in line 13

Step 5.2: write the final result to output
In the final step of the algorithm, the result $z = x\,y$ is produced and written to output. The product is constructed from 2^{l+1} blocks in each of the local arrays Zl; the block numbers are determined by the Master Indices in MI. All blocks that make up the final result are sent to processor 0, and before writing the global blocks to output, some postprocessing is needed with respect to the overflow-blocks (see step 3.1 of the algorithm) and a possible carry from the previous (global) block. The carries originate from processor $p-1$ and are already owned by processor 0 (see the carry handling on page 16).

This postprocessing proceeds as follows: an auxiliary array is filled with digits αp through $\beta p - 1$ from a global block of size βp. This block then contains the 2 overflow-digits generated in the first part of the algorithm, possibly a few additional zeroes (if an odd block is processed, it only contains zeroes). This overflow block is added to the next block. Possible carries are processed in the same procedure. After this procedure, the global block is written to output.

2.4 Communication costs

There are 4 major communication steps, 2 in the sequential part of the algorithm and 2 in the parallel part. The two communication steps in the sequential part are: (i) distributing the input over the available processors and (ii) writing the final result to output. The ones in the parallel part are: (i) redistributing the result of the parallel additions as preparation for execution of the local sequential Karatsuba's and (ii) redistributing the locally calculated products as preparation for the parallel assembling of the higher level products. There are 2 minor communication steps for processing the carries at the end of steps 1 and 5.

Sequential parts of the algorithm: since all data words come from or go to the same processor 0, the number of words this processor sends in the first sequential part and receives in the second sequential part is equal to $2n$ (assuming each digit is one word): n words for distributing x and the same amount for distributing y, about $2n$ words for distributing $z = x\,y$ (in fact slightly more than $2n$ due to some extra trailing zeroes added in the process of the algorithm). The first sequential part requires one synchronization, in the last sequential part synchronization takes place $2^{l+1}p$ times (at level $k = l-1$ the product is made up of $2^{l-k+1} = 4$ global blocks of size βp, at the top level $L = 0$ the product $z = x\,y$ is made up of 2^{l+1} blocks of size βp). To estimate the lowest level l, assume the number of products at level l is about equal to $3p$ to reach sufficient computational efficiency for the local sequential Karatsuba's. Then $l = \lceil \log_3 p \rceil + 1$. The communication costs for the sequential parts of the algorithm total $T_{c,seq} = 4ng + (2^{\lceil \log_3 p \rceil + 2}p + 1)l_{sync}$, where g determines the communication cost per data word, and l_{sync} the global synchronization cost, both machine-dependent parameters[2]. Ignoring the ceiling-function, one gets $T_{c,seq} > 4ng + (2^2 p^{log_3 2} p + 1)l_{sync} \approx 4ng + (4p^{1.63} + 1)l_{sync}$. For large p, synchronization costs during output can therefore be high.

Parallel parts of the algorithm: in the first major redistribution each processor sends and receives $2 * 3^l$ blocks of size $\alpha = \frac{n}{2^l p}$, in total $2 * 3^l * \frac{n}{2^l p} = \frac{2n}{p}(\frac{3}{2})^l$ words. In the second redistribution each processor sends and receives $2 * 3^l$

blocks of size $\beta \approx \alpha$: another $\frac{2n}{p}(\frac{3}{2})^l$ words. Both redistributions require one synchronization. For handling carries, the amount of data communicated is equal to $\frac{2n}{\alpha p}(\frac{3}{2})^l$ (negligible if $\alpha \gg 1$), synchronization takes place $2 * 2 = 4$ times on average. Assuming $\alpha \gg 1$ and estimating $l = \lceil \log_3 p \rceil + 1$, one obtains for the total communication costs for the parallel parts of the algorithm $T_{c,par} = \frac{4n}{p}(\frac{3}{2})^{\lceil \log_3 p \rceil + 1} g + 6l_{sync} < \frac{4n}{p}(\frac{3}{2})^2(\frac{3}{2})^{\log_3 p} g + 6l_{sync} = \frac{9n}{p^{\log_3 2}} g + 6l_{sync}$. These communication costs are therefore essentially linear in n.

3 Results

3.1 Verification tests

In order to show the correctness of the implementation, the produced result $z = x * y$ has been checked for several values of the input size n ($1 \leq n \leq 10^6$), and the number of processors p used ($1 \leq p \leq 244$).

For this purpose, x, y, and z were imported in *Mathematica* and the results have been verified using only standard *Mathematica*-functions.

For al other tests (tuning parameters and determining parallel speedup), the correctness of the implementation is assumed without further verification. The radix is set to $r = 2^{32}$, the largest possible value for using the built-in C-product function (the implementation is found to work correctly for other values of r as well, but smaller radices are less efficient).

3.2 Tuning parameters

The critical value to switch to the classical multiplication In section 1.2.2 the theoretical value for switching to the classical multiplication algorithm is found to be $nCs = 8$: as soon as the size n' of x and y after consecutive splits becomes smaller than or equal to this value ($n' \leq nCs$), the product is computed using the classical algorithm.

A sample of 40 different values for n, $100000 \leq n < 200000$ is used to determine the optimal value for nCs. The results are shown in figure 1.

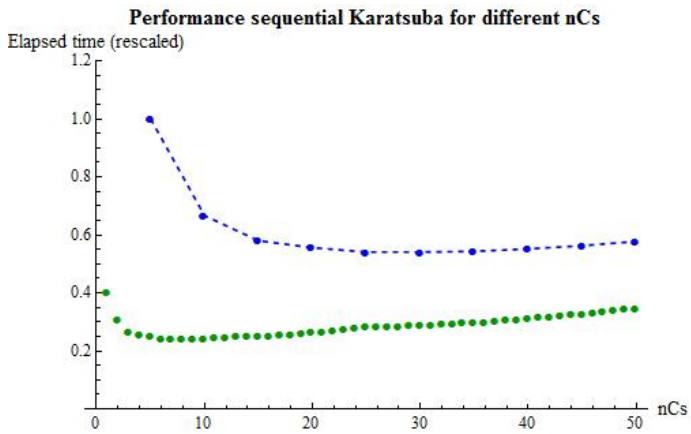


Figure 1: Test results (dashed lines) and theoretical values (dotted green line)

The figure shows a flat behavior in the range 25-35. This can be explained by the fact that theoretical values (the dotted green line) show the same flat behavior. The theoretical values do not refer to run times, but are based on the total number of operations, as derived in section 1.2.2. The only purpose here is to show the shape of the function. It is scaled such, that test values and theoretical values can easily be compared.

Based on these results, $nCs = 30$ is picked as the critical value for switching to the classical multiplication.

The computational efficiency in the sequential Karatsuba's The efficiency reached when executing the local sequential Karatsuba's is the main factor for the total efficiency, and thus for the parallel speedup. It is therefore important to tune the parameters that determine this efficiency accurately.

As outlined in section 2.3 (page 12), two parameters play a role here: E_{min} is the minimal required efficiency, l_e is the maximum number of extra splits that are allowed to reach this efficiency. In the test we set no limit to the number of extra splits allowed, so the required minimal efficiency was always reached. The test is performed using a sample of 3 values of n (1×10^7, 2×10^7, and 3×10^7) for $p = 24$, $p = 64$, and $p = 240$.

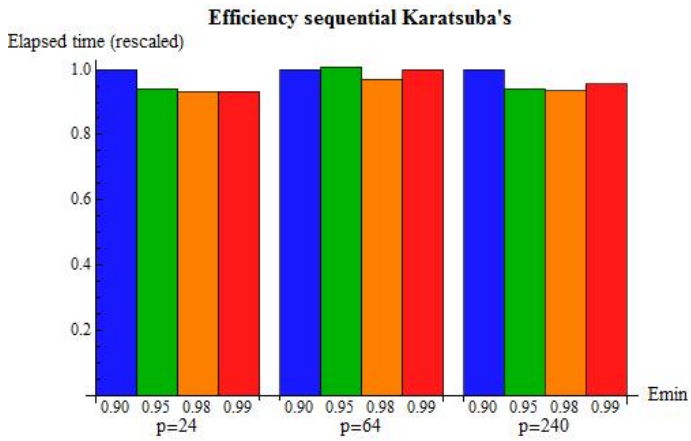


Figure 2: Test results for different values of $Emin$ and p

For each split, the amount of memory allocated grows by a factor $\frac{3}{2}$. It is therefore possible, that a larger minimal efficiency indeed leads to a better load balance when performing the local sequential Karatsuba's, but gives equal or worse results on the overall performance. This may be caused by increased memory access time, or increased communication costs. If the number of processors is relatively small, and the size n of x and y relatively large, too many splits may even cause an abort due to insufficient memory. From the figure above, an efficiency between 0.95 and 0.98 seems optimal.

To better judge the best values for the parameters E_{min} and l_e, the efficiencies that will be actually reached, are computed and given in the table on the right, assuming $E_{min} = 0.95$. The efficiencies are calculated for $l_e = 3$ and $l_e = 4$, for values of p that are relevant for subsequent tests to determine the parallel speedup.

Efficiencies E for $E_{min} = 0.95$

p	E ($l_e = 3$)	E ($l_e = 4$)
1	1	1
4	0.964	0.964
10	0.972	0.972
24	0.920	0.980
27	1	1
32	0.990	0.990
48	0.949	0.990
64	0.949	0.976
81	1	1
96	0.990	0.990
240	0.911	0.976
243	1	1
320	0.976	0.976
480	0.976	0.976
960	0.976	0.976
Mean	0.972	0.984

Based on both test results and computed efficiencies, the parameters that determine the efficiency for the execution of the local sequential Karatsuba's are set to $E_{min} = 0.95$ and $l_e = 4$. This guarantees almost optimal efficiencies at a minimum number of splits.

3.3 Test results

For running programs on the Cartesius parallel computer at the SurfSARA computer center in Amsterdam, two types of batch nodes are available: the so-called thin nodes and fat nodes. The thin nodes have 24 cores and 64GB of memory (per node), whereas the fat nodes have 32 cores and 256 GB of memory (per node). The program is mainly tested on thin nodes (for test results on fat nodes: see section 3.7), for values of p that can be found in the table on page 27. These values are selected such, that a whole number of nodes is reserved. There are two reasons for this approach: (i) Cartesius always gives a job exclusive access to the allocated nodes, so actually using only part of a node is very cost-inefficient (the job is charged for its run time, i.e. wall-clock time, times the number of processors in the node, irrespective of the number of cores actually used), and (ii) using a whole number of nodes might give a better overall performance. Exceptions are $p = 27$, $p = 81$, and $p = 243$, which are selected because they yield a computational efficiency $E = 1$ in the local sequential Karatsuba's. Other exceptions are $p = 4$ and $p = 10$, which are added to have two tests with a small number of processors. One test run for $p = 1$ is required for determining the parallel speedup for the chosen p.

The numbers x and y to be multiplied are randomly generated in all tests.

3.3.1 Run times

Figure 3 shows the run times $\log_{10} T$ (T in milliseconds) for values $1 \leq p \leq 240$, $10^5 \leq n \leq 10^7$. For each p, 10 different problem sizes n are shown. It gives an impression of the processor time this parallel implementation of the Karatsuba algorithm needs to multiply numbers of this size, represented in radix $r = 2^{32}$.

3.3.2 Parallel speedup

Let T_p be the total executing time using p processors and T_{seq} and T_{par} be the sequential and parallel execution time respectively: $T = T_{seq} + T_{par}$. Suppose that r_1 is the relative sequential execution time (the sequential fraction) and $r_2 = 1 - r_1$ the relative parallel execution time (the parallel fraction). The total execution time T_p using p processors now becomes (T_1 is the overall execution time using one processor):

$$T_p = T_{seq} + T_{par} = r_1 T_1 + \frac{1 - r_1}{p} T_1 = \frac{r_1(p-1)+1}{p} T_1$$

By Amdahl's law[1], the parallel speedup S_p is then defined as:

$$S_p = \frac{T_1}{T_p} = \frac{p}{r_1(p-1)+1}$$

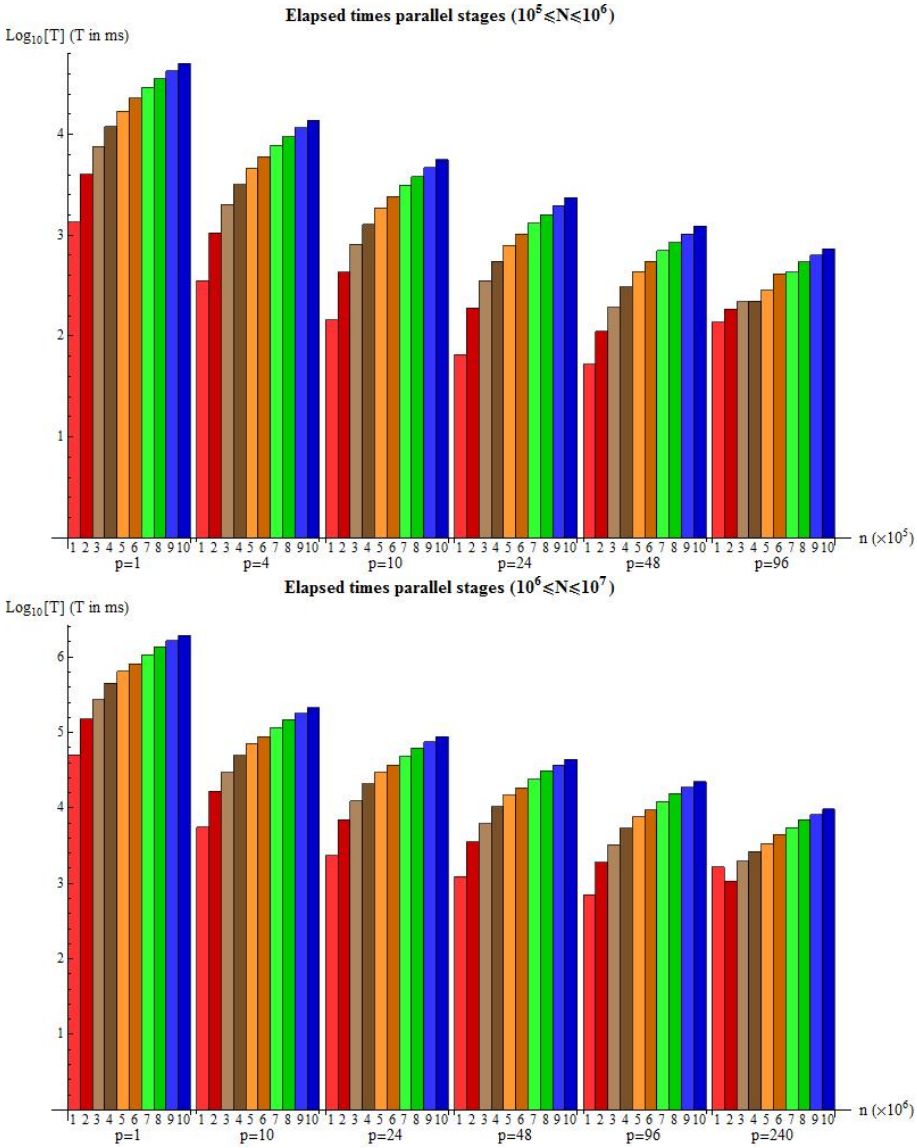


Figure 3: Run times for different p and n

The two sequential communication steps (for distributing the input x and y and collecting the output z) are not taken into account in the calculations for the parallel speedup. For an analysis of this communication, see section 3.5.

In figure 4, the parallel speedups are given for the different n used in the test. In general the speedups reached are satisfactory, provided the number of processors p matches the size of the numbers n. The 3 smallest numbers $n = 1 \times 10^5$, $n = 2 \times 10^5$, and $n = 3 \times 10^5$ (represented in light–red, dark–red and light–brown respectively) are too small to obtain good parallel speedup when $p = 96$; compared to $p = 48$, the parallel speedup is even less, implying that using twice as much resources slows down the process instead of speeding

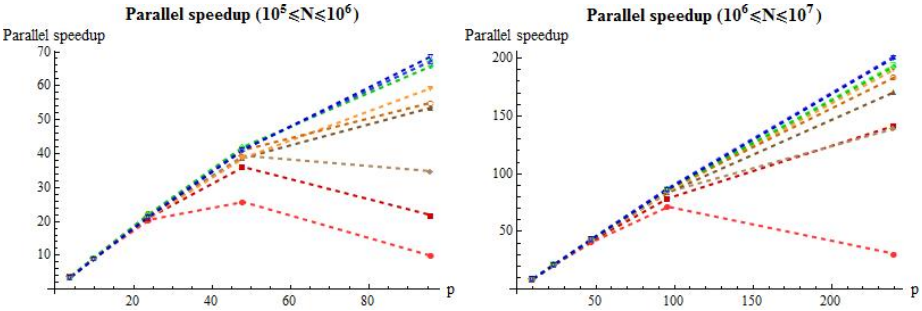


Figure 4: Increasing N: light/dark Red, –Brown, –Orange, –Green, –Blue

it up! For $n = 1 \times 10^5$, doubling the resources from $p = 24$ to $p = 48$ hardly affects the run time.

For $10^6 \leq n \leq 10^7$ reasonable to good parallel speedups are reached for all p, with one exception: for $n = 1 \times 10^6$, the process is slowed down by about a factor 2 when resources are more than doubled, from $p = 96$ to $p = 240$.

3.3.3 Parallel efficiency

To better compare the efficiency of the use of the available resources, the parallel efficiency (relative parallel speedup), defined as $E_{par} = \frac{S_p}{p}$, is shown in the charts in figure 5. An optimal use of resources is obtained when $E_{par} = 1$. For some of the smaller values for n, the poor use of resources for $p = 96$ and $p = 240$ becomes apparent immediately.

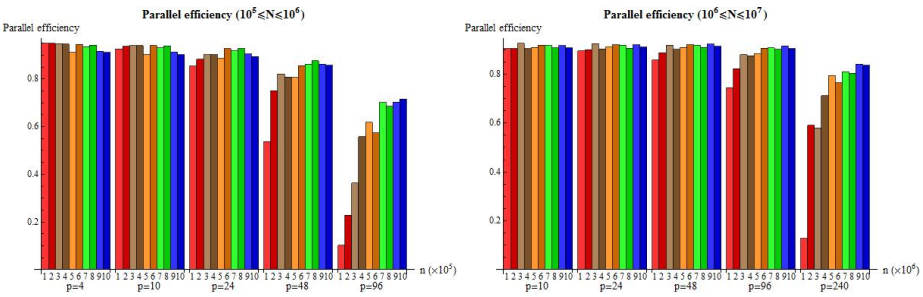


Figure 5: Parallel efficiency

3.3.4 Comparison with results from the literature

In 1996 Cesari and Maeder tested several Karatsuba parallel implementations[3], the method depending on the number of processors p used. For p a power of 2, the results were very poor (the parallel efficiency $E_{par} < 0.4$ in all tests). Their best performing algorithm, with p a power of 3, is used here to compare the results; test results were reported for numbers of size $2^{10} \leq n \leq 2^{20}$, with n a power of 2. It is worth noting, that the implementation as described in section 2.3 principally has no restrictions with respect to p and n; it is therefore more generally applicable.

In tables 1 and 2 the results Cesari and Maeder published, are compared with the results reported in sections 3.3.1 and 3.3.3. Cesari and Maeder tested their implementation on an Intel Paragon with 96 processors, type 50MHz Intel 860. Of course absolute run times cannot be compared with Cartesius' 2.4GHz Intel Xeon processors, but it is interesting to compare the parallel efficiencies for the different values for p and n. These values do not exactly match, but comparison is meaningful for p and n about the same values.

	Absolute run time T in seconds				Parallel efficiency E_{par}		
n	$p = 1$	$p = 9$	$p = 27$	$p = 81$	$p = 9$	$p = 27$	$p = 81$
65536	100	11.4	4.04	1.61	0.975	0.917	0.767
131072	300	33.9	11.8	4.49	0.983	0.942	0.825
262144	906	101	35.1	12.9	0.997	0.956	0.867
524288	2702	304	105	38.5	0.988	0.953	0.866
1048576	8106	916	322	114	0.983	0.932	0.878

Table 1: Cesari & Maeder 1996, Intel Paragon 96 processors (50MHz Intel 860)

	Absolute run time T in seconds				Parallel efficiency E_{par}		
n	$p = 1$	$p = 10$	$p = 24$	$p = 96$	$p = 10$	$p = 24$	$p = 96$
100000	1.345	0.145	0.066	0.137	0.926	0.855	0.102
200000	4.015	0.428	0.189	0.184	0.937	0.883	0.227
300000	7.619	0.810	0.352	0.219	0.940	0.903	0.363
500000	16.929	1.876	0.796	0.285	0.902	0.887	0.619
1000000	50.471	5.595	2.352	0.734	0.902	0.894	0.716

Table 2: 2014, Cartesius 12960 cores (2.4GHz Intel Xeon)

In general, Cesari & Maeder reach parallel efficiencies that are slightly better. For $n \leq 3 \times 10^5$ however, Cesari & Maeder obtain much better parallel efficiencies for $p = 81$ compared to $p = 96$. The poor results for $p = 96$ will be analyzed in more detail in the next section.

3.4 Analysis of the stages in the algorithm

3.4.1 Parallel stages

To explain the behavior found in the previous section, an analysis is made of the parallel stages in the implementation. Six different stages are distinguished in this analysis: (i) processing additions and carries 'downwards' (to generate the input for the local sequential Karatsuba's), (ii) the first parallel redistribution of data, (iii) the execution of the local sequential Karatsuba's, (iv) the second parallel redistribution of data, (v) processing additions and subtractions 'upwards' to generate the higher-level products, and (vi) general overhead throughout all stages, mainly (de-)registering variables and (de-)allocating memory. In figure 6, the results are shown as percentages with respect to the total parallel time for selected values for p.

For $10^5 \leq n \leq 10^6$, $p \leq 24$, and for $10^6 \leq n \leq 10^7$, $p \leq 96$ the available resources are almost exclusively used for executing the local sequential Karatsuba's. Since the computational efficiency reached in this stage is highly tuned

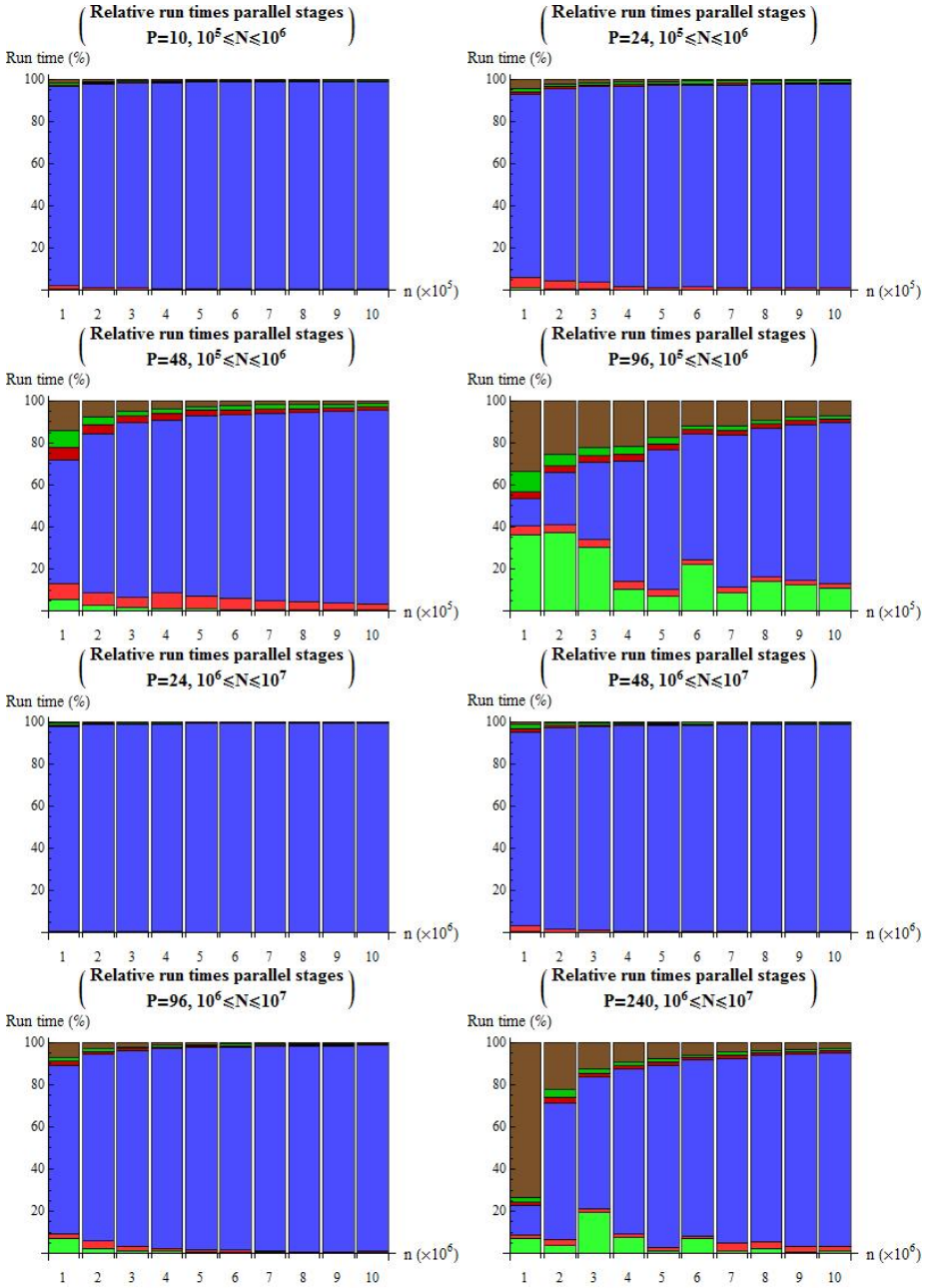


Figure 6: The parallel stages in the algorithm
Stages i through vi are shown from bottom to top in the colors light green, light red, blue, dark red, dark green, and brown

($E \geq 0.95$), the parallel speedup is excellent as well. Note that the total processor time used for these local sequential Karatsuba's hardly depends on the number of processors used, or to put it differently: the number of times the data is split in the first step of the algorithm (the cut-off level l). In each split the

number of multiplications grows by a factor of 3, but the size of each of them halves, which yields a processing time of $\frac{1}{3}$ for each of the 3 smaller sized multiplications (the latter following from the complexity of the algorithm: $\mathcal{O}(n^{\log_2 3})$, see section 1.2.2).

For $10^5 \leq n \leq 10^6$ and $p = 96$ relatively much time is consumed on overhead tasks (the brown colored region), especially for the smaller n. This partly explains the poor parallel efficiency. Relatively much time is spent on the first stage of the algorithm as well, processing additions and carries 'downwards' (the light green colored region). The level l only depends on p, not on n, so for larger n there should be more work to do. Looking into the data more closely reveals, that this is indeed the case for $p \leq 48$, but the run times are quite irregular for $p = 96$. It also reveals, that for $p = 96$ over 90% of the time in this stage is spent on the additions, and only 10% on processing carries. Communication costs therefore can not explain this peculiar behavior. Remarkable is also, that the observed effect is much less in the fifth stage of the algorithm, processing additions and carries 'upwards' (the dark green colored region). This means that the small block-size cannot provide an explanation as well, since these block-sizes are about equally small in the first and fifth stage (for the three smallest n, α equals 9, 17 and 25 respectively; for β these values are 10, 18 and 26). An explanation for the observed effect can not be found in the data.

For $n = 1 \times 10^6$ and $p = 240$ the time spent on overhead tasks is quite remarkable. For larger n it becomes smaller, which may be explained by cache-effects. However, compared to $n = 1 \times 10^6$ and $p = 96$, the difference is extreme: overhead costs grow by more than a factor 23: from $T_{ov} = 0.052$ to $T_{ov} = 1.203$ seconds. A reason for this inefficiency is not found.

In all cases the communication costs are the same for the first and the second redistribution, as expected.

In any case, the conclusion is justified, that it is important to accurately tune the number size n and the number of processors p allocated for the job: the size of the problem must match p.

3.4.2 Parallel accumulated versus sequential

The overall execution time of the implementation can be split into the execution time for the two sequential (distribution) steps and the execution time for the parallel part (accumulated run times for the 6 parallel stages). In the end, this determines the total number of computing hours that are charged. The results are shown in figure 7.

As a reminder: the first sequential distribution step concerns the distribution of the input over the processors (the input can be from disk or be randomly generated), the second sequential distribution concerns collecting the output and write the result to disk (although the actual writing is not done, only the collection of the output is performed).

In these sequential communication steps, all data is communicated from (first distribution) and to (second distribution) processor 0, making them extremely inefficient. However, these communication steps can be avoided, if applications using this implementation deliver the data in block-cyclic format, and if the output is delivered according to specifications set by these applications. For the latter, the program must be adapted (especially with respect to the final processing of overflow blocks, see section 2.3, page 24).

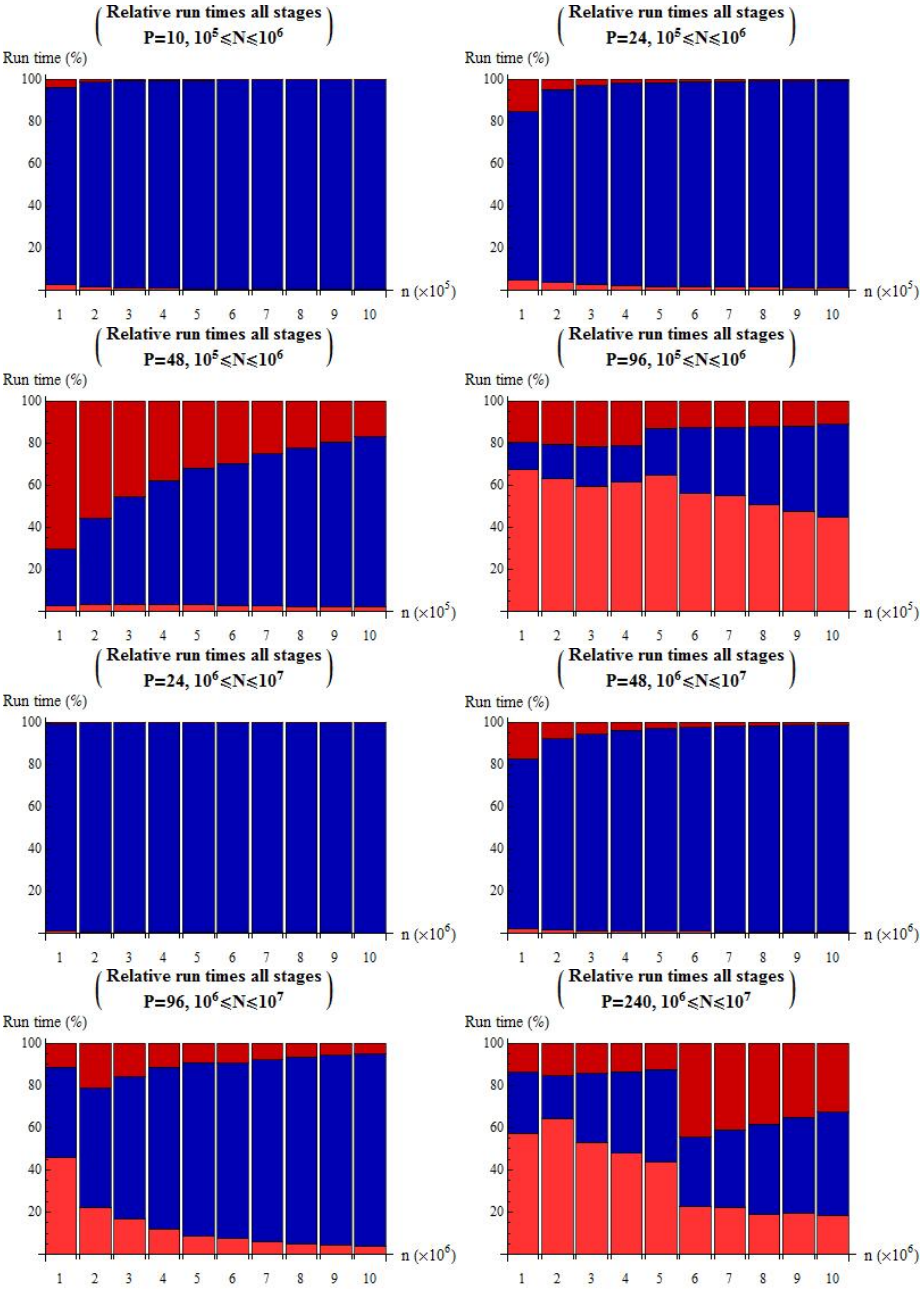


Figure 7: The sequential distributions and the parallel stages accumulated
The sequential distributions in light and dark red, the parallel part in blue

For larger p the communication costs in the sequential distributions can grow excessively. It is not clear, why for some p the first sequential communication step (colored light red) is extremely costly, whereas for other p the second sequential communication step (in dark red) may take extremely much time (the amount of data to be communicated is the same, since the output z has twice

the size of the input x and y). In collecting the output, synchronization takes place after each (global) block received, whereas in distributing the input synchronization only takes place once, so one might expect the second distribution to be more expensive, but clearly in general this is not the case.

The random number generator, a rather expensive function, cannot explain the increase in the first sequential communication step, since there is no significant increase in the first sequential communication for smaller p; moreover, the same number of digits must be generated independent of p.

3.5 Analysis of the communication costs

In figure 8 the communication costs for $10^5 \leq n \leq 10^7$ are plotted against theoretical values, obtained by using results from the benchmark.

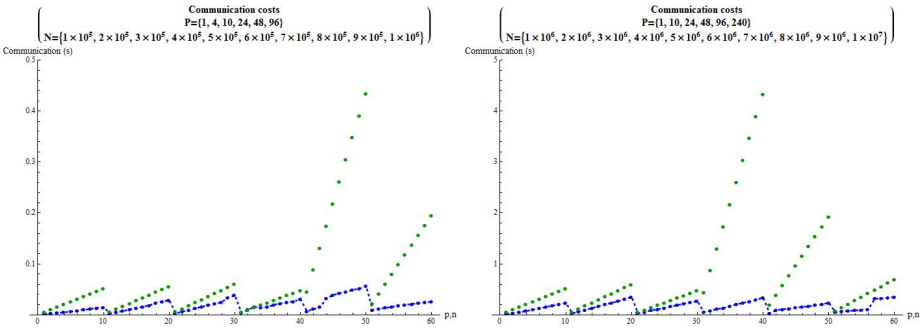


Figure 8: Communication costs test (blue) versus theoretical (green) values

The real communication costs are generally lower compared to theoretical values based on the benchmark. This is due to differences in the way data is communicated in this implementation and in the benchmark used: in the benchmark small chunks (8 bytes) of data are put to determine the value for g, while in this implementation much larger amounts of data are put before synchronization takes place. This apparently positively influences (i.e. lowers) the communication costs considerably. However, of more importance is the linearity the figure shows: for each p the costs are more or less linear in n. This is in accordance with the costs theoretically predicted in section 2.3.

There is a remarkable growth in theoretical costs for $p > 24$. This is caused by an extreme increase in the benchmark values for g and l_{sync} for larger p: for $p = 24$ the benchmark returned $g = 415.4$ and $l_{sync} = 277229.3$, for $p = 48$ these values jumped to $g = 5056.9$ and $l_{sync} = 2025071.8$. The reason for this sudden increase probably is the use of more than 1 node (a thin node has 24 cores).

3.6 Test Ultra Long Integers

To test the scalability of the implementation, an additional test is done for $10^8 \leq n \leq 10^9$, $p = 240$. To avoid too much data is traveling through the system at one time (and thus an insufficient memory condition), the program is adapted by splitting up the communication into smaller chunks of data (for smaller n this program change has no consequences, so the test results discussed before also hold for the new version).

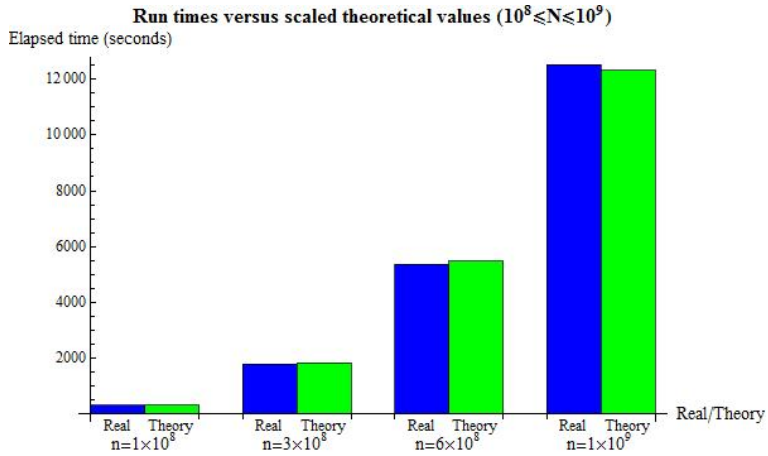


Figure 9: Run times versus theoretical values; $N = 10^8 \Rightarrow T_{theory} \equiv T_{test}$

In figure 9 the run times are compared with theoretical values, calculated from the complexity of the algorithm ($\mathcal{O}(n^{\log_2 3})$). Test values almost perfectly match theoretical values; the parallel efficiency can therefore be compared with values observed in section 3.7 for $n = 10^8$ and $p = 240$ (thin nodes), which were highly satisfactory.

In order to boost the performance, it is worth considering to replace the local sequential Karatsuba's by a (for long integers) faster convolution-based algorithm. Run times can be estimated by making an assumption for the break-even point (or threshold): $n \approx 10^4$ is the value at which convolution-based algorithms become more efficient compared to Karatsuba-like methods[7].

In this test the local multiplications have size 390720 ($n = 1 \times 10^8$), 1171920 ($n = 3 \times 10^8$), 2343840 ($n = 6 \times 10^8$), and 3906480 ($n = 1 \times 10^9$) respectively. Applying the complexity for the Karatsuba method ($\mathcal{O}(n^{\log_2 3})$) and convolution-based algorithms ($\mathcal{O}(n \log n \log \log n)$), one obtains boost factors 5.40 ($n = 1{\times}10^8$), 9.21 ($n = 3{\times}10^8$), 12.95 ($n = 6{\times}10^8$), and 16.69 ($n = 1{\times}10^9$).

These factors are used to correct the run times for the third stage in the algorithm; figure 10 shows the result after this correction. On the left the real relative run times of the different stages, on the right the estimated relative run times, assuming a convolution-based algorithm in the third stage of the algorithm. As can be concluded from this figure, communication costs remain relatively small.

The above observations raise the question: is it possible to parallelize convolution-based algorithms efficiently this way? For generating lower level, smaller-sized multiplications and recombining higher level multiplications from

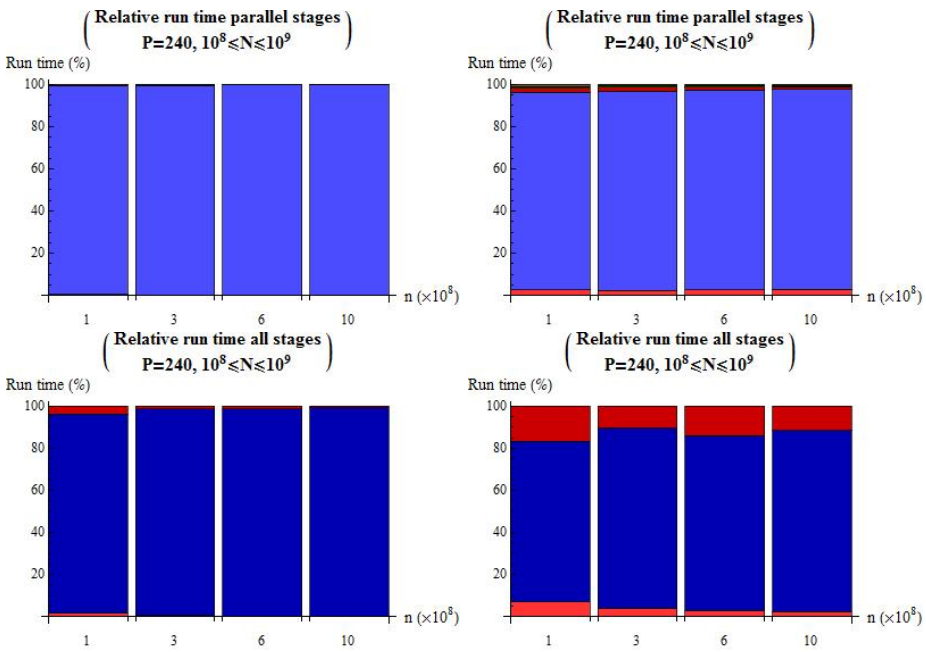


Figure 10: Stage analysis (estimated times) using a convolution-based algorithm
On the left the Karatsuba, on the right a convolution-based algorithm

the lower level results, the Karatsuba method is applied, while the lower level multiplications are locally computed using a convolution-based method.

Combining both algorithms as proposed above, affects the parallel efficiency in the sense that it becomes bounded. The table on the right reveals, that an efficient parallelization is possible for small p only. It shows the maximum possible parallel efficiency quickly drops when more processors are used to do the job. This drop in parallel efficiency can be explained as follows. Each split generates 3 new multiplications of half the size. For algorithms faster than the Karatsuba algorithm this is adverse: each of the smaller sized multiplications takes more than $\frac{1}{3}T$, where T is the time to compute the higher level product. The total computation time for the 3 lower level products is therefore larger than the computation time for the higher level product. Each subsequent split reduces the parallel efficiency further. In the table only powers of 3 are used for p, in order to reduce the number of splits as much as possible (for p a power of three the computational efficiency in the third stage is 1, as explained in section 2.3 on page 12).

Maximum parallel efficiencies

p	E_{par} $(n = 10^9)$	E_{par} $(n = 10^{12})$	E_{par} $(n = 10^{15})$
3	0.697	0.689	0.684
9	0.486	0.475	0.468
27	0.340	0.327	0.320
81	0.238	0.226	0.219
243	0.167	0.156	0.150
729	0.118	0.108	0.103

Clarification: for simplicity assume an algorithm with complexity linear in n is used (instead of $n \log n \log \log n$). Suppose the higher level (before the

split) takes $T_1 = t$ seconds. Then the lower level, using p processors, takes $T_p = \frac{3}{2p}t$ seconds at best (so $T_p > \frac{3}{2p}t$ seconds). The parallel speedup is $S_p = \frac{T_1}{T_p} = \frac{t}{\frac{3}{2p}t} = \frac{2p}{3}$ at best (so $S_p < \frac{2p}{3}$). The parallel efficiency can not exceed $E_{par} = \frac{S_p}{p} = \frac{2}{3}$. Since a convolution based algorithm is a little bit worse than linear, the disadvantage of splitting is a little less as well, hence the value $E_{par} \approx 0.69$ in the table. In general: let $C(n)$ be the complexity of the (convolution-based) algorithm replacing the Karatsuba algorithm in the third stage of the algorithm. Let $T_C(n,l)$ be the time needed to compute the level-l multiplications, $T_C(n,0) = t$. Then $T_C(n,l) = 3^l \frac{C(\frac{n}{2^l})}{C(n)} t$, and the maximum parallel efficiency using $p = 3^l$ processors is $E_{par} = \frac{T_C(n,0)}{T_C(n,l)}$.

3.7 Comparison thin and fat nodes

To compare the performance and behavior of the thin nodes and the fat nodes of the Cartesius system, tests were executed with $n = 1 \times 10^7$, $n = 3 \times 10^7$, $n = 6 \times 10^7$, and $n = 1 \times 10^8$. Choosing larger values would consume too much computing hours, since for determining the parallel speedup and efficiency, one cost-inefficient job with $p = 1$ had to be run on both nodes (as noted before, charged hours are based on the exclusive use of the whole node). The test is executed for 11 values for p on the thin and 10 values for p on the fat nodes (the configuration requesting 960 processors on fat nodes appeared not to be available). The fat nodes might be needed for its larger memory when running the program for large n; it is therefore useful to know the behavior of the fat nodes when running this implementation.

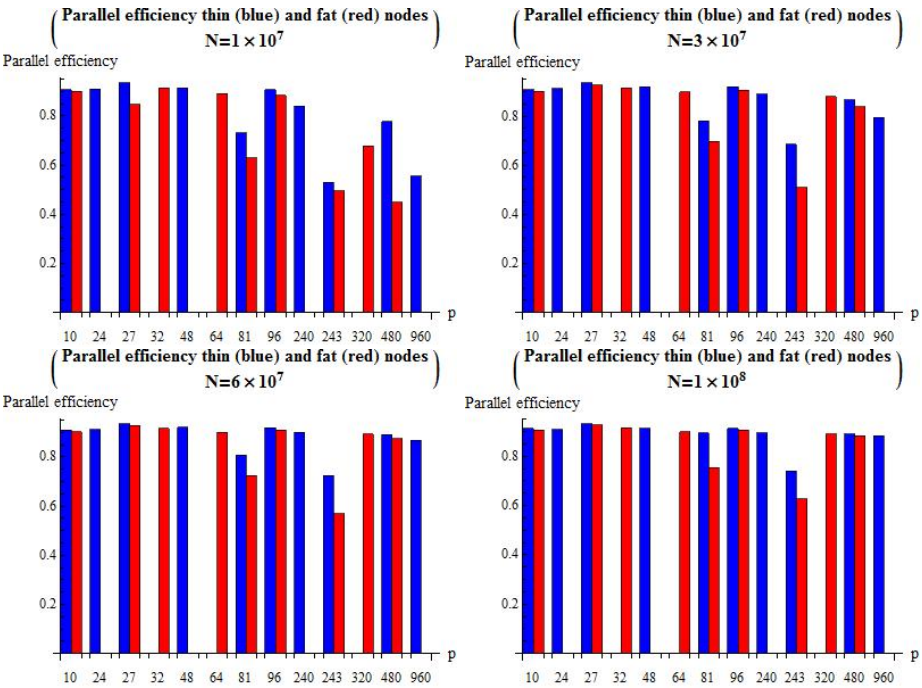


Figure 11: Parallel efficiency thin (blue) and fat (red) nodes

The fat nodes consumed much more processor time for the same amount of work, irrespective of the number of processors used. Apparently the larger amount of memory available makes the fat nodes slower (not surprising because accessing a larger memory takes more time), but the difference is quite large: on average the fat nodes consumed about 28% more processor time for the same amount of work.

In figure 11 the parallel efficiency is shown for the different configurations. In general the results are quite satisfactory, with two exceptions: (i) for $p = 81$ and $p = 243$ there is a significant drop in the parallel efficiency (the drop is most significant for the fat nodes), and (ii) for the smallest $n = 1 \times 10^7$, for $p = 960$ (thin nodes) and $p = 480$ (fat nodes) there is hardly an increase, or even a decrease in parallel efficiency compared to the next lower p. The first exception can be explained by the fact that p is not a multiple of the number of nodes accessed, which apparently gives a significantly lower performance (in spite of the fact that the computational efficiency in the local sequential Karatsuba's for these values of p is equal to 1). To explain the second exception, one can say the number of processors used is too large for this relatively small n. A similar behavior has been observed in section 3.3.

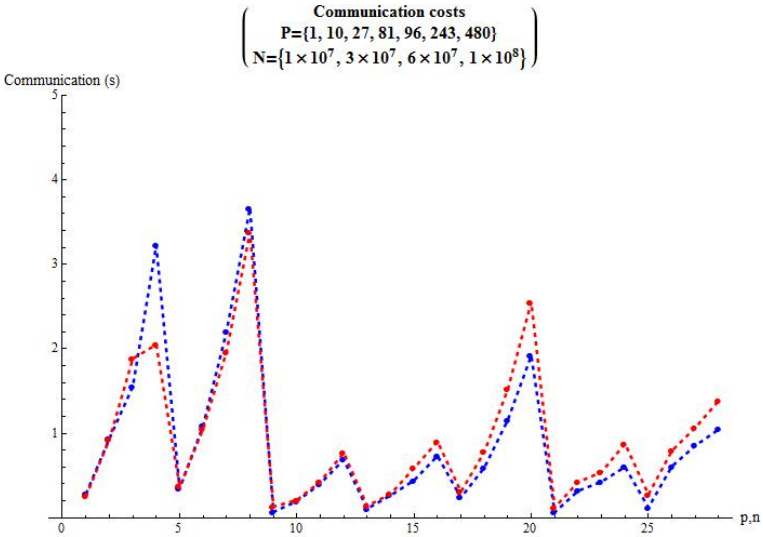


Figure 12: Communication costs thin nodes (blue) versus fat nodes (red)

The parallel efficiency (and thus the parallel speedup) is significantly less for the fat nodes. This does not follow from their slower performance, since the larger processor time used by the fat nodes also holds for $p = 1$. The lower parallel efficiency is (partly) due to higher communication costs, as can be seen in figure 12.

For larger p, communication costs for the fat nodes are generally higher; only for $p = 1$ (irrelevant for determining the parallel efficiency) and $p = 10$ the thin nodes turn out to be more expensive.

4 Discussion and Conclusion

4.1 Performance of the implementation

This parallel implementation of the Karatsuba multiplication algorithm has shown nice results for parallel speedup, provided the problem size grows sufficiently with the number of processors. From the stage analysis in the previous section the conclusion is justified, that for $10^5 \leq n \leq 10^7$ satisfactory parallel speedups are obtained for all p satisfying $\frac{n}{p} > 10^4$.

How the algorithm behaves for smaller numbers and moderate values for p, has not been part of the research: the main goal was to show the scalability and efficiency of the implementation, also for large values for p (and as a consequence, for larger n). However, it can be interesting to test the behavior for smaller values for p and n, since algorithms based on the Fast Fourier Transform (and their parallel implementations) are more fit to deal with large number multiplication. On the other hand, one may wonder if a data-distributed approach makes sense for small(er)-sized problems: after splitting the input l times to generate the lower level multiplications, the numbers to multiply become very small, and the result comes closer to parallelizing the classical multiplication algorithm instead. However, if one wishes to use the implementation for $10^4 \leq n \leq 10^5$, $p \leq 24$ is likely to be a restriction to obtain good parallel speedup.

It turned out to be possible to set the minimal efficiency for the local sequential Karatsuba's to a high level, and efficiencies larger than 0.95 were easily reached. This is important, since a significant drop in performance showed up for p a power of 3 (for these values for p the efficiency is always 1). The reason for this drop is probably a consequence of the architecture of the Cartesius system the implementation was tested on, and is not a consequence of the implementation itself. The high efficiencies reached in the local sequential Karatsuba's was the main factor for the satisfying parallel speedups in the tests.

Another factor that usually determines the parallel speedup, is the amount of communication in the process. Overall communication costs in the parallel part of the algorithm turned out to be very limited. The delayed processing of carries certainly helped to limit communication as much as possible.

4.2 Parallelizing convolution based algorithms

This implementation can also be used to parallelize convolution based algorithms, provided the number of processors is small. For $p > 9$ the maximum parallel efficiency that can be reached drops quickly. This implies the scalability of the implementation using convolution based algorithms is limited.

4.3 Suggestions for improvement

Two findings need further discussion: (i) communication problems that showed up for $n \geq 3 \times 10^8$ (actually a scalability problem), and (ii) increasing costs for the two sequential communication steps (distributing input and collecting output).

Communication problems for large n When executing the multiplication of numbers of size $n \approx 3 \times 10^8$ and larger, problems occurred in the com-

munication steps of the algorithm. In the first sequential and both parallel communication steps, all data is distributed at one time, synchronizing only once at the end of the step. This approach turned out to be a problem for large numbers: apparently too much data traveled through the system in one time.

In fact this is a scalability issue, and the program is adapted in order to test the multiplication of numbers up to $n = 10^9$: for large n the communication is split into smaller chunks of data with synchronization after each block in order to avoid memory problems. A parameter is used to make the splitting up more flexible: the parameter determines the maximum amount of data that is transmitted in each step. This splitting up increases communication costs slightly, but only for large n (depending on the parameter).

However, a further improvement is probably needed if convolution based algorithms are applied in the local multiplications (the third stage of the algorithm). When applying these algorithms, the number of times the data is split must be minimal in order to obtain a reasonable parallel efficiency: the number of multiplications at the lowest level is equal to p (with p a power of 3), thus each processor has exactly 1 multiplication to execute. The current splitting is based on a larger number of multiplications to be executed by each processor, so a different way of splitting the communication must be adapted: the β-sized blocks must be split into smaller chunks (only for communication, the block-size β need not be changed).

Communication costs for distributing input and collecting output
When the size n of the numbers x and y and the number of processors p become large, costs for the sequential communication steps can grow to extreme proportions. All data goes from and to one single processor (only processor 0 has access to the file system to read input and write output), which makes these sequential communication steps very cost-inefficient.

Integrating this implementation with applications that use it can avoid these expensive communication steps: the input should be delivered in the desired block-cyclic distribution, the output should be delivered according to specifications set by the application. One might think of implementing a few standard output distributions, e.g. the block, the cyclic, and the block-cyclic distribution.

To accomplish this, the post-processing of overflow blocks (as described in section 2.3, page 24) must be integrated in a final redistribution step to deliver the data as specified by the application. In the current implementation this post-processing is part of the collection of all output.

4.4 Practical applicability

This implementation can be used for the multiplication of medium-sized integers ($10^5 < n < 10^7$). For the multiplication of ultra long integers ($n \geq 10^9$), a convolution based algorithm should be used for executing the local multiplications, but usage of this implementation only makes sense for small values for p because of the limited parallel efficiency for larger p (consult the table in section 3.6). For the multiplication of shorter integers ($n < 10^5$) the implementation is likely to be efficient for $p \leq 24$ only. However, using numbers this size almost comes down to parallelizing the classical multiplication.

5 References

1. G. M. Amdahl, "Validity of the single-processor approach to achieving large-scale computing capability", in *Proceedings of AFIPS Conference*, Volume **30** (1967) pp. 483–485.

2. R. H. Bisseling, *Parallel Scientific Computation, A structured approach using BSP and MPI*, Oxford University Press (2004).

3. G. Cesari and R. Maeder, "Performance Analysis of the Parallel Karatsuba Multiplication Algorithm for Distributed Memory Architectures", Journal of Symbolic Computation, Volume **21**, Issue 4-6 (1996) pp. 467–473.

4. Z. Chen and P. Schaumont, "A Parallel Implementation of Montgomery Multiplication on Multicore Systems: Algorithm, Analysis, and Prototype", IEEE Transactions on Computers, Volume **60**, Issue 12 (2011) pp. 1692–1703.

5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to algorithms*, 3rd edition, The MIT Press (1998).

6. B. S. Fagin, "Large Integer Multiplication on Hypercubes", Journal of Parallel and Distributed Computing, Volume **14**, Issue 4 (1994) pp. 426–430.

7. gmplib.org, http://gmplib.org/devel/MUL_FFT_THRESHOLD.html.

8. T. Jebelean, "Using the Parallel Karatsuba Algorithm for Long Integer Multiplication and Division", Lecture Notes in Computer Science, Volume **1300** (1997) pp. 1169–1172.

9. A. Karatsuba and Y. Ofman, "Multiplication of Multiplace Numbers on Automata", Doklady Akad. Nauk SSSR, Volume **145**, Issue 2 (1962) pp. 293–294. Translation in Soviet Physics-Doklady, Volume **7**, Issue 7 (1963) pp. 595–596.

10. A. Karatsuba, "The complexity of Computations", Proceedings of the Steklov Institute of Mathematics, Volume **211** (1995) pp. 169–183. Translated from Trudy Matematicheskogo Instituta imeni V. A. Steklova, Volume **211** (1995) pp. 186–202.

11. D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, Volume **2**, 3rd edition, Addison-Wesley, Reading, Mass. (1997).

12. H. J. Nussbaumer, *Fast Fourier Transform and Convolution Algorithms*, Springer-Verlag Berlin Heidelberg New York (1981).

13. A. Schönhage and V. Strassen, "Schnelle Multiplikation großer Zahlen", Computing, Volume **7**, Issue 3/4 (1971) pp. 281–292.

14. D. Zuras, "On Squaring and Multiplying Large Integers", in *Proceedings of the 11th Symposium on Computer Arithmetic*, IEEE Computer Society Press, Los Alamitos, California (1993) pp. 260–271.

15. D. Zuras, "More On Squaring and Multiplying Large Integers", IEEE Transactions on Computers, Volume **43**, Issue 8 (1994) pp. 899–908.