Table of Contents

Chapter 1. Introduction

1.1 Prologue

In our days, is hard to imagine our life without computers. We are facing with the necessity of huge data processing and computations. But, any computation can be reduced to the four basic operations: addition, subtraction, multiplication, and division. Therefore, if we want to substantially improve the execution time of our computations, we must focus first on these basic operations.

When we want to improve the execution time of one of our programs, we have two choices: “work hard” and “work smart”. The first one implies the using of several computers in the same time (or using a parallel computer with several processors) to getting the result, while the second one require the improvement of the algorithm used in computations. The ideal case is when we succeed to combine these two methods in one, obtaining a very good time.

In this paper we will study a method for improving the execution time of the multiplication, using a combination of the “work hard” and “work smart” approaches. More exactly, we will study the Parallel Karatsuba Algorithm for Long Integer Multiplication and it’s implementation (made by us) on Origin 2000, a MIMD shared-memory supercomputer.

Using our implementation on Origin 2000, we obtained comparing with classical multiplication a speed-up of 90 at 16384 words per operand. Comparing with the recursive Karatsuba algorithm we obtained a speed-up of 6.84 at 16384 words per operand. The dimension of the words, used in our implementation, is 32 bits.

1.2 Acknowledgments

I’m grateful to my advisor, Tudor Jebelean, for his constant support, kind advice and excellent lectures.

I thank to Mircea Drăgan, for giving me all his previous works in this field.

I thank my office colleagues for understanding and stimulating me in this work. I thank to all RISC members who helped me to complete my knowledge in one way or another.

Socrates program provided me a scholarship from February, 2000 to July, 2000 at RISC, which gave me the possibility to make this research. Thanks.

Chapter 2. Sequential and Parallel Karatsuba Algorithm

If we want to multiply large numbers, it's possible to use other techniques than the traditional multiplication, which is time expensive. In this case the Karatsuba Multiplication Algorithm [8], [9] is one of the choices.

2.1 The Karatsuba Integer Multiplication Algorithm

For convenience, we will focus on long integers in binary representation [9]. If we have the numbers

$$U = (u_{2n-1}...u_1u_0)_2 \text{ and } V = (v_{2n-1}...v_1v_0)_2 \qquad (1)$$

represented on 2n bits, we can write then

$$U = 2^nU_1 + U_0 \quad \text{and} \quad V = 2^nV_1 + V_0 \text{ , where}$$

$$U_1 = (u_{2n-1}...u_n),\ U_0 = (u_{n-1}...u_1u_0),$$
$$V_1 = (v_{2n-1}...v_n),\ V_0 = (v_{n-1}...v_1v_0).$$

Now, we have:

$$\begin{aligned} UV &= (2^nU_1 + U_0)(2^nV_1 + V_0) = \\ &= 2^{2n}U_1V_1 + 2^n(U_1V_0 + U_0V_1) + U_0V_0 = \\ &= 2^{2n}U_1V_1 + 2^n((U_1 + U_0)(V_1 + V_0) - U_1V_1 - U_0V_0) + U_0V_0 \qquad (2) \end{aligned}$$

This formula reduce the original 2n bits operands multiplication to three n bits operands multiplication: U_1V_1, $(U_1 + U_0)(V_1 + V_0)$ and U_0V_0, and few simple operations (like shift and addition).

The main advantage of using the formula (2) is that we can define a recursive process for multiplication, which is faster than the traditional multiplication. The traditional multiplication has the complexity $O(n^2)$ [9]. Proceeding recursively we obtain the bit complexity $O(n^{\lg 3})$, where

$$\lg 3 = 1.58... < 2 \text{ . [10]}$$

When this technique is recursively applied to multidigit numbers, a point is reached in the recursion when the overhead of addition and subtraction makes it more efficient to use the usual $O(n^2)$ multiplication algorithm to evaluate the partial products. We call this point "Karatsuba threshold". The most efficient overall method therefore relies on a combination of Karatsuba and conventional multiplication [6]. Namely, we'll have $n^{\log3/\log2}$ digit products for operands of length n, not n^2 like in the traditional multiplication. (see [9], p. 233)

It's convenient to see this algorithm in terms of a ternary tree. Each node has three children that compute the partial products and at each level the input length is divided with two. The leaves perform the classical multiplication. The threshold for switching to the traditional multiplication method is determined experimentally [3].

2.2 The Parallel Karatsuba Multiplication Algorithm

Parallelization of the Karatsuba algorithm is intuitive, if we look to (2). We just have to make three parallel tasks to compute the three n bits products and finally, to combine the results. However, there are three interesting approaches of this (proposed by Cesari and Maeder in [3]).

2.2.1 Algorithm 1

The first algorithm follows a master-slave approach: the master processor starts the Karatsuba algorithm and creates all the tasks down to a specified level in the tree. Once these tasks have been created, they are sent to the slaves and computed there sequentially. After the slaves have finished to compute, the master process collects the results and traverses the tree back to the root forming the result at a node by combining the results of the children.

2.2.2 Algorithm 2

The second algorithm follows a fork-join approach: one processor starts at the root and recursively assigns separate nodes in the tree to different processors. Each node assigns three subnodes for the three recursive multiplications and combines the results according to (2) after the three subnodes have completed their computation. (One of the three subtasks is performed by the parent process, which would otherwise wait idly for the completion of the three subprocesses.) When a certain cut-off criterion is reached (the “parallel threshold”), no further parallel splitting occurs: the work remaining in a given task is done sequentially.

2.2.3 Algorithm 3

In the third algorithm at a given level of the tree only two tasks are computed in parallel, while the third is delayed and is computed when the first two return.

2.2.4 Comparisons of the three algorithms

In Algorithm 2, if one of the three parallel subtasks is performed by the parent node, the total number of processors is given by 3^{λ}, where $\lambda = \log_2 n$ is the depth of the tree and n is the input size of the operands. Even if Algorithms 1 and 2 have the same complexity, Algorithm 2 achieves better performance than Algorithm 1. For Algorithm 3, while the asymptotic speed-up is worst, the efficiency is better than that of Algorithm 1 and 2. But, this algorithm is interesting only from a theoretical point of view because it shows worse behavior than Algorithm 2 [3].

Chapter 3. Implementation of Sequential and Parallel Karatsuba Algorithm on Origin 2000

3.1 Description of SGI Cray Origin 2000

Figure 3.1. Image of our SGI Cray Origin 2000

The SGI Cray Origin 2000 is the central Compute Server at the switchboard of informatics service of the Johannes Kepler University Linz. [11]

3.1.1 Hardware equipment of the SGI Origin 2000

Computer type:
    Parallel computer with shared memory (MIMD);
Processing units:
    64 processors R12000 at 300MHz;
Storage:
    20 GB RAM of joint functioning storages;
Harddisk:
    202 GB;
Magnetic tape:
    DLT4000 30GB + DLT7000 Stacker 70GB;
CD-ROM:
    4 Drives;
Network:
    2 xs ATM 155 Mb/s,
    FDDI 100 Mb/s,
    Ethernet 10Mb/s.

3.1.2 Performance tuning of parallel programs on an Origin 2000 server

An Origin 2000 server provides cache-coherent, shared memory in the hardware. Memory is physically distributed across processors. Consequently, references to locations in the remote memory of another processor take substantially longer (by a factor of two or more) to complete than references to locations in local memory. This can severely affect the performance of programs that suffer from a large number of cache misses.

To obtain good performance in such programs, it is important to schedule computation and distribute data across the underlying processors and memory modules, so that most cache misses are satisfied from local rather than remote memory. The primary goal of the programming support, therefore, is to enable user control over data placement and computation scheduling.

If the misses are to data referenced primarily by a single processor, then data placement may be able to convert remote references to local references, thereby reducing the latency of the miss. The possible options for data placement are automatic page migration or explicit data distribution, either regular or reshaped.

Automatic page migration requires no user intervention and is based on the run-time cache miss behavior of the program. It can therefore adjust to dynamic changes in the reference patterns. However, the page migration heuristics are deliberately conservative, and may be slow to react to changes in the reference patterns. They are also limited to performing page-level allocation of data.

3.2 Implementation of Karatsuba Multiplication Algorithm on Origin2000

When we planned to implement this algorithm on Origin 2000, our first thought was the portability. So, we used for parallelization only the POSIX threads in our implementation (the implementation is written in C). Therefore, this program can run on any shared memory parallel computer, not only on Origin 2000. In fact, it can run on any Unix-based computer, even if it has only one processor. This is, of course, a big advantage, because we all know that the "life" of a parallel computer is very short. The best parallel computer we can have today will became probably tomorrow too slow (comparing with the tomorrow's parallel computers).

In our implementation we chose the Algorithm 2 [3] (it has a short presentation in the Chapter 2 of this paper). Every process (in our case is a thread) creates, if necessary, two more processes. Based on formula (2), the first one will compute U_1V_1, the second one U_0V_0, and the current process will compute $(U_1 + U_0)(V_1 + V_0)$. When all these three results are ready, they will be combined in this process (the current one).

All running processes access the same memory spaces, respectively the U, V, and the result R. This can be possible because we have a shared memory computer. When all threads are finished, the R will contain the result of the multiplication of U by V.

The parallel Karatsuba algorithm used can be written in the pseudo-code in the following way (to be simpler, we assume that we have operands with the same length):

```
function parallel_karatsuba(U, V, R, depth)
    n = length(U)  [in this case n = length(V) , too]
    if (n < PARALLEL_THRESHOLD) then
        sequential_karatsuba(U, V, R)
        return
    if (depth >= MAX_DEPTH) then
        sequential_karatsuba(U, V, R)
        return
    create_thread_1 with parallel_karatsuba(U1, V1, S, depth + 1)
    create_thread_2 with parallel_karatsuba(U0, V0, T, depth + 1)
    X ← U1 + U0
    Y ← V1 + V0
    call parallel_karatsuba(X, Y, Z, depth + 1)
    join thread_1
    Z ← Z – S
    join thread_2
    Z ← Z – T
    R ← 2^(2n)S + 2^(n)Z + T
    return
```

One can observe that if the depth is under the maximum value (value set by the user), and the length of the operands is bigger than a certain parallelization threshold, there are created only two new processes, and the current process will start a new execution of the function. This can be observed in the execution graph from Figure 3.2.

The depth is incremented on each macro-step through function call.

The sequential Karatsuba algorithm used is similar with the parallel one. But, in this algorithm we call recursively the function, instead of creating threads.

function sequential_karatsuba(U, V, R)
    n = length(U) [in this case n = length(V) , too]
    if (n < SEQUENTIAL_THRESHOLD) then
        traditional_karatsuba(U, V, R)
        return
    call sequential_karatsuba(U_1, V_1, S)
    call sequential_karatsuba(U_0, V_0, T)
    $X \leftarrow U_1 + U_0$
    $Y \leftarrow V_1 + V_0$
    call sequential_karatsuba(X, Y, Z)
    $Z \leftarrow Z - S$
    $Z \leftarrow Z - T$
    $R \leftarrow 2^{2n}S + 2^{n}Z + T$
    return

In the case of sequential Karatsuba multiplication algorithm, we will apply it until the length of the operands are under a certain sequential threshold determined experimentally. When this threshold is reached, the computation will continue with the traditional multiplication.

The user can choose the maximum number of threads to be created, setting the depth of the paralelization. If is chose 0 the program will multiply the U by V using the sequential Karatsuba algorithm. For example, if we choose 2 (it means 3^2 = 9 threads), we will obtain the following execution graph (if the length of the operands are big enough):

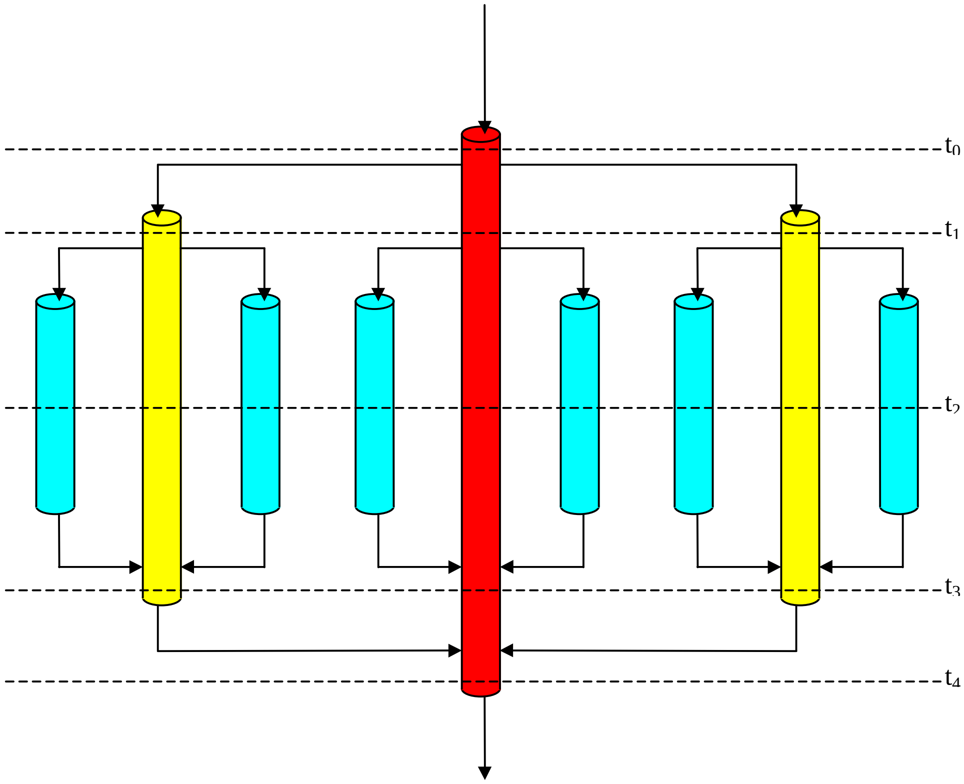

Figure 3.2. Execution graph with 9 parallel processes

One can observe, based on this graph, that the “9 processes” represent the maximum processes involved in the computation. For example, at time t_0 and t_4, there is only one process running, while at time t_1 and t_3 there are 3 processes running. Only in a part of the total execution time the computation runs on 9 processors in the same time.

The next problem, which we had, was related to the threshold value. At which length of the operands it is worth to start the parallelization? In fact, we have two thresholds: the first one is between traditional multiplication and sequential Karatsuba multiplication, and the second one is between the sequential Karatsuba multiplication and the parallel Karatsuba multiplication.

These values for thresholds must be, as we mentioned in Chapter 2, determined experimentally. In fact, we have to test an interval of values and choose the threshold which gives the minimum execution time.

For the first threshold, we tested the values from 4 to 257 for operands with 1024 words each (see Figure 3.3).

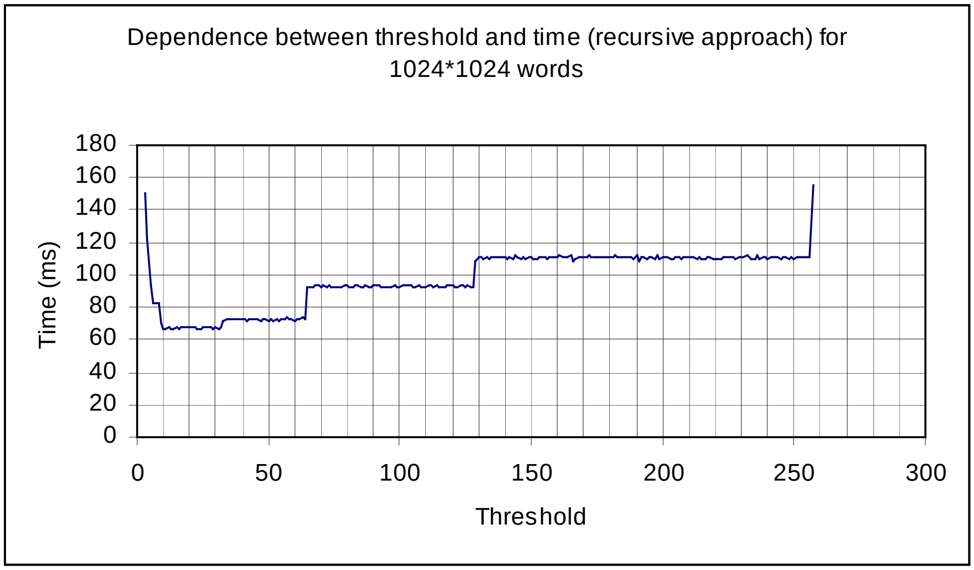

Figure 3.3. Dependence threshold-time (recursive approach) for 1024*1024 words per operand

One can observe that the execution time is almost constant between two consecutive powers of 2. From our tests, we conclude that for Origin 2000, in our implementation the optimal threshold for starting the sequential Karatsuba algorithm is 32.

Based on this threshold value, we tested the second optimal threshold value, for starting the parallelization. We used for tests values from 64 to 1024 for threshold, applied on operands of length 2048 words each (see Figure 3.4).

From this test, we conclude that the optimal threshold value for starting the parallelization is 512 .

So, we obtained the thresholds pair (32, 512). This pair of thresholds can be changed by the user in the program, for testing and setting the appropriate values for each particular machine.

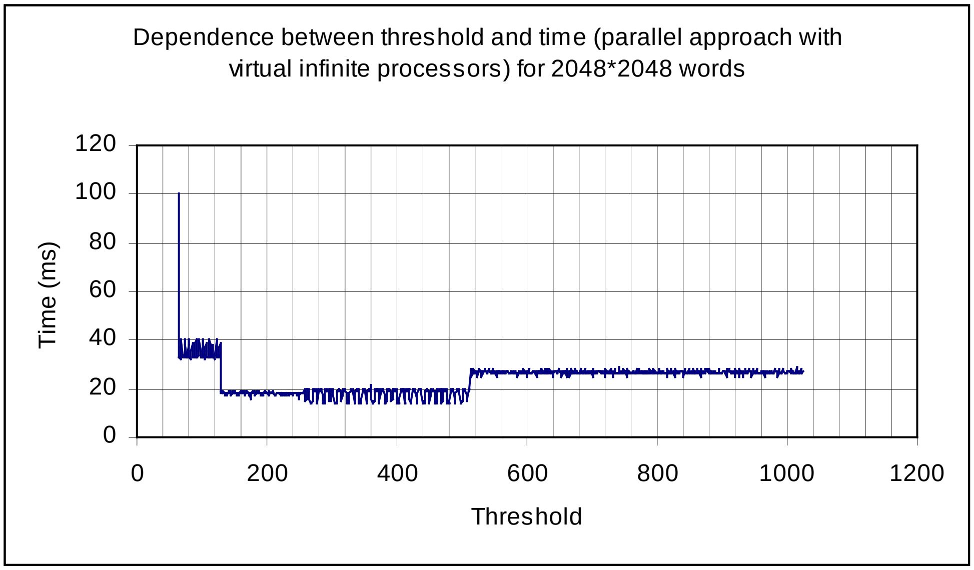

Figure 3.4. Dependence threshold-time (parallel approach) for 2048*2048 words per operand

We have to mention that we chose the best maximum values for the thresholds (32 and not 17; 512 and not 129), because those fits better with long operands (equilibrate the work between processes).

In our implementation the size of one word is 32 bits, and we work only with positive long numbers.

3.3 Practical results

For our experiments we used operands with the same length (from 32 to 16384 words). Even if Origin 2000 has only 64 processors, we tested multiplication using 1, 3, 9, 27 and 81 parallel threads. Even if it's possible to set in our program the explicit processor to use for every thread, we didn't do this, letting the operating system to choose the best solution. We have to mention that we are using automatic page migration, so, data location and execution control is made by the operating system. Effectively, we used 27 processors. For 81 parallel threads, they are shuffled on the 27 processors used by the operating system. We compiled our program with option "–O2", for optimizing the generated code.

The execution time of our multiplication's tests is given in milliseconds, but in fact, we used for every result 10 pairs of operands, randomly generated, and the result is the average of the resulting times.

Using the thresholds pair (32, 512) and operands from 32 to 16384 words (of 32 bits), we obtained the following results:

Words	Traditional	1 proc	3 proc	9 proc	27 proc	81 proc
32	1	1	1	1	1	1
64	1	1	1	1	1	1
128	1	1	1	1	1	1
256	2	1	1	1	1	1
512	11	4	7	5	5	7
1024	42	11	9	17	17	14
2048	170	30	21	20	35	38
4096	674	93	40	26	51	132
8192	2928	280	106	56	79	156
16384	11709	842	413	132	130	216

Figure 3.5. Comparative execution times using Karatsuba Multiplication Algorithm

Using these values, we can generate the graphical interpretation of the results:

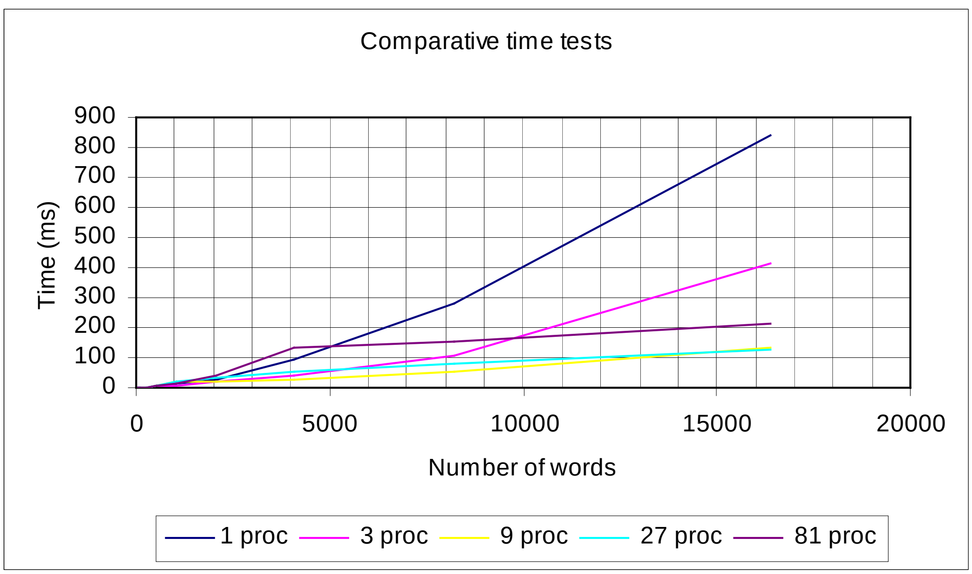

Figure 3.6. Graphical interpretation of the execution times

One can observe from this two figures that the execution time is minimal when we use 1 process with operands less than 512 words length each. If we want to multiply two operands of 16384 words length, the minimum execution time is given when we use 27 processes.

We can compute the speed-up of our parallelization, using the following formula:

$$Speed-up = \frac{Sequential\_time}{Parallel\_time}$$

Applying this formula to our data we obtained following results:

Words	3 proc	9 proc	27 proc	81 proc
32	1.000	1.000	1.000	1.000
64	1.000	1.000	1.000	1.000
128	1.000	1.000	1.000	1.000
256	1.000	1.000	1.000	1.000
512	0.571	0.800	0.800	0.571
1024	1.222	0.647	0.647	0.786
2048	1.429	1.500	0.857	0.789
4096	2.325	3.577	1.824	0.705
8192	2.642	5.000	3.544	1.795
16384	2.039	6.379	6.477	3.898

Figure 3.7. Speed-up results over the sequential Karatsuba algorithm

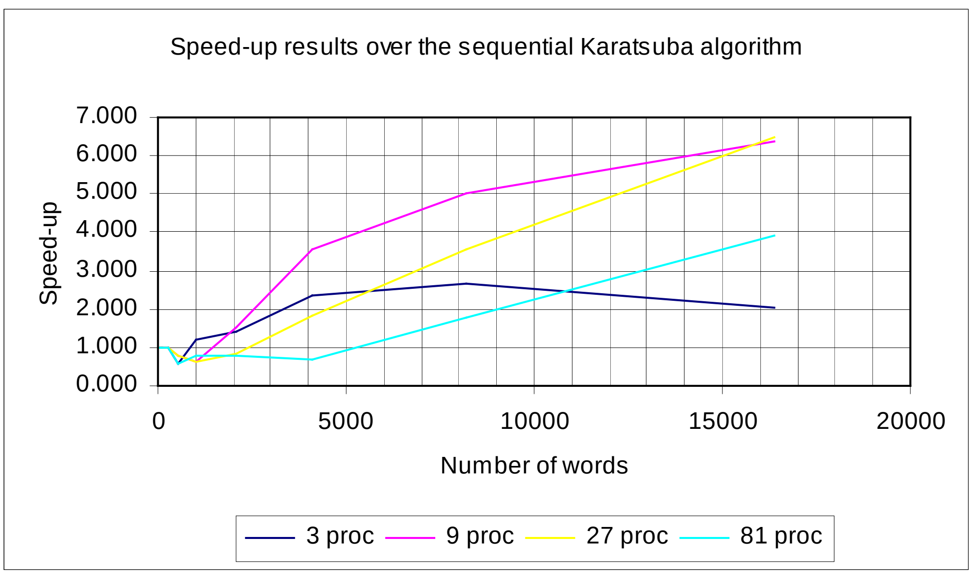

Figure 3.8. Graphical interpretation of the speed-up results over the sequential Karatsuba algorithm

If we take as a reference time the execution time of the traditional multiplication, we will obtain the following combined speed-up values:

Words	1 proc	3 proc	9 proc	27 proc	81 proc
32	1.000	1.000	1.000	1.000	1.000
64	1.000	1.000	1.000	1.000	1.000
128	1.000	1.000	1.000	1.000	1.000
256	2.000	2.000	2.000	2.000	2.000
512	2.750	1.571	2.200	2.200	1.571
1024	3.818	4.667	2.471	2.471	3.000
2048	5.667	8.095	8.500	4.857	4.474
4096	7.247	16.850	25.923	13.216	5.106
8192	10.457	27.623	52.286	37.063	18.769
16384	13.906	28.351	88.705	90.069	54.208

Figure 3.9. Combined speed-up over the sequential classical multiplication algorithm

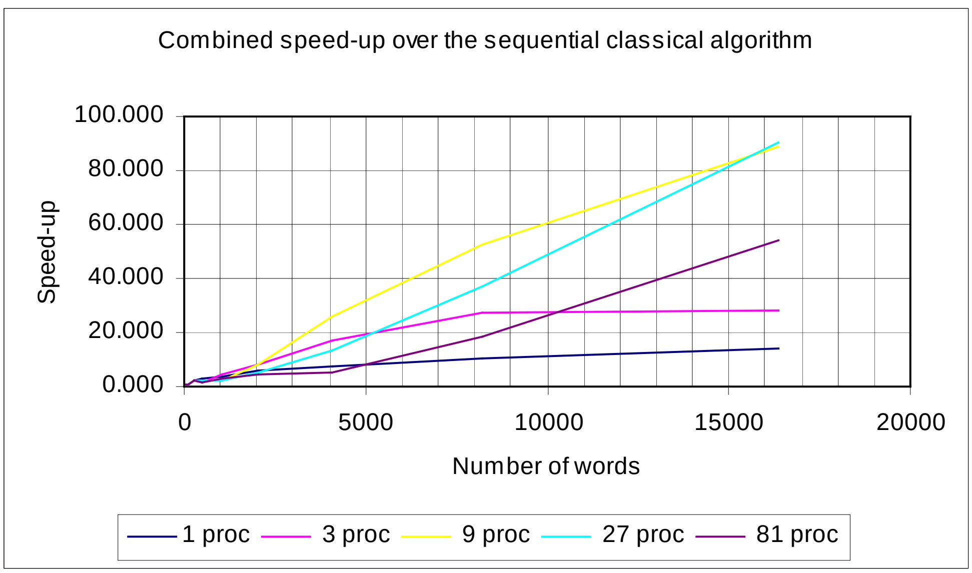

Figure 3.10. Graphical interpretation of combined speed-up over the sequential classical multiplication algorithm

One can observe from these tables and graphics, that the maximum speed-up over the sequential Karatsuba algorithm is given by using 27 proceses, at 16384 words per operand, and it is 6.48. If we look to the combined speed-up, the maximum value is, as we expected, in the same place and have the value 90.1.

Now, we can compute the efficiency of the processors (given in percents), using the following formula:

$$Efficiency = \frac{Speed - up}{Number\_of\_parallel\_threads} * 100$$

Applying this formula to our data we obtain:

Words	3 proc	9 proc	27 proc	81 proc
32	33	11	4	1
64	33	11	4	1
128	33	11	4	1
256	33	11	4	1
512	19	9	3	1
1024	41	7	2	1
2048	48	17	3	1
4096	78	40	7	1
8192	88	56	13	2
16384	68	71	24	5

Figure 3.11. Efficiency of the parallelization

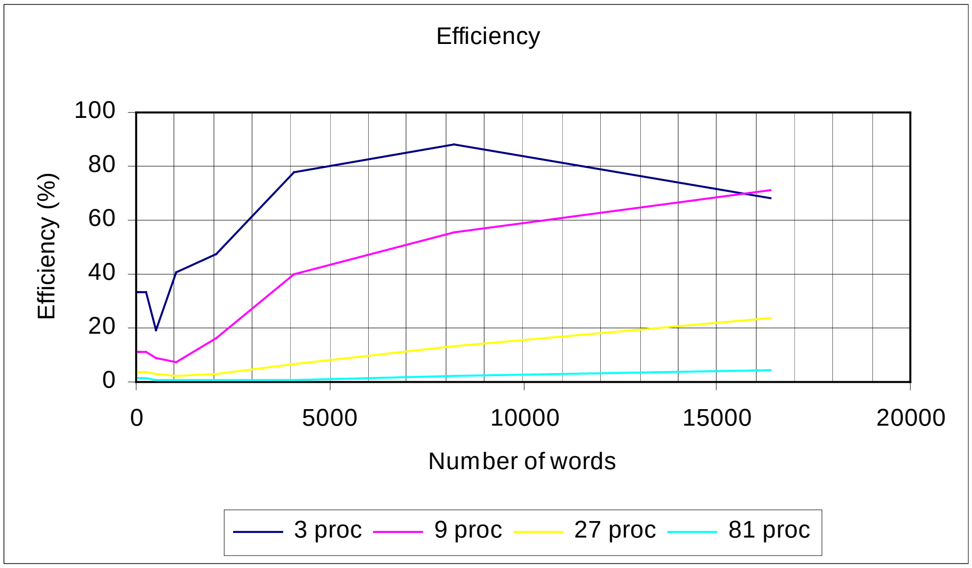

Figure 3.12. Graphical interpretation of the parallelization efficiency

Looking at the results, we can see that the maximum efficiency is given by using 3 processors when we multiply two numbers with the length

8192. The worse efficiency is given, as we expected, when we use 81 parallel threads, because of insufficient number of processors available.

Conclusions and future work

We don't guarantee that our implementation is the best one what can be made. For all that, our results are good enough to be mentioned.

Increasing the number of processes (or threads) over the number of available processors do not give better execution time results. The best result is given when the number of threads is less than the number of available processors.

We see that the speed-up is better for relative small numbers (less than 8192 words) when we use 3 or 9 processes. Only for numbers over the 16384 words it is worth to use 27 processes. This behavior is reflected by the efficiency, too.

Now, we are focus to the other three basic operations (addition, subtraction and division). For division we are working on a new parallel implementation of the Karatsuba algorithm which we hope that will give us notable results.

References

1. Jebelean, T. (1997). Using the Parallel Karatsuba Algorithm for Long Integer Multiplication and Division. Europar ’97 (European Conference on Parallel Processing, Passau, 26-29 August 1997), pp. 1169-1172. Springer-Verlag.
2. Jebelean, T. (1997). Application of the parallel Karatsuba algorithm to long integer multiplication and division. Technical Report no. 97-08. RISC – Linz, Austria.
3. Cesari, G.; Maeder, R. (1996). Performance Analysis of the Parallel Karatsuba Multiplication Algorithm for Distributed Memory Architectures. Journal of Symbolic Computation, 21, pp. 467-473.
4. Krandick, W.; Jebelean, T. (1996). Bidirectional Exact Integer Division. Journal of Symbolic Computation, 21, pp. 441-455.
5. Jebelean, T. (1996). Integer and rational arithmetic on MasPar. DISCO ’66, pp. 162-173. Karlsruhe, Germany, September 1996. Springer-Verlag LNCS 1128.
6. Weisstein, E. W. Karatsuba Multiplication. Available on-line at: http://mathworld.wolfram.com/KaratsubaMultiplication.html (Wolfram Research, Inc.)
7. Jebelean, T. (1994). Systolic multiprecision arithmetic. Technical Report no. 94-37, RISC-Linz, Austria. PhD Thesis.
8. Karatsuba, A.; Ofman, Yu. (1962). Multiplication of multidigit numbers on automata. *Sov. Phys. Dokl.*, 7:595-597.
9. Knuth, D. E. (1981). The art of computer programming, volume 2. Addison-Wesley, 2 edition, 1981.
10. Borwein, J. M.; Borwein, P. B.; Bailey, D. H. (1989). “Ramanujan, Modular Equations, and Approximations to Pi, or How to Compute One Billion Digits of Pi.” Amer. Math. Monthly 96, pp. 201-219, 1989.
11. Our SGI Cray Origin 2000 Documentation. Available on-line at http://origin.edvz.uni-linz.ac.at/hardware/ORIGIN.html and in the Insight Viewer.