Parallel integer multiplication

Samuel Vivien

Parallel integer multiplication

VIVIEN Samuel[*][†]

[*] *Département d'informatique de l'ÉNS, École normale supérieure, CNRS, PSL Research University, 45 rue d'Ulm,* Paris, France
samuel.vivien@ens.psl.eu
[†]*Université de Lorraine, CNRS, Inria, LORIA,* Nancy, France

January 24, 2022

***Abstract*—Multiplication is a fundamental step in many algorithms. If the multiplication of two integers of n words has a complexity of $M(n)$, divisions and squares can be computed in $O(M(n))$ as well and the greatest common divisor can be computed in $O(M(n)\log n)$. Thus being able to have a small value for $M(n)$ is extremely important.**

To this day, the best known algorithm for reachable values is the Schönhage-Strassen algorithm which is implemented by a few arithmetic libraries. Asymptotically faster algorithms exist, however no computer is able to hold numbers big enough for those algorithms to outrun Schönhage-Strassen.

The GNU Multiple Precision (GMP) library has a sequential-only implementation of Schönhage-Strassen.

However some algorithms contains a step which is a single big multiplication. Thus when trying to parallelize such an algorithm, one requires a parallel algorithm for multiplication. An example of such an algorithm is the batch factorization for Number Field Sieve. Thus people trying to implement a parallel version of such algorithms need to find an arithmetic library that implements a parallel integer multiplication.

An example of such a library is the Flint (Fast LIbrary for Number Theory) library that contains a parallel implementation of Schönhage-Strassen. In this article we present an implementation of Schönhage-Strassen, that reaches a speedup of 20 for the multiplication of two integers of 10^7 words of 64 bits using a Xeon Gold with 32 cores.

I. INTRODUCTION

Multiplication is a major arithmetic operation as many algorithms depend on this operation. Schönhage-Strassen's algorithm [1, 2] multiplies two integers of size n in $O(n \cdot \log n \cdot \log\log n)$ and is to this day the best algorithm known for integers of reachable sizes. Asymptotically faster algorithms exist [3], however no computer is able to hold numbers big enough for those algorithms to outrun Schönhage-Strassen [4]. If $M(n)$ is the cost of multiplying two integers of size n, many arithmetic operations can be computed in either $O(M(n))$ or $O(M(n)\cdot\log n)$ such as the greatest common divisor [5], square root, and division [6].

On the other hand, twenty years ago, the first multicore processors have been revealed (IBM POWER4). Since then, such processors have appeared on the market for the general public and today processors with multiple cores are ubiquitous. This means that in order to harness the performance of a computer, parallelization is required.

Some algorithms require a parallel multiplication algorithm in order to be parallelized efficiently. An example of such an algorithm is the batch cofactorization for the Number Field Sieve [7]. This algorithm implemented in CADO-NFS [8] contains a step which is one huge multiplication. Thus, the only way to parallelize this step of the algorithm is to have a parallel algorithm for multiplication. However the literature on parallel multiplication is sparse. There exist only a few articles presenting parallel multiplication algorithms for integers [9] or polynomials [10]. The only known implementations are a parallel version of Karatsuba [11] and the multiplication routine from Flint [12].

In this article we present a parallel implementation of Schönhage-Strassen's algorithm, based on the sequential implementation of this algorithm from the GMP library. This implementation is compared to the parallel implementation from the Flint library in order to assess its performance.

First of all, we present different existing implementations. This will be followed by a presentation of the modifications made to the implementation of Schönhage-Strassen in order to parallelize it. Then the global performances will be presented followed by a brief conclusion.

II. EXISTING IMPLEMENTATIONS

To this day there exist three implementations of Schönhage-Strassen's algorithm in C-libraries. The most notorious one is in GMP [13]. Another important library implementing it is Flint [12] that contains a multithreaded multiplication routine.

A third important implementation to acknowledge is an optimized version of GMP's implementation [14]. From now on we will refer to this code as GKZ.

Every figure in this article has been done by performing measures on a node from the cluster *grvingt* of the Grid'5000 infrastructure [15]. Such a node contains two Intel Xeon Gold 6130 CPUs and 192 GiB of RAM. Each CPU has 16 cores, leading to 32 cores in total and the capacity to hold 64 threads when using hyper-threading.

Figure 1 shows that out of the three implementations, GKZ is the fastest one. But it requires tuning in order to achieve such a performance resulting in a long overhead time before its first use on a computer. This figure also shows that Flint's code is slightly slower than the one from GMP. But as said before, the

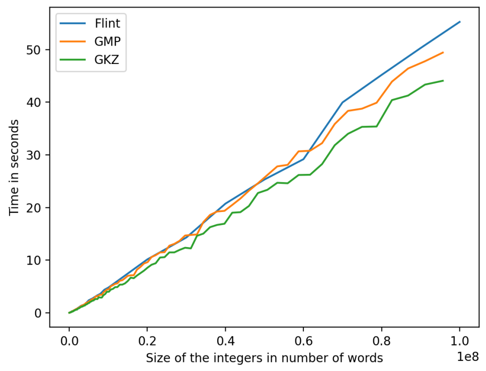

Figure 1. Time in seconds taken to multiply two integers of n words for GMP 6.2.1, GKZ's code, and the single threaded version of Flint.

code from the Flint library is multithreaded. Thus we can expect a better performance when we increase the number of threads.

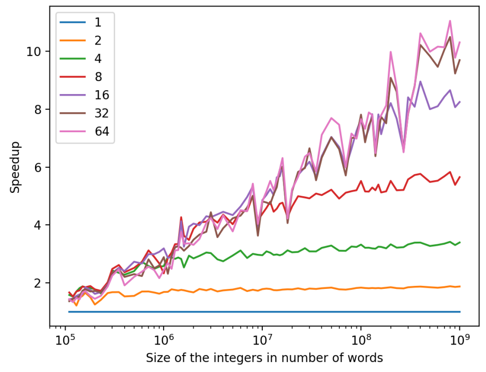

Figure 2. Speedup obtained when multiplying two integers of n words with the Flint library using t threads.

On Figure 2 we see that the speedup achieved when using multiple threads in Flint is rising when the size of integers increases. When using 32 threads, which means the whole computing power, it is only for integers of more than 10^8 words that the speedup manages to rise above 8. And for such sizes, the speedup is almost the same for 16, 32 or 64 threads. This shows that the Flint library restricts the amount of threads for small integers.

Another parallel implementation of Schönhage-Strassen is the one presented by Tsz-Wo Sze [16]. However he faced memory management difficulties because of the architecture of the computer he used as it had 6GB of RAM per node. Thus he is only 3 time faster than the performances we measured for GMP on our computer. This is why our results can hardly be compared.

III. Design of a parallel integer multiplication

The code described below is a modified version of the original code used inside the GMP 6.2.1 library. Some details have been knowingly omitted as they were not modified between the version of the code presented below and the original version from GMP. For further details about the original algorithm the reader should refer to the code inside GMP 6.2.1 and the existing literature on the subject [1, 2].

The Schönhage-Strassen algorithm is an evaluation-interpolation algorithm such as Karatsuba's and Toom-Cook's algorithms. This means that they rely on converting the integers into polynomials, on computing the product of the polynomials through evaluation and interpolation, and finally on converting the resulting polynomial back to an integer. This algorithm computes the result of the multiplication modulo $b^N + 1$ where N and b are detailed below and can be decomposed into 6 steps.

Decomposition: Each integer is rewritten as a vector of N integers. This vector represents the decomposition in some base b of the given integer.

Two direct FFTs: Perform a FFT on each vector. This is a integer FFT, thus we don't have any problem about the precision of the result.

Convolution: Compute the pointwise product of the two vectors.

Inverse FFT: Perform an inverse FFT on the vector obtained at the previous step.

Division: Division of each coefficient by the size of the vector; this step is usually counted inside the inverse FFT but we have separated it as those steps are different in term of implementation.

Evaluation: The vector now represents the polynomial resulting from the product of the polynomials computed in step ”decomposition”. Thus evaluating this polynomial in b will give the result of the integer multiplication.

In algorithm 1 and the following ones, bigInt are numbers of arbitrary size represented as an array of integers.

Algorithm 1: Multiplication routine

```
Data: Operand1, Operand2 : bigInt
Result: Out : bigInt
Op1Vector = decompose(Operand1);
Op2Vector = decompose(Operand2);
in_place_fft(Op1Vector);
in_place_fft(Op2Vector);
convolution(Op1Vector, Op2Vector);
in_place_inverse_fft(Op1Vector);
division(Op1Vector);
evaluation(Out, Op1Vector);
```

Those steps are detailed below, some of them being much more important than others in computing time.

A. *Decomposition*

In the decomposition step, one needs to decompose each integer in the base $b = 2^\ell$ into a vector of n-bit coefficients where $n > \ell$. Thus the program just needs to cut the integers into chunks of ℓ bits, copy those chunks into the vector and fill the remaining bits with zeroes. This is done in a for-loop where iterations are completely independent. One just needs to distribute the iterations to the threads and the parallelization is done. However this step raises two problems:

- The first problem is when the operands are larger than the modulo given for the multiplication. In the sequential code, it is just an Euclidean division, this means that a parallel subtraction routine had to be implemented (see III-F).
- The second problem is more complex. The author has observed that adding a synchronization barrier after the creation of the threads did in fact *speed up* the computation. The most probable reason for this difference is that accessing memory, while the memory mapping is being modified, slows down the whole computation drasticaly.

In algorithm 2 and the other parallel steps, we suppose in the pseudo code that nbThreads divides every number we want in order to simplify the presented code. Full case handling is available in the code available at the end of the paper.

Algorithm 2: Parallel decomposition

```
Data: Number : bigInt; $l$, $n$, vectorSize, threadId,
      nbThreads : int
Result: Vector : bigInt array
modulo = $2^l$ * vectorSize + 1;
compute_remainder(Number, modulo);
chunckSize = vectorSize / nbThreads;
for $i$ = chunckSize $\times$ threadId to chunckSize $\times$ (threadId + 1) - 1 do
    Vector[i] = [Number[i $\times$ b], ..., Number[(i + 1) $\times$ b - 1], 0, ..., 0];
```

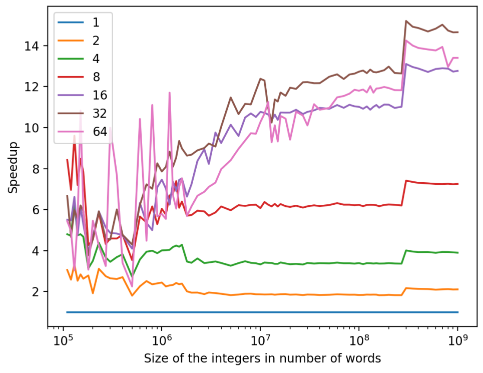

Figure 3. Speedup for the decomposition step when multiplying two integers of n words.

On Figure 3 we can see that the decomposition reaches a speedup of 12 for integers of more than 10^7 words with 32 threads. We could have hoped for a better performance but this is a memory intensive step, thus the limits also come from the capability of the cache to bring all the data to the processing unit. Another interesting thing we can see on this figure is that the performance with 64 threads is worse than with 32 threads. In other words, using hyper-threading decreases the computation power. This is not surprising as many multi-threaded linear algebra libraries advise against using hyper-threading.

B. FFTs

The three FFTs (two direct and one inverse) are very important steps because they take a large part in the total computation time. Different approaches have been tested to parallelize the FFTs.

Currently the code inside GMP is a recursive implementation of Cooley-Tukey's FFT [17]. The first approach we followed was to rewrite the recursive function with a loop. However, this worsened the performances by a factor 1.5. This was the consequence of an increase in the amount of cache misses.

There exists other FFT algorithms that are easier to parallelize and reduce the amount of cache misses such as the one presented by Bailey [18]. This algorithm considers the vector as a matrix and computes FFTs on each of the rows of the matrix, transposes it and computes FFTs on each row once again. Bayley's algorithm [18] was designed for matrices with floating-point coefficients, which take a few machine words, whereas here we have integer coefficients with hundreds or thousands of machine words. Vectors of such coefficients exceed the cache size, leading to much more cache miss than expected. And this algorithm requires transposing matrices which is quite difficult to parallelize and creates lots of cache misses. As we did not need the FFT to return an ordered FFT but only to be able to invert it, the choice has been made to remove the transposition step from Bailey's algorithm. This method gave the best results out of all the methods tested.

Two optimizations were also added in order to improve the performance. Each thread was bound to a CPU core in order to reduce thread migration. Also, each direct FFT is computed on a single CPU leading to the two direct FFTs happening at the same time.

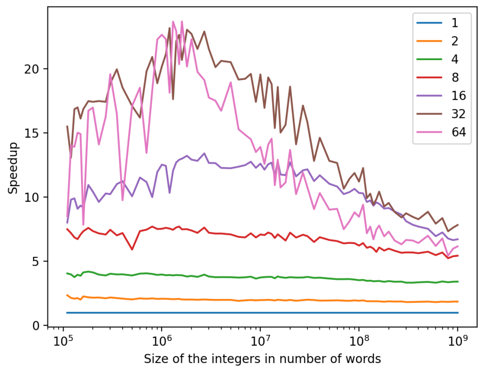

Figure 4. Speedup for the direct FFT step when multiplying two integers of n words.

However as we can see on Figure 4 the performance decreases for huge integers. This comes from the fact that the transposition step existed for a reason. It was there to reduce the number of cache misses. Thus removing this step increases the amount of cache misses for the sub-FFTs. The impact of them increases with the size of the integers. However the speedup goes above 20 for integers of $2 \cdot 10^6$ words and stays above 10 for integers of less than 10^8 words when using 32 threads. This is good because FFTs take a big part of the global computation time and achieving good performance on this part leads to better global performance.

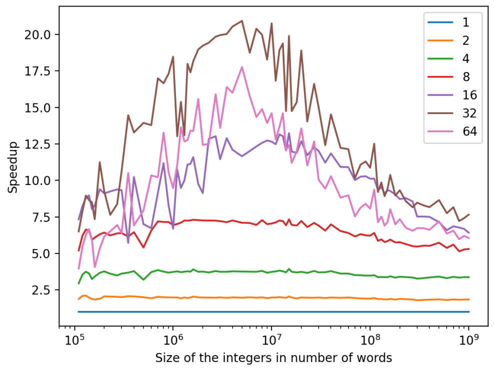

Figure 5. Speedup for the inverse FFT step when multiplying two integers of n words.

C. Convolution

The convolution is also a very important step as it takes roughly as much time as the 3 FFTs combined in the single-threaded version.

This step is only iterating on the vectors to compute the pointwise products. Thus we just need to cut the vectors into equal shares of consecutive indices and give each thread its share in order to parallelize this step.

Algorithm 3: Parallel convolution

```
Data: OutV, InV : bigInt vectors; vectorSize, threadId,
      nbThreads, modulo : int
chunckSize = vectorSize / nbThreads;
for $i$ = $chunckSize \times threadId$ to $chunckSize \times (threadId + 1) - 1$ do
    mpn_mul_mod(OutV[i], InV[i], modulo);
```

Due to its simplicity and the size of the data, the parallelization went smoothly and the performance followed. This step achieves good speedups that remain stable for sizes between 10^5 and 10^9 words as shown on Figure 6.

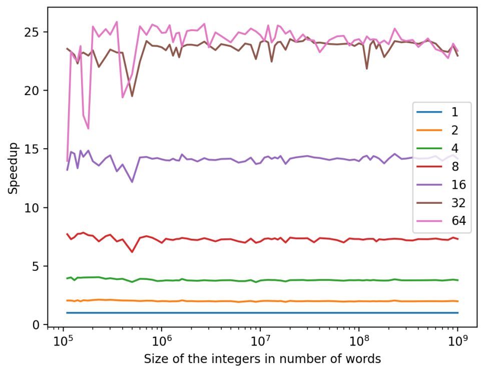

Figure 6. Speedup for the convolution step when multiplying two integers of n words.

Here again we see that hyper-threading is not of much help for huge sizes. However for this step, hyper-threading does not have a negative impact on performance.

D. Division

As the computation is done modulo $b^N + 1$, division is only a multiplication by the inverse. Thus, in this step, every coefficient of the output vector is multiplied by a power of two. However a problem arises, as this multiplication by a power of two has not been implemented in place.

In the sequential version of the code, the solution is to send each coefficient on the previous coefficient, when the first one was sent to an additional memory address. For the multi-threaded code it has been chosen that each thread would have its own additional memory and that every division will be done from the vector to the additional memory, before being copied back to the vector.

This approach allows to prevent allocating any additional memory as other parts of the code also need the same amount of additional memory (the size of a single coefficient) while the time trade-of caused by the additional copy was small because all the data is already inside the closest cache thanks to the operations that happened just before.

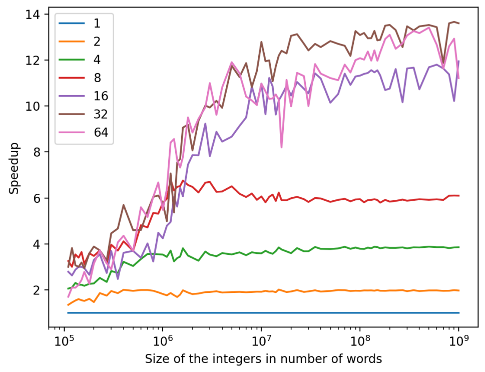

Figure 7. Speedup for the division step when multiplying two integers of n words.

On Figure 7 we can see that the division step reaches a speedup of 13 with 32 threads and as we have already noticed before, hyper-threading worsens performance. We can also see that for small integers, reducing the number of threads might improve performance. Some tuning might be useful for this step, in order to use more efficiently the amount of threads available.

E. Evaluation

The next step of the multiplication is the evaluation of the polynomial. In other words, every coefficient from the vector needs to be added to the result. Every coefficient takes at most $2\ell + \log_2 K$ bits where K is the dimension of the vector. This upper bound comes from the formulation when computing the product of two polynomials. If we cut the resulting number into chunks of ℓ bits, we can see that each coefficient can take up to 2 chunks and a few additional bits. Thus, this step might cause many data races if not handled well.

The best approach to minimize the amount of mutexes in this part of the computation is to copy as many data as possible inside the vector without creating any possible collisions (to avoid using locks). Afterward, a sequential or quasi-sequential code takes care of the carries. On the one hand one does not know where a carry will stop, thus the code requires lots of locks to guarantee its correctness. On the other hand, probabilities show that carries have a short life-time expectancy. Thus the impact of having a sequential carry propagation code is small on the overall performance except in a few worst case scenarios.

For this step, two different approaches are possible:

- The first one is currently used inside the code described in this article. It consists of cutting the coefficients into multiple pieces.
    1) One copies the first 2ℓ bits of each odd-indexed coefficient into the result.
    2) One adds the last $\log_2 K$ bits of each odd-indexed coefficient to the next coefficient (whose index is even), except for the last coefficient whose exceeding bits are added to the result.
    3) Now one adds the first 2ℓ bits of each even-indexed coefficient to the result and adds the carry to the exceeding bits of the coefficient.
    4) Sequentially add exceeding bits left in the even-indexed coefficients.

  The idea behind this approach is that the exceeding bits are usually one machine word only, when each coefficient takes around a thousand words for operands of 10^7 words. Thus the last part of this step is much faster than the other ones. However, as shown on Figure 8, the performance for this approach is much worse than the ones from the other steps, reaching a speedup of only 6-7 with 32 threads.

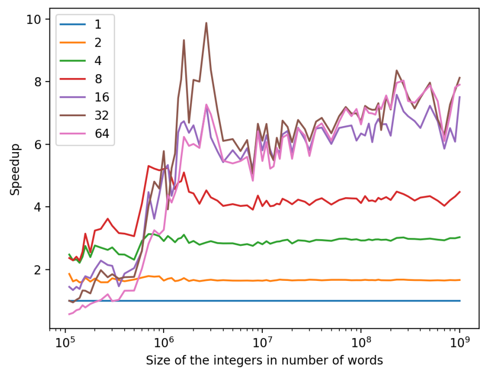

Figure 8. Speedup for the evaluation step when multiplying two integers of n words.

  As on Figure 7, we see on Figure 8 that for small integers it would be useful to limit the number of threads in order to achieve better performance. We can clearly see that for integers of $2 \cdot 10^5$ words, using 8 threads is better than using 32 threads which achieves the same performance as a single thread!

  However, this step does not take as much time as the FFTs and the convolution. When multiplying two integers of 10^8 words each, this step only uses 3% of the overall computation time. Thus efforts were concentrated on more time-consuming steps.

- The second approach was thought to be better than the first one.

  The idea was to split the vectors in equal share and that each thread computes the evaluation step on his share before adding the overlapping parts at the end.

  However even if this approach was more efficient theoritically we didn't managed to observe any gain.

F. *Chinese Remainder*

The Schönhage-Strassen algorithm is a modular multiplication algorithm as it returns the result of the multiplication modulo $b^N + 1$ where N is the size of the vectors. The Chinese Remainder Theorem says that if we know the result of the multiplication for different coprime moduli we can deduce the result of the multiplication modulo the product of the moduli. This means that we can change one large multiplication into multiple smaller ones. This adds a last step to the multiplication routine, called Chinese Remainder, whose effect is to reunite the results on various rings into the overall result. Inside GMP there exist two implementations for this Chinese Remainder step: a default version and an older one that can be reactivated.

Let M be an integer larger than the sum of the sizes of the operands.

- By default the multiplication routine used for big integers is `mpn_mul_nussbaumer` (in GMP 6.2.1) which computes the results of the multiplication modulo $2^{M/2} + 1$, $2^{M/4} + 1$ and so on, until the modulo is small enough for another algorithm to outrun Schönhage-Strassen. Then the result is computed via a series of Chinese Remainder steps.
- The original version of the code can be enabled by the flag `--enable-old-fft-full` when configuring GMP. This version computes multiplications modulo $2^{3M/5} + 1$ and $2^{2M/5} + 1$. Then the result is computed via a single Chinese Remainder step.

Both codes call the same function afterward that computes the product of the two operands with the given modulo. Thus the previous work of parallelization works for both at the same time. But experiments have shown that small multiplications are much harder to parallelize as shown in previous figures. Thus the code used for the performances shown in this article is using the original version of the code. This choice also allowed a much easier parallelization of the Chinese Remainder step of the computation since only two rings were used.

However, the Chinese Remainder step is still quite a tricky step because it does not contain any loop, and every arithmetic operation uses the same piece of data. Thus, everything has to be done sequentially. The only way to improve the performance of this part of the computation when increasing the number of threads is to implement a parallel version of the basic arithmetic functions used in this step. Thus the following operation have been multi-threaded:

Copy: This one was easy to implement, each thread calls the normal copy routine on its part of the number.
  We also have a routine `mpn_copy_n`$(out, in, size)$ that copy an integer of $size$ machine words starting at address in into the integer starting at address out.

Logical shift right of s bits (with s smaller than a word size): This operation requires synchronization between the

Algorithm 4: Parallel copy

```
Data: Out, In : bigInt; size, threadId, nbThreads : int
chunckSize = size / nbThreads;
mpn_copy_n(Out + threadId × chunckSize, In +
  threadId × chunckSize, chunckSize);
```

threads. First of all, every thread stores the s smallest bits of the share of the next thread. Then, when everybody is done computing this, every thread calls the normal logical shift right routine on its part of the number. And finally each thread places the saved bits from the first step to the s highest bits of its share.

Algorithm 5: Parallel logical shift right (rshift)

```
Data: Out, In : bigInt; size, $s$, threadId, nbThreads : int
Result: OutLimb : shifted out bits if threadId is 0, 0
        otherwise
chunckSize = size / nbThreads;
saved = $(2^s - 1)$ & In[chunckSize × (threadId + 1)];
if threadId = 0 then
  | OutLimb = $(2^s - 1)$ & In[0];
else
  L OutLimb = 0;
sync();
mpn_rshift(Out + threadId × chunckSize, In +
  threadId × chunckSize, chunckSize, $s$);
Out[r * chunckSize - 1] = Out[r * chunckSize - 1] |
  (saved $\times 2^{\text{word size}-s}$ )
```

Addition: This operation also requires synchronization, but at the end. First, every thread calls the normal addition routine on his share. Then, when every thread completed its task, it adds to the result the potential carry that came from the previous chunk.

Algorithm 6: Parallel addition

```
Data: Out, In : bigInt; size, $s$, threadId, nbThreads : int
Result: OutCarry : int
chunckSize = size / nbThreads;
carry = mpn_add_n(Out + threadId × chunckSize, In
  + threadId × chunckSize, chunckSize);
OutCarry = 0;
sync();
mutex_lock();
OutCarry += mpn_incr(Out + (threadId + 1) ×
  chunckSize, OutCarry, size - (threadId + 1) ×
  chunckSize);
mutex_unlock();
```

Subtraction: The approach for this operation could have been similar to the one for the addition but it has been done differently with an approach closer to the one for the logical shift right.

First of all, each thread computes the incoming borrow of its share. This means that if its share is from the n-th word to the m-th word, it compares the two numbers given in input restricted to their n first words to see whether the result of the subtraction is negative or not in order to predict a potential incoming borrow.

Once this is done for every thread, they can all compute the subtraction on their share using a GMP routine computing subtraction with a given carry.

The parallelization of the above operations allows to improve the performance of the Chinese Remainder step. However this code requires much more synchronization than the previous steps. This might be the reason why the performance on small integers is so bad when using many threads as shown on Figure 9.

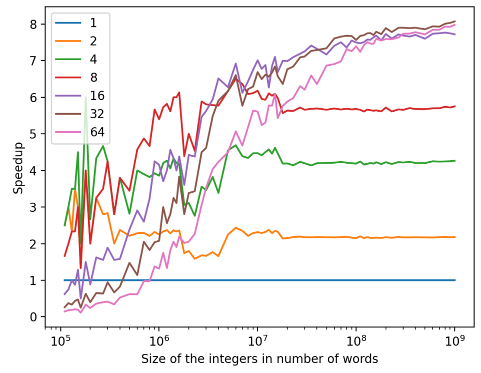

Figure 9. Speedup for the Chinese Remainder step when multiplying two integers of n words.

When working on this part of the code, the author noticed that binding each thread to a specific core had a huge impact[1] on the time required by the threads to wake up after a barrier.

Even if Figure 9 shows that this part of the code still requires some work to be efficient when using many threads, it is nice to notice that the speedup with 2 or 4 threads is already optimal.

A last solution to improve performance is to reduce the number of threads used for smaller integers because having too many threads degrades the performance. For example, we clearly see on Figure 9 that for integers of less that 10^6 words, using 8 threads is much better than using 32. The need of reducing the number of threads for small integers could already be seen on Figure 8.

IV. Assessing the global performance

Figure 10 shows a chronogram of the multiplication between two integers of 10^7 words each. Each of the 32 horizontal stripes represents a thread and the upper stripe represents the part of the code currently running. A white rectangle means that the thread is idle while a colored rectangle means that the thread is currently active. The goal of this graph is to detect sequential parts of the code that have a significant impact on the overall performance. This graph also permits to detect parallel parts of the code that take a significant part of the overall computation time. In the highest line, the two blue parts represent the two calls of the Schönhage-Strassen algorithm (modulo $2^{3M/5} + 1$

[1]There exists a factor 10 for the time required by this step between the binded and the non-binded versions when multiplying two integers of 10^7 to 10^8 words.

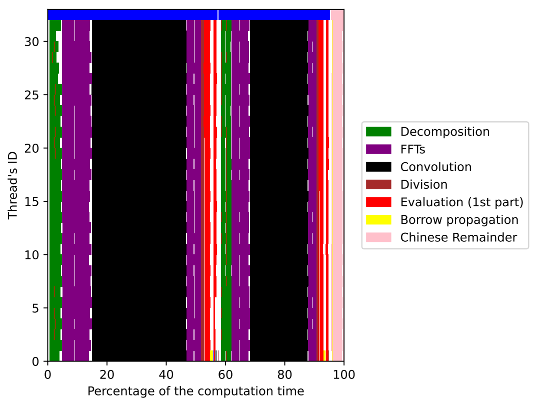

Figure 10. Chronogram of the multiplication of two integers of 10^7 words with 32 threads.

first and $2^{2M/5}+1$ afterward) and the pink part is the Chinese Remainder step.

This chronogram helps us see different informations about the performance. First of all, we clearly see that the evaluation step might require some additional work as the yellow part uses only a single core.

We can also see that the workload for convolution (black) is well distributed as we do not see much white at the end, meaning that we do not have threads finishing this part before others. The optimum would be for the whole graph to look like that.

On the other hand, some work should maybe be done in the decomposition and the FFTs, by distributing the workload better in order to prevent one thread being idle while the others are still working.

We can also see from the evaluation and the Chinese Remainder steps that adding too many barriers for synchronizing the threads has a negative effect on performance as we can see many white parts at the end of the chronogram.

Now that we have an idea of the performance of the different steps of the multiplication we look at the performance of the whole computation.

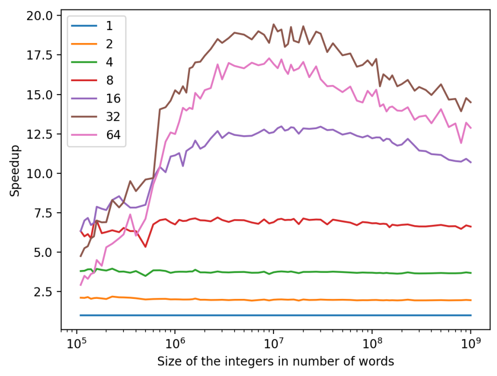

Figure 11. Global speedup when multiplying two integers of n words.

Figure 11 represents the time taken between the multiplication routine with 1 thread divided by the time for t threads for various sizes. It shows and confirms different things about the code.

- First of all we can see that the speedup almost reaches a value of 20 with 32 threads when given two integers of 10^7 words in input. Even if only a few steps reached a ratio above 15, this confirms that concentrating efforts on the convolution and the FFTs is important.
- The impact of the FFTs is once again confirmed by the decrease in speedup with integers of more than $2 \cdot 10^7$ words. A decrease that is a direct consequence of Figure 4.
- As already seen on every figure shown above, hyper-threading has a negative impact on the performance.
- The speedup is quite small for integers of less than 10^6 words. In fact restricting the number of threads used could even improve the performance. For example we can see on Figure 9 that the Chinese Remainder step would benefit greatly of using a smaller number of threads.

One last thing to acknowledge is that every measure was done without any other program running. Thus, the measures done with a single thread had the advantage of having the whole computer for themselves when threads had to share the L3 cache with one another when the program was run with 32 threads.

However all the modifications made to achieve such performances came with a cost. The single-threaded computation time has increased by 10% compared to the GMP sequential version.

Another indicator than the speedup would be to compare the performance of this new code with the Flint library.

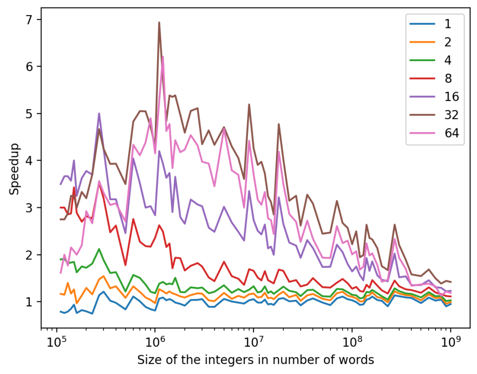

Figure 12. Time taken to multiply two integers of n words with Flint divided by the time for our code for different amounts of threads.

As we can see on Figure 13, our code completely outruns Flint for integers from 10^5 to 10^8 for more than 16 threads. However the ratio above 5 for integers of around 10^6 words decreases toward a mere 1.5 for 10^9 words. This decrease is due to two effects that we have already seen before: the increase of the speedup with the size of the integers for Flint shown on Figure 2 and the decrease of speedup for our code that comes from the performance of the FFT on big sizes as shown on Figure 4.

V. Conclusion

In this article we have seen how it is possible to modify the existing multiplication code inside GMP in order to reach outstanding parallel performance. This code is a proof of

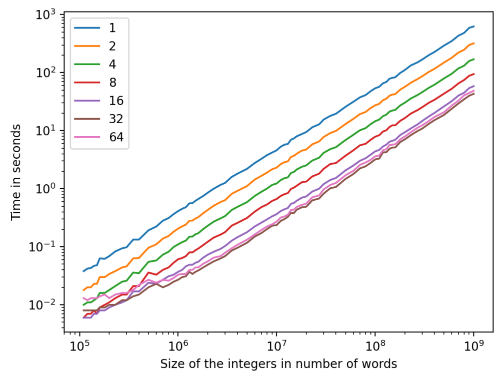

Figure 13. Time taken to multiply two integers of n words with our code.

concept that greater parallel performance can be reached for integer multiplication. While some work is still required in order to improve the portability of the code, performance can be improved even more.

The code is downloadable at https://github.com/samsa1/integer-para-mul

Acknowledgement

The author would like to thank Emmanuel Thomé and Paul Zimmermann for their guidance and the rest of Caramba team at LORIA for hosting him on the internship that led to this paper.

References

[1] A. Schönhage and V. Strassen. "Schnelle Multiplikation großer Zahlen". In: *Computing* 7 (1971), pp. 281–292. DOI: 10.1007/BF02242355. URL: https://doi.org/10.1007/BF02242355.

[2] Richard Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Ed. by Cambridge University Press. 2010.

[3] David Harvey and Joris Van Der Hoeven. "Integer multiplication in time O(n log n)". In: *Annals of Mathematics* 193 (2021), pp. 563–617. DOI: https://doi.org/10.4007/annals.2021.193.2.4. URL: https://hal.archives-ouvertes.fr/hal-02070778.

[4] Christoph Lüders. "Implementation of the DKSS Algorithm for Multiplication of Large Numbers". In: *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation*. ISSAC '15. Bath, United Kingdom: Association for Computing Machinery, 2015, pp. 267–274. ISBN: 9781450334358. DOI: 10.1145/2755996.2756643. URL: https://doi.org/10.1145/2755996.2756643.

[5] Damien Stehlé and Paul Zimmermann. "A Binary Recursive Gcd Algorithm". In: *Algorithmic Number Theory, 6th International Symposium, ANTS-VI, Burlington, VT, USA, June 13-18, 2004, Proceedings*. Vol. 3076. June 2004, pp. 411–425. ISBN: 978-3-540-22156-2. DOI: 10.1007/978-3-540-24847-7_31.

[6] Daniel J. Bernstein. "Fast multiplication and its applications". In: (2004). URL: http://cr.yp.to/papers.html#multapps.

[7] Daniel J. Bernstein. "How To Find Small Factors Of Integers". In: (2000). URL: cr.yp.to/factorization.html.

[8] The CADO-NFS Development Team. *CADO-NFS, An Implementation of the Number Field Sieve Algorithm*. Release 2.3.0. 2017. URL: http://cado-nfs.gforge.inria.fr/.

[9] Viktor Bunimov and Manfred Schimmler. "Efficient Parallel Multiplication Algorithm for Large Integers". In: *Euro-Par 2003 Parallel Processing*. Ed. by Harald Kosch, László Böszörményi, and Hermann Hellwagner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 923–928. ISBN: 978-3-540-45209-6.

[10] Changbo Chen et al. "Parallel Integer Polynomial Multiplication". In: *SYNASC 2016 - 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. Timisoara, Romania, Sept. 2016, pp. 72–80. DOI: 10.1109/SYNASC.2016.024. URL: https://hal.archives-ouvertes.fr/hal-01520021.

[11] Tudor Jebelean. "Using the parallel Karatsuba algorithm for long integer multiplication and division". In: vol. 1300. Apr. 2006, pp. 1169–1172. ISBN: 978-3-540-63440-9. DOI: 10.1007/BFb0002869.

[12] William Hart and Flint development team. *Flint (Fast Library for Number Theory)*. Version c1acfda772b20a55c525ec5418bf934dc34f5e45. July 2021. URL: https://github.com/wbhart/flint2.

[13] T. Granlund and the GMP development team. *GNU MP : The GNU Multiple Precision Arithmetic Library*. Version 6.2.1. 2020. URL: http://gmplib.org/.

[14] Pierrick Gaudry, Alexander Kruppa, and Paul Zimmermann. "A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm". In: *ISSAC 2007*. Ed. by C. W. Brown. Proceedings of the 2007 international symposium on Symbolic and algebraic computation. Waterloo, Ontario, Canada: ACM Press, July 2007, pp. 167–174. DOI: 10.1145/1277548.1277572. URL: https://hal.inria.fr/inria-00126462.

[15] Daniel Balouek et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In: *Cloud Computing and Services Science*. Ed. by Ivan I. Ivanov et al. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20. ISBN: 978-3-319-04518-4. DOI: 10.1007/978-3-319-04519-1_1.

[16] Tsz-Wo Sze. "SchöNhage-Strassen Algorithm with MapReduce for Multiplying Terabit Integers". In: *Proceedings of the 2011 International Workshop on Symbolic-Numeric Computation*. SNC '11. San Jose, California: Association for Computing Machinery, 2012, pp. 54–62. ISBN: 9781450305150. DOI: 10.1145/2331684.2331693. URL: https://doi.org/10.1145/2331684.2331693.

[17] James W. Cooley and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". In: *Mathematics of Computation* 19.90 (1965), pp. 297–301. URL: http://www.jstor.org/stable/2003354.

[18] David Bailey. "FFTs in External or Hierarchical Memory." In: vol. 4. Jan. 1989, pp. 211–224. DOI: 10.1007/BF00162341.