CiSJ

computing@tanet.edu.te.ua
www.tanet.edu.te.ua/computing

ISSN 1727-6209
International Scientific
Journal of Computing

DESIGNING A MULTI-PRECISION NUMBER THEORY LIBRARY

Cătălin Hriţcu, Iulian Goriac, Raluca Mihaela Gordân, Elena Erbiceanu

Faculty of Computer Science, "Al. I. Cuza" University of Iaşi, Romania
Email: mpnt@infoiasi.ro, Web address: www.infoiasi.ro/~mpnt

Abstract: *The aim of this paper is twofold. First, we present the basic principles and point out the main difficulties in writing a library supporting operations with arbitrarily large numbers. Aspects such as library structure, number representation, algorithm selection, memory management, etc., are discussed and exemplified on the most efficient libraries developed. Secondly, we present work in progress regarding the design of a new multi-precision library, MpNT. Comparisons between our library and the existing ones show that it achieves high performance.*

Keywords: *large numbers, multiple precision, high performance, unlimited precision, library, design principles, number theory algorithms, cryptography*

1. INTRODUCTION

Cryptography applications and research require multiple precision computations at high speed. Internet security protocols, probabilities and statistics and numerical calculus are other domains where very large numbers are often involved. Therefore, computations with large numerical data (having more than 10 or 20 digits, for example) need specific treatment.

Most programming languages, as C and C++, provide only limited precision numerical data types. This precision is architecture dependent and is often not high enough. Recent programming languages, like Java™ and Python, have build-in multi-precision capabilities, but being highly portable often also implies an unacceptable efficiency loss.

Mathematical software, such as Maple or Mathematica, also offers the possibility to work with unlimited precision. Such software can be used to easily prototype algorithms or to compute constants but it is usually neither very efficient nor portable.

The most efficient solution for multiple precision computing is the use of a multi-precision library. Several such libraries have been proposed, most of them being free software released under the GNU General Public License.

LIP [1] is one of the first libraries for arbitrary length integer arithmetic. It was originally written by Arjen K. Lenstra and was later maintained by Paul Leyland. Being written in pure ANSI C, it is highly portable but not very efficient.

LiDIA [2] is a library for computational number theory, developed at the Technical University of Darmstadt and organized by Thomas Papanikolau. LiDIA provides a collection of highly optimized implementations of various multi-precision data types and time-intensive algorithms.

CLN [3] was written by Bruno Haible and is currently maintained by Richard Kreckel. It is a C++ library that implements elementary arithmetical, logical and transcendental functions and has a rich set of classes. CLN is memory and speed efficient.

NTL [4] is written and maintained mainly by Victor Shoup. It is portable and can be used in conjunction with GMP for enhanced performance.

PARI [5] was developed at Bordeaux by a team led by Henri Cohen and is capable of performing formal computations on recursive types at high speed. It is primarily aimed at number theorists and has an extensive algebraic number theory module.

GMP [2] was developed by Törbjord Granlund and the GNU free software group. GMP is a C library for arbitrary precision arithmetic with a general emphasis on speed. It uses highly optimized assembly code for the most common inner loops for a lot of CPUs. In fact GMP is generally faster than any other multi-precision library.

MpNT is a multi-precision number theory package developed at the Faculty of Computer Science, "Al. I. Cuza" University of Iaşi under the guidance of Professor, Ph.D. Ferucio Laurenţiu Ţiplea. This new ISO C++ library was started as a base for cryptographic applications. However, it may be used in any other domain where efficient large number computations are required. For the time being the library supports integer, modular and floating point arithmetic with practically unlimited precision. It is both speed efficient and highly portable without disregarding code structure and clarity. MpNT is freely available according to the GNU Lesser General Public License. Therefore, any criticism and suggestions are warmly welcomed.

In this paper we present some basic principles in writing a multi-precision library. Three main goals are to be achieved when implementing such a library: **efficiency**, **portability** and **functionality**. Developing involves making a series of choices and tradeoffs that will essentially affect the characteristics of the final product. For a number theory library it is hard to completely satisfy all these requirements. Thus, many products of this kind have been developed, giving the user the possibility to choose. There is no unanimously accepted solution to this matter, and new approaches are still found every day. Nevertheless, certain common lines should be followed while designing such a library.

The paper is organized into two parts. The first part presents basic principles for designing a multi-precision number theory library. The second part provides comparisons between MpNT and four well-known libraries: GMP, CLN, PARI and NTL.

2. PROGRAMMING LANGUAGE

A well-written program using only assembly code is very fast, but lacks portability and is very hard to maintain. On the other side, developing the same program in a high-level language will make it easily portable, easy to understand and maintain, but some efficiency will be lost. Therefore, a compromise solution is to use both. As a high-level language, C++ makes a good choice because it retains C's ability to deal efficiently with the fundamental objects of the hardware (bits, bytes, words, addresses, etc.), while providing the flexibility of an object-oriented language. Assembly language should be used only for the most frequently called functions.

Therefore, MpNT uses ISO C++ for the main part of the library because it is highly portable and the fastest high-level programming language available. A clean and intuitive interface was built using OOP. Classes provide data hiding, guaranteed initialization of data, implicit type conversion for user defined types and mechanisms for overloading operators. We also took advantage of the superior type checking, default arguments and inline substitution of functions, and the reference type provided by C++.

Assembly language is used only for the small machine-dependent kernel that is intended to increase the performance of the library, because it is closest to what the hardware architecture really provides. For portability purposes, this set of routines is also available in plain C++.

3. LIBRARY STRUCTURE

Developing an easy to maintain and extend library requires some sort of modular structure. The best approach is to group the functions in layers, each of them having a different level of abstraction. It is desirable that only low-level functions have direct access to number representation.

The MpNT library is structured into two layers: the kernel and the C++ classes.

The MpNT kernel contains small, carefully optimized routines that are easy to rewrite for different architectures. Most of the kernel functions operate on arrays of digits, such as: comparisons, bitwise operations and basic arithmetical operations. However, they are risky to use because they assume that certain relations between operands hold and that enough memory has been allocated for the results. Special optimizations apply for the Intel IA-32 and compatible processors under Windows and Linux. Because of similarities in the number representation, the capability of using the GMP [6] or even CLN [3] kernel as an alternative may be easily added.

The application programming interface is intended to be as intuitive, consistent, and easy-to-use as possible. This can be achieved by providing the classes that best model the mathematical concepts, hiding the actual implementation. In our case, the C++ classes, such as **MpInt**, **MpMod** and **MpFloat**, provide a safe and easy to use interface. These classes also hide the functions of the kernel; therefore any code relying upon them will have a high level of independence. Backward binary compatibility throughout the library development is more than desirable.

The **MpInt** class provides multi-precision integer arithmetic: addition, subtraction, multiplication, division, greatest common divisor, bit operations etc. All available operators for the **int** type are also defined for objects of the **MpInt** class; therefore they can be regarded as normal integers, but with no size restrictions.

The **MpMod** class provides multi-precision modular arithmetic. Only one modulus can be used at a specific time, and the numbers are automatically reduced. Functions determining the multiplicative inverse and performing modular multiplication and exponentiation are provided along with other basic modular operations (addition, subtraction etc.).

The **MpModulus** and **MpLimLee** classes offer high performance modular reduction, multiplication and exponentiation using pre-computed modulus or base information.

The **MpFloat** class provides floating point arithmetic with user-selectable precision. Each object has its own precision limited only by available memory.

4. NUMBER REPRESENTATION

Number representation highly depends on the features provided by the hardware architecture, including: registers' dimensions, instruction set, cache sizes, parallelism level provided etc.

MpNT uses signed-magnitude representation for its multi-precision integers (members of the MpInt class). The current implementation of the class includes four private attributes:
- a field that uses every bit independently to store a logical value (a flag). One bit stores the sign of the number. Two more bits keep special status information to avoid unnecessary copying by the overloaded operators. The other bits are yet unused.
- the magnitude of the number, an array of digits stored "little-endian". For best performance digits have the size of the microprocessor's word.
- the number of digits used for the magnitude. The number zero is represented by setting this field to zero.
- the number of digits allocated for the magnitude.

This representation provides quick access to class information and is easily extendible; the yet unused flag bits may also store other information regarding a multi-precision integer.

Floating point numbers (MpFloat class) currently have three private attributes:
- the mantissa of the number, a multi-precision integer.
- the precision of the number, a simple-precision integer, determining the location of the radix point.
- the virtual precision of the number, a simple-precision integer, determining the number of digits past the radix point used for further computations.

This allows us to change the precision of the number very fast and without truncation. The use of whole digits also facilitates fast floating point computations.

5. ALGORITHM SELECTION

In many cases several algorithms may be used to perform the same operation depending on the length of the operands. Of course, the ones with the best *O*-complexity are preferred when dealing with huge numbers, but on smaller operands a simpler, highly optimized algorithm may perform much better. This is why careful performance testing is required to find out the limits of applicability.

Even though in MpNT we implemented more than one algorithm for some operations, the interface functions will use only the routines or the combination of routines proved to be most efficient. A detailed correctness and complexity analysis of the implemented algorithms can be found in [7].

Usually, while looking for efficient implementations various tricks are used and a new problem arises: the correctness of implementation. This can be regarded as an instance of a more general problem, the software validation problem. Much work has been devoted to this problem, especially to find automatic procedures for validation. The Coq proof assistant [8] is one such tool. It has been used, for instance, to validate the GMP implementation of the Zimmermann's square root algorithm [9]. Proofs are developed using the correctness tool to deal with imperative features of the program. The formalization is rather large (more than 13000 lines) and requires some advanced techniques for proof management and reuse (see [10] for other attempts of GMP procedure validations).

6. MEMORY MANAGEMENT

The most frequently used memory allocation policy is on-demand allocation (allowing the user to explicitly allocate memory). Additional space may be transparently allocated whenever a variable does not have enough (e.g., GMP, NTL). This is easy to implement but the user has responsibilities in managing memory. This drawback may be eliminated by using a garbage collector (e.g., CLN), but the speed overhead could be unacceptable. Memory leaks may also be prevented by the use of class destructors. Some libraries give the user the possibility to choose the allocation technique that best suits his application or even to use his/her own memory management routines (e.g., LiDIA).

The memory management policy adopted in MpNT is based on explicit allocation of memory. To avoid frequent reallocation, when the exact amount of necessary memory is known, the user may make such a request. For the same reason, whenever reallocation occurs, we provide a little more space than needed. Memory may be released either on demand or automatically by the class destructors.

7. ERROR HANDLING

The desirable approach is to signal the occurred errors, allowing the user to choose the actual handling policy. This involves supplementary checking, it is time consuming and can make the code harder to read and maintain. Therefore a frequent approach is to ignore errors, which surely involves some risks, but eliminates the overhead.

We chose not to ignore errors so MpNT uses the throw-try-catch mechanism provided by C++ to signal exceptions to the user.

8. COMPARISONS

The comparisons were performed on a small set of basic functions: multiplication (Fig.1), greatest common divisor (Fig.2), modular reduction (Fig.3) and modular exponentiation (Fig.4). The time versus operand size (measured in 32 bits digits) graphs below illustrate that the best results belong to GMP and CLN immediately followed by our library.

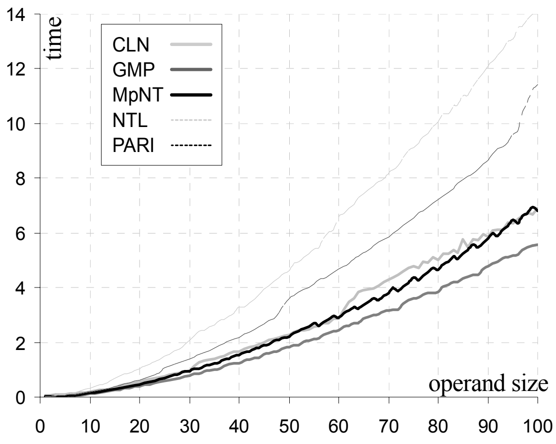

Fig.1 – Multiplication

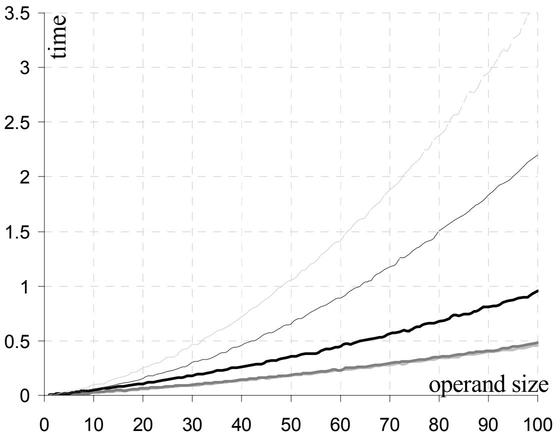

Fig.2 – Greatest Common Divisor

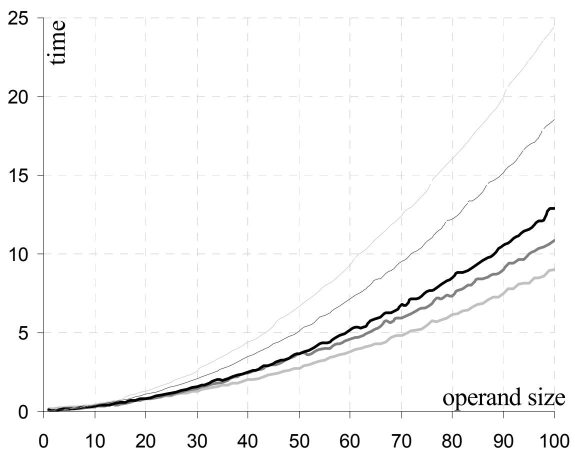

Fig.3 – Modular Reduction

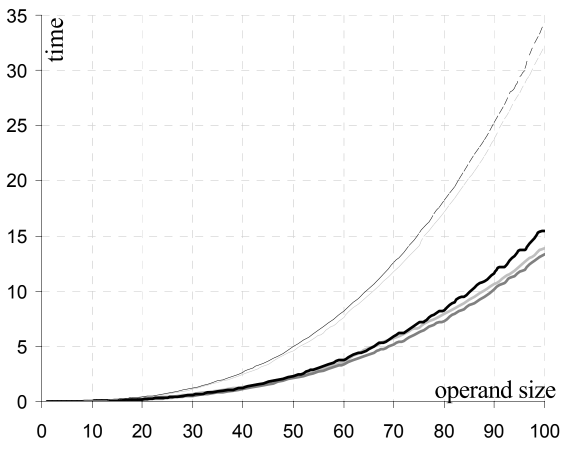

Fig.4 – Modular Exponentiation

The versions of the libraries compared were: CLN 1.1.5, GMP 4.1, MpNT 0.1pre1, NTL 5.2 and PARI 2.2.4.alpha. Default options were used for building and installing these libraries from sources. The test system had an 800MHz AMD processor and was running Linux (Mandrake 9.0).

9. REFERENCES

[1] A. Lenstra. LIP – Long Integer Package, Bellcore, http://usr/spool/ftp/pub/lenstra/LIP.

[2] LiDIA Group. LiDIA: A C++ Library for Computational Number Theory, Darmstadt University of Technology, http://www.informatik.tu-darmstadt.de/TI/LiDIA/.

[3] B. Haible, R. Kreckel. CLN – Class Library for Numbers, http://www.ginac.de/CLN/.

[4] V. Schoup. NTL: A Library for Doing Number Theory, http://www.shoup.net/ntl/

[5] PARI-GP Group. PARI-GP, http://www.parigp-home.de/.

[6] GMP Group. GMP – The GNU Multiple Precision Arithmetic Library, www.swox.com/gmp/.

[7] F.L. Ţiplea, S. Iftene, C. Hrițcu, I. Goriac, R.M. Gordân, E. Erbiceanu. MpNT: A Multi-precision Number Theory Package. Number-Theoretic Algorithms (I), Faculty of Computer Science, "Al. I. Cuza" University of Iaşi, Technical Report TR03-02 (2003), http://thor.info.uaic.ro/~tr/tr.pl.cgi

[8] Coq Site. The Coq Proof Assistant, http://pauillac.inria.fr/coq/

[9] P. Zimmermann. A Proof of the GMP Square Root Using the Coq Assistant, Rapport de recherche 4475, INRIA, 2002.

[10]P. Zimmermann. A Proof of the GMP Fast Division and Square Root Implementations, Rapport de recherche, INRIA, 2000.

Cătălin Hrițcu *was born in 1982 in Suceava, România. He is currently a student in Computer Science at the "Al. I. Cuza" University of Iaşi. His research interests include mathematics of computation, compiling techniques, distributed operating systems, peer-to-peer networks and software development.*

Iulian Goriac *was born on July 11th, 1978 in Suceava, România. He is currently a student of the Faculty of Computer Science of "Al. I. Cuza" University of Iaşi. His areas of interests include artificial intelligence, cryptography, software engineering and software validation techniques.*

Raluca Mihaela Gordân *was born in 1982 in Paşcani, România. She is currently a student of the Faculty of Computer Science of "Al. I. Cuza" University of Iaşi. Her areas of interests are cryptography, software development and the design of algorithms.*

Elena Erbiceanu *was born on May 23rd, 1982, in Paşcani, România, and is currently a third year student at the Faculty of Computer Science, "Al. I. Cuza" University of Iaşi. Her areas of interests include coding theory and cryptography, graphic interfaces design, algorithm design and software development.*