A Runtime Test of Integer Arithmetic and Linear Algebra in LEDA

Michael Seel

December 19, 1996

Abstract

In this Research Report we want to clarify the current efficiency of two LEDA software layers. We examine the runtime of the LEDA big integer number type *integer* and of the linear algebra classes *integer_matrix* and *integer_vector*.

Contents

1 Introduction

The integer arithmetic package and the linear algebra package in LEDA [MNU95] is heavily used in exact geometric calculations. To give an impression of the efficiency of this two software modules is the objective of this report. We want to test the two layers against different "competitors". In the area of big integer arithmetic it was very difficult to find a package which had the same functionality as LEDA's *integers*. We finally ended only with the GNU *Integer* class from the *g*++ library [Lea92]. Thus the tests were limited with respect to the compiler technology to *g*++. To get an impression of the performace of the linear algebra package we faced two mathematical allround competitors from the professional league, which of course are not limited to linear algebra and where we cannot know any internal design issues. However we considered *Maple* [CGG+91] and *Mathematica* [Wol96] at least as experienced judges of the game.

This is basically a test protocol. We provide the code which allowed us to set up the test environment and write down our experience.

2 Runtime Results

This section contains the test descriptions for the LEDA-*integer*– GNU-*Integer* comparison and for the linear algebra examination between LEDA, Mathematica and Maple.

In a first test we compared the execution time of the basic arithmetic operations $+$, $-$, $*$ and $/$ operating of numbers of a certain bit length of both libraries. Iterating the test gives us measurable run time results. We proceed as follows. We created two tables with 100 rows and entries of a certain bit length. Then we iterated on adjacent rows a million times. See the implementation section for details. We received the following results on an Ultrasparc 140:

#Bits	#Ops	LEDA | GNU in seconds							
		$+$		$-$		$*$		$/$	
32	10^6	1.1	5.1	1.3	5.5	1.6	7.5	4.7	26.6
64	10^6	1.2	5.0	1.3	5.4	4.2	10.9	24.8	36.6
100	10^6	1.6	5.3	1.7	5.5	7.8	17.3	31.0	47.9
500	10^6	1.8	7.3	1.86	7.8	158.8	272.7	225.7	375.3
1000	10^6	2.7	10.4	2.8	13.4	591.6	1063.4	736.5	1281.3
10000	10^6	20.8	76.6	21.1	77.7	4.5 h	28.3 h	16.4 h	30.3 h
		machine arithmetic: double | long							
53|32	10^6	0.2	0.2	0.2	0.2	0.2	0.4	0.3	0.6

In the second test we examined the solution of non homogenous linear systems calculated by the three different software products. We set up the following scenario: We created sqared matrices and a suitable column vector of a certain dimension and with randomly generated integer entries of 32 Bit (the *long* data type). After converting the created test data into input data for Mathematica and Maple, we took the time of the linear solver module of each software package. The timing test of Maple and Mathematica was done within each package by the respecting timing function for the calculation. The test was executed on a Sparc 10 with 40 MHz processor speed:

dimension	LEDA	Maple V R3	Mathematica 2.0
20	1.3 s	15.8 s	31.6 s
30	7.6 s	92.4 s	211.3 s
40	28.4 s	363.7 s	840.3 s
100	32 min	$>$ 1 h	$>$ 1 h

2.1 Testing the LEDA integer package

We compare the LEDA big integer number type *integer* (from now on *leda_integer* with GNUs type *Integer* (from now on *gnu_integer* from the GNU C++ Library. No other package of their functionality was generally available at the time of this runtime test. Thus we are limited to make the arithmetic tests with the GNU compiler. We create tables with 101 entries and do the runtime tests by iteration over this table. The accumulation test is appended to demonstrate the only weakness of the LEDA implementation. GNU beats LEDA there, as GNU has a special `operator+=(long)` whereas LEDA uses the standard operator after an automatic cast.

⟨*integer_test.c*⟩≡

```
#include <stdio.h>
#include <Integer.h>
#include <LEDA/integer.h>
#include <LEDA/random_source.h>

#define TRACE(t) cout << " " << t << " "
#define TRACEN(t) cout << " " << t << "\n"

#define TABLE_SIZE 101

typedef integer leda_integer;
typedef Integer gnu_integer;

⟨the timing routines⟩

main(int argc, char* argv[])
{
  if (argc != 3)
    error_handler(1,"usage: integer_test #bit_length #iterations");

  long bit_length, iteration_num, iterations;

  sscanf(argv[1],"%il",&bit_length);
  sscanf(argv[2],"%il",&iteration_num);
  random_source S(31);

  leda_integer* leda_arr = new leda_integer[TABLE_SIZE];
  gnu_integer* gnu_arr   = new gnu_integer[TABLE_SIZE];
  leda_integer* leda_divarr = new leda_integer[TABLE_SIZE];
  gnu_integer* gnu_divarr   = new gnu_integer[TABLE_SIZE];
  double* double_arr =     new double[TABLE_SIZE];
  long* long_arr     =     new long[TABLE_SIZE];

  leda_integer ledai;
  gnu_integer gnui;

  ⟨table creation of leda\_arr and gnu\_arr⟩

  ⟨summation test of equal bit length⟩
  ⟨subtraction test of equal bit length⟩
  ⟨multiplication test of equal bit length⟩
  ⟨division test of equal bit length⟩

  ⟨summation accumulation test with random longs⟩
}
```

We create two tables of size `TABLE_SIZE` for each type *leda_integer* and *gnu_integer* filled with random (but for both tables equal) numbers of demanded bitlength *bit_length*. We create each entry by a shifting summation of random longs ($< 2^{31}$) which basically concatenates 31-Bit patterns. For the division test we additionally need a table of double bit length.

⟨*table creation of* `leda\_arr` *and* `gnu\_arr`⟩≡

```
  long actnum;
  int shiftnum = bit_length / 31;
  int dshiftnum = 2*shiftnum;
  unsigned long shift = 1;
  for (int i = 0; i< 31; i++)
    shift *= 2;

  cout << "creating tables...\n";
```

```
  for (int i = 0; i < TABLE_SIZE; i++) {
    leda_arr[i]=0;
    gnu_arr[i]=0;
    for (int j = 0; j < shiftnum; j++) {
      S >> actnum;
      leda_arr[i] = leda_arr[i]*shift + actnum;
      gnu_arr[i] = gnu_arr[i]*shift + actnum;
    }
    for (int j = 0; j < dshiftnum; j++) {
      S >> actnum;
      leda_divarr[i] = leda_divarr[i]*shift + actnum;
      gnu_divarr[i] = gnu_divarr[i]*shift + actnum;
    }
    S >> actnum; while (actnum == 0) S >> actnum;
    double_arr[i] = double(actnum);
    long_arr[i] = actnum;
  }
  cout << "one array row:\n" << leda_arr[0] << "\n" << gnu_arr[0] << "\n";
  cout << "one array row:\n" << leda_divarr[0] << "\n" << gnu_divarr[0] << "\n";
  cout << "now testing...\n";
```

The timing routines are based on an iteration over a fixed sized array for a number of times.

⟨*the timing routines*⟩≡

```
template <class ARITH>
float time_sumop(ARITH* A_array, long iterations)
{
  ARITH erg;
  int outerit = iterations / (TABLE_SIZE-1);
  float end, start = used_time();
  for (int i=0; i<outerit; i++)
    for (int j=0; j<TABLE_SIZE-1; j++)
      erg = A_array[j] + A_array[j+1];
  end = used_time(start);
  cout << iterations << " adds in time " << end << "\n";
  return end;
}
template <class ARITH>
float time_subop(ARITH* A_array, long iterations)
{
  ARITH erg;
  int outerit = iterations / (TABLE_SIZE-1);
  float end, start = used_time();
  for (int i=0; i<outerit; i++)
    for (int j=0; j<TABLE_SIZE-1; j++)
      erg = A_array[j] - A_array[j+1];
  end = used_time(start);
  cout << iterations << " subs in time " << end << "\n";
  return end;
}
template <class ARITH>
float time_mulop(ARITH* A_array, long iterations)
{
  ARITH erg;
  int outerit = iterations / (TABLE_SIZE-1);
```

```
    float end, start = used_time();
    for (int i=0; i<outerit; i++)
      for (int j=0; j<TABLE_SIZE-1; j++)
        erg = A_array[j] * A_array[j+1];
    end = used_time(start);
    cout << iterations << " muls in time " << end << "\n";
    return end;
  }
```

For the division routine we use a second array which provides arguments of double length.

⟨*the timing routines*⟩+≡

```
  template <class ARITH>
  float time_divop(ARITH* A_array1, ARITH* A_array2, long iterations)
  {
    ARITH erg,divisor;
    int outerit = iterations / (TABLE_SIZE-1);
    float end, start = used_time();
    for (int i=0; i<outerit; i++)
      for (int j=0; j<TABLE_SIZE-1; j++) {
        erg = A_array1[j] / A_array2[j];
      }
    end = used_time(start);
    cout << iterations << " divs in time " << end << "\n";
    return end;
  }
```

Here we show the only case when LEDA is less efficient than the GNU arithmetic. If we accumulate longs into the respective integer type and use the +=-operation, GNU uses only half the time. This is due to the fact that GNU provides an adapted +=-operator for long arguments, whereas LEDA enforces the usage of the +=-operator with argument type `integer` after invoking a cast operation. For the follwing accumulation loop over 10^6 +=-operations LEDA needed 3.18 seconds but GNU only 1.36 seconds.

⟨*summation accumulation test with random longs*⟩≡

```
    cout << "LEDA long acc add speed: #" << iteration_num;
    float start = used_time();
    ledai = S();
    for (int i = 0; i < iteration_num; i++) {
      ledai += S.get();
    }
    float end1 = used_time(start);
    cout << " result in " << end1 << " seconds\n";

    cout << "GNU long acc add speed: #" << iteration_num;
    start = used_time();
    gnui;
    for (int i = 0; i < iteration_num; i++) {
      gnui += S.get();
    }
    float end2 = used_time(start);
    cout << " result in " << end2 << " seconds\n\n";
```

The following chunks just contain all bundled timing tests.

⟨*summation test of equal bit length*⟩≡

```
cout << "LEDA integers: " << bit_length << " Bits ";
time_sumop(leda_arr,iteration_num);
cout << "GNU integers: " << bit_length << " Bits ";
time_sumop(gnu_arr,iteration_num);
cout << "longs: ";
time_sumop(long_arr,iteration_num);
cout << "doubles: ";
time_sumop(double_arr,iteration_num);
cout << "\n";
```

⟨*subtraction test of equal bit length*⟩≡

```
cout << "LEDA integers: " << bit_length << " Bits ";
time_subop(leda_arr,iteration_num);
cout << "GNU integers: " << bit_length << " Bits ";
time_subop(gnu_arr,iteration_num);
cout << "longs: ";
time_subop(long_arr,iteration_num);
cout << "doubles: ";
time_subop(double_arr,iteration_num);
cout << "\n";
```

⟨*multiplication test of equal bit length*⟩≡

```
cout << "LEDA integers: " << bit_length << " Bits ";
time_mulop(leda_arr,iteration_num);
cout << "GNU integers: " << bit_length << " Bits ";
time_mulop(gnu_arr,iteration_num);
cout << "longs: ";
time_mulop(long_arr,iteration_num);
cout << "doubles: ";
time_mulop(double_arr,iteration_num);
cout << "\n";
```

⟨*division test of equal bit length*⟩≡

```
cout << "LEDA integers: " << bit_length << " Bits ";
time_divop(leda_divarr,leda_arr,iteration_num);
cout << "GNU integers: " << bit_length << " Bits ";
time_divop(gnu_divarr,gnu_arr,iteration_num);
cout << "longs: ";
time_divop(long_arr,long_arr,iteration_num);
cout << "doubles: ";
time_divop(double_arr,double_arr,iteration_num);
cout << "\n";
```

2.2 Testing the Linear Algebra Module

In this part we examine the solution of non homogenous linear systems calculated by three different software products. We set up the following scenario: We create sqared matrices and a suitable column vector of a certain dimension and with randomly generated integer entries of 32 Bit (the *long* data type). We can easily do this in LEDA with the types *integer_matrix* and *integer_vector* of the LEDA linear algebra module and the *random_source*. Then we convert the created test data types into input data in two files, which are each in the suitable input format for Mathematica and Maple. Afterwards we take the time of the linear solver module of each software.

⟨*la_test.c*⟩≡

```
#include <LEDA/stream.h>
#include "integer_matrix.h"
#include <LEDA/integer.h>

void print_in_maple_format(ostream&, const integer_matrix&);
void print_in_maple_format(ostream&, const integer_vector&);
void print_in_mathematica_format(ostream&, const integer_matrix&);
void print_in_mathematica_format(ostream&, const integer_vector&);

⟨maple output format⟩
⟨mathematica output format⟩
⟨a random matrix generator⟩
⟨setting up the test⟩
```

This chunk provides the file output routines for maple matrix and vector format.

⟨*maple output format*⟩≡

```
void print_in_maple_format(ostream& O, const integer_matrix& M)
{
  O << "matrix(" << M.dim1() << "," << M.dim2() << ",[";
  for (int i = 0; i < M.dim1(); i++) {
    for (int j = 0; j < M.dim2(); j++) {
      O << M(i,j);
      if (i == (M.dim1()-1) && j == (M.dim2()-1))
        O << "]";
      else
        O << ",";
    }
  }
  O << ")";
}

void print_in_maple_format(ostream& O, const integer_vector& v)
{
  O << "vector(" << v.dim() << ",[";
  for (int i = 0; i < v.dim(); i++) {
    O << v[i];
    if (i == (v.dim()-1))
      O << "]";
    else
      O << ",";
  }
  O << ")";
}
```

This chunk provides the file output routines for mathematica matrix and vector format.

⟨*mathematica output format*⟩≡

```
void print_in_mathematica_format(ostream& O, const integer_matrix& M)
{
  O << "{";
  for (int i = 0; i < M.dim1(); i++) {
    O << "{";
    for (int j = 0; j < M.dim2(); j++) {
      O << M(i,j);
```

```
      if (j != (M.dim2()-1))
        O << ",";
    }
    O << "}";
    if (i != M.dim1()-1)
      O << ",";
  }
  O << "}";
}
```

```
void print_in_mathematica_format(ostream& O, const integer_vector& v)
{
  O << "{";
  for (int i = 0; i < v.dim(); i++) {
    O << v[i];
    if (i != (v.dim()-1))
      O << ",";
  }
  O << "}";
}
```

We easily fill the matrix and the vector by longs from the LEDA random number generator.

⟨*a random matrix generator*⟩≡

```
void fill_with_random_entries(integer_matrix& M, int entry_size)
{
  random_source ranso(-entry_size,entry_size);
  for (int i = 0; i < M.dim1(); i++)
    for (int j = 0; j < M.dim2(); j++)
      M(i,j) = ranso.get();
}
```

```
void fill_with_random_entries(integer_vector& b, int entry_size)
{
  random_source ranso(-entry_size,entry_size);
  for (int i = 0; i < b.dim(); i++)
    b[i] = ranso.get();
}
```

We do the following:

1. we create the matrix and vector instance of the required dimension
2. we fill it with random long entries
3. we create the Maple and Mathematica input files
4. we trigger the LEDA test

⟨*setting up the test*⟩≡

```
main(int argc, char* argv[])
{
  if (argc != 2)
    error_handler(1,"usage: la_test #matrix_size");
  long mdim;
  sscanf(argv[1],"%il",&mdim);
  integer_matrix M(mdim,mdim);
  integer_vector b(mdim);
```

```
    fill_with_random_entries(M,mdim);
    fill_with_random_entries(b,mdim);
```

⟨*creating a maple input file*⟩
⟨*creating a mathematica input file*⟩

```
    float end;
    integer D;
    integer_vector x;
    float start = used_time();
    cout << "M =\n" << M << "\n";
    cout << "b =\n" << b << "\n";
    bool solvable = linear_solver(M,b,x,D);
    end = used_time(start);
    cout << "LEDA time = " << end << "\n";
    if (solvable)
      cout << "solution = " << x << "\n divided by " << D << "\n";
    else
      cout << "not solvable\n";
  }
```

For Maple we have to create a file `mp1` which looks like the following example:

```
with(linalg):
M := matrix({\tt 12,13,24],[233,34,456],[234,32,1});
b := vector([23,34,33]);
linsolve(M,b);
```

This file can be executed in Maple by the command `read mp1; <return>`.

⟨*creating a maple input file*⟩≡

```
  {
    file_ostream maple_f("mp1");
    maple_f << "with(linalg):\n";
    maple_f << "M := ";
    print_in_maple_format(maple_f,M);
    maple_f << ";\nb := ";
    print_in_maple_format(maple_f,b);
    maple_f << ";\nlinsolve(M,b);\n";
  }
```

For Mathematica we have to create a file `mt1` which looks like the following example:

```
M = {{4,3,5},{123,34,54},{12,17,1111}};
b = {34,21,111};
Timing[LinearSolve[M,b]]
```

This file can be executed in Mathematica by the command `<<mt1 <return>`.

⟨*creating a mathematica input file*⟩≡

```
  {
    // the mathematica output:
    file_ostream math_f("mt1");
    math_f << "M = ";
    print_in_mathematica_format(math_f,M);
    math_f << ";\nb = ";
    print_in_mathematica_format(math_f,b);
    math_f << ";\nTiming[LinearSolve[M,b]]";
  }
```

References

[CGG+91] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. *Maple V Language reference manual.* Springer, 1991.

[Lea92] D. Lea. *User's Guide to the GNU C++ Library*, 1992.

[MNU95] K. Mehlhorn, S. Näher, and C. Uhrig. *The LEDA User Manual*, 1995.

[Wol96] S. Wolfram. *The Mathematica Book, 3rd ed.* Wolfram Media/Cambridge University Press, 1996.