POSIX Threads Polynomials (PTPol): a scalable implementation of univariate arithmetic operations

Mohab Safey El Din
UPMC/CNRS LIP6, Equipe SPIRAL
Projet SALSA, INRIA
61 avenue Kennedy
75016 Paris (France)
Mohab.Safey@lip6.fr

Philippe Trébuchet
UPMC/CNRS LIP6, Equipe SPIRAL
Projet SALSA, INRIA
61 avenue Kennedy
75016 Paris (France)
Philippe.Trebuchet@lip6.fr

ABSTRACT

In this paper, we describe the design of a `C` library named `PTPol` implementing arithmetic operations for univariate polynomials and report on practical experiments showing the relevance of using threads on recent multi-core computers.

We show how to use efficiently an API named `OpenMP` and POSIX Threads to achieve scalability. On multi-core architectures, we obtain a speed-up equivalent to the number of cores on addition and multiplication on univariate polynomials for degrees **a completer**.

Categories and Subject Descriptors

G.0 [**Mathematics of Computing**]: General

General Terms

Algorithms, application

Keywords

Univariate polynomial, symbolic-numeric computation, scalability

1. INTRODUCTION.

PTPOL stands for POSIX Thread POLynomials. It is a `C99` library implementing arithmetic operations on univariate polynomials. Our goal is to reach peak performances on modern multi-processors machines. This question is of first importance since now, most popular architectures are based on multi-core processors and/or multi-processors. First of all let us say that we have focused on shared memory models because :
1) it avoids costly Inter Process Communication (IPC),
2) most of our testing machines have enough memory for performing the needed computations,
3) multi-core and multiprocessors are quickly spreading and it seemed to us fundamental to be able to exploit them efficiently, and
4) it is necessary to efficiently deal with this model for doing multi scale parallel computation (e.g. computations over a grid where tasks are themselves done in parallel using threads).

In [5], a theoretical approach of parallel programming using the PRAM model can be found. Following these results, one can expect a speed-up on arithmetic operations on univariate polynomials which is equivalent to the number of cores of a multi-core architecture if we develop programming techniques to reduce the cost of data fetching and processor communication.

Efficient use of multi-core processors can be done by different ways. The *classical* one is to use `OpenMP` which is a multi-platform API providing a shared-memory parallel programming model. It works with widespread languages as `C/C++` and `Fortran`, and it is available on many architectures. This API aims at achieving scalability without making a strong effort on the implementation. The main drawback of `OpenMP` is that, when using it naively, it does not allow fine control of the total (i.e. global) number of threads. This is very inconvenient on recursive algorithms and it is important to notice that the over-head introduced by a naive use of `OpenMP` is not negligible.

To reach our aim, we developed PTPOL in two ways. First, we succeeded in using efficiently `OpenMP` by numbering all the thread created and forbidding the creation of too many threads at the same time. Since this induces non-negligible modifications of the sequential code, we also studied the use of POSIX Threads to check if one can obtain a speed-up appearing for polynomials of smaller degree.

Finally, using cleverly `OpenMP` or POSIX Threads, we succeeded to obtain a speed-up on arithmetic operations on univariate polynomials which is equivalent to the number of cores of the machine used to perform the computations. The degree at which this speedup is reached depend mostly on the cost of the arithmetic of the ground field: more the cost of the arithmetic is expensive, bigger the speedup is.

In the following, we first describe the library, then we develop the different ways we achieved scalability. At last, we

PASCO'07, July 24–27, 2007, Waterloo, Canada.

report on practical experiments.

2. DESCRIPTION OF THE LIBRARY

PTPOL is written in plain C99 to allow maximum portability which is a key feature for a library that aims at running on the most recent architectures. More particularly, we try to take advantage of multi-core architectures to obtain a speed-up related to the number of cores.

It is independent of the ground field arithmetic. This feature is achieved using the C pre-processor to generate *on the fly* the correct functions. Choosing C as an implementation language for PTPOL has been done regarding three aspects. First, multi-threading a polynomial arithmetic means multi-threading operations that may become quite fine grain on small degrees. Hence, for obtaining an improvement of the performances on these small degrees one has to decrease as far as possible the cost of calling arithmetic functions. Though there are quite lightweight, C++ interfaces to GMP induce an overhead that must be avoided. Second, PTPOL was designed to construct *dynamic* libraries that could be loaded at running time. C++ ABI (Application Binary Interface) is not constant across compilers. Thus, each time one changes the used compiler, a reconstruction of the interface is required. Last, we wanted PTPOL to be able to run using GMP of course, but also MPFR (that, up to very recently, had no working C++ interface), MPFI, MPAI, MPC and some other libraries that do not have any C++ interface.

PTPOL assumes that the ground field arithmetic follows the GMP interface. In other words, the ground field functions are assumed to be named as *field_name*_operation_name, and the macro PTPOL_PREFIX will be substituted to *field_name*.

PTPOL implements dense univariate polynomials. This is dictated by experience. Indeed it appears that almost all situations in which intensive univariate polynomial computations are requested, are such that it is dense univariate polynomials that are requested. The same constatation led Shoup in [6] to implement its univariate polynomials also with dense storage.

The implemented operations are addition/soustraction, naive multiplication, Karatsuba's multiplication and euclidean division. An ongoing work is to implement asymptotically optimal operations such as FFT.

For user-friendliness the authors took a great care in mimicing GMP behavior when implementing PTPOL . That is to say that anybody minimally familiar with GMP will be able to use PTPOL without any problem.

A polynomial is a complex structure that might be reallocated during the computations you are performing with. Hence you must take care to perform a correct initialization and to free resources allocated to a polynomial when it is no more used. This scheme is exactly the one that is customary using GMP. All the functions are such that their first argument is the argument in which the result of the function is stored.

3. DIFFERENT WAYS TO ACHIEVE SCALABILITY.

We study here the use of OpenMP on the one hand, and the direct use of the POSIX Threads API on the other hand. OpenMP is an API allowing the introduction of parallel execution with minimal code modification. Our goal is to allow efficiency and scalability without inducing a strong development time.

Using OpenMP. We restrict ourselves here to give a hint on the mechanism of OpenMP only for the C programming language, the interested reader is urged to have a look at [3] for a comprehensive presentation of this API. OpenMP in C is introduced by inclusion of the file openmp.h and by annotations in the C code, by means of pragma, to tell the compiler that a section of code shall be run in parallel.

OpenMP automates the parallelization of some sections of code. Hence, it introduces some overhead in the computation that makes it significantly heavier. To expect some practical improvements, the granularity of the threads created by OpenMP has to be the coarsest possible. Otherwise, the total computational time is dominated by the time needed for creating the threads. We have used the GOMP implementation of OpenMP.

OpenMP allows only to control the number of threads which are created in a loop or for dealing with a section. There is no global way to control the *total* number of threads that are run on the machine by OpenMP. Controlling the total number of created threads is more difficult and without special care about that, it may become important which would induce a great loss of efficiency. Recursive algorithms are a case where this phenomenon occurs. Thus, to avoid getting plethora of threads, we have modified our program: instead of defining a parallel section which contain recursive calls, we have programmed two functions, one that effectively has a parallel section and the other that is purely sequential. The recursive calls that need not create new threads are all done calling the purely sequential function. A shared counter on the number of threads allows to control it .

Finally, let us mention that although this technique allows significant speedup in most cases, it induces significant changes in the source code and its deployment is tedious. Thus, it is natural to try achieving scalability with POSIX Threads (inducing also some changes in the sequential code) at least to check if one can obtain a significant speedup for polynomials of smaller degree.

Posix Threads by hand. The POSIX threads API does not aim at automatic parallelization. It just allows the programmer to create new threads. We refer the reader to [1, 2] for a description of the POSIX thread API. As the programmer has to choose what will be performed in parallel, it allows a full control on what occurs in each thread. Most importantly the programmer can make use of assumptions the OpenMP API cannot be told of: no aliasing on some pointers, read only expressions on some part of the code, no need for synchronization for accessing particular variable etc.

The main drawback of using directly POSIX thread API inside the arithmetic is that it leads to perform many system

call that may inder the performances on small examples. The authors' experience with the POSIX thread API shows that, once you have defined the correct structure for argument passing the deployment of POSIX thread is rather simple. For launching the different threads we constructed an argument passing structure, i.e. a structure containing references to all the parameters.

In practice, it took us almost the same time to make use of the POSIX thread API as the one we spent to introduce the correct thread counting methods using `OpenMP`.

4. EXPERIMENTAL BEHAVIOR

Experiments reported here have been performed on an Intel Core Duo 2GHz, with 2GB of DDR2 667MHz RAM. The programs where compiled with `gcc (GCC) 4.3.0 2007/02/16` which provides an implementation of `OpenMP` called `GOMP`.

We consider here addition and multiplication of univariate polynomials with different ground fields. The timings of our sequential implementation compares favorably with the ones of `NTL`.

We compared the obtained timings our sequential implementation with the parallel implementations we did using `OpenMP`, as described above, and POSIX Threads.

It appears that when the coefficients have an expensive arithmetic (e.g. rational `mpq_t`, big integers `mpz_t`) `OpenMP` and `Posix` Threads behave more or less the same, i.e. quickly reach the desired speedup in addition and Karatsuba's multiplication.

More precisely, for polynomials with coefficients in `mpz_t` with 16 digits integer (resp. `mpq_t` with 16 digits integer as numerator and denominator) we obtain a speedup on the addition of 2 for polynomials having degree at least 4000 (resp, 2500). For the Karatsuba's multiplication, this speedup appears for polynomials having degree at least 2000 (resp. 700) for coefficients in `mpz_t` (resp. `mpq_t`). If the coefficients are bigger the obtained speedup appears earlier.

Note that up to degree 100 (resp. 70) one obtains a speedup of 1.5 for Karatsuba's multiplication of polynomials with coefficients in `mpz_t` (resp. `mpq_t`). Up to degree 900 (resp. 900) one obtains a speedup of 1.5 for the addition of polynomials with coefficients in `mpz_t` (resp. `mpq_t`).

Using `OpenMP`, With `float` coefficients, the speedup of 2 appears earlier using `OpenMP`, up to degree 10000 for Karatsuba's multiplication and up to degree 9000 for the addition. For `double` coefficients, the speedup of 2 appears up to degree 10000 for the addition.

Note that up to degree 2000 (resp. 3000) one obtains a speedup of 1.5 for Karatsuba's multiplication of polynomials with `float` (resp. `double`) coefficients. Up to degree 4000 (resp. 5000) one obtains a speedup of 1.5 for the addition of polynomials with (resp. `double`) coefficients.

5. REFERENCES

[1] Single Unix Specification ISBN 1931624437

[2] POSIX standard ISO/IEC 9945-3: 2003 (IEEE Std. 1003.3: 2001)

[3] `OpenMP` Open MP Specification 2.5 May 2005 (http://www.openmp.org/drupal/node/view/75)

[4] C programming language Norm ISO-Standard ISO/IEC 9899:1999

[5] D. Bini, V. Pan Polynomial and matrix computation. *Progress in Theoretical Computer Science*, Birkhaüser, 1994

[6] Shoup, V., NTL: A Library for doing Number Theory, (http://shoup.net/ntl/)