Reliable Computing 1 (3) (1995), pp. 317–323

A parallel complex zero finder

Mark J. Schaefer and Tilmann Bubeck

A recent paper [7] describes a variable precision interval arithmetic algorithm for the computation of the zeros of an analytic function inside a given rectangle and to a user-specified accuracy. The algorithm is based on the argument principle in the set of complex numbers **C** and carries much potential for parallelization at various levels of granularity. Here we explain how to modify the sequential algorithm to take advantage of parallelism at four levels ranging from coarse to medium grain. The algorithm is tested in a distributed environment consisting of eight SPARC workstations. The underlying software environment is also discussed.

Параллельная процедура поиска комплексных нулей

М. Шефер, Т. Бубек

В недавней работе [7] описан интервально-арифметический алгоритм переменной разрядности для вычисления нулей аналитической функции внутри заданного прямоугольника с точностью, определяемой пользователем. Этот алгоритм основан на принципе аргумента на множестве комплексных чисел **C** и обладает большим потенциалом параллелизации на разных уровнях гранулированности. В настоящей работе предлагается модификация последовательного алгоритма с использованием параллелизации на четырех уровнях гранулированности: от грубого до среднезернистого. Алгоритм тестировался в распределенной вычислительной сети, состоящей из восьми рабочих станций SPARC. Опеределенное внимание уделено программному обеспечению нижнего уровня.

1. Introduction

The verification of zeros of analytic functions in the complex plane is commonly based on the argument principle and computation of winding numbers (see [6], Remarks to Chapter 5). The winding number of an analytic function f with respect to a simple closed contour S in the complex plane is the number of times the point $f(z)$ winds around the origin in the image plane as z traces S once in the positive direction. The argument principle applied to analytic functions states that, in the absence of zeros of f on S, the winding number equals the number of zeros of f interior to S, counting multiplicities. It is clear that given an initial rectangle R in the complex plane whose sides contain no zeros of f, a simple bisection strategy coupled with an algorithm for computing winding numbers can serve as a basis for locating zeros of f inside R.

The algorithm in [7], hereafter referred to as the *sequential algorithm*, is a hybrid method: given a starting rectangle and a suitable elementary function, the first phase of the program serves to isolate a function's zeros within individual subrectangles. This is accomplished through the computation of winding numbers using interval methods and rectangular bisection. The

second phase employs Newton's iteration to locate individual zeros rapidly and is only applied to first order zeros. The first phase is needed to ensure that no zeros are missed and to provide, for each first order zero, a reasonable starting location and bounded search area for Newton's method. Higher order zeros are handled correctly by the program but their presence can increase execution times significantly because convergence to them is only linear.

When interval methods are used to compute the winding number of an analytic function f with respect to some rectangle R, three outcomes are possible: R is found not to contain any zeros, R is found to contain one or more zeros, or else the winding number could not be computed and the test for zeros was inconclusive. The third case can occur either because a zero lies directly on the boundary of R or else because a zero lies close to the boundary and the current precision of computation is insufficient to resolve the behavior of $f(\partial R)$, where ∂R denotes the boundary of R (more details are given in [7]). The sequential algorithm monitors the number of rectangles encountered for which the winding number could not be computed and periodically adjusts the precision of computation to keep this number in check.

During its first phase, the sequential algorithm manipulates one or several lists of rectangles known or suspected to cover zeros of the problem function f. (Initially, there is only one list containing one rectangle, the starting rectangle supplied by the user.) Occasionally, a list is found to consist of a number of mutually disjoint sets of rectangles. If each such set can be enclosed in its own rectangular region, disjoint from other regions containing other sets, then the list is split into a number of independent sublists, each of which is processed in turn. Within each list, each rectangle is first removed from the list, bisected, and its two halves checked for zeros (the starting rectangle is first processed as a whole to determine the number of zeros it contains before it is bisected). Those halves that are found not to contain any zeros are discarded while the others are appended to the end of the list. The number of zeros contained in a rectangle is found by obtaining the *winding amount* (see [7]) of f for each side of that rectangle. A rectangle side s typically requires several recursive bisections to resolve the location of $f(s)$ relative to the origin in the image plane, which then translates into the winding amount.

The first phase of the sequential algorithm is by far the most time consuming but offers good opportunities for parallel computations. We use a form of domain decomposition where the input to each job consists of a list of rectangles, a single rectangle, a side of a rectangle or just one half of a side. None of these inputs are known a priori but rather are determined dynamically, except for the four sides of the starting rectangle initially provided by the user. This is to be expected since the computations will tend to concentrate near the zeros, and since there is no advance knowledge of their distribution in the rectangle supplied.

Currently, our hardware platform is made up of eight SPARC ELC stations coupled loosely by a network bus. The software basis consists of the Range Arithmetic package [1] coupled with the Distributed Thread System (DTS) [2]. The latter makes possible the distributed computation of individual functions in a C or C++ program. Using this system, one generates only a single executable program, an image of which resides on each machine that participates in the computations. A more complete introduction to DTS appears in Section 4. Section 2 provides a detailed description of the parallel version of the algorithm and Section 3 gives numerical results for an example problem.

2. Four levels of parallelism

This section describes the modifications to the sequential algorithm that were made in the parallel version. Perhaps the most obvious opportunity for distributing the computation occurs when a list of rectangles is split into a number of independent sublists. A thread of execution causing such a split retains one of the new lists but sends the others off for remote processing. It then continues execution, processing the list it retained for itself. An exception to this scheme occurs when a new list's rectangles are known to cover only a single (first order) zero. Typically, this kind of zero is found and verified rapidly using Newton's method, in which case it does not pay to send off the list for remote processing. Instead, the local thread attempts to find the zero itself using Newton's method. Only when this fails will the list be shipped out to another machine. The importance of this level of parallelization clearly depends on the number and geographical distribution of zeros in the starting rectangle.

Next we consider the simultaneous processing of rectangles in a single list. The two halves of a newly bisected rectangle should not be processed in parallel since they share a new side that has not previously been processed and must be processed now. The rectangles on a list currently awaiting their turn should, however, be processed in parallel. If more than one rectangle is available, all but one are broadcast to other machines for remote processing. This level of parallelism usually kicks in shortly before a list is split into several sublists or before the precision of computation is increased. Recall that a low precision setting can result in rectangles for which the number of zeros covered is unknown and which cannot be discarded.

The third level of parallelism occurs when several sides of the same rectangle are processed simultaneously. For reasons given below, however, most rectangles require processing only one of their sides, and this level is not as important as may first appear. A notable exception is of course the starting rectangle. When this rectangle is processed, neither of the earlier two levels of parallelism are active, and processing all four sides in parallel is clearly beneficial.

The final level of parallelism implemented consists of the simultaneous processing of the two halves of a subinterval, in case that subinterval requires bisection. The data in [7] shows that the average depth of bisection varies considerably for different problems, but that the potential degree of parallelism at this level can often be expected to exceed eight. Unfortunately, as the depth of bisection increases the individual jobs quickly become too light to warrant export to other machines. Therefore, in our program this level of parallelism is only employed for sides of rectangles and not for any subintervals of these sides. Nevertheless, on a more tightly coupled multicomputer this level could well be the most rewarding as the number of processors available increases beyond just a few.

The sequential version of the algorithm uses binary trees for the purpose of retaining winding information on each processed subinterval of a rectangle's side. This avoids unnecessary recomputation in case this information is ever needed again. Each side of a processed rectangle points to such a tree, whose structure reflects the bisections that were carried out before. Rectangles that share sides or even just parts of sides share a tree or subtree. The implementation of this idea made the program more complicated and less suitable to parallelization, but it did result in a significant reduction of execution time, sometimes by more than a factor of three.

In the parallel version, it is not assumed that two threads of execution have any shared memory available, and binary trees are no longer shared between rectangles in this version. Moreover, the trees used have a fixed maximum depth. This simplifies the programming of

functions responsible for broadcasting and receiving such trees to and from other machines. It also eliminates the risk of incurring too much communication overhead from broadcasting deep trees. The trees used are two levels deep, which can be motivated by the following argument.

Consider the four sides of a new rectangle R_c which has been obtained through bisection of a parent rectangle R_p. The aim is to find the number of zeros contained in R_c, but how many of R_c's sides need to be processed? We assume here that the sibling of R_c, the other subrectangle emerging from the bisection of R_p, has not already been processed. If absolutely no winding information about R_p's sides is forwarded from R_p to R_c, then it is clear that all four sides of R_c need to be processed, which constitutes the worst case. In contrast, the best possible scenario requires processing just one side of R_c, with enough information inherited from R_p to deal with the other three sides. It is interesting to derive the expected number of sides of R_c to be processed as a function of the maximum depth of inherited binary trees. Naturally, this number will lie between one and four, and will decrease with increasing depth, because more information can then be inherited by R_c from R_p.

We will not carry out this derivation here but only quote the results for binary trees of depths 0, 1, and 2. For simplicity we assume that the computation of the winding amount associated with a given side always succeeds, using as many recursive bisections of that side as necessary. First suppose that the binary trees contain only root nodes, which means that R_c inherits from R_p only information about R_p's sides, but no information about any subintervals of these sides. In this case it is clear that the expected number of sides of R_c to be be processed is three, one less than the worst case but two more than the best case. Next assume the binary trees are one level deep: R_c inherits from R_p winding information about its sides and possibly information about the two halves of each side. It can be shown that the expected number of sides to be processed now equals $\frac{43}{24} \approx 1.79$. The derivation assumes that whenever a side is processed, it has to be bisected at least once, which means that winding information will then be available not only for the side itself but also for its two halves. From the data in [7], it is clear that this is a fair assumption in practice.

Finally, assume the binary trees are two levels deep. The expected number of sides to be processed turns out to be $\frac{843}{720} \approx 1.17$, which is optimistically close to the best case. Here the derivation assumes that whenever a side is processed, it has to be recursively bisected at least twice, which is still a fair assumption to make, at least for the test cases in [7]. For the example of Section 3, the observed average number of sides processed is 1.43. Contrary to our earlier hypothesis, the computation of winding amounts is not always successful, either because the precision is too low or because a zero lies on the rectangle's boundary. Therefore, child rectangles do not always inherit as much information about their parents' sides as we assumed in the derivation of the expected value.

3. Numerical experience

A loosely coupled workstation cluster implies significant communication overhead when used as a parallel computer. Moreover, network traffic can vary considerably during the course of a day, and this directly affects the overhead. The data below is based on measurements obtained on three separate days and subsequently averaged. These measurements are actual observed time intervals that elapsed during program execution and lie in the range of several minutes. Although the machines used in the computations are part of a much larger departmental

network and subject to remote logins, at least none of them were used by other users at the start of program execution.

The problem solved is to find all zeros of

$$f(z) = \sin\left(\frac{z^2 + \pi^2}{z + \pi(2i - 3)}\right)$$

in the rectangle defined by the corner points $-10 - 5i$ and $10 + 10i$. The desired number of guaranteed decimal places is 20. This problem was already solved in [7] where it was seen that 27 zeros exist in the specified rectangle. About half of these zeros are clustered near the lower right corner point. Figure 1 shows the speedup obtained as a function of the number of machines used. Clearly, it is far less than ideal but certainly not neglible. We deliberately chose a problem that was originally conceived without parallel execution in mind, to avoid the temptation of "discovering" a problem ideally suited to our environment.

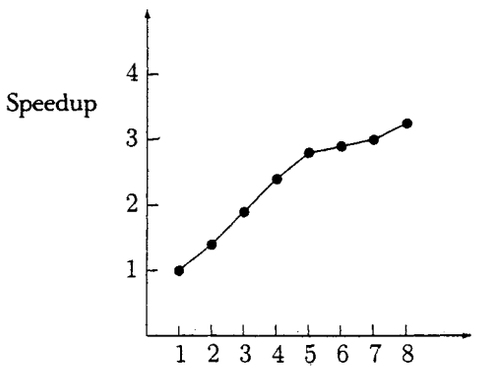

Figure 1.

The version run on a single workstation, which constitutes the comparison base for the other cases, is identical to that run on multiple machines. By removing all forks, decoupling the program from DTS, and storing the precision of computation in a global variable as opposed to the thread's system control block (cf. Section 4), the program can be made to run as fast on one machine as on two without these changes. The original sequential version (with its full binary trees) runs as fast on one machine as the parallel version using all eight workstations. Nevertheless, we believe the parallel algorithm could be sped up significantly beyond what is shown here for reasons explained below.

It is interesting to consider the number of forks that take place at each of the four levels of parallelism. (Here the term *fork* refers to the creation of a new independent thread of execution by an existing thread, which basically means that some particular function call can be off-loaded to another processor.) For the problem above, there are six forks at the first level, 154 forks at the second level, 88 forks at the third level, and 255 forks at the fourth level. Slightly more than half of all forks occur at the fourth level. When the number of processors is small (say two or three), the overhead incurred by this level appears to outbalance the reduction in execution time gained through added parallelism: preventing these forks then reduces the average execution time by about 10%. With eight machines available, however, the

average execution time increases by about 20% when fourth level forks are absent. It is hoped that in the future, the algorithm can be tested on a more tightly coupled multicomputer. The fourth level forks could then be extended deeper into the tree of recursive bisections. In the example problem above, the average depth of bisection for rectangle sides equals 6.3, and there is the potential for keeping dozens of processors occupied.

4. Software basis

Relatively little effort was required in adapting the Range Arithmetic package to the Distributed Thread System (DTS). The most interesting aspect of this conversion concerns the precision setting associated with each thread. As more than one thread may be concurrently active on the same machine, this information cannot be stored in the global address space shared by the threads. It cannot be passed via an additional parameter argument to the functions that most frequently access the precision setting, because these functions are overloaded C++ operators which use a predefined number of arguments. We store a thread's current precision setting directly inside its own system control block.

We conclude this Section with an overview of DTS, a system which allows a user to distribute a program over a network of loosely coupled workstations. As described above, this can lead to significant improvements in program execution time. DTS uses Parallel Virtual Machine (PVM) [8] as the underlying message passing system and CThreads [3] for parallel execution on a single node. It basically offers the usual thread functions *fork* and *join*, here extended to distributed computing in a SPMD (single program over multiple data streams) [4] programming environment.

In contrast to the client-server model, there is only one executable program, containing all required functions. The programer does not write separate programs for client and server, as for instance in RPC (Remote Procedure Call) [4] applications. The semantics and syntax of DTS correspond to what is typical of many thread packages, such as CThreads or Posix Threads.

Unlike the case of PVM applications, a user of DTS does not manually start his program on each host but instead starts execution on just one of the participating machines. The program then calls a DTS function which in turn loads and executes the same program image on all other machines. The initiating machine continues execution following this call while the others wait for job assignments. When a machine forks to concurrently execute some function in the program, DTS chooses the machine on which the execution takes place. There is no need to specify the executing machine by hand or in advance. The choice is based on the current individual loads of the machines, and an attempt is made to balance the overall load as much as possible. After execution of a forked job, the results are sent back to the caller's machine and saved in a special buffer until a subsequent call to the corresponding *join* occurs. All sending and receiving of input parameters and results is done in a non-blocking buffered way, in order to establish as much parallelism as possible. This means that a forking thread can continue execution without concern about whether outgoing messages have already been delivered. Furthermore, DTS automatically recovers whenever a machine goes down unexpectedly, because the system remembers which node executed which jobs and reassigns crashed jobs to other machines. This can be a very important reliability enhancement in a distributed environment that consists of many independent computers.

Under DTS, all participating nodes are allowed to use *fork* and *join*. It is not unusual even for the initiating host to execute jobs forked from other machines. The entire network, which may consist of many heterogeneous machines, can be made to join in the computations. In the experiments above we have avoided using heterogeneous computers only for the sake of obtaining meaningful speedup data. Apart from the present program, DTS has been successfully used to parallelize a number of different applications. Among these are a RSA crypto system in PARSAC–2 [5], a linear equation solver, and others resulting in system efficiencies topping 80% (system efficiency is defined as speedup over number of processors).

To summarize, DTS features (1) automatic load-balancing, (2) parallel execution on many machines, (3) single sourcecode, and (4) dynamic recovery and reconfiguration of the working pool.

References

[1] Aberth, O. and Schaefer, M. J. *Precise computation using range arithmetic, via C++*. ACM Transactions on Mathematical Software (December 1992).

[2] Bubeck, T. *Eine Systemumgebung zum verteilten funktionalen Rechnen*. Internal Report WSI–93–8, Wilhelm-Schickard-Institut der Universität Tübingen, 1993.

[3] Cooper, E. C. and Draves, R. P. *C Threads*. Technical Report, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, July 1987.

[4] Hwang, K. *Advanced computer architecture*. McGraw-Hill, 1993.

[5] Küchlin, W. W. *PARSAC–2: a parallel SAC–2 based on threads*. In: Sakata, S. (ed.) "Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes: 8th International Conference, AAECC–8", Lecture Notes in Computer Science **508** (1990), Springer-Verlag, Tokyo, pp. 206–217.

[6] Neumaier, A. *Interval methods for systems of equations*. Cambridge University Press, 1990.

[7] Schaefer, M. J. *Precise zeros of analytic functions using interval arithmetic*. Interval Computations 4 (1993), pp. 22–39.

[8] Sunderam, V. S. *PVM: a framework for parallel distributed computing*. Concurrency: Practice & Experience (December 1990), pp. 315–339.

Received: February 28, 1994
Revised version: November 20, 1994

Universität Tübingen
Wilhelm-Schickard-Institut
Sand 13
72076 Tübingen
Germany