Reliable Computing 1 (3) (1995), pp. 265–274

Optimizing INTBIS on the CRAY Y-MP

Chenyi Hu, Joe Sheldon*, R. Baker Kearfott, and Qing Yang

INTBIS is a well-tested software package which uses an interval Newton/generalized bisection method to find all numerical solutions to nonlinear systems of equations. Since INTBIS uses interval computations, its results are guaranteed to contain all solutions. To efficiently solve very large nonlinear systems on a parallel vector computer, it is necessary to effectively utilize the architectural features of the machine In this paper, we report our implementations of INTBIS for large nonlinear systems on the Cray Y-MP supercomputer. We first present the direct implementation of INTBIS on a Cray. Then, we report our work on optimizing INTBIS on the Cray Y-MP

Оптимизация INTBIS для CRAY Y-MP

Ч. Ху, Дж. Шелдон, Р. Б. Кирфотт, К. Янг

INTBIS — прошедший тщательное тестирование пакет программного обеспечения, использующий интервальный метод Ньютона и обобщенный метод половинного деления для нахождения всех численных решений систем нелинейных уравнений. Благодаря тому что INTBIS использует интервальные вычисления, получаемые им результаты гарантированно содержат все решения. Для эффективного решения очень больших нелинейных систем на параллельном векторном компьютере необходимо максимально использовать особенности архитектуры машины. В настоящей работе описаны реализации INTBIS для больших нелинейных систем на суперкомпьютере Cray Y-MP Сначала представлена прямая реализация INTBIS для компьютеров Cray, а затем излагаются результаты работы по оптимизации INTBIS для Cray Y-MP

1. Introduction

INTBIS [10] is a fully documented well-tested portable software package written in standard FORTRAN 77. The software package includes 46 subroutines and functions with a total length of over 10,000 lines. The software finds *all* numerical solutions to nonlinear systems of equations

$$F(X) = \Big(f_0(x_0, x_1, \ldots, x_{n-1}), f_1(x_0, x_1, \ldots, x_{n-1}), \ldots, f_{n-1}(x_0, x_1, \ldots, x_{n-1})\Big) = 0 \qquad (1)$$

in a given box[1] $\mathbf{X} = (\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$, where $x_i \in \mathbf{x}_i$ and $\mathbf{x}_i = \{x_i | \underline{x}_i \leq x_i \leq \overline{x}_i\}$. The result of applying INTBIS is a collection of small boxes that are guaranteed to contain solutions of the equation. This package takes into consideration the finite precision of the computer arithmetic operations.

In this paper, we report our work on vectorization and optimization of INTBIS on the Cray Y-MP/864 at San Diego Supercomputer Center (SDSC). Efficient implementations of the

This research was partially supported by NSF Grants No. DMS-9205680 and No. MIP-9208041

*This co-author is a NASA supported undergraduate research assistant at the University of Houston-Downtown.

[1]Throughout the paper we will use boldface letters and capital letters to denote interval quantities and vectors, respectively We use $\underline{x}$ and $\overline{x}$ to denote the lower bound and the upper bound for an interval variable $\mathbf{x}$, respectively

software package require a clear understanding of the system environment of the machine as well as the algorithm. The underlying algorithm in INTBIS is based on the interval Newton/generalized bisection method. Detailed information about the algorithm can be found in [2, 6, 7, 9]. Readers may also refer to an outline of the algorithm in [4] which appears in this issue.

2. System environment

The Cray Y−MP at SDSC is a shared memory MIMD (multiple instruction multiple data) supercomputer. It has 8 CPUs, each with a 6-nanosecond clock period, providing a theoretical peak speed of 2.7 Gflops (billions of floating-point operations per second) across all processors.

Each CPU of the Cray is a vector processor which contains a number of functional units. Each of these functional units is designed for a special purpose and operates independently of the others. Therefore, operations can occur in parallel on different functional units. The functional units of the Cray are capable of *chaining*, i.e., the output from one operation can become the input to another functional unit. Furthermore, each of these functional units is fully *pipelined*, i.e., provides a way to start a new task on the functional unit before an old one has been completed, so that the intermediate steps required to complete an operation can be broken down into time slots of one-clock-period duration.

Here is an example to explain how the pipeline principle works. Suppose it requires 6 clock cycles (steps) to add two floating point numbers in a computer. Then, adding two floating point N-vectors to form a new N-vector (i.e. $C(i) = A(i) + B(i)$, for $i = 1, 2, \ldots, N$) on a conventional (non-pipelined) computer requires $6N$ clock periods. A pipelined floating point addition unit consists of 6 elements, and each element performs only one of the 6 steps required, then provides its intermediate result to the next element, all in one clock cycle. The 6 elements of the addition unit can work concurrently. The unit needs 6 clock cycles (called the initiation time) to fill the pipeline (produce the first result), then it produces a result in every clock cycle. The total time required by the pipelined addition functional unit to add two vectors of length N will be $6 + (N - 1)$. In general, if a task requires c clock periods to complete; then it will require $c + N - 1$ clock periods on a fully pipelined functional unit to complete N similar tasks, compared to cN clock periods on a conventional (non-pipelined) computer. The time saved would be $cN - c - N + 1 = (c - 1)(N - 1)$. Since $c \geq 1$ is fixed for a given pipelined functional unit, the larger$N$, the more time will be saved. Therefore, the major optimization technique used in this paper is to increase the length of vectors to be accessed by the fully pipelined Cray functional units.

In the above discussion, we assumed that there is no memory accessing delay for the fully pipelined functional unit to work. However, it is not always true in practice. In fact, memory accessing latency (i.e., time necessary to fetch data from the memory) is a major bottleneck in scientific computations nowadays. To reduce memory accessing latency, Cray's memory is partitioned into 4 sections; each of these sections is partitioned into 8 subsections, and each subsection, in its turn, is divided into 8 separate subsubsections called *banks*. This construction gives the Cray 256 memory banks. This memory system structure is called *interleaved memory*. Each Cray Y−MP CPU has its own path to each section of the memory. Each CPU has four ports: two for reading, one for writing, and one for instruction buffers or I/O request. For a CPU to access memory, it must have both an available port and an available path to that section of memory. The bank cycle time for a Cray Y−MP is 5 clock periods. This means that

each memory reference by a CPU makes the referenced memory bank unavailable to all ports in all other CPUs for 5 clock periods Furthermore, each memory reference makes an entire memory subsection unavailable to all ports of the same CPU for 5 clock periods. The time that a CPU-memory reference must wait to re-access a memory bank is 5 clock periods. Hence, attempts to access the same bank or subsection each tick of the clock (called *bank conflict*) results in an effective memory speed of 1/5 of the possible speed. In our optimization, we also tested memory accessing to see if bank conflicts happened in our numerical experiments.

The Cray Y–MP at SDSC runs UNICOS, a UNIX operating system from Cray Research, Inc Most portable programs written in standard FORTRAN 77 can be compiled by the Cray *cf77* compiler and run on the Cray Y–MP without any changes. The Cray *cf77* compiler can also analyze the source code and automatically perform some optimizations based on the program. If there are no data dependencies, the *cf77* compiler can vectorize the innermost loop of a nested loop to be run on a vector processor, and can parallelize the outermost loop of a nested loop to be run on different processors. Our work reported in this paper only involves the optimization of INTBIS on a single vector processor of the Cray Y–MP. In the following two sections, we first report the result obtained by compiler optimization, and then report additional optimizations requiring changes in the source code.

3. Direct implementation

Although written in standard FORTRAN 77, INTBIS simulates interval operations, and so, special consideration must be given to the Cray's special floating point arithmetic and architecture.

All current digital computers store real numbers in a floating point format and perform only floating point operations. To perform interval operations with both mathematical and numerical rigor, INTBIS implements the interval arithmetic in software. A subroutine RNDOUT is employed to simulate directed roundings in a reasonably transportable way. It is called for each elementary operation involving intervals. The endpoints of the result interval are first computed with the machine's usual floating point arithmetic. Then, the routine RNDOUT may decrease the left endpoint of that approximate interval result by the absolute value of that endpoint times a rigorous estimate for the maximum relative error in an elementary operation. The routine RNDOUT may also similarly increase the right endpoint of that approximate interval.

For RNDOUT to work properly, a machine-dependent parameter must be set by calling the routine SIMINI. The routine SIMINI obtains certain machine parameters by calling the SLATEC routines D1MACH and I1MACH. In particular, SIMINI sets the amount by which the left endpoint is to be decreased the right endpoint is to be increased after the end points are obtained with the usual floating point arithmetic, to guarantee that the resulting interval will contain the result which would have been obtained with true directed roundings. If the machine has so-called *guard digits* (meaning that all arithmetic operations are correct up to the last binary digit), then the results will have a maximum error of one ULP (unit in the last place). However, Cray machines do not have guard digits, and on such machines, the use of D1MACH is not rigorous (i.e., does not lead to an interval guaranteed to contain the desired result) [11]. In our implementation, to get guaranteed estimates, we used a larger value of the largest relative distance between floating point numbers than that given in D1MACH. This can be done by properly setting the value of MAXERR, which is the maximum number of ULP's.

In our implementation, we set MAXERR = 20 which was recommended by H. Schwandt and P. Tang [8]. In effect, the technique uses part of the computer word as guard digits, to avoid the type of subtraction error illustrated in [11].

Due to the Cray memory architecture, the Cray performs optimally with single precision. Using double precision will severely degrade the overall performance. In addition, the Cray Y−MP is 64-bit machine. Single precision on the Cray is 64 bits so the actual precision is equivalent to double precision on most other machines. Therefore, we converted all double precision variables in INTBIS into single precision.

In runing INTBIS on the Cray, we found for some sets of test data the program crashed The problem was traced to the interval division routine XDIV, in which a mathematical operation was being performed on a variable that had not been initialized. After making this very minor change, the INTBIS was properly implemented on the Cray Y−MP and ran well. In this direct implementation, we compiled INTBIS with machine optimization and vectorization.

We chose Broyden's problem [7] to analyze the behavior of the direct implementation of INTBIS on the Cray. Broyden's problem is defined by:

$$f_i = x_i(2 + 5x_i^2) + 1 - \sum_{j \in J_i} x_j(1 + x_j)$$

where $J_i = \{j : j \neq i, \max(1, i-5) \leq j \leq \min(n, i+1)\}$; the initial box is $[-1, 1]^n$. We chose Broyden's problem because n (the dimension of the system of equations) can be changed easily. By using the Cray *flowtrace* utility, we generated a report detailing the time used by each subroutine for Broyden's problem with $n = 16$.

Here is the report generated by the Cray flowtrace for the direct implementation; we only list the top ten most expensive routines.

Routine Name	Total Time	Number of Calls	Percentage
SCLMLT	2.38	592294	21.14
RNDOUT	1.88	761646	16.69
MULT	1.42	206098	12.56
POLFUN	1.16	370	10.25
ADD	1.12	256915	9.92
POLJAC	0.77	180	6.80
INTGS	0.53	112	4.68
NBLPCW	0.50	1347	4.44
CASPIV	0.47	2697	4.15
POWER	0.32	88320	2.84
...	...	...	...
Totals	11.3	2044457	100

We have also run a set of other test problems. The program behavior information provided by the Cray flowchart utility is comparable with the above.

Analyzing the behavior of the software package, we found that the five subroutines, SCLMLT, RNDOUT, MULT, POLFUN, and ADD cost over 70% of the machine time. The number of calls to SCLMLT, RNDOUT, MULT and ADD are about 90% of total subroutine calls. To improve the overall efficiency of INTBIS, we definitely should take full advantage of Cray architecture to optimize these subroutines, which consume most of the computation time.

4. Optimization

The original flowtrace report showed that SCLMLT, which multiplies an interval by a point value, used the most computation time, followed by RNDOUT. We worked on the SCLMLT routine first. The structure of the routine SCLMLT is

```
SUBROUTINE SCLMLT(A, B, RESULT)
  REAL A, B(2), RESULT(2)
  ...
END
```

In the subroutine SCLMLT, variable A is a scalar and B is an array that stores the upper and lower bounds of an interval. The resulting interval is put into the two-element array RESULT. The fully pipelined Cray multiplication function unit will not speed up SCLMLT significantly; because of the pipeline initialization time, it is not advantageous to process arrays with two elements. To take full advantage of fully pipelined Cray functional units, we searched for calls to SCLMLT in INTBIS, and found that 11% of the total calls of SCLMLT was in a loop inside the routine INTGS (interval Gauss-Seidel linear solver).

```
DO 10 I = 1, N
  A = ...
  B = ...
  CALL SCLMLT(A, B, RESULT)
  ... = A
  ... = B
10  CONTINUE
```

Such loops are really interval vector operations analogous to level-1 BLAS (Basic Linear Algebra Software) routines. The Cray compiler cannot vectorize loops that contain function calls. To achieve higher speedup, we pushed the loops inside SCLMLT to form a new routine VSCLMLT. To push the loop into SCLMLT, the parameters to SCLMLT are put into temporary arrays, the loop is placed in VSCLMLT, and the multiple calls to SCLMLT are replaced by a single call to VSCLMLT.

The new code has the following structure:

```
DO 10 I = 1, N
  TEMP1(I) = ...
  TEMP2(1,I) = ...
  TEMP2(2,I) = ...
10  CONTINUE
CALL VSCLMLT(N, TEMP1, TEMP2, TEMP3)
```

Routine VSCLMLT:

```
SUBROUTINE VSCLMLT(N, A, B, RESULT)
INTEGER N
REAL A(N), B(2,N), RESULT(2,N)
```

```
DO 10 I = 1, N
  RESULT(1,N) = ...
  RESULT(2,N) = ...
10  CONTINUE
END
```

In INTBIS, the routine RNDOUT is called by each single interval (a 2-element array) operation to guarantee rigor. In addition to the vectorization of SCLMLT in INTGS, RNDOUT was also vectorized with the same method; we used VRNDOUT (Vector RNDOUT) in VSCLMLT instead of RNDOUT. We obtained the following statistics after vectorization of SCLMLT and RNDOUT in INTGS:

Routine Name	Total Time	Number of Calls	Percentage
SCLMLT	2.22	527711	19 86
RNDOUT	2.01	762798	17.98
MULT	1.39	206082	12.40
POLFUN	1.17	370	10.49
ADD	1.09	256781	9.77
POLJAC	0.78	180	7.00
NBLPCW	0.50	1346	4.51
CASPIV	0.49	2697	4.34
INTGS	0.39	112	3.51
POWER	0.31	88320	2.81
...	...	...	. .
Totals	11.2	1989176	100

Comparing the flowcharts, we found that the number of function calls to SCLMLT went down from 2044457 to 1989176. This optimization saved 55,281 function calls to SCLMLT. This means that we only vectorized about 11% of the calls to subroutine SCLMLT in another routine INTGS. However, we can see an improvement: the total time (as reported by flowtrace) fell from 11.3 seconds to 11.2 seconds. Although the time saved is insignificant (about 1%), it indicates that our approach is in a right direction.

The same method was applied to the routine MULT in CASPIV and in INTGS, and to the routine ADD in INTGS; calls to RNDOUT in MULT and ADD were replaced with calls to VRNDOUT in VMULT and VADD. After each vectorization the final result was compared to the original to guarantee accuracy of the changes, and a flowtrace report was generated.

After vectorization of MULT and RNDOUT in CASPIV, we have the following measurements:

Routine Name	Total Time	Number of Calls	Percentage
SCLMLT	2.00	441137	18.66
RNDOUT	1.96	734572	18.28
POLFUN	1.16	370	10.79
ADD	1.13	256883	10.56
MULT	1.13	162792	10.54
POLJAC	0.77	180	7.19
NBLPCW	0.51	1346	4.72
INTGS	0.40	112	3.75
CASPIV	0.37	2697	3.48
POWER	0.32	88320	2.95
...	...	...	...
Totals	10.7	1845578	100

This optimization saved 143.598 function calls.

After vectorization of MULT and RNDOUT in INTGS, we have the results shown below:

Routine Name	Total Time	Number of Calls	Percentage
RNDOUT	1.87	694194	17.99
SCLMLT	1.87	400757	17.98
POLFUN	1.19	370	11.47
ADD	1.10	256883	10.54
MULT	0.99	142602	9.55
POLJAC	0.78	180	7.50
NBLPCW	0.51	1346	4.86
CASPIV	0.38	2697	3.68
INTGS	0.35	112	3.36
POWER	0.33	88320	3.15
...	...	...	...
Totals	10.4	1751360	100

This optimization saved 94,218 function calls.

After vectorization of ADD and RNDOUT in INTGS the measurements become:

Routine Name	Total Time	Number of Calls	Percentage
SCLMLT	1.86	400757	18.91
RNDOUT	1.70	629666	17.26
POLFUN	1.17	370	11.91
MULT	1.00	142602	10.14
ADD	0.85	192355	8.62
POLJAC	0.77	180	7.82
NBLPCW	0.51	1346	5.15
CASPIV	0.37	2697	3.78
POWER	0.33	88320	3.39
DAXPY	0.22	60993	2.20
...	...	...	...
Totals	9.83	1630370	100

This optimization saved 120,990 function calls.

The total number of function calls saved by these five optimizations is 414,087; approximately 20 percent of the original number of function calls was eliminated. The total time savings as reported by flowtrace was 1.47 seconds; a speedup of 1.15. By comparing the above table with the table in Section 3, we find that the speedup was obtained by only vectorizing 32% of SCLMLT calls, 17% of RNDOUT calls, 31% of MULT calls, and 25% of ADD calls. Together with POLFUN, these four subroutines cost about 67% of total computation time and about 84% of the total number of function calls. Much higher speedup is expected if these routines are optimized over the entire package INTBIS. At this time, we think that would be better to redesign a vector version of INTBIS rather than modify these routines over the entire package. Therefore, we stopped any further vectorizations.

We also tested the effects of different n for Broyden's problem, with the original version and our partially vectorized version of INTBIS. The following table compares the times (with compiler optimizations - cf77 -Zp -Wf"-o aggress"). The array size (MN2) was set to 64. Original program time is the time for the non-vectorized version. Vectorized program time is the time for the vectorized version.

n	Original (seconds)	Vectorized (seconds)	Speedup
8	0.4677	0.4547	1.03
16	4.7684	4.4597	1.07
32	20.8699	18.7790	1.11
64	119.2850	105.4345	1.13

The above data tell us that the larger n, the greater the speedup. This is consistent with our analysis in Section 2. It also shows that, as the problem size increases, the cost of computation increases rapidly. It suggests that parallelization may be required in solving large-scale nonlinear systems of equations with INTBIS.

The following table was generated as above, but MN2 was set to 65 in both the original program and the vectorized version to see if possible memory bank conflicts were a problem.

Problem Size	Original (seconds)	Vectorized (seconds)	Speedup
8	0.4569	0.4538	1.01
16	4.6886	4.4196	1 06
32	20.6211	18.7981	1.10
64	117.6258	105.5723	1.11

There is no significant difference in runing time among the above two tables. Hence, we believe there is no serious memory bank conflict for our test problem.

While the time savings reported by flowtrace do not fully reflect advantages obtained by compiling for full vectorization, they do show the improvement that can be gained by reducing the overhead caused by many function calls. Along this line, an optimization technique called "inlining" moves the contents of a very small subprogram into the calling program to reduce the overhead of subroutine calls. The following table was generated as above, (MN2 = 65) and RNDOUT was inlined in the vectorized program.

Problem Size	Original (seconds)	Vectorized (seconds)	Speedup
8	0.4569	0.3684	1.24
16	4.6886	3.7130	1.26
32	20.6211	16.3853	1.26
64	117.6258	95.6795	1.23

The above table shows that inlining RNDOUT significantly improves the speed and at the same time, does not add much to the program size: The vectorized program size (without RNDOUT inlined) was 993552 bytes, and the size with RNDOUT vectorized was 994408 bytes.

5. Conclusions and future work

Our work of implementing INTBIS on the Cray shows that the interval software package INTBIS can be implemented and further optimized on the Cray Y–MP supercomputer to efficiently solve nonlinear systems of equations.

Generally, our observations and vectorization work follow a well-known pattern: replacement of subroutine calls and loops by higher-level linear algebra routines. This suggests that interval versions of the level 1, 2, and 3 BLAS would be appropriate, when restructuring codes such as INTBIS to run on high-performance computers. Indeed, we anticipate that, using such structure, with interval BLAS routines optimized to particular routines, we will obtain more impressive speedups than those reported here. We intend eventually to include portable versions of such BLAS routines with INTLIB [8]. Eventually, we can even provide language support for an interval vector data type in our Fortran 90 module system of [5]. Such vector support is also being written into Fortran–XSC [12], though details of how it will be implemented on particular machines are not yet known.

Since the original INTBIS was designed for conventional machines, there is much room for modification to achieve ultra performance on the Cray or other multi-vectorprocessor computers. To continue this research, we will create a new version of INTBIS which takes full advantage of the hardware architecture. First, we will redefine the data structure. Instead of defining N intervals as N 2-element arrays, we will define them as an $(N, 2)$ array. The new data structure can be processed most efficiently by the fully pipelined functional units of vector processors. Operations on a single vector processor will be vectorized as much as possible, and the length of vector operands will be taken as large as possible. In solving large-scale problems, sparse Jacobian matrices are often involved. The original INTBIS does not provide special considerations for general sparse systems. We will incorporate our recent result in [4] into the new version of INTBIS. We expect the revised INTBIS to be able to solve large-scale nonlinear systems of equations on multi-vectorprocessor supercomputers efficiently in parallel.

Acknowledgement

The authors wishes to acknowledge the referees for their careful reading and useful suggestions, and to the San Diego Supercomputer Center for providing us time allocation on the Cray Y–MP to carry out this research.

References

[1] Gan, Q., Yang, Q., and Hu, C. *Parallel all-row preconditioned interval linear solver for nonlinear equations on multiprocessor*. Parallel Computing **20** (1994), pp. 1249–1268

[2] Hansen, E. R. and Sengupta, S. *Bounding solutions of systems of equations using interval arithmetic*. BIT **21** (1981), pp. 203–211.

[3] Hu, C., Bayoumi, M., Kearfott, R. B., and Yang, Q. *A parallelized algorithm for all-row preconditioned interval Newton/generalized bisection method*. In: "Proc. SIAM 5th Conf. on Paral. Proc for Sci. Comp.", 1991, pp. 205–209.

[4] Hu, C., Frolov, A., Kearfott, R. B., and Yang, Q. *A general iterative sparse linear solver and its parallelization for interval Newton's methods*. This issue of J. Reliable Computing.

[5] Kearfott, R. B. *A Fortran 90 environment for research and prototyping of enclosure algorithms for nonlinear equations and global optimization*. ACM Trans. Math. Software, to appear.

[6] Kearfott, R. B. *Abstract generalized bisection and a cost bound*. Math. Comput. **49** (179) (1987), pp. 187–202.

[7] Kearfott, R. B. *Some tests of generalized bisection*. ACM Trans. Math. Software **13** (3) (1987), pp. 197–220.

[8] Kearfott, R. B., Dawande, M., Du, K., and Hu, C. *INTLIB: A portable interval FORTRAN–77 standard function library*. ACM Trans. Math. Software, to appear.

[9] Kearfott, R. B., Hu, C., and Novoa, M. *A review of preconditioners for the interval Gauss-Seidel method*. Interval Computations 1 (1991), pp. 59–85.

[10] Kearfott, R. B. and Novoa, M. *A program for generalized bisection*. ACM Trans. Math. Software **16** (2) (1990), pp. 152–157.

[11] Kulisch, U. and Miranker, W. L. *A new approach to scientific computation*. Academic Press, 1983.

[12] Walter, W. V. *Fortran–XSC: A Fortran-like language for verified scientific computing*. In: "Scientific Computing with Automatic Result Verification", Academic Press, New York, etc, 1993.

Received: February 28, 1994
Revised version: January 9, 1995

C. HU, J. SHELDON
Department of Computer and Mathematical Sciences
University of Houston-Downtown
Houston, Texas 77002, USA

R. B. KEARFOTT
Department of Mathematics
University of Southwestern Louisiana
U.S.L. Box 4–1010
Lafayette, LA 70504–1010, USA
E-mail: rbk@usl.edu

Q. YANG
Department of Electrical and Computer Engineering
University of Rhode Island
Kingston, Rhode Island 02881, USA