Intervals and OpenMP: Towards an Efficient Parallel Result-Verifying Nonlinear Solver*

Thomas Beelitz[1], Christian H. Bischof[2], and Bruno Lang[1]

[1] University of Wuppertal, Mathematics Department,
Gaußstr. 20, D-42097 Wuppertal, Germany
[2] Aachen University, Institute for Scientific Computing,
Seffenter Weg 23, D-52064 Aachen, Germany

Abstract. Nonlinear systems occur in diverse applications, i.e., in the steady state analysis of chemical processes. If safety concerns require the results to be provably correct then result-verifying algorithms relying on interval arithmetic should be used for solving these systems. Since such algorithms are very computationally intensive, parallelism must be exploited to make them feasible in practice. In this lab session we want to discuss the unexpected behaviour of our implemented nonlinear solver.

1 Desires and expectations for the OMP*lab* session

Currently we are porting an interval–based branch–and–bound nonlinear solver to C++. As the following sections provide a more detailed overview of the background and the key features of our application, we just want to summarize the most important issues.

The branch–and–bound algorithm lends itself naturally to a recursive implementation. All subboxes occuring at some fixed recursion depth can be handled independently from each other. Let L_i be a list containing the boxes at recursion depth i. In our first approach we distribute the work on L_i over all available processors. With shared memory and OpenMP this requires just parallelizing the main loop and one additional synchronization to prevent the same L_{i+1} entry being written to by different processors.

The provided file `Solver.tar.gz` contains all sources of our solver, a suitable test problem and an easy–to–use makefile mechanism. Please notice that the procedure `RunSolver` in the source file `NonlinearSolver.cpp` contains the parallelized main loop. In order to use interval arithmetic together with OpenMP constructs one has to compile these sources with version 5.5 of the Sun C++ compiler. This can easily be done by typing `gmake build`. If the solver should run on <`nthreads`> processors one has to type `gmake omp NTHREADS=`<`nthreads`>. The test problem is a slighty changed version of the non–public system in section 5.

* This work is supported by VolkswagenStiftung within the project "Konstruktive Methoden der Nichtlinearen Dynamik zum Entwurf verfahrenstechnischer Prozesse", Geschäftszeichen I/79 288.

The most important topic we want to discuss during the OMP*lab* session is the performance of our parallel code. Currently we cannot explain why the speedup of our algorithm seems to be limited. We hope that the participation in this lab session gives us a deeper insight into the behaviour of our parallel code. At the moment it seems to be impossible for us to decide where the parallel speedup is lost (in theory it should approach the number of processors). If our approach cannot provide sufficient speedup we want to discuss possible strategies for a hybrid parallelization scheme using OpenMP and MPI. Please notice that there will be only one person (probably Thomas Beelitz) representing our project in the lab session.

2 Introduction

Nonlinear systems arise in a variety of applications, one example being the search for singularities in a chemical process. Let $\mathbf{p} \in \mathbb{R}^k$ denote the adjustable *parameters* controlling the process (such as heating, inflow concentrations, etc.), and let $\mathbf{x} \in \mathbb{R}^\ell$ describe its internal *state* (current temperature, reaction rates, etc.). Then the steady states of the process can be described by a set of equations, $\mathbf{f}(\mathbf{p}, \mathbf{x}) = \mathbf{0}$, and singularities mark those parameter values $\mathbf{p}^*$ where transitions from unique steady states $\mathbf{x} = \mathbf{x}(\mathbf{p})$ to multiple steady states occur. As multiple steady states can lead to fluctuations in the quality of the resulting product or can even cause severe damage to the facility, being able to *guarantee* the absence of singularities in the parameter range $[\mathbf{p}] = [\underline{p_1}, \overline{p_1}] \times \cdots \times [\underline{p_k}, \overline{p_k}]$ intended for operating the process is an important goal during process analysis and design.

One approach to achieve this goal first augments the system $\mathbf{f}(\mathbf{p}, \mathbf{x}) = \mathbf{0}$ with equations characterizing a specific type of singularity [1], resulting in a larger system, $\mathbf{F}(\mathbf{z}) = \mathbf{0}$. Here, $\mathbf{z} \in \mathbb{R}^n$ comprises the variables $\mathbf{p}$ and $\mathbf{x}$, as well as auxiliary variables introduced during the augmentation, and $\mathbf{F}$ consists of the functions $\mathbf{f}$ and additional functions involving derivatives such as $\partial\mathbf{f}/\partial\mathbf{x}$; see [2] for more details.

In a second step, a result-verifying nonlinear solver is applied to the augmented system; see, e.g., [3]. These solvers are based on interval arithmetic [4] and are able to either *guarantee* that the system has no solution (i.e., there are no singularities of a specified type in the process), or to yield sharp bounds for the parameter combinations that might lead to singular behaviour and therefore must be avoided.

Even if the result-verifying algorithms have been improved substantially during the last years, the solution of nonlinear systems remains a computationally intensive task, in particular for realistic problems with several dozens of parameters. Therefore, exploiting parallelism is essential for the successful solution of such problems.

In the following section we briefly describe some key features of our framework for the verified solution of nonlinear systems. Then we discuss how the solver's inherent coarse-grained parallelism can be exploited with OpenMP. In Sect. 5 we report on numerical experiments.

3 The Result-Verifying Solver

Our framework for the solution of nonlinear systems consists of three modules. The central **solver** module implements an interval–based branch–and–bound nonlinear solver. The augmented system to be solved is set up in a symbolic **preprocessing** step, and the function and derivative values needed in the **solver** are provided in a third **evaluation** module.

The basic branch–and–bound algorithm works as follows. Given some "box" $[\mathbf{z}] = [\underline{z_1}, \overline{z_1}] \times \ldots \times [\underline{z_n}, \overline{z_n}] \subset \mathbb{IR}^n$, evaluating the components F_i of the function $\mathbf{F}$ with appropriate interval-based methods yields intervals $[F_i]$ that are *guaranteed* to enclose the ranges of the F_i over the box $[\mathbf{z}]$, even in the presence of rounding errors. Therefore, if $0 \notin [F_i]$ for some i, then the system $\mathbf{F}(\mathbf{z})$ cannot have a zero in $[\mathbf{z}]$, and the box can be excluded from further consideration. If $0 \in [F_i]$ for all i then the system may, but need not, have a zero in $[\mathbf{z}]$. In this case the box is split into two (or more) subboxes, and the above test is applied recursively to these.

Starting with an initial box corresponding to the intended ranges of the process parameters, this procedure results in a list of small boxes, which are guaranteed to contain all singularities that might be contained in the initial box. Several acceleration techniques are available [2, 3, 5, 6] that, added to the basic branch–and–bound algorithm, enhance its efficiency. These techniques rely on Taylor expansions of the functions F_i and therefore require the (interval) evaluation of derivatives such as $\partial F_i / \partial \mathbf{z}$. Some of these techniques are also able to *prove* that the resulting boxes indeed do contain singularities.

The branch–and–bound algorithm lends itself naturally to a recursive implementation. Considering the dynamic call-tree as a whole, however, reveals a large amount of coarse-grained parallelism: All subboxes occuring at some fixed recursion depth can be handled independently from each other.

In addition to the coarse-grained inter-box parallelism, the work done for each box can also be spread over multiple processors. Here, medium-to-fine-grained parallelism is obtained by distributing the evaluation of the n functions F_i and the n^2 partial derivatives $\partial F_i / \partial z_j$, as well as the solution of linear systems occuring in the acceleration techniques, etc. Finally, the evaluation of each function bears limited potential for a third level of very fine-grained parallelism.

4 Exploiting the Coarse-Grained Parallelelism with OpenMP

To exploit the inter-box parallelism, the dynamic call-tree is traversed in breadth-first order. Let L_i be a list containing the boxes at recursion depth i. Then, working on the boxes in L_i one-by-one (or in parallel), subboxes resulting from splitting a box are not handled immediately but placed in the list L_{i+1}, which is read only when the work on L_i is completed.

In a distributed memory environment with message passing [7, 8], the lists L_i may either be managed by a single processor using a standard master–slave

approach, or the processors may work on private sub-lists following some "mediated" scheme [9]. Since the amount of work spent for a box can vary significantly, simply assigning the same number of boxes to each processor is not sufficient to keep the workload roughly balanced. The best strategy depends on the number of processors and on the relative speeds of communication and computation.

With shared memory and OpenMP [10, 11], the situation is much simpler. Distributing the work on L_i over several processors requires just parallelizing the main loop and one additional synchronization to prevent the same L_{i+1} entry being written to by different processors. To account for the different amount of work associated with the boxes, we mimic the behaviour of the master–slave approach by using scheduling with chunksize 1, i.e., the boxes are distributed one-by-one, and as soon as a thread has finished its work on a box it is assigned the next box.

As the OpenMP parallelization requires fewer changes to the code than an MPI-based version and in addition can be done incrementally, we have decided to first use OpenMP for the inter-box parallelism. If this approach cannot provide sufficient speedup we may switch to a hybrid parallelization scheme later, using MPI for the inter-box parallelism and OpenMP for distributing the work associated with a single box.

5 Numerical Results

The numerical experiments were performed on a Sun Fire 6800 server with 24 processors (900MHz) and 24GB of main memory running Solaris 9. The programs were written in C++ and compiled with version 5.5 of the Sun C++ compiler. This compiler provides support for interval arithmetic and allows intervals to be used together with OpenMP constructs.

Table 1 gives the timings for several versions of the code and different numbers of processors. The data refer to solving a system with 29 equations and 29 unknowns. This system results from augmenting the model of a hydration reactor with additional equations characterizing a saddle–node singularity; cf. [2] for details.

First, we note that compiling the same *serial* code with the `-xopenmp` flag slows down its execution by roughly 20%, due to some aggressive optimizations being disabled by this flag. Parallelizing the code requires some restructuring and in particular introduces additional data movements and constructor calls. Taking both effects together, the parallel code—run on a single processor—is about 1.3 times slower than the optimized serial code.

This performance loss is compensated when the parallel code is run on $p > 1$ processors. The data indicate that the scheduling scheme (`static, 1`) is superior to the standard scheme, which partitions the list L_i into p equally-sized chunks and assigns one chunk to each processor.

Surprisingly, `( static, 1 )` scheduling (i.e., thread k is assigned the tasks ℓ where $\ell \equiv k \bmod p$) seems at least competitive to `dynamic` scheduling, which in theory should adapt better to the different complexity of the parallel tasks. This

Table 1. Timings (minutes:seconds) for solving a system with $n = 29$ unknowns.

Program version	#Procs	Time	Speedup
Serial	1	17:32	
Serial, compiled with `-xopenmp`	1	21:18	
Parallel	1	22:55	1.00
Parallel, `schedule( static, 1 )`	2	13:52	1.65
Parallel, `schedule( static, 1 )`	3	11:20	2.02
Parallel, `schedule( static, 1 )`	4	10:46	2.13
Parallel, standard scheduling	4	11:41	1.96
Parallel, `schedule( dynamic, 1 )`	4	12:00	1.91

indicates that the average complexity of each thread's tasks is approximately equal, so that the lower administrative overhead of `static` scheduling is dominating in the case $p = 4$. (For $p = 2$, `dynamic` scheduling is slightly better than the `static` scheme.)

We cannot yet explain why the speedup seems limited to about 2, even if more than four processors are used. We expect to obtain significantly better results with deeper insight into the behaviour of the parallel code.

References

1. Golubitsky, M., Schaeffer, D.G.: Singularities and Groups in Bifurcation Theory, Volume I. Springer-Verlag, New York (1985)
2. Bischof, C.H., Lang, B., Marquardt, W., Mönnigmann, M.: Verified determination of singularities in chemical processes. In Krämer, W., Wolff von Gudenberg, J., eds.: Scientific Computing, Validated Numerics, Interval Methods, New York, Kluwer Academic/Plenum Publishers (2001) 305–316
3. Kearfott, R.B.: Rigorous Global Search: Continuous Problems. Kluwer Academic Publishers, Dordrecht, The Netherlands (1996)
4. Alefeld, G., Herzberger, J.: Introduction to Interval Computations. Academic Press, New York (1983)
5. Neumaier, A.: Interval Methods for Systems of Equations. Cambridge University Press, Cambridge, UK (1990)
6. Schulte Althoff, K.: Algorithmen zum verifizierten Lösen nichtlinearer Gleichungssysteme. Diploma thesis, Aachen University, Germany (2002)
7. Message Passing Interface Forum: MPI: A message-passing interface standard (1994) `http://www.mpi-forum.org`.
8. Message Passing Interface Forum: MPI-2: Extensions to the message-passing interface (1997) `http://www.mpi-forum.org`.
9. Berner, S.: Parallel methods for verified global optimization: Practice and theory. J. Global Optim. **9** (1996) 1–22
10. OpenMP Architecture Review Board: OpenMP C and C++ application program interface, version 1.0 (1998) `http://www.openmp.org`.
11. OpenMP Architecture Review Board: OpenMP C and C++ application program interface, version 2.0 (2002) `http://www.openmp.org`.