Shared-Memory Parallelization of an Interval Equations Systems Solver – Comparison of Tools

Bartłomiej Jacek Kubica[1*]

[1] Warsaw University of Technology, Institute of Control and Computation Engineering, ul. Nowowiejska 15/19, 00–665 Warsaw, Poland, email: bkubica@elka.pw.edu.pl

Key words: interval computations, underdetermined systems of equations, parallelization, multi-threaded programming

Abstract This paper considers the shared-memory parallelization of an interval solver of underdetermined systems of nonlinear equations. Four threading libraries are investigated: OpenMP, POSIX threads, Boost threads and TBB. Directions for further investigations on multi-threaded interval algorithms are outlined.

1 Introduction

The majority of papers about parallel interval branch-and-bound/branch-and-prune methods concentrates on distributed-memory systems, mostly using MPI. As multi-core architectures become more and more popular, the interest in shared-memory implementations increase. Such implementations of interval methods are usually based on OpenMP (e.g. [1], [2], [7], [3]) or less frequently POSIX threads (e.g. [4]). The purpose of this paper is to analyze other parallelization tools to learn about their usability for interval computations. Obviously, the basic concern is efficiency, but other features (like the presence or not of some useful primitives) are considered, too.

2 Benchmark problem

As a benchmark an interval solver of underdetermined equations systems is considered. Solvers for other problems could also be used, but underdetermined problems are particularly intensive computationally, as there is a continuum –instead of a finite number – of solutions. The solver has been developed by the author and its parallelization using OpenMP did not give expected speedup (see [3]).

Interval methods are based on interval arithmetic operations and basic functions operating on intervals instead of real numbers (so that result of an operation on numbers belong to the result of operation on intervals, containing the arguments). We shall not define interval operations here; interested reader is referred to several papers and textbooks (see e.g. [1], [3], [6] and references therein).

*The research has been supported by the Polish Ministry of Science and Higher Education under grant N N514 416934. The computer on which experiments were performed is shared with the Institute of Computer Science of our University. Thanks to Jacek Błaszczyk for maintaining it.

The basic meta-algorithm of the solver was the branch-and-prune method that can be expressed as follows:

```
IBP (x^(0); f)
// x^(0) is the initial box, f(·) is the interval extension of the function f: R^n → R^m
// L_ver is the list of boxes verified to contain a segment of the solution manifold
// L_pos is the list of boxes that possibly contain a segment of the solution manifold
L = L_ver = L_pos = ∅ ;
x = x^(0) ;
loop
    process the box x, using the rejection/reduction tests ;
    if (x does not contain solutions) then discard x ;
    else if (x is verified to contain a segment of solution manifold) then push (L_ver, x) ;
    else if (tests subdivided x into x^(1) and x^(2)) then
        x = x^(1) ;
        push (L, x^(2)) ;
        cycle loop ;
    else if (x is small enough) then push (L_pos, x) ;
    if (x was discarded or stored) then
        x = pop (L) ;
        if (L was empty) then exit loop ;
    else
        bisect (x), obtaining x^(1) and x^(2);
        x = x^(1) ;
        push (L, x^(2)) ;
    end if ;
end loop
```

Different variants of the "rejection/reduction tests" were considered in [3]. They are not going to be described here. Now, we consider three variants of the algorithm in computational experiments: the Hansen variant, the Neumaier variant and the variant using componentwise Newton operator with Herbort and Ratz heuristic.

3 Investigated tools

Four tools were compared: OpenMP, POSIX threads, Boost threads and TBB.

3.1 OpenMP

OpenMP [10] is considered to be the standard for shared-memory parallel computations, nowadays. It is an extension of C, C++ and Fortran languages, that adds compiler directives to them.

The API is known to be very dense and – as compiler directives are usually ignored on compilers not using OpenMP – serial and parallel variants of the program might be identical or almost identical (which practically happens for extremely simple programs only, though).

OpenMP gives us the following tools:

- distributing a program into parallel sections,

- parallelizing simple `for` loops,
- basic synchronization primitives – locks, barriers and very simple atomic operations,
- functions to control the number of threads, etc.

It is worth noticing that atomic operations include incrementing and decrementing a variable, and arithmetic and bitwise operations. Unfortunately, many popular atomic primitives, like `compare-and-swap()`, do not belong to OpenMP standard.

3.2 POSIX threads

The Pthreads library [11] describes low-level general purpose threads. The module contains several functions to control the work of threads – many of them of low usability for parallel computations (e. g. manipulating the thread priority or thread cancellation). POSIX threads is a C library and it is unnatural to use it in object-oriented programs as some concepts are incompatible (see e.g. [5]); such cooperation is possible, though.

The library gives us the following tools:

- creating and joining a thread; several functions to manipulate it,
- synchronization primitives: mutexes (i. e. mutual exclusive locks), readers-writer locks, barriers and condition variables,
- functions to manipulate data placed in thread-local storage.

There is no POSIX API for atomic operations; we must apply other tools to use them.

3.3 Boost threads

Boost threads [9] are a portable threads variant for C++. They are simply an object-oriented envelope over POSIX threads. In particular we have scoped locks here (unlike POSIX mutexes they are compatible with the exception-throwing mechanism). Other than that the API is quite analogous to Pthreads. Again – no atomic operations.

3.4 TBB

The Intel Threading Building Blocks [12] is a tool for building multi-threaded programs working in multi-core environments and aiming applications in parallel computing. This library is very different from the three above.

The module is a C++ template library, designed as a very high-level API. We do not use threads directly here, but rather some tasks that are mapped onto physical threads somehow. Consequently, the library does not have some basic primitives like the barrier or a condition variable. Rather than that, it offers some parallel algorithms concepts, like `parallel_for`, `parallel_do`, `parallel_reduce`, etc.

Writing programs with this tool requires a different approach than with the other three techniques.

The library contains:

- parallel programming concepts,
- several variants of locks – ordinary mutexes, spin mutexes, queueing mutexes, etc.
- a very rich set of atomic operations,
- several other useful tools, like the memory allocators classes, containers for parallel data structures and a user-space task scheduler.

According to the documentation, as TBB are designed for parallel computations, they attempt to make an efficient use of the cache and other processor resources.

4 Implementation

The implementation for OpenMP, Pthreads and Boost threads is very similar and based on the scheme we encounter in [1] and [4]. As in the pseduocode (Section 2) we have a linked list of boxes to consider, a list of verified solution boxes and possible solution boxes. Access to each of the lists is guarded by a mutex.

In all three variants we use a global variable `num_working_threads`, showing how many threads are not idle. When this variable becomes zero, the computations should be finished.

A difference is that for POSIX and Boost threads we use a condition variable instead; OpenMP does not have this primitive and an active waiting loop must be used here.

The implementation is quite different for TBB. The author used the concept of `parallel_do` (i. e. a few tasks execute some sort of a `do...while` loop), though some other concepts could be utilized too, probably. The initial box is passed to one of the tasks and results of its bisection are distributed between tasks using the so-called "feeder". So, we do not have a list of boxes to consider there, i. e. we do not keep it explicitly.

The Reference Manual (on web page [12]) is not clear about how the work items are stored or in what way acquiring them is synchronized. We can only learn that adding additional work items by the feeder results in better scalability than in the situation when "all of the items come from the input stream" (which would be impossible in our program, anyway).

As for other implementations we have two lists of solutions with dedicated locks – spin mutexes are used there, as the operation of inserting a box is not time consuming.

5 Test problems

Three problems are considered in numerical experiments – all of them used in [6] and [3].

The first one is called the hippopede problem:

$$\begin{aligned} & x_1^2 + x_2^2 - x_3 = 0 \ , \\ & x_2^2 + x_3^2 - 1.1 x_3 = 0 \ . \\ & x_1 \in [-1.5, 1.5], \ x_2 \in [-1, 1], \ x_3 \in [0, 4] \ . \end{aligned} \tag{1}$$

Accuracy $\varepsilon = 10^{-3}$ was set.

The following problem, called Puma, arose in the inverse kinematics of a 3R (three revolute joint) robot and is one of typical benchmarks for nonlinear system solvers:

$$\begin{aligned} & x_1^2 + x_2^2 - 1 = 0 \ , \ x_3^2 + x_4^2 - 1 = 0 \ , \\ & x_5^2 + x_6^2 - 1 = 0 \ , \ x_7^2 + x_8^2 - 1 = 0 \ , \\ & 0.004731 x_1 x_3 - 0.3578 x_2 x_3 - 0.1238 x_1 - 0.001637 x_2 - 0.9338 x_4 + x_7 = 0 \ , \\ & 0.2238 x_1 x_3 + 0.7623 x_2 x_3 + 0.2638 x_1 - 0.07745 x_2 - 0.6734 x_4 - 0.6022 = 0 \ , \\ & x_6 x_8 + 0.3578 x_1 + 0.004731 x_2 = 0 \ , \\ & -0.7623 x_1 + 0.2238 x_2 + 0.3461 = 0 \ , \\ & x_1, \ldots, x_8 \in [-1, 1] \ . \end{aligned} \tag{2}$$

In the above form it is a well-determined (8 equations and 8 variables) problem with 16 solutions that are easily found by several solvers. To make it underdetermined the

last equation was dropped (as in [3]). The variant with 7 equations was considered in numerical experiments as second test problem. Accuracy $\varepsilon = 10^{-4}$ was set.

The last system arose in aircraft equilibrium problems:

$$\begin{aligned}
&-3.933x_1 + 0.107x_2 + 0.126x_3 - 9.99x_5 - 45.83x_7 - 7.64x_8 + \\
&-0.727x_2x_3 + 8.39x_3x_4 - 684.4x_4x_5 + 63.5x_4x_7 = 0\ , \\
&-0.987x_2 - 22.95x_4 - 28.37x_6 + 0.949x_1x_3 + 0.173x_1x_5 = 0\ , \\
&0.002x_1 - 0.235x_3 + 5.67x_5 + 0.921x_7 - 6.51x_8 - 0.716x_1x_2 + \\
&-1.578x_1x_4 + 1.132x_4x_7 = 0\ , \\
&x_1 - x_4 - 0.168x_6 - x_1x_2 = 0\ , \\
&-x_3 - 0.196x_5 - 0.0071x_7 + x_1x_4 = 0\ .
\end{aligned} \tag{3}$$

This problem has 5 equations in 8 variables. As in [6] no bounds were given, we take – as we did in [3] – $x_i \in [-2, 2]$, $i = 1, \ldots, 8$. Accuracy $\varepsilon = 10^{-1}$ was set.

6 Numerical experiments

Numerical experiments were performed on a computer with 16 cores, i. e. 8 Dual-Core AMD Opterons 8218 with 2.6GHz clock. The machine ran under control of a Fedora 10 Linux operating system. The solver was implemented in C++, using C-XSC 2.2.3 library [8] for interval computations. The GCC 4.3.2 compiler was used.

Results are given in Tables 1–3. The larger font marks the shortest computation time for all tools and threads numbers for a variant of the algorithm.

Other experiments. Results for a few experiments are not listed in the tables, but are worth mentioning. To check if the reason of relatively poor performance of OpenMP-based implementation is the lack of condition variable, the author implemented a variant of program using Pthreads, where active waiting is used instead of the condvar. Performance of this variant did not differ from the performance of the other Pthreads-based variant. So the problem is not active waiting loop, but something in the internal implementation of OpenMP. The author also tried to optimize the performance of TBB-based variant, by using lock-free synchronization; instead of having mutexes guarding the two lists of solutions, we use the `compare-and-swap()` instruction to add boxes to them, retrying the insertion in the case of collision. Measurements showed that collisions (and retrials) had been relatively rare, but the improvement of program performance was negligible if any.

7 Conclusions

Comparison of different threading tools is not simple as several criteria might be considered. Even comparison of computation times must take into account that programs using different linbraries have to be diffferent, as it was discussed in previous sections. All four tools allow to obtain some speedup, but far below the linear one. Results are comparable, but in several cases (all methods for problem (1) and methods “Hansen” and “cmp Newton” for problem (3)) TBB performed best (obtained shortest computation times). As for other tools, OpenMP seems to scale worse than other tools for 8

Table 1. Parallelization of algorithms for Problem (1).

Hansen method							
tool \ threads num.		1	2	4	6	7	8
OpenMP	time (sec.)	268	184	109	95	91	87
	speedup	1	1.46	2.46	2.82	2.95	3.08
POSIX	time (sec.)	275	171	106	90	90	87
	speedup	1	1.61	2.59	3.06	3.06	3.16
Boost	time (sec.)	276	182	113	88	90	92
	speedup	1	1.52	2.44	3.14	3.07	3.00
TBB	time (sec.)	273	158	89	**68**	69	84
	speedup	1	1.73	3.07	4.01	3.96	3.25
Neumaier method							
tool \ threads num.		1	2	4	6	7	8
OpenMP	time (sec.)	221	128	83	72	67	64
	speedup	1	1.73	2.66	3.07	3.30	3.45
POSIX	time (sec.)	235	130	78	69	67	63
	speedup	1	1.80	3.01	3.41	3.51	3.73
Boost	time (sec.)	227	138	82	76	70	67
	speedup	1	1.64	2.77	2.99	3.24	3.39
TBB	time (sec.)	212	121	64	55	59	**54**
	speedup	1	1.75	3.31	3.85	3.59	3.93
cmp Newton method							
tool \ threads num.		1	2	4	6	7	8
OpenMP	time (sec.)	197	129	78	66	62	61
	speedup	1	1.53	2.53	2.98	3.18	3.23
POSIX	time (sec.)	207	125	75	65	60	55
	speedup	1	1.66	2.76	3.18	3.45	3.76
Boost	time (sec.)	202	130	81	68	63	62
	speedup	1	1.55	2.49	2.97	3.21	3.26
TBB	time (sec.)	197	114	66	50	**47**	54
	speedup	1	1.73	2.98	3.94	4.19	3.65

threads – the difference seems especially significant for test problem (3). Boost threads perform noticeably worse than POSIX threads for problem (1), but for other two problems results are less clear – reasons of this behavior require further investigation. TBB seems particularly interesting as a parallelization tools and further investigations of this library for use in interval computations are planned.

Bibliography

[1] T. Beelitz, B. Lang, C. H. Bischof, "*Efficient Task Scheduling in the Parallel Result-Verifying Solution of Nonlinear Systems*", *Reliable Computing* 12(2006), pp.141-151.

Table 2. Parallelization of algorithms for Problem (2) with 7 equations.

Hansen method							
tool \ threads num.		1	2	4	6	7	8
OpenMP	time (sec.)	1628	1283	488	353	321	**314**
	speedup	1	1.27	3.34	4.61	5.07	5.18
POSIX	time (sec.)	1645	880	489	412	399	314
	speedup	1	1.87	3.36	3.99	4.12	5.24
Boost	time (sec.)	1757	894	479	380	375	320
	speedup	1	1.96	3.67	4.62	4.69	5.49
TBB	time (sec.)	1660	866	446	332	326	317
	speedup	1	1.92	3.72	5.00	5.09	5.24
Neumaier method							
tool \ threads num.		1	2	4	6	7	8
OpenMP	time (sec.)	162	86	44	33	32	31
	speedup	1	1.88	3.68	4.91	5.06	5.23
POSIX	time (sec.)	168	87	47	39	35	32
	speedup	1	1.93	3.57	4.31	4.80	5.25
Boost	time (sec.)	166	86	47	40	36	33
	speedup	1	1.93	3.53	4.15	4.61	5.03
TBB	time (sec.)	161	82	44	33	31	**30**
	speedup	1	1.96	3.66	4.88	5.19	5.37
cmp Newton method							
tool \ threads num.		1	2	4	6	7	8
OpenMP	time (sec.)	1168	644	377	288	283	280
	speedup	1	1.81	3.10	4.06	4.13	4.17
POSIX	time (sec.)	1235	654	357	303	270	**267**
	speedup	1	1.88	3.46	4.08	4.57	4.63
Boost	time (sec.)	1243	672	397	358	341	314
	speedup	1	1.85	3.13	3.57	3.65	3.96
TBB	time (sec.)	1211	643	322	306	314	291
	speedup	1	1.88	3.76	3.96	3.86	4.16

[2] T. Beelitz, C. H. Bischof, B. Lang, "*Intervals and OpenMP: Towards an Efficient Parallel Result-Verifying Nonlinear Solver*", in: An Mey, D. (ed.), Proc. EWOMP'03, September 22-26, 2003, Aachen, Germany, Aachen, 2003, pp. 119-125.

[3] B. J. Kubica, "*Interval methods for solving underdetermined nonlinear equations systems*", presented at SCAN 2008, El Paso, Texas, 2008.

[4] B. J. Kubica, A. Wozniak, "*A multi-threaded interval algorithm for the Pareto-front computation in a multi-core environment*", presented at PARA 2008 Conference, Trondheim, Norway, 2008.

[5] N. Maclaren, "*Why POSIX Threads Are Unsuitable for C++*", a C++ Standard Committee Paper, 2006, `http://www.open-std.org/jtc1/sc22/wg21/` .

Table 3. Parallelization of algorithms for Problem (3).

Hansen method							
tool \ threads num.		1	2	4	6	7	8
OpenMP	time (sec.)	4250	2249	1201	895	894	893
	speedup	1	1.89	3.54	4.75	4.75	4.76
POSIX	time (sec.)	4486	2329	1231	907	881	846
	speedup	1	1.93	3.64	4.95	5.09	5.30
Boost	time (sec.)	4398	2282	1212	924	863	859
	speedup	1	1.93	3.63	4.76	5.10	5.12
TBB	time (sec.)	4441	2269	1160	887	801	**784**
	speedup	1	1.96	3.83	5.01	5.54	5.66
Neumaier method							
tool \ threads num.		1	2	4	6	7	8
OpenMP	time (sec.)	1759	923	489	369	371	362
	speedup	1	1.91	3.60	4.77	4.74	4.86
POSIX	time (sec.)	1793	976	491	403	354	352
	speedup	1	1.84	3.65	4.45	5.06	5.09
Boost	time (sec.)	1767	947	479	377	348	**345**
	speedup	1	1.87	3.69	4.69	5.08	5.12
TBB	time (sec.)	1822	930	478	365	364	359
	speedup	1	1.96	3.81	4.99	5.01	5.08
cmp Newton method							
tool \ threads num.		1	2	4	6	7	8
OpenMP	time (sec.)	1118	585	325	240	228	221
	speedup	1	1.91	3.44	4.66	4.90	5.06
POSIX	time (sec.)	1183	620	327	246	230	210
	speedup	1	1.91	3.62	4.81	5.14	5.63
Boost	time (sec.)	1125	593	307	232	210	214
	speedup	1	1.90	3.66	4.85	5.36	5.26
TBB	time (sec.)	1165	591	309	244	228	**204**
	speedup	1	1.97	3.77	4.77	5.11	5.71

[6] A. Neumaier, "*The Enclosure of Solutions of Parameter-Dependent Systems of Equations*", in *Reliability in Computing* (ed. Moore, R), Academic Press, 1988.

[7] R. van der Pas, "*Using OpenMP to parallelize interval algorithms*", presented at SCAN 2008, El Paso, Texas, 2008.

[8] C-XSC interval library, `http://www.xsc.de` .

[9] Boost `http://www.boost.org` .

[10] OpenMP `http://www.openmp.org` .

[11] POSIX Threads Programming `https://computing.llnl.gov/tutorials/pthreads` .

[12] Intel Threading Building Blocks `http://www.threadingbuildingblocks.org` .