Bergische Universität
Wuppertal

Using C-XSC in a Multi-Threaded Environment

Michael Zimmer

Preprint
BUW-WRSWT 2011/2

Wissenschaftliches Rechnen/
Softwaretechnologie

Impressum

Herausgeber:	Prof. Dr. W. Krämer, Dr. W. Hofschuster Wissenschaftliches Rechnen/Softwaretechnologie Fachbereich C (Mathematik und Naturwissenschaften) Bergische Universität Wuppertal Gaußstr. 20 42097 Wuppertal, Germany

Internet-Zugriff

Die Berichte sind in elektronischer Form erhältlich über die World Wide Web Seiten

`http://www2.math.uni-wuppertal.de/wrswt/literatur.html`

Autoren-Kontaktadresse

Michael Zimmer
Bergische Universität Wuppertal
Gaußstr. 20
D-42097 Wuppertal

E-mail: `zimmer@math.uni-wuppertal.de`

Using C-XSC in a Multi-Threaded Environment

Michael Zimmer

June 1, 2011

C-XSC is a C++ library providing many tools for numerical computation with a focus on interval arithmetic and verified methods. Since C-XSC is now nearly twenty years old, thread-safety has not been a focus in its development for a long time. In recent years, a lot of work has been invested into making C-XSC fit for high performance computing and the thread-safety of C-XSC has been vastly improved in the process. This document explains these advancements and especially gives advice concerning the remaining pitfalls when using C-XSC in a multi-threaded environment.

Note: This paper is based on C-XSC version 2.5.1

1 Introduction

The C-XSC library [6, 5] provides a vast amount of numerical tools, especially for interval arithmetic and reliable computing, ranging from basic data types for real or complex intervals and (interval-)vectors and matrices to a toolbox of preimplemented verified algorithms [4]. The first version of C-XSC was published in 1992 and is now nearly twenty years old. Some parts of the C-XSC code reach back to Pascal-XSC and are even older.

Obviously, not much thought was put into making the library thread safe in the beginning, since multi-processor systems at that time were very expensive. In recent years, processors with multiple cores have become the norm and are now built into computing devices of every price range. Thus the importance of shared memory parallelization, for example using OpenMP, has grown tremendously.

The focus of C-XSC development has also shifted in the last couple of years and many improvements have been made to make C-XSC fit for high performance computing, including support for BLAS- and LAPACK routines [12], dot product computations in K-fold `double` precision [1], data types for sparse matrices [11], MPI communication functions for C-XSC data types and parallelization of some routines for clusters and

shared memory systems [7, 3]. This has also lead to huge improvements in the thread safety of C-XSC.

Section 2 explains some of the problems of older C-XSC versions that have now been rectified. Section 3 deals with some remaining pitfalls and best practices for writing multi-threaded programs using C-XSC version 2.5.1 or higher. The examples in this paper use OpenMP [13] for the parallelization, which usually makes the code more readable than common libraries for thread programming and therefore makes the examples easier to comprehend.

2 Problems with older C-XSC versions

Nearly all of the problems concerning the thread-safety of older C-XSC versions stem from the use of global variables that hold temporary data during some computations. This was especially true for the use of operators that involve dot products, such as matrix-matrix or matrix-vector multiplication. Since older versions C-XSC use an object of the class `dotprecision` (representing a long accmulator [8, 9, 10]) to compute dot products which has a size of about 540 bytes, a global array of such objects was used in the operators to avoid allocation and deallocation in each operator call.

However, on modern systems the impact of using a local variable instead is neglible, while the use of such a global array often leads to problems when using multiple threads. Listing 1 shows an example for this.

```
#include <iostream>
#include <imatrix.hpp>  //Header for interval matrices
#include <omp.h>        //OpenMP header

using namespace cxsc;
using namespace std;

int main() {
  int threads = omp_get_num_threads(); //Number of threads used
  imatrix A(10,10)[threads]; //Create an array with one 10x10 matrix for
                             //each thread

  //Set all elements of matrix i to the interval [i,i+1]
  for(int i=0 ; i<threads ; i++)
    A[i] = interval(i,i+1);

  #pragma omp parallel
  {
    int id = omp_get_thread_num(); //id of this thread
    A[id] = A[id]*A[id]; //Leads to wrong result in older C-XSC versions
  }

  return 0;
}
```

Listing 1: Example program for the problems occuring in older C-XSC versions when using matrix- and vector-operators

The computation in the parallel block in this example will lead to wrong results, since every thread calls the `operator*` of `imatrix` to compute its matrix-matrix product, but there is only one global `dotprecision`-variable used for all computations. Since the memory access is not synchronized (and cannot be by the user without modifying the library itself) one gets random results. This was true for many operators, especially those computing one or more dot products.

A similar problem appears when using the staggered precision data types (`l_real`, `l_interval`, ...). In older versions, these also used global variables at different points, making the use of the staggered data types also not thread safe. Both of these problems have been recitified in C-XSC version 2.3.0, so these problems do **not** occur in the current C-XSC version.

Another problem occured during the use of the automatic differentiation and Hesse-arithmetic modules from the C-XSC toolbox. Both internally used a global variable depicting the order of the derivative to compute. If one called the computation of a function (`fEval`), its derivative (`dfEval`) or its second derivative (`ddfEval`) from multiple threads (even if the computations were meant for different functions) this could easily lead to wrong results for the derivative and second derivative and longer than necessary computation times for the function evaluation, since the global variable for the order of the derivative was switched both at the beginning and at the end of each of those functions.

This problem has been solved in C-XSC version 2.5.1 by using Thread Local Storage (more details on that in the next section). Note however that when using Mac OS X, Thread Local Storage is disabled by default since it is not supported by the systems current default compiler (GCC 4.2.1 with modifications by Apple). However, when compiling the current version of the GNU Compiler (4.6.0) from source, Thread Local Storage seems to work (other compilers like the Intel Compiler might also support it on Mac OS X).

If you are using Mac OS X and you are sure that your compiler does support Thread Local Storage, you can force its usage by compiling with the option `-D_CXSC_FORCE_TLS` (during installation, add this option to the optimization options and also use it when compiling your own programs). Note that this means that by default the automatic differentiation and Hesse-arithmetic are **not** thread-safe when using C-XSC under Mac OS X.

3 Writing multi-threaded programs using the current C-XSC Version

This section deals with remaining issues when using C-XSC in a multi-threaded environment that have to be taken into account. The advice given here can be essential for a correct program, but can also lead to faster computation times. C-XSC users that want to write multi-threaded programs should be aware of the issues presented here.

3.1 Global precision variables

C-XSC uses two global variables that set the precision of the according computations:

- `opdotprec`: Defines the precision of the dot product computations when using the respective operators for vector-vector, matrix-vector an matrix-matrix multiplication.
- `stagprec`: Sets the precision for the staggered data types like `l_interval` [2].

When compiling C-XSC with default settings, these two variables are normal global variables. That means that changing the value of one of these variables in one thread changes the value (and thus the precision of the according computations) for all threads. Listing 2 shows an example for this.

```
#include <omp.h>
#include <ivector.hpp>     //interval vectors (uses opdotprec)
#include <l_interval.hpp> //staggered intervals (uses stagprec)

using namespace cxsc;
using namespace std;

int main() {

  opdotprec = 5;
  stagprec = 5;

  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    printf("A: %d, %d, %d\n", id, opdotprec, stagprec);
    opdotprec = stagprec = id;
    printf("B: %d, %d, %d\n", id, opdotprec, stagprec);
  }

  return 0;

}
```

Listing 2: Effect of changing a global precision variable inside a parallel region

This example program first sets both precision variables to 5, then changes the current value in parallel to a different value per thread and puts out the current value before and after this change. The way this program is normally meant by the programmer should yield the following output when running with four threads (the order of the rows in this and all following examples can be more or less random due to the priority given to the threads by the operating system):

```
A: 0, 5, 5
B: 0, 0, 0
A: 1, 5, 5
B: 1, 1, 1
A: 2, 5, 5
```

```
B: 2, 2, 2
A: 3, 5, 5
B: 3, 3, 3
```

Instead, since the thread overwrites the current values of the global variables in a non-deterministic way, the above program gives results like the following:

```
A: 0, 5, 5
B: 0, 0, 0
A: 1, 0, 0
B: 1, 1, 1
A: 2, 1, 1
B: 2, 2, 2
A: 3, 2, 2
B: 3, 3, 3
```

One way to avoid this problem of course is to only change the precision outside of parallel blocks. However, depending on the problem that one wants to parallelize this might not always be possible. For example if one thread has to compute an approximation with precision one, while another thread tries to compute a residuum as exact as possible with maximum precision, the precision setting for each thread must be different. In this case, we provide the option of using Thread Local Storage for the global precision variables.

Thread Local Storage means that every thread uses its own copy of a global variable, such that changes to its value in one thread do not affect the value of all other threads. All modern compilers support this storage scheme, which is usually activated by using the `__thread` directive (for Windows systems it is usually `__declspec(thread)`), for example

```
static __thread int stagprec
```

when declaring or defining the global variable. To activate Thread Local Storage, one has to compile C-XSC with the option `-DCXSC_USE_TLS_PREC` (for example by adding it to the optimization options) and one must also use this option when compiling own programs. Note that when using Mac OS X, the limitations to Thread Local Storage described at the end of Section 2 also apply here, meaning that this option should only be activated if your compiler supports Thread Local Storage.

Every thread will then automatically receive its own copy of the two global precision variables. However, there is one caveat. When this option is activated, changes to the two precision variables outside of a thread (in the serial part of the program) will *only* affect the value of the copy of the precision variables of thread 0. This means that the output of the above example would be:

```
A: 0, 5, 5
B: 0, 0, 0
A: 1, 0, 0
B: 1, 1, 1
```

```
A: 2, 0, 0
B: 2, 2, 2
A: 3, 0, 0
B: 3, 3, 3
```

Thus the programmer has to take care that the desired accuracy is set at the beginning of *each* parallel block and must be aware that changes in the serial part of the program do *not* affect every thread automatically.

Thread Local Storage is similar to the `threadprivate` directive in OpenMP. Using the `copyin` clause, which tells OpenMP to copy the value of the global variable to all the copies of this variable of the respective threads, one can even avoid the above mentioned problem. However, OpenMP requires that a variable is declared `threadprivate` before its first use. In our tests, the threadprivate directive had to be used directly after the first declaration of the variable, directly in the C-XSC source files (only the Intel Compiler allows the use of the `threadprivate` directive directly in the program source without changing the library). Therefore, and because using the `threadprivate` directive would only support OpenMP, we decided to only give the option of using Thread Local Storage for the global precision variables.

3.2 Matrix/Vector operations

As described in Section 2, the operators for matrix/vector operations have been thread safe since C-XSC version 2.3.0. However, we want to give some advice on the parallelization of such operations, focussing especially on matrix-matrix multiplication.

As mentioned before, C-XSC provides dot product computations with choosable precision K (K-fold double precision, with $K = 0$ meaning maximum precision using the long accumulator). For all operators containing dot products, this precision is set using the `opdotprec` variable. Its use in multi-threaded programs has been described in Section 3.1. For precision $K = 1$ (simple floating point computation in double precision), there is also the option of using highly optimized BLAS libraries by compiling with the option `-D_CXSC_USE_BLAS` and linking to the BLAS and CBLAS libraries.

Since most BLAS libraries are already multithreaded (and very effectively so), one should use them for all computations requiring only `double` precision, instead of trying to parallelize the operations explicitly by hand. Listing 3 gives an example.

```
#include <rmatrix.hpp>     //real matrix

using namespace cxsc;
using namespace std;

int main() {
  rmatrix A(1000,1000), B(1000,1000);


  //Example data: Sets a_ij to max(i,j)
  for(int i=1 ; i<=1000 ; i++)
    for(int j=1 ; j<=1000 ; j++)
      A[i][j] = (i>j) ? i : j;
```

```
  opdotprec = 1;

  //If BLAS is activated, this is already a parallel operation:
  B = A * A;

  //Parallelization of the for loop
  #pragma omp parallel for
  for(int i=1 ; i<=1000 ; i++)
    for(int j=1 ; j<=1000 ; j++)
      B[i][j] = A[i] * A[Col(j)];

  return 0;

}
```

Listing 3: Parallelization of matrix-matrix product

On our test system (2 Intel Xeon 2.26 GHz with four cores each and 24 GB RAM) using the Intel Compiler 11.1 and the Intel MKL, the BLAS version takes 0.07 seconds, while the naive for loop parallelization takes 1.4 seconds. Of course, there are better ways to parallelize a matrix-matrix product by hand, but the use of a good BLAS library is nearly always preferrable.

However, when using higher or even maximum precision, one has to do the parallelization by hand. In the above example, when changing the precision to 2 and running the for-loop with 1, 2, 4 and 8 threads on the above mentioned test system, one gets the timings shown in the following table.

Cores	Time in s	Speed Up
1	17.7	1.00
2	8.9	1.99
4	4.5	3.93
8	2.3	7.69

As the table shows, the naive parallelization here works very well and gives nearly optimal speed ups. This is due to the much greater amount of computations necessary and the ability to utilize the cache much more efficient in `double`-precision than with simulated higher precision when using a smart parallelization algorithm.

Some more information on the topic of dot product computations of choosable precision in C-XSC can be found in [1].

3.3 In- and output of C-XSC types

C-XSC provides in- and output operators for alle C-XSC data types using the stream concept of C++. Using output streams in combination with multi-threading can often lead to problems due to the nature of streams. Listing 4 gives an example of such problems.

```
#include <iostream>
#include <iomanip>
#include <omp.h>
```

```
using namespace std;

int main() {

  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    if(id % 2 == 0) {
      cout << "id=" << id << ": ";
      for(int i=2 ; i<=10 ; i+=2)
        cout << i << " ";
      cout << endl;
    } else {
      cout << "id=" << id << ":";
      cout << showpos;
      for(int i=1 ; i<10 ; i+=2)
        cout << i << " ";
      cout << endl;
    }
  }

  return 0;
}
```

Listing 4: Problems when using output streams and multi-threading

In this program, every thread with an even id number shall give out all even numbers between 1 and 10 and every thread with an odd id number shall give out all odd numbers between one and ten with a leading plus sign (using the respective stream manipulator `showpos`). However, running this program with four threads one gets an output such as the following:

```
id=id=02: : 2 id=2 4 6 8 10 1:id=+14  +3+6  +3+8+5  :+7
+10
+1 +3 +5 +7 +9
 +9
```

The problem here is that every thread writes data to the same stream (`cout`) and since the data is not written as a whole, but in small portions, it arrives on the stream in more or less random fashion. Also, the stream manipulator `showpos` changes a global IO-flag, which affects all other threads.

C-XSC in- and output with streams has the same problems, but the global IO-flags (C-XSC has its own) are especially problematic. When putting out intervals, one has to consider that the bounds of the interval are stored as binary floating point numbers, which are usually converted to decimal numbers for the output as a string. Thus, the bounds of the interval can be rounded inwards for the output, potentially putting out a wrong interval. To avoid this, C-XSC uses directed rounding activated by stream manipulators in the output operators, as shown in Listing 5.

```
std::ostream & operator << (std::ostream &s, const interval& a) throw()
{
```

```
    //Stores the current IO flags, then puts out the infimum
    //with downward rounding and the supremum with upward
    //rounding, and restores the previous IO flags
    s << '['          << SaveOpt << RndDown
      << a.inf << ',' << RndUp
      << a.sup        << RestoreOpt
      << ']';

    return s;
}
```

Listing 5: Interval output operator

Since the stream manipulators change global IO-flags, this causes similar problems like in the example at the beginning of this subsection when using multi threading, where the threads overwrite the global flags in a non-deterministic way, causing the output to be wrong in most cases.

Also, all output routines are based on legacy code, which formats the output in the desired way. This legacy code however is more than 20 years old and frequently uses global variables. For these reasons, the in- and output operators for C-XSC data types are in general not thread safe.

Because of this, if in- and output inside a thread is necessary, C-functions like `printf` should be used. These of course do not directly support the C-XSC types, but since (nearly) all C-XSC data types are based on the `double`-format, they can be used with these C-function nonetheless. However, we suggest to use hexadecimal output to avoid the rounding problems for the interval bounds as explained above. For this, we provide a function `realToHex(const real&)`, which converts a `real` value to a string of the hexadecimal representation of the respective floating point number. Listing 6 gives an example of how to put out some C-XSC data types in a multi-threaded program using `printf`.

```
#include <cstdio>
#include <cinterval.hpp> //includes all basic data types
#include <omp.h>

using namespace cxsc;

int main() {
  real r = 2.0;
  interval i(2.0,4.4);
  complex c(1.0,2.0);
  cinterval ci(i,-i);

  //Parallel output
  //All output is hexadecimal to avoid rounding errors
  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    //real output
    printf("%d: %s \n", id, realToHex(r).c_str());
    //interval output
    printf("%d: [%s,%s] \n", id, realToHex(Inf(i)).c_str(),
```

```
                                    realToHex(Sup(i)).c_str());
    //complex output
    printf("%d: (%s,%s) \n", id, realToHex(Re(c)).c_str(),
                                    realToHex(Im(c)).c_str());
    //cinterval output
    printf("%d: ([%s,%s],[%s,%s]) \n", id, realToHex(InfRe(ci)).c_str(),
                                           realToHex(SupRe(ci)).c_str(),
                                           realToHex(InfIm(ci)).c_str(),
                                           realToHex(SupIm(ci)).c_str());
  }

  return 0;
}
```

Listing 6: Output of some C-XSC data types using `printf`

A possible output of the above example program is:

```
1: +10000000000000e400
0: +10000000000000e400
1: [+10000000000000e400,+1199999999999Ae401]
1: (+10000000000000e3FF,+10000000000000e400)
0: [+10000000000000e400,+1199999999999Ae401]
1: ([+10000000000000e400,+1199999999999Ae401],[-1199999999999Ae401,-10000000000000e400])
0: (+10000000000000e3FF,+10000000000000e400)
0: ([+10000000000000e400,+1199999999999Ae401],[-1199999999999Ae401,-10000000000000e400])
```

The hexadecimal output here is in the same format as the hexadecimal output of the C-XSC output operators: The sign, followed by the mantissa (the leading 1 is always displayed, `10000000000000` means `1.0000000000000`) in hexadecimal format, followed by an `e` and the exponent in hexadecimal format including the bias 1023 (or 3FF in hexadecimal).

An alternative that allows the use of streams is to put a lock on all regions that contain in- or output statements. This way, the respective parts of the program will only be executed by one thread at once. However, this forces the other threads to wait till they can execute the code, which might not be desired depending on the circumstances. In OpenMP, the example in Listing 4 can be rewritten as seen in Listing 7 to work this way.

```
#include <iostream>
#include <iomanip>
#include <omp.h>

using namespace std;

int main() {

  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    if(id % 2 == 0) {
      #pragma omp critical
```

```
      {
        cout << "id=" << id << ": ";
        for(int i=2 ; i<=10 ; i+=2)
          cout << i << " ";
        cout << endl;
      }
    } else {
      #pragma omp critical
      {
        cout << "id=" << id << ":";
        cout << showpos;
        for(int i=1 ; i<10 ; i+=2)
          cout << i << " ";
        cout << noshowpos;
        cout << endl;
      }
    }
  }

  return 0;
}
```

Listing 7: Using a lock to make output streams in a parallel region work correctly

This method works nicely if only a final result of a lengthy computation is put out by each thread, since the synchronization will then have only a neglible effect on the performance. In all other cases, the uses of `printf` and other C-style in- and output functions should be preferred.

4 Conclusion

A lot of work went into making C-XSC as thread safe as possible. The remaining known issues described in this paper should not hamper the creation of multi-threaded C-XSC programs in a meaningful way, but only require some consideration by the programer. The following points summarize the current state of the thread-safety of C-XSC:

- The global precision variables should either not be changed inside of a thread or synchronized using Thread Local Storage.
- In- and output of C-XSC data types using streams is NOT thread safe.
- The rest of the C-XSC library, and especially all operators for C-XSC data types, should be thread safe.
- However, when using Mac OS X, the limitations to Thread Local Storage as described at the end of Section 2 can cause additional problems, especially in the automatic differentiation and Hesse-arithmetic modules.
- Some additional software packages from the C-XSC website might not be thread-safe yet, but will be checked and, if necessary, corrected in the near future.

The future C-XSC version 3.0 should rectify most of the remaining problems due to major rewrites, especially of the legacy code parts that create most of the issues.

Should you encounter any issues with the thread safety of C-XSC not mentioned in this paper, please feel free to contact us at
`xsc@math.uni-wuppertal.de`

References

[1] Zimmer, M.; Krämer, W.; Bohlender, G.; Hofschuster, W.: Extension of the C-XSC Library with Scalar Products with Selectable Accuracy, to appear in Serdica Journal of Computation

[2] Blomquist, F.; Hofschuster, W.; Krämer, W.: A Modified Staggered Correction Arithmetic with Enhanced Accuracy and Very Wide Exponent Range. Lecture Notes in Computer Science LNCS 5492, pp. 41–67, Springer, 2009.

[3] Grimmer, M.: Selbstverifizierende Mathematische Softwarewerkzeuge im High-Performance Computing. Konzeption, Entwicklung und Analyse am Beispiel der parallelen verifizierten Loesung linearer Fredholmscher Integralgleichungen zweiter Art. Logos Verlag, 2007.

[4] R. Hammer and M. Hocks and U. Kulisch and D. Ratz: Numerical Toolbox for Verified Computing I: Basic Numerical Problems. Springer Verlag, 1993

[5] Hofschuster, W.; Krämer, W.: C-XSC 2.0: A C++ Library for Extended Scientific Computing. Numerical Software with Result Verification, Lecture Notes in Computer Science, Volume 2991/2004, Springer-Verlag, Heidelberg, pp. 15 - 35, 2004.

[6] Klatte, Kulisch, Wiethoff, Lawo, Rauch: C-XSC - A C++ Class Library for Extended Scientific Computing. Springer-Verlag, Heidelberg, 1993.

[7] Krämer, W., Zimmer, M.: Fast (Parallel) Dense Linear System Solvers in C-XSC Using Error Free Transformations and BLAS. Lecture Notes in Computer Science LNCS 5492, pp. 230–249, Springer, 2009.

[8] Kulisch, U.; Miranker, W.: The arithmetic of the digital computer: A new approach. SIAM Rev., 28(1):1-40, 1986.

[9] Kulisch, U.: Die fünfte Gleitkommaoperation für Top-Performance Computer. Berichte aus dem Forschungsschwerpunkt Computerarithmetik, Intervallrechnung und numerische Algorithmen mit Ergebnisverifikation, 1997.

[10] Kulisch, U.: Computer Arithmetic and Validity: Theory, Implementation, and Applications, de Gruyter Studies in Mathematics, 2008.

[11] Zimmer, M.; Krämer, W.; Hofschuster, W.: Sparse Matrices and Vectors in C-XSC, BUW-WRSWT 2009/7, Preprint 2009/7, Universität Wuppertal, 2009

[12] Zimmer, M.; Krämer, W.; Hofschuster, W.: Using C-XSC in High Performance Computing BUW-WRSWT 2009/5, Preprint 2009/5, Universität Wuppertal, 2009

[13] OpenMP specification:
http://openmp.org/wp/openmp-specifications/

[14] C-XSC Website:
http://www2.math.uni-wuppertal.de/wrswt/xsc/cxsc_new.html