Multi-threaded Rigorous Global Optimization with Intel Threading Building Blocks and Thread-safe Parts of C-XSC

Brendan Bycroft
brb44@student.canterbury.ac.nz

Raazesh Sainudiin*
Department of Mathematics and Statistics,
University of Canterbury, Private Bag 4800,
Christchurch 8041, New Zealand
r.sainudiin@math.canterbury.ac.nz

Abstract

Rigorously enclosing the global minimum and minimizer of a function can take a long time especially if the function is complicated. A parallel global optimization algorithm was created in C++ using algorithmic skeletons readily available from the Intel Threading Building Blocks (TBB) template library. The parallel algorithm aimed to be faster than the standard global optimization algorithm implemented in the popular C-XSC library for interval computations by utilizing CPUs with multiple cores so that parts of the algorithm could run in parallel. When run on a single core, the parallel algorithm is slower than the standard algorithm, due to constraints of having to run in parallel. However, it is successful in that the speed of the algorithm that relies on the `parallel_for` construct in TBB increases almost linearly with the number of cores used. In contrast, the standard algorithm does not improve in speed when additional cores are utilised. Our experiments demonstrate the ease with which performance can increase in close proportion to the number of CPU cores used in fully portable multi-threaded programs written using thread-safe parts of C-XSC library and parallel algorithmic skeletons provided by TBB library.

Keywords: parallel global optimization, TBB, C-XSC, parallel_for and parallel_while
AMS subject classifications: 65G20,65Y05,90C26

*Corresponding Author

1 Introduction

There is an increasing emphasis on multicore chip design, due to engineering challenges posed by any further significant increase in processor clock speeds. The general trend in processor development has moved from dual-, tri-, quad-, hexa-, octo-core chips to ones with tens or even hundreds of cores. In order to take advantage of these new chips there is an urgent need for genuinely multithreaded software. Algorithms will stagnate at a performance ceiling if they are unable to exploit the resources available on machines with multiple cores by using multiple threads. Intel Threading Building Blocks (also known as TBB) [9], a new abstraction for C++ parallelism, is a C++ template library developed by Intel Corporation. Programs written with TBB take advantage of multi-core processors that are becoming common today. In this paper we investigate two of the simplest parallel global optimization algorithms using the `parallel_while` and the `parallel_for` skeletons in TBB and the thread-safe parts of C-XSC library [12] for interval computations.

Rigorous parallel global optimization algorithms in previous studies focussed on distributed memory computers, such as cluster of Linux machines (for example see [4, 7, 3, 1, 11, 5, 6]). Such techniques generally use message passing such as MPI to communicate between processors. Truly multi-threaded shared-memory algorithms have shown promise [10, 2]. The POSIX Thread NPTL library (version 2.3.5) was used in [2] to create threads in a thread-safe reimplementation of the C-XSC 2.1.1, the widely used interval class library. Here we focus on abstract multi-threaded programming using the the thread-safe parts of the publicly distributed C-XSC 2.5.0 library and the parallel algorithmic skeletons readily available in TBB (version 2.1) as opposed to more tedious programming with POSIX threads.

This paper is organized as follows. We give a brief introduction to parallel computing in § 1.1 and global optimization using interval methods in § 1.2. We describe the two parallel global optimization algorithms along with some results using TBB's `parallel_while` and `parallel_for` in § 2 and § 3, respectively. We conclude in § 4 and include the code in § 5.

1.1 Parallel Computing

Parallel computing is doing multiple things at once using multiple CPU cores on a single computer. The reason for having multiple cores in a computer is almost entirely for speed. Being able to utilise more than one simultaneously means that a program can do all of its calculations in a shorter time.

In software, CPU cores are accessed by threads. These are controlled by the underlying operating system, and are available as part of the C standard library. Threads always have access to the same memory space in a program but they each have their own set of local variables. The shared memory means it is possible for two threads to access the same global variable, which can be problematic if that variable is modified. To get around this, locks are used. Locks make sure that only one thread can read and modify a particular variable and so enforce mutual exclusion. If two threads try and access a variable under a lock at the same time, one of them is forced to wait until the lock is removed by the first one.

Some existing parallel programming models that allow multi-threaded programming such as Cilk++, FastFlow, OpenMP, MPI and Skandium can be used on multi-core platforms. Intel Threading Building Blocks (also known as TBB) [9], a new abstraction for C++ parallelism, is a C++ template library developed by Intel Cor-

poration. The TBB library consists of data structures and algorithms that circumvent the manual and tedious creation, synchronization and termination of individual threads of execution with native threading packages such as POSIX threads, Windows threads, or the portable Boost Threads. Instead TBB abstracts access to multiple processors by allowing the operations to be treated as *tasks*. These tasks are then assigned to threads by a task-scheduler. The advantage of this approach is that any number of threads may be created, and the tasks will be assigned dynamically by TBB library's run-time engine to individual cores while efficiently using cache. However, it does require that tasks of a sufficient work load are used. This is needed so that the work of the task-scheduler is small compared to the work done in the tasks. TBB's solution for parallel programming decouples the programming from the particulars of the underlying machine. As of March 2010, TBB is available in FreeBSD, several popular GNU/Linux, Microsoft Windows, Mac OS X and Sun Solaris distributions.

Thus a TBB program creates, synchronizes and destroys graphs of dependent tasks according to high-level parallel programming paradigms called *algorithmic skeletons*. Two algorithms which implemented global optimization in parallel were created. They were largely based around the constructs available as part of the TBB library. The second method in § 3 was created as the first one in § 2 was found to not be sufficient in solving the problem. All of our programs were run on a machine with two Intel X5482 3.2Ghz quad core Xeon CPUs, 32GB of ram, 2 x 320GB 15K Seagate SAS hard drives and openSuSE 11.1 (x86_64) OS.

1.2 Global Optimization

Global optimization is the task of finding the global minimum of a scalar function f in some domain D. That is,

$$\check{y} = \min_{x \in D} f(x) \tag{1}$$

A naïve approach to this would be to evaluate the function at a number of points and pick the lowest. This is not very good, as there is no guarantee that anything close to the true minimum would be found. A better approach uses intervals and interval arithmetic[8] to find the global minimum. An interval is defined as a pair of numbers which bound a region on the real line. We write

$$\boldsymbol{x} = [\underline{x}, \overline{x}] \tag{2}$$

where $\underline{x}, \overline{x} \in \mathbb{R}$ and $\underline{x} \leq \overline{x}$. We have $\boldsymbol{x} \in \mathbb{IR} := \{\boldsymbol{x} = [\underline{x}, \overline{x}] : \underline{x} \leq \overline{x}, \underline{x}, \overline{x} \in \mathbb{R}\}$.

For global optimization, a function $\boldsymbol{y} = F(\boldsymbol{x})$ is used which has $F : \mathbb{IR} \mapsto \mathbb{IR}$. This is the natural interval extension of $f : \mathbb{R} \to \mathbb{R}$. This means that F will give a range enclosure of the possible values f can take for the values in $\boldsymbol{x}$. That is,

$$F(\boldsymbol{x}) \supseteq \{f(x) : x \in D\} \tag{3}$$

The basic algorithm is as follows. The domain which is being looked at is divided up into a number of sub-intervals $\boldsymbol{x}_i$. For each of the sub-intervals, the range enclosure is computed as

$$\boldsymbol{y}_i = F(\boldsymbol{x}_i) \tag{4}$$

Now, if the minimum of one enclosure is larger than the maximum of another enclosure, then the interval with the higher enclosure can be ruled out as containing the global minimum. The remaining intervals are then subdivided into smaller ones. These smaller intervals then have corresponding smaller range enclosures. The removal

process can then be repeated. This is done until some sort of tolerance is reached. At this point, the remaining intervals give a location of the global minimum, which can be made to be arbitrarily small. The key idea is that the final bounds are guaranteed to contain the global minimum. This strategy to find the global minimum is called the *branch-and-bound* method.

It is also possible to discard intervals by other means under further assumptions on the function f, such as smoothness. If a function is monotone in some region, it cannot contain a minimum in its interior. Also, if the function's second derivative is always negative in a region, then that region cannot contain a minimum in its interior either. Both of these situations can be handled robustly using automatic differentiation. Automatic differentiation makes it possible to calculate the range enclosures of the first and second derivatives of a function.

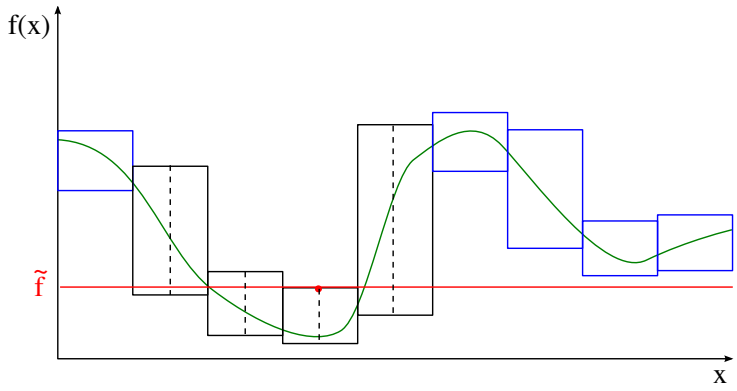

Figure 1: Global optimization. The blue intervals can be discarded as they lie entirely above the $\tilde{f}$ line. The remaining intervals are then bisected.

The program was written in C++ using the popular C-XSC 2.5.0 library[12]. This provided an interval class, and various operations on them. While there was extensive support for intervals and arithmetic, it had major problems as a lot of the code was not safe for use in threads. This was mainly due to many global variables being used to hold temporary information. As such, the computation of the derivatives of functions was not possible, and only functions with a one-dimensional domain could be used. It would be possible to make such a library thread-safe, as done in [2], but that is beyond the scope of this project. This project aims to use TBB's readily available parallel algorithmic skeletons and the thread-safe parts of the publicly distributed C-XSC 2.5.0 library to investigate the extent of speedup from fully portable and multi-threaded variants of the simplest branch-and-bound method. We hope that our experiments with the thread-safe parts of C-XSC library will increase interest in making C-XSC entirely thread-safe.

2 Method 1: While-loop Method

This method was based around the `parallel_while` construct in TBB. It uses a working list of intervals, where the intervals are continuously added and removed from the

list as the algorithm progresses. The intervals represent domains of the function which are candidates for containing the global minimum. The intervals in the list are able to be tested in parallel, and could be ruled out as containing the global minimum. Intervals were ruled out simply by their range enclosures. A global variable `f_min` was held which contained a value that was known to be above the function at some point. Thus, intervals whose range enclosures lay entirely above `f_min` could be removed. If this wasn't the case, the interval was bisected, and the two new intervals were added to the working list. Also, the central point of the interval was checked to see if it was possible to make `f_min` lower. Finally, if an interval was considered to be too small to subdivide further, it was added to a result list. The algorithm could then terminate when there were no more intervals in the working list. The working list was handled internally by TBB and had a queue like structure. The variable `f_min` had a lock on it, so that only one thread could read or write to it at a time. Also, the result list could be added to by multiple threads at a time.

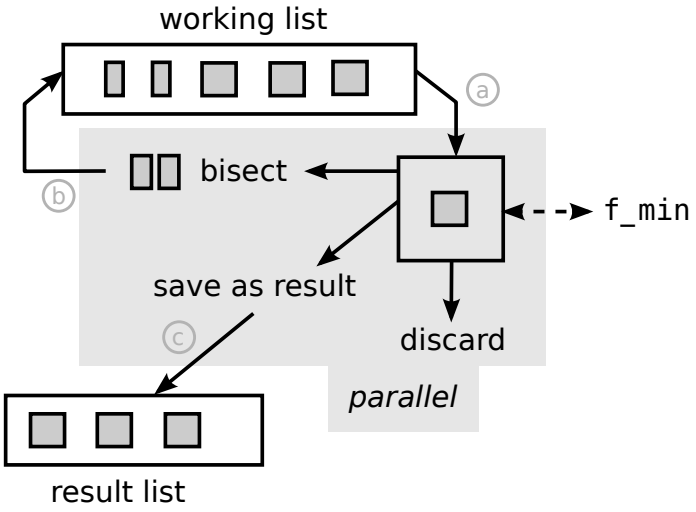

Figure 2: While-loop method schematic. a) Intervals are taken from the working list and checked. b) The interval is bisected and added to the back of the working list. c) Small intervals are saved in the result list.

Results

This method had a number of shortcomings. The core problem was the global `f_min` variable. This caused different sets of candidate intervals to be operated on, depending only on the evaluation order of those intervals. This could happen if one interval which had a low range enclosure was stuck in a thread. The other thread could then run, and subdivide substantially. If the first thread was not stuck, the intervals in the second thread might have been removed early on by the low `f_min` value. So in this example, when the first thread is stuck, a lot of unnecessary work is done. This makes for an unstable algorithm, and the parallelization can often make the global optimization process take somewhat longer than when single threaded.

3 Method 2: For-loop Method

This method was based around the `parallel_for` construct in TBB. This is because at each stage in the process, the number of calculations to carry out is known in advance, so work can be assigned to threads easily. As in the first method, intervals which are candidates for containing the global minimum are kept in a working list. Each interval in the list is tested to decide what to do with it. It is compared with the global variable `f_min`, and if the range enclosure is entirely above it, the interval is discarded. Otherwise, it is bisected. The two new intervals are then put in a new list. While running through this working list, the `f_min` variable remains unchanged. If it is possible to lower this value, the new value is stored in a different variable `f_min_new`. Once the working list has been run through, the working list is replaced with the new list, and `f_min` is replaced with `f_min_new`. Now, the latest set of intervals can be run through. If an interval is sufficiently small, it is flagged as such, and passed straight through into the new list. The algorithm then terminates when there are only these flagged intervals in the working list. The variable `f_min_new` is made thread-safe by adding a lock to it. Also, the container which contained the new list was one which could be added to by multiple threads running in parallel.

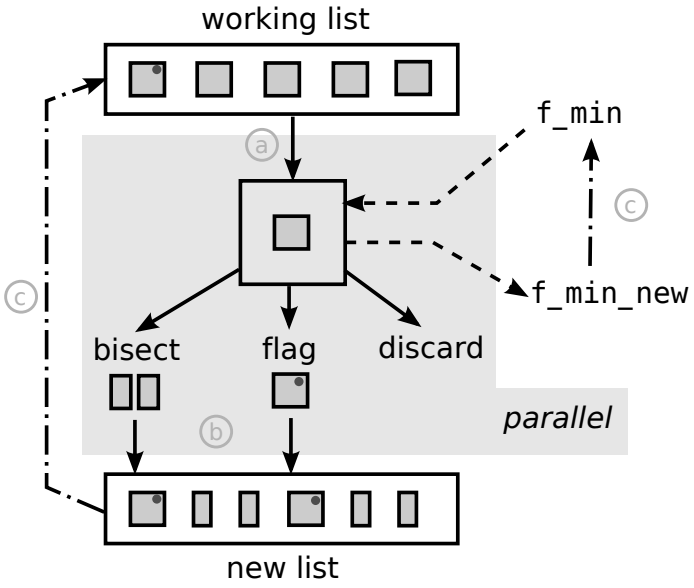

Figure 3: For-loop method schematic. a) Intervals are taken from the working list and checked. b) Checked intervals are passed into the new list. c) After all intervals in the working list have been checked, `f_min` becomes `f_min_new`, and the working list becomes the new list.

Results

The most important aspect of this algorithm is that the output will always remain the same for a given problem. In particular, the working list and `f_min` will be consistent across tests, independent of the order of execution in the program. This makes it much easier to reason about and check for correctness. This algorithm was successful, as its speed consistently increased as the number of cores used increased as shown in Figure 4. Therefore, it was taking full advantage of the hardware available on the computer.

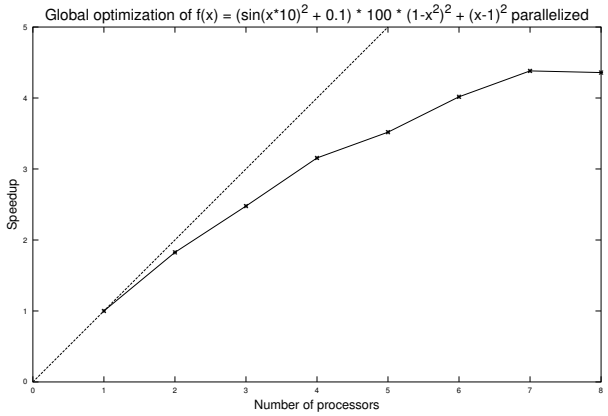

Figure 4: This graph shows that the algorithm speeds up as the number of processors utilised grows. Speedup for p processors or threads is defined as the time taken for p threads divided by the time taken for 1 thread.

4 Conclusion

Two methods were implemented to increase the performance of the global optimization algorithm when run on CPUs with multiple cores by using two of the simplest alorithmic skeletons in TBB library, namely `parallel_while` and `parallel_for`. The while-loop method was found to be unreliable as the computations performed depended on the order of execution of threads. In some circumstances, the number of computations changed a 100 fold on the same problem.

The for-loop method, on the other hand, got around these problems, and operated reliably on any number of CPU cores. This method also showed increases in speed as the number of cores increased. The major stumbling-block is that the popular C-XSC library is largely unsafe for use in programs with multiple threads. If this were not the case, then the algorithms presented here with parallel algorithmic skeletons from TBB could be trivially improved for significant performance gains on conventional

multi-core hardware. Adding thread-safety by avoiding global variables in `hess_ari` modules that underly multidimensional optimization routines in C-XSC would greatly help toward taking full advantage of multi-core machines. When C-XSC is thread-safe we can also add various accelerators such as the monotonicity and concavity tests. Finally, the abstractions provided by TBB library along with the portability naturally allows for multi-threaded programs written with C-XSC and TBB libraries to take full advantage of the multi-core machines that are becoming the norm.

References

[1] S. Berner. Parallel methods for verified global optimization practice and theory. *Journal of Global Optimization*, 9:1–22, 1996. 10.1007/BF00121748.

[2] L. G. Casado, J. A. Martinez, I. Garcia, and E. M. T. Hendrix. Branch-and-bound interval global optimization on shared memory multiprocessors. *Optimization Methods Software*, 23:689–701, October 2008.

[3] L. C. W. Dixon and M. Jha. Parallel algorithms for global optimization. *J. Optim. Theory Appl.*, 79:385–395, November 1993.

[4] J. Eriksson. *Parallel Global Optimization Using Interval Analysis.* pHd thesis, University of Umea, Sweden, 1991.

[5] S. Ibraev. *A New Parallel Method for Verified Global Optimization.* pHd thesis, Wuppertal University, Germany, 2001.

[6] S. Kahou. *Kahoue, J.H.T.* pHd thesis, Wuppertal University, Germany, 2005.

[7] R. Moore, E. Hansen, and A. Leclerc. Rigorous methods for global optimization. In C.A. Floudas and P.M. Pardalos, editors, *Recent advances in global optimization*, Princeton series in computer science, pages 321–342, Princeton, NJ, USA, 1992. Princeton University Press. Papers presented at a conference held at Princeton University, May 10–11, 1991.

[8] RE Moore. *Interval analysis.* Prentice-Hall, 1967.

[9] James Reinders. *Intel Threading Building Blocks.* O'Reilly, 2007.

[10] N. Revol, Y. Denneulin, J-F Méhaut, and B. Planquelle. A methodology of parallelization for continuous verified global optimization. In *Proceedings of the th International Conference on Parallel Processing and Applied Mathematics-Revised Papers*, PPAM '01, pages 803–810, London, UK, UK, 2002. Springer-Verlag.

[11] A. Wiethoff. *Verifizierte globale optimierung auf Parallelrechnern.* pHd thesis, Karlsruhe University, Germany, 1997.

[12] Universität Wuppertal. C-XSC 2.0: A C++ library for extended scientific computing. `http://www.math.uni-wuppertal.de/~xsc/xsc/cxsc.html`.

5 Code

gopt_while.cpp

```
/*
 * Copyright (C) 2009, 2010, 2011 Brendan Bycroft <brb44@uclive.ac.nz>
 * Description: Source code to implement the "while-loop" method for global
     optimization in parallel.

 * This program is free software; you can redistribute it and/or modify
```

```
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or (at
 * your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#include <iostream>

#include <ddf_ari.hpp>
#include <tbb/task_scheduler_init.h>
#include <tbb/parallel_while.h>
#include <tbb/spin_mutex.h>
#include <tbb/atomic.h>
#include <tbb/concurrent_vector.h>

typedef tbb::spin_mutex GlobalMaxMType;
GlobalMaxMType GlobalMaxM;
real global_max;

typedef tbb::spin_mutex CullResultMType;
CullResultMType CullResultM;

typedef struct {
    interval x;
    real fx_inf;
} Pair;

typedef tbb::concurrent_vector<Pair> PairList;

PairList result_list;

typedef interval (*myfunctype)(const interval&);

interval my_func(const interval& x) {
    return 100 * sqr (1.0 - sqr (x)) + sqr (x - 1.0);
}


void cull_results(real fmax, PairList& results) {
    PairList new_list;
    cout << results.size() << endl;
    for (unsigned int i=0; i<results.size(); i++) {
        if (fmax >= results[i].fx_inf) {
            new_list.push_back(results[i]);
        }
    }
    results = new_list;
}


real epsilon;
int max_results;
int n;
tbb::atomic<int> num_checked;
int max_num = 10;

class Body {
    tbb::parallel_while<Body>& my_while;
    myfunctype func;
public:
```

```
Body(tbb::parallel_while<Body>& w, myfunctype func_p) : my_while(w) {
    func = func_p;
    epsilon = 1e-8;
    max_results = 20000;
};

typedef interval* argument_type;

/* this is called by tbb::parallel_while */
void operator()(interval* item) const {
    check_interval(*item);
};
/* Splits an interval into two. */
void bisect_interval(interval& Y, interval& A, interval& B) const {
    real c;
    c = (Sup(Y) + Inf(Y))*0.5;
    //cout << Sup(Y) << " " << Inf(Y) << " " << c << endl;
    A = interval(Inf(Y), c);
    B = interval(c, Sup(Y));
};

/* this is called be check_interval, provided that the interval is still
   a candidate for the global minimum */
void bisect_or_save(interval &Y, interval &fY) const {
    /* stopping condition.  */
    if (RelDiam(fY) <= epsilon) {
        /* add interval to result list */
        Pair temp;
        temp.x = Y;
        temp.fx_inf = Inf(fY);

        result_list.push_back(temp);

    } else {
        /* bisect interval and add results to the work list. */
        interval *A, *B;
        A = new interval;
        B = new interval;
        bisect_interval(Y, *A, *B);

        my_while.add(A);
        my_while.add(B);
    }
};

void update_global_max(real newx) const {
    {
        GlobalMaxMType::scoped_lock lock(GlobalMaxM);
        if (newx < global_max) {
            global_max = newx;
        }
    }
}

void check_interval(interval U) const {
    num_checked++;

    interval fU, fC, fV, fcV;
    real c, cV;

    fU = func(U);

    {
        GlobalMaxMType::scoped_lock lock(GlobalMaxM);
        if (global_max < Inf(fU)) return;
    }

    /* would do monotonicity and concavity tests here, if they
```

```
           were thread-safe. */

        cV = (Sup(U) + Inf(U))*0.5;

        fcV = func(interval(cV));

        /* updating f_max */
        update_global_max(Sup(fcV));

        bisect_or_save(U, fU);

    };
};

//for globopt, there is only a single thing in the item stream, and it is
//the  whole interval.
class ItemStream {
    interval* my_ptr;
    bool ran_once;
public:
    ItemStream(interval* domain) {
        my_ptr = domain;
        ran_once = false;
    };

    bool pop_if_present(interval*& item) {
        if (!ran_once) {
            item = my_ptr;
            ran_once = true;
            return true;
        } else {
            return false;
        }
    };
};

void ParallelApplyGlobOpt(myfunctype f, interval& domain) {
    num_checked = 0;
    result_list.clear();
    global_max = Infinity;

    tbb::task_scheduler_init init(2);

    tbb::parallel_while<Body> w;

    ItemStream stream(&domain);

    /* the algorithm is applied here */
    w.run(stream, Body(w, f));

    /* remove results that aren't needed */
    cull_results(global_max, result_list);

    cout << "results:\n";
    for (unsigned int i=0; i<result_list.size(); ++i) {
        cout << result_list[i].x << "\n";
    }
    int a = num_checked;
    cout << "num_checked = " << a << endl;
}


int main(int argc, char* argv[]) {
    interval a(-3, 2);
    for (int i=0; i<100; ++i) {
        ParallelApplyGlobOpt(my_func, a);
    }
    return 0;
```

```
}
```

gopt_for.cpp

```
/*
 * Copyright (C) 2009, 2010, 2011 Brendan Bycroft <brb44@uclive.ac.nz>
 * Description: Source code to implement the "for-loop" method for global
     optimization in parallel.

 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or (at
 * your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#include <iostream>

#include <ddf_ari.hpp>
#include <tbb/task_scheduler_init.h>
#include <tbb/parallel_for.h>
#include <tbb/spin_mutex.h>
#include <tbb/atomic.h>
#include <tbb/concurrent_vector.h>
#include <tbb/blocked_range.h>
#include <tbb/tick_count.h>

typedef tbb::spin_mutex GlobalMaxMType;
GlobalMaxMType GlobalMaxM;

typedef ddf_FctPtr myfunctype;

/* some sample functions */
DerivType my_func(const DerivType& x)
{
    return (sqr(sin(x*10)) + 0.1) * 100 * sqr (1.0 - sqr (x)) + sqr (x - 1.0);
}

DerivType f1(const DerivType& x)
{
    return exp(0.2 * x) + power(x, 2)*sin(x - 2);
}

DerivType bell_func(const DerivType& x)
{
    real n = 123;
    return exp(exp(cos(x))*cos(sin(x))) * sin(exp(cos(x))*sin(sin(x)))*sin(n*x);
}

struct Pair {
    interval x;
    real ymin;
    bool is_small; /* if is_small is false, ymin is undefined. */
};

typedef tbb::concurrent_vector<Pair> PairVector;

/* This struct is called by parallel_for, and intervals are created and removed
   using the simple interval-based global optimization method.
*/
struct CheckIntervals {
```

```
/* constant through an iteration */
myfunctype func;
real *f_max;
PairVector *input;

/* will be updated through an iteration (must be thread-safe) */
real *f_max_next;
PairVector *output;
tbb::atomic<size_t> *any_left;

/* this is called by tbb::parallel_for */
void operator()(const tbb::blocked_range<size_t>& r) const {

    for (size_t i=r.begin(); i!=r.end(); ++i) {
        check_interval((*input)[i]);
    }
}

/* will either cull or bisect an interval. */
void check_interval(const Pair& in) const {

    if (in.is_small) {
        /* if interval is too small to be subdivided, it still
           has the possibility of being removed from the list. */
        if (*f_max < in.ymin) {
            return;                /* cull */
        } else {
            output->push_back(in); /* keep it in the list */
        }
    }

    interval x, y, dy, fc;
    real c;

    x = in.x;

    fEval(func, x, y); // y = func(x);

    /* checking with f_max of the previous interation set */
    if (*f_max < Inf(y))
        return;

    /* This function can not be called because it is not thread-safe */
    //dfEval(func, x, y, dy); // y = func(x); dy = dfunc(x);

    //if (is_monotone(dy))
    //    return;

    /* Do concavity test here if ddfEval function is thread-safe */

    /* using the central point to get an f_max_next */
    c = (Sup(x) + Inf(x))*0.5;
    fEval(func, interval(c), fc); // fc = func(interval(c));

    {
        GlobalMaxMType::scoped_lock lock(GlobalMaxM);
        if (Sup(fc) < *f_max_next) {
            *f_max_next = Sup(fc);
        }
    }

    /* stopping condition */
    if (RelDiam(x) <= 1e-14 || RelDiam(y) <= 1e-8) {
        /* interval no longer bisects. Keep in working set. */
        Pair a;
        a.x = x;
        a.ymin = Inf(y);
        a.is_small = true;
```

```
            output->push_back(a);
        } else {
            /* bisect and add two to working set. */
            Pair a, b;
            a.is_small = false;
            b.is_small = false;
            bisect_interval(x, a.x, b.x);
            (*any_left)++;
            output->push_back(a);
            output->push_back(b);
        }
    }

    void bisect_interval(const interval& x, interval& a, interval& b) const {
        real c;
        c = (Sup(x) + Inf(x))*0.5;
        a = interval(Inf(x), c);
        b = interval(c, Sup(x));
    }

    /* if region is monotone, it will not contain a minimum */
    bool is_monotone(interval& df) const {
        if (Inf(df) > 0.0 || Sup(df) < 0.0)
            return true;
        return false;
    }

    /* if region is concave_down, it will not contain a minimum */
    bool is_concave_down(interval& ddf) const {
        if (Sup(ddf) < 0.0)
            return true;
        return false;
    }

};

/* finds the global minimum in the interval of some function */
void run_loop(interval range, myfunctype function) {
    PairVector input, output;
    tbb::atomic<size_t> any_left;
    real f_max = Infinity, f_max_next = Infinity;
    int num_loops = 0;

    /* initializing interval-checker */
    CheckIntervals checker;
    checker.input = &input;
    checker.output = &output;
    checker.any_left = &any_left;
    checker.f_max = &f_max;
    checker.f_max_next = &f_max_next;
    checker.func = function;

    /* creating initial interval */
    Pair start;
    start.x = range;
    start.is_small = false;
    checker.input->push_back(start);

    while (num_loops++ < 100) {

        checker.output->clear();
        *(checker.any_left) = 0;

        /* run a single iteration */
        tbb::parallel_for(tbb::blocked_range<size_t>(0, checker.input->size(), 1), checker);

        /* continue loop if any remaining intervals are not yet small enough */
        if (*checker.any_left == 0) {
```

```
            break; /* the while loop should always exit via this break. */
        }

        /* swap input & output */
        *checker.input = *checker.output;
        *checker.f_max = *checker.f_max_next;

    }

    if (checker.output->size()) {
        Pair ans = (*checker.output)[0];
        cout << "bisected " << num_loops << " times. " << checker.output->size()
             << " results. " << ans.x << endl;
    } else {
        cout << "No answer found" << endl;
    }
}


int main() {

    tbb::tick_count t0, t1;

    int n = tbb::task_scheduler_init::default_num_threads();

    /* Testing the algorithm on the bell function using different numbers of
       threads.*/
    for(int p=1; p<=n; ++p) {
        /* Construct task scheduler with p threads */
        tbb::task_scheduler_init init(p);

        double min_t = 10000;
        for (int i=0; i<5; i++) {

            t0 = tbb::tick_count::now();
            run_loop(interval(0, 3.2), bell_func);
            t1 = tbb::tick_count::now();

            double t = (t1 - t0).seconds();
            if (t < min_t) min_t = t;
        }

        cout << "time = " << min_t << " with " << p << " threads\n";
    }
    cout << "default " << n << " threads\n";
    return 0;
}
```