Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение
высшего образования
Национальный исследовательский Нижегородский государственный
университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Отчет по по практике по получению первичных профессиональных умений и навыков

«Алгоритм глобального поиска для одномерных многоэкстремальных задач оптимизации»

Выполнил:
студент группы 381806-1
Стрельцова Я. Д.
Проверил:
доцент кафедры МОСТ,
кандидат технических наук
Сысоев А. В.

Нижний Новгород
2021

Оглавление

Введение

Многие задачи принятия оптимальных решений, возникающие в различных сферах человеческой деятельности, такие как моделирование климата, генная инженерия, проектирование интегральных схем, создание лекарственных препаратов и др., могут быть сформулированы как задачи оптимизации. Увеличение числа прикладных проблем, описываемых математическими моделями подобного типа, и бурное развитие вычислительной техники инициировали развитие глобальной оптимизации. При этом большое практическое значение имеют не только многомерные, но и одномерные задачи поиска глобально-оптимальных решений, часто встречающиеся, например, в электротехнике и электронике. Также одномерные алгоритмы поиска глобального экстремума могут быть использованы в качестве основы для конструирования численных методов решения многомерных задач оптимизации посредством применения схем редукции размерности.

Усложнение математических моделей оптимизируемых объектов затрудняет поиск оптимальной комбинации параметров, и как следствие, не представляется возможным найти такую комбинацию аналитически, поэтому возникает необходимость построения численных методов для ее поиска. Численные методы глобальной оптимизации существенно отличаются от стандартных локальных методов поиска, которые часто неспособны найти глобальное решение рассматриваемых задач, т.к. не в состоянии покинуть зоны притяжения локальных оптимумов и, соответственно, упускают глобальный оптимум.

Проблема численного решения задач оптимизации, в свою очередь, может быть сопряжена со значительными трудностями. Такие задачи могут характеризоваться многоэкстремальной, недифференцируемой или же заданной в форме черного ящика (т.е. в виде некоторой вычислительной процедуры, на вход которой подается аргумент, а на выходе наблюдается соответствующее значение) целевой функцией. Многоэкстремальные оптимизационные модели обладают высокой трудоемкостью численного анализа, поскольку для них характерен экспоненциальный рост вычислительных затрат с ростом размерности (количества параметров модели). Именно поэтому разработка эффективных параллельных методов для численного решения задач многоэкстремальной оптимизации и создание программных средств их реализации на современных многопроцессорных системах является стратегическим направлением по существенному развитию проблематики исследования сложных задач оптимального выбора.

Одним из методов решения задач одномерной оптимизации является алгоритм глобального поиска, носящий имя доктора физико-математических наук Романа Григорьевича Стронгина. Этот метод работает для функций, удовлетворяющих условию Липшица и основан на вероятностной модели функций, заданных на конечном множестве точек.

1. Постановка задачи

Задача:

Необходимо разработать и реализовать последовательный и параллельный варианты алгоритма глобального поиска для одномерных многоэкстремальных задач оптимизации, проверить корректность работы алгоритмов, провести вычислительные эксперименты и сравнить эффективность их работы в зависимости от различных входных данных и параметров.

Работа должна содержать следующие модули:

- последовательная реализация;
- параллельная реализация с помощью MPI;
- параллельная реализация с помощью TBB.

Входные данные:

- минимизируемая функция: $\phi(x)$;
- отрезок: $x \in Q = [a, b]$;
- заданная точность поиска: ϵ;
- максимальное количество испытаний: k_{max}.

Выходные данные:

- минимальное вычисленное значение функции: ϕ_k^*;
- координата этого значения: x_k^*.

2. Алгоритм глобального поиска для одномерных многоэкстремальных задач оптимизации

Задачей оптимизации будем называть задачу следующего вида:

Найти точную нижнюю грань $\phi^* = inf\{\phi(x) : x \in Q\}$ *и, если множество точек глобального минимума* $Q^* \equiv Arg\ min\{\phi(x) : x \in Q\}$ *не пусто, найти хотя бы одну точку* $x^* \in Q^*$.

Рассмотрим одномерную задачу минимизации функции на отрезке:

$$\phi(x) \to min, x \in Q = [a, b]$$

Вычислительная схема АГП:

Дадим детальное описание вычислительной схемы АГП (алгоритма глобального поиска), применяемого к решению сформулированной задачи, рассматривая при этом в качестве поисковой информации множество:

$$\omega = \omega_k = \{(x_i, z_i), 1 \le i \le k\}.(*)$$

Согласно алгоритму, два первых испытания проводятся на концах отрезка $[a, b]$, т.е. $x^1 = a, x^2 = b$, вычисляются значения функции $z^1 = \phi(a), z^2 = \phi(b)$, и количество k проведенных испытаний полагается равным 2.

Пусть проведено $k \ge 2$ испытаний и получена информация $(*)$. Для выбора точки x^{k+1} нового испытания необходимо выполнить следующие действия:

1. Перенумеровать нижним индексом (начиная с нулевого значения) точки $x_i, 1 \le i \le k$, из $(*)$ в порядке возврастания, т.е. $a = x_0 < x_1 < \dots < x_{k-1} = b$.
2. Полагая $z_i = \phi(x_i), 1 \le i \le k$, вычислить величину $M = \max\limits_{1 \le i \le k-1} |\frac{z_i - z_{i-1}}{x_i - x_{i-1}}|$ и положить

$$m = \begin{cases} rM, & M > 0 \\ 1, & M = 0 \end{cases}$$

   , где $r > 1$ является заданным параметром метода.
3. Для каждого интервала $(x_{i-1}, x_i), 1 \le i \le k - 1$ вычислить характеристику $R(i) = m(x_i - x_{i-1}) + \frac{(z_i - z_{i-1})^2}{/} m(x_i - x_{i-1}) - 2(z_i + z_{i-1})$.
4. Найти интервал (x_{t-1}, x_t), которому соответствует максимальная характеристика $R(t) = max\{R(i) : 1 \le i \le k - 1\}$ (в случае нескольких интервалов выбирается интервал с наименьшим номером t).
5. Провести новое испытание в точке $x^{k+1} = \frac{1}{2}(x_t + x_{t-1}) - \frac{z_t - z_{t-1}}{2m}$, вычислить значение $z^{k+1} = \phi(x^{k+1})$ и увеличить номер шага поиска на единицу: $k = k + 1$.

Правило остановки задается в форме:

$$H_k(\Phi, \omega_k) = \begin{cases} 0, & x_t - x_{t-1} \le \epsilon \text{ or } k \ge k_{max} \\ 1, & x_t - x_{t-1} > \epsilon \text{ or } k < k_{max} \end{cases}$$

, где $\epsilon > 0$ - заданная точность поиска (по координате), k_{max} – максимальное количество испытаний.

Наконец, в качестве оценки экстремума выбирается пара $e^k = (\phi_k^*, x_k^*)$, где ϕ_k^* - минимальное вычисленное значение функции, т.е. $\phi_k^* = \underset{1 \leq i \leq k}{min} \phi(x^i)$, а x_k^* - координата этого значения: $x_k^* = arg \underset{1 \leq i \leq k}{min} \phi(x^i)$

3. Параллельный алгоритм глобального поиска

Распараллеливание АГП можно производить различными способами:

1. разделить отрезок, на котором производится поиск глобального минимума, на подотрезки, внутри которых запустить последовательный алгоритм глобального поиска;

2. распараллелить вычисление внутренних характеристик последовательного алгоритма глобального поиска;

3. распараллеливание и модифицирование алгоритма глобального поиска, обеспечивая одновременное выполнение нескольких испытаний.

Реализация параллельного алгоритма с помощью функций библиотеки межпроцессного взаимодействия MPI:

1. способ:
   - вычислить новые границы отрезков, на которых будет производиться поиск глобального минимума;
   - создать структуру, содержащую в себе 2 поля типа double: первое – значение локального минимума y, а второе – значение локального минимума x;
   - в каждом процессе вызвать последовательную реализацию глобального поиска на соответствующем новом отрезке;
   - с помощью функции MPI_Reduce вычислить глобальный минимум по типу MPI_2DOUBLE_PRECISION с помощью операции MPI_MINLOC.
2. способ:
   - в каждом из доступных процессов найти локальный максимум оценки константы M для каждых двух соседних элементов;
   - с помощью функции MPI_Allreduce операцией MPI_MAX во все процессы коммуникатора положить максимум из всех локально вычисленных значений M;
   - создать структуру, содержащую в себе 2 поля: первое – типа double, локально на каком-либо процессе вычисленное значение характеристики R_i, а второе – типа int, номер отрезка, на котором производилось вычисление;
   - в каждом из доступных процессов найти локальный максимум оценки константы R для каждых двух соседних элементов;
   - с помощью функции MPI_Allreduce операцией MPI_MAXLOC по типу MPI_DOUBLE_INT во все процессы коммуникатора положить максимум из всех локально вычисленных значений R.
3. способ:
   - в каждом из доступных процессов найти локальный максимум оценки константы M для каждых двух соседних элементов;
   - с помощью функции MPI_Allreduce операцией MPI_MAX во все процессы коммуникатора положить максимум из всех локально вычисленных значений M;
   - создать структуру R, содержащую в себе 2 поля: первое – типа double, локально на каком-либо процессе вычисленное значение характеристики R_i, а второе – типа

int, номер отрезка, на котором производилось вычисление;

- создать локальный и глобальный вектора структур R, содержащие в себе оценку константы R_i для каждых двух соседних элементов;
- в каждом из доступных процессов заполнить локальный вектор структур R_i;
- с помощью функции MPI_Allgatherv собрать во все процессы коммуникатора вычисленные значения характеристики R в глобальный вектор и отсортировать его по убыванию;
- создать структуру *Point*, содержащую в себе 2 поля типа double: первое – значение x новой точки, а второе – значение y новой точки;
- в каждом из доступных процессов посчитать локальные значения x и y новых точек, на первых n отрезках с максимальной характеристикой R_i;
- с помощью функции MPI_Allgatherv собрать во все процессы коммуникатора вычисленные локальные значения в глобальный вектор новых точек;
- добавить новые точки в конец вектора исходных точек и отсортировать его.

Реализация параллельного алгоритма с помощью функций библиотеки Threading Building Blocks:

1. способ:
   - создать функтор *Segment_split*, который содержит следующие методы:
     - метод operator, выполняющий вычисления;
     - метод join, выполняющий редукцию;
     - метод result, возвращающий результат.
   - создать объект класса функтора и вызвать метод parallel_reduce, передав в него объект класса функтора;
   - вызвать у объекта класса функтора метод result, чтобы получить результаты.
2. способ. Способ параллельного вычисления внутренних характеристик не реализовывался, т.к. TBB не поддерживает распараллеливание цикла с редукцией и итерационным пространством, заданным итераторами.
3. способ:
   - для хранения точек будем использовать структуру данных concurrent_map<double, double> для потокобезопасного доступа;
   - для хранения значений характеристик R_i нак каждом отрезке будем использовать структуру данных concurrent_map<double, double> для потокобезопасного доступа;
   - написать лямбда-функцию для параллельного вычисления значений характеристик R_i нак каждом отрезке;
   - написать лямбда-функцию для параллельного вычисления и добавления новых точек на первых n отрезках с максимальной характеристикой R_i.

4. Программная реализация алгоритма глобального поиска

Описание структуры программы

Код программы содержится в следующих файлах:

- MPI - проект, содержащий в себе реализацию последовательного и параллельного алгоритмов глобального поиска с помощью библиотеки межпроцессного взаимодействия MPI;
  - global_search.h – включает объявление функций последовательного и параллельных алгоритмов глобального поиска;
  - global_search.cpp – включает реализацию функций последовательного и параллельных алгоритмов глобального поиска;
  - hansen_functions.h - включает объявления вспомогательных математических функций, а также массивы отрезков, на которых производится поиск, и значений глобального минимума;
  - hansen_functions.cpp - включает реализацию вспомогательных математических функций;
  - test.cpp – включает функцию main, тестирующую алгоритм глобального поиска.
- TBB - проект, содержащий в себе реализацию последовательного и параллельного алгоритмов глобального поиска с помощью библиотеки Threading Building Blocks;
  - Segment_split.h - включает объявление и реализацию функтора;
  - global_search.h – включает объявление функций последовательного и параллельных алгоритмов глобального поиска;
  - global_search.cpp – включает реализацию функций последовательного и параллельных алгоритмов глобального поиска;
  - hansen_functions.h - включает объявления вспомогательных математических функций, а также массивы отрезков, на которых производится поиск, и значений глобального минимума;
  - hansen_functions.cpp - включает реализацию вспомогательных математических функций;
  - test.cpp – включает функцию main, тестирующую алгоритм глобального поиска.

4.1 Описание последовательной версии

Описание функций

```
void sequential_global_search(double (*fcnPtr)(double), double a, double b, int
    kmax, double precision , double& xmin, double& ymin)
```

- назначние: осуществляет последовательный поиск глобального минимума;
- входные данные:

– fcnPtr – указатель на минимизируемую функцию, принимающую и возвращающую вещественное число;

– a – вещественное число, левая граница отрезка;

– b – вещественное число, правая граница отрезка;

– kmax – целое число, максимальное количество испытаний;

– precision – вещественное число, заданная точность поиска.

- выходные данные:
  – xmin – вещественное число, координата минимального значения функции;

  – ymin – вещественное число, минималное значение функции.

4.2 Описание MPI-версии

Описание функций

```
void segment_split(double (*fcnPtr)(double), double a, double b, int kmax,
    double precision, double& xmin, double& ymin)
```

- назначние: осуществляет параллельный поиск глобального минимума первым способом;

```
void parallel_operations(double (*fcnPtr)(double), double a, double b, int
    kmax, double precision, double& xmin, double& ymin)
```

- назначние: осуществляет параллельный поиск глобального минимума вторым способом;

```
void parallel_global_search(double (*fcnPtr)(double), double a, double b,
    int kmax, double precision, double& xmin, double& ymin)
```

- назначние: осуществляет параллельный поиск глобального минимума третьим способом;

- входные данные:
  – fcnPtr – указатель на минимизируемую функцию, принимающую и возвращающую вещественное число;

  – a – вещественное число, левая граница отрезка;

  – b – вещественное число, правая граница отрезка;

  – kmax – целое число, максимальное количество испытаний;

  – precision – вещественное число, заданная точность поиска.

- выходные данные:
  – xmin – вещественное число, координата минимального значения функции;

  – ymin – вещественное число, минималное значение функции.

4.3 Описание TBB-версии

Описание классов

```
class Segment_split
{
    double (*fcnPtr)(double);
    int kmax;
    double precision;
    double xmin, ymin;
public:
    explicit Segment_split(double (*_fcnPtr)(double), int _kmax, double
        _precision);
    Segment_split(const Segment_split& tmp, split);
    void operator()(const blocked_range<double>& r);
    void join(const Segment_split& tmp);
    void result(double& _xmin, double& _ymin);
};
```

- назначние: осуществляет параллельный поиск глобального минимума первым способом;

Описание функций

```
void parallel_global_search(double (*fcnPtr)(double), double a, double b,
   int kmax, double precision, double& xmin, double& ymin)
```

- назначние: осуществляет параллельный поиск глобального минимума третьим способом;
- входные данные:
  - fcnPtr – указатель на минимизируемую функцию, принимающую и возвращающую вещественное число;
  - a – вещественное число, левая граница отрезка;
  - b – вещественное число, правая граница отрезка;
  - kmax – целое число, максимальное количество испытаний;
  - precision – вещественное число, заданная точность поиска.
- выходные данные:
  - xmin – вещественное число, координата минимального значения функции;
  - ymin – вещественное число, минималное значение функции.

5. Результаты экспериментов

Вычислительные эксперименты для оценки эффективности последовательного и параллельного алгоритмов глобального поиска проводились на оборудовании со следующей аппаратной конфигурацией:

- Процессор: Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz, 4 ядра;
- Оперативная память: 8 ГБ;
- ОС: Microsoft Windows 10 Home 64-bit.

Вычисления производились с заданной точностью $\epsilon = 0.001$ и максимальным количеством шагов $k_{max} = 1000$ на 4 процессах (MPI) / потоках (TBB).

В таблице 1 представлены результаты, подтверждающие корректность работы реализованных алгоритмов. Первый столбец содержит порядковый номер тестируемой функции. Во втором столбце перечислены все координаты x, в которых тестируемая функция принимает своё минимальное значение. В последующих столбцах представлена координата x, найденная соответсвующим алгоритмом поиска глобального минимума.

№	Real x	Sequential	MPI 1st way	MPI 2nd way	MPI 3rd way	TBB 1st way	TBB 3rd way
1	10	9.99648	10.0012	9.99648	10.0018	10.0012	10.0018
2	5.14575	5.14863	5.14671	5.14863	5.14557	5.14671	5.14557
3	-0.49139, -6.77458, 5.79179	-6.78118	-0.48949	-6.78118	-0.49448	-0.48949	-0.49448
4	2.868	2.87484	2.8702	2.87484	2.86091	2.8702	2.86091
5	0.966086	0.9671	0.9671	0.9671	0.967406	0.9671	0.967406
6	0.679578	0.684807	0.688281	0.684807	0.684807	0.688281	0.684807
7	5.199776	5.19677	5.1967	5.19677	5.20088	5.1967	5.20088
8	-0.80032, -7.08351, 5.48286	5.4838	5.48359	5.4838	5.48518	5.48359	5.48518
9	17.0392	17.0381	17.0432	17.0381	17.0419	17.0432	17.0419
10	7.97867	7.98166	7.98245	7.98166	7.98143	7.98245	7.98143
11	2.09444, 4.18879	4.19226	4.18751	4.19226	4.18381	4.18751	4.18381
12	4.71239, 3.14159	4.71	3.14	4.71	4.71	3.14	4.71
13	0.70711	0.712251	0.712251	0.712251	0.71264	0.712251	0.71264
14	0.22488	0.219278	0.219278	0.219278	0.219278	0.219278	0.219278
15	2.41421	2.41259	2.42266	2.41259	2.41259	2.42266	2.41259
16	1.590721	1.59762	1.59243	1.59762	1.59244	1.59243	1.59244
17	-3, 3	3.0057	-2.99769	3.0057	2.9978	-2.99769	2.9978
18	2	1.99946	2.00019	1.99946	2.00041	2.00019	2.00041
19	5.8728656	5.87662	5.87848	5.87662	5.87849	5.87848	5.87849
20	1.195137	1.19489	1.19277	1.19489	1.19903	1.19277	1.19903

Таблица 1: Координата x глобального минимума

В таблице 2 представлено время работы в секундах каждого реализованного алгоритма поиска глобального минимума. Первый столбец содержит порядковый номер тестируемой функции.

№	Sequential	MPI 1st way	MPI 2nd way	MPI 3rd way	TBB 1st way	TBB 3rd way
1	1.98684	0.418129	0.0765019	0.746855	0.809128	9.52045
2	0.351726	0.025808	0.0106301	0.0674728	0.0779781	0.856633
3	1.98613	0.312087	0.101185	1.09666	0.969963	13.3693
4	0.0580657	0.0109359	0.0016188	0.0098205	0.0338061	0.205782
5	0.0314042	0.0025829	0.0011057	0.0057835	0.0256853	0.118677
6	2.12284	1.04983	0.0918233	0.762338	1.89965	11.9006
7	0.515828	0.0325687	0.0138859	0.0785869	0.0914991	1.12353
8	2.35078	0.28207	0.0889648	1.11156	0.994286	15.5373
9	2.1886	0.31177	0.0824311	0.959739	0.945486	13.7701
10	1.53283	0.184499	0.0317334	0.378982	0.488411	5.24417
11	0.582684	0.0387124	0.0199062	0.133934	0.246798	1.9169
12	0.434674	0.0321488	0.0133553	0.107945	0.155449	2.00538
13	0.0175562	0.0020435	0.000674	0.0026628	0.0202261	0.0521011
14	0.27692	0.01448	0.007001	0.0582406	0.0416089	0.926343
15	1.06724	0.171527	0.0230642	0.266156	0.352577	4.1833
16	0.507303	0.0561889	0.0124977	0.103419	0.251011	1.45336
17	0.69582	0.0723363	0.0200514	0.147168	0.341592	3.46478
18	0.437395	0.0753891	0.0103295	0.0895955	0.214612	1.54216
19	1.00528	0.0669877	0.019015	0.186115	0.163427	2.99287
20	2.06142	0.987243	0.103779	0.875112	2.34018	11.3986

Таблица 2: Время работы алгоитмов глобального поиска

По данным экспериментов видно, что значение координаты x глобального минимума, найденного последовательным алгоритмом, полностью совпадает с координатой x, найденной вторым способом распараллеливания с помощью MPI. Это происходит потому, что при одинаковой заданной точности и максимальном количестве шагов второй способ распараллеливания по характеристикам не подразумевает увеличения общего количества испытаний, как это происходит в 1-ом и 3-ем способе распараллеливания.

Также можно заметить, что возвращаемые значения глобального минимума в 1-ом и 3-ем способе, реализованных с помощью MPI, совпадают с возвращаемыми значениями глобального минимума соответствующих способов, реализованных с помощью TBB. Это подтверждает корректность распараллеливания, т.к. при одинаковой заданной точности, максимальном количестве шагов и способе распараллеливания общее количество испытаний должно быть неизменным для реализаций с использованием различных технологий.

Из таблицы 2 видно, что наиболее эффективно работает 2-ой способ распараллеливания по характеристикам с помощью MPI.

Заключение

Был изучен алгоритм глобального поиска для одномерных многоэкстремальных задач оптимизации. Также были разработаны последовательная и параллельная реализации данного алгоритма и протестированы на различных минимизируемых функциях с разными входными параметрами.

Основной задачей данной работы была реализация эффективной параллельной версии. Эта цель была успешно достигнута, что подтверждается результатами экспериментов, проведенных в ходе работы. Из результатов тестирования можно сделать вывод, что реализованные алгоритмы работают корректно.

В ходе дальнейшей работы планируется реализовать последовательный алгоритм глобального поиска для многомерных многоэкстремальных задач оптимизации.

Литература

1. Стронгин Р.Г., Гергель В.П., Гришагин В.А., Баркалов К.А. Параллельные вычисления в задачах глобальной оптимизации: Монография / Предисл.: В.А. Садовничий. – М.: Издательство Московского университета, 2013. – 280 с., илл.

2. Сергеев Я.Д., Квасов Д.Е. Краткое введение в теорию липшицевой глобальной оптимизации: Учебно-методическое пособие. – Нижний Новгород: Изд-во ННГУ, 2016. – 48с.

3. Документация по TBB [Электронный ресурс] // URL: https://software.intel.com/content/www/ru/ru/develop/articles/tbb_async_io

Приложение

MPI проект:

global_search.h

```
#ifndef __GLOBAL_SEARCH_H__
#define __GLOBAL_SEARCH_H__

void sequential_global_search(double (*fcnPtr)(double), double a, double b, int
    kmax, double precision, double& xmin, double& ymin);

void segment_split(double (*fcnPtr)(double), double a, double b, int kmax,
    double precision, double& xmin, double& ymin);

void parallel_operations(double (*fcnPtr)(double), double a, double b, int
    kmax, double precision, double& xmin, double& ymin);

void parallel_global_search(double (*fcnPtr)(double), double a, double b, int
    kmax, double precision, double& xmin, double& ymin);

#endif
```

global_search.cpp

```
#include <mpi.h>

#include <algorithm>
#include <cmath>
#include <iterator>
#include <limits>
#include <map>
#include <vector>

#include "global_search.h"

void sequential_global_search(double (*fcnPtr)(double), double a, double b, int
    kmax, double precision, double& xmin, double& ymin)
{
    if (a > b)
        throw "Incorrect bounds: a must be less than b";
    std::map<double, double> x = { {a, fcnPtr(a)}, {b, fcnPtr(b)} };
    int k = 2;
    bool prec = 1;
    while ((k < kmax) && prec) {
        double M = 0;
        for (auto it1 = x.begin(), it2 = ++x.begin(); it2 != x.end(); it1++,
            it2++) {
            M = fmax(M, abs((it2->second - it1->second) / (it2->first -
                it1->first)));
        }
        double r = 2;
        double m = r * M;
        if (M == 0)
            m = 1;
        double R = 0, i1 = 0, i2 = 0;
```

```
        for (auto it1 = x.begin(), it2 = ++x.begin(); it2 != x.end(); it1++,
            it2++) {
            double Ri = m * (it2->first - it1->first) + (it2->second -
                it1->second) * (it2->second - it1->second) /
                (m * (it2->first - it1->first)) - 2 * (it2->second -
                    it1->second);
            if (Ri > R) {
                R = Ri;
                i1 = it1->first;
                i2 = it2->first;
            }
        }
        double new_x = 0.5 * (i2 + i1) - (x[i2] - x[i1]) / (2 * m);
        x[new_x] = fcnPtr(new_x);
        k++;
        prec = i2 - i1 <= precision ? 0 : 1;
    }
    xmin = a;
    ymin = x[a];
    for (const auto& i : x) {
        if (ymin > i.second) {
            ymin = i.second;
            xmin = i.first;
        }
    }
}

void segment_split(double (*fcnPtr)(double), double a, double b, int kmax,
    double precision, double& xmin, double& ymin)
{
    if (a > b)
        throw "Incorrect␣bounds:␣a␣must␣be␣less␣than␣b";

    int size, rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    double segment = static_cast<double>(b - a) / size;
    double local_a = a + segment * rank;
    double local_b = a + segment * (rank + 1);

    struct {
        double ymin = std::numeric_limits<double>::max();
        double xmin = std::numeric_limits<double>::max();
    }local_min, global_min;

    sequential_global_search(fcnPtr, local_a, local_b, kmax, precision,
        local_min.xmin, local_min.ymin);

    MPI_Reduce(&local_min, &global_min, 1, MPI_2DOUBLE_PRECISION, MPI_MINLOC,
        0, MPI_COMM_WORLD);

    xmin = global_min.xmin;
    ymin = global_min.ymin;
}

void parallel_operations(double (*fcnPtr)(double), double a, double b, int
    kmax, double precision, double& xmin, double& ymin)
{
    if (a > b)
```

```cpp
        throw "Incorrect bounds: a must be less than b";

    int size, rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    std::vector<std::pair<double, double>> x = { {a, fcnPtr(a)}, {b, fcnPtr(b)}
        };
    int k = 2;
    bool prec = 1;
    while ((k < kmax) && prec) {
        double local_M = 0, global_M;
        for (int i1 = rank, i2 = rank + 1; i2 < x.size(); i1 += size, i2 +=
            size) {
            local_M = fmax(local_M, abs((x[i2].second - x[i1].second) /
                (x[i2].first - x[i1].first)));
        }
        MPI_Allreduce(&local_M, &global_M, 1, MPI_DOUBLE, MPI_MAX,
            MPI_COMM_WORLD);
        double r = 2;
        double m = r * global_M;
        if (global_M == 0)
            m = 1;

        struct {
            double val = 0;
            int pos = 0;
        }local_R, global_R;
        for (int i1 = rank, i2 = rank + 1; i2 < x.size(); i1 += size, i2 +=
            size) {
            double Ri = m * (x[i2].first - x[i1].first) + (x[i2].second -
                x[i1].second) * (x[i2].second - x[i1].second) /
                (m * (x[i2].first - x[i1].first)) - 2 * (x[i2].second -
                    x[i1].second);
            if (Ri > local_R.val) {
                local_R.val = Ri;
                local_R.pos = i1;
            }
        }
        MPI_Allreduce(&local_R, &global_R, 1, MPI_DOUBLE_INT, MPI_MAXLOC,
            MPI_COMM_WORLD);

        double new_x = 0.5 * (x[global_R.pos + 1].first +
            x[global_R.pos].first) - (x[global_R.pos + 1].second -
            x[global_R.pos].second) / (2 * m);
        x.emplace(x.begin() + global_R.pos + 1, std::pair<double,
            double>(new_x, fcnPtr(new_x)));
        k++;
        prec = x[global_R.pos + 1].first - x[global_R.pos].first <= precision ?
            0 : 1;
    }
    xmin = x[0].first;
    ymin = x[0].second;
    for (const auto& i : x) {
        if (ymin > i.second) {
            ymin = i.second;
            xmin = i.first;
        }
    }
}
```

```cpp
void parallel_global_search(double (*fcnPtr)(double), double a, double b, int
    kmax, double precision, double& xmin, double& ymin)
{
    if (a > b)
        throw "Incorrect bounds: a must be less than b";

    int size, rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    std::vector<std::pair<double, double>> x = { {a, fcnPtr(a)}, {b, fcnPtr(b)}
        };
    int k = 2;
    int prec = 1;
    while ((k < kmax) && prec) {

        double local_M = 0, global_M = 0;
        for (int i1 = rank, i2 = rank + 1; i2 < x.size(); i1 += size, i2 +=
            size) {
            local_M = fmax(local_M, abs((x[i2].second - x[i1].second) /
                (x[i2].first - x[i1].first)));
        }
        MPI_Allreduce(&local_M, &global_M, 1, MPI_DOUBLE, MPI_MAX,
            MPI_COMM_WORLD);
        double r = 2;
        double m = r * global_M;
        if (global_M == 0)
            m = 1;

        struct R {
            double val = 0;
            int pos = 0;
        };
        std::vector<R> local_Ri;
        for (int i1 = rank, i2 = rank + 1; i2 < x.size(); i1 += size, i2 +=
            size) {
            local_Ri.push_back(R{ m * (x[i2].first - x[i1].first) +
                (x[i2].second - x[i1].second) * (x[i2].second - x[i1].second) /
                 (m * (x[i2].first - x[i1].first)) - 2 * (x[i2].second -
                    x[i1].second), i1 });
        }

        std::vector<R> global_Ri(x.size() - 1);
        int part = (x.size() - 1) / size;
        std::vector<int> scounts(size, part);
        for (int i = 0; i < (x.size() - 1) % size; i ++)
            scounts[i]++;
        std::vector<int> displs(size, 0);
        for (int i = 1; i < size; i++)
            displs[i] = displs[i - 1] + scounts[i - 1];
        MPI_Allgatherv(local_Ri.data(), local_Ri.size(), MPI_DOUBLE_INT,
            global_Ri.data(), scounts.data(), displs.data(), MPI_DOUBLE_INT,
            MPI_COMM_WORLD);
        int n = global_Ri.size() < size ? global_Ri.size() : size;
        std::sort(global_Ri.begin(), global_Ri.end(), [](R a, R b)->bool
            {return a.val > b.val; });

        struct Point {
            double x;
```

```
            double y;
        } local_point;
        int local_prec = prec;
        for (int i = rank; i < n; i += size) {
            local_point.x = 0.5 * (x[global_Ri[i].pos + 1].first +
                x[global_Ri[i].pos].first) - (x[global_Ri[i].pos + 1].second -
                x[global_Ri[i].pos].second) / (2 * m);
            local_point.y = fcnPtr(local_point.x);
            if (local_prec)
                local_prec = x[global_Ri[i].pos + 1].first -
                    x[global_Ri[i].pos].first <= precision ? 0 : 1;
        }
        MPI_Allreduce(&local_prec, &prec, 1, MPI_INT, MPI_LAND, MPI_COMM_WORLD);

        std::fill(scounts.begin(), scounts.end(), n / size);
        for (int i = 0; i < n % size; i++)
            scounts[i]++;
        for (int i = 1; i < size; i++)
            displs[i] = displs[i - 1] + scounts[i - 1];
        std::vector<Point> global_point(n);
        MPI_Allgatherv(&local_point, scounts[rank], MPI_2DOUBLE_PRECISION,
            global_point.data(), scounts.data(), displs.data(),
            MPI_2DOUBLE_PRECISION, MPI_COMM_WORLD);

        for (int i = 0; i < n; i++)
            x.push_back(std::pair<double, double>(global_point[i].x,
                global_point[i].y));
        std::sort(x.begin(), x.end(), [](std::pair<double, double> a,
            std::pair<double, double> b)->bool {return a.first < b.first; });
        k++;
    }
    xmin = x[0].first;
    ymin = x[0].second;
    for (const auto& i : x) {
        if (ymin > i.second) {
            ymin = i.second;
            xmin = i.first;
        }
    }
}
```

hansen_functions.h

```
#ifndef __HANSEN_FUNCTIONS_H__
#define __HANSEN_FUNCTIONS_H__

#include <cmath>
#include <vector>

static double intervals[][2] = { {-1.5, 11}, {2.7, 7.5}, {-10.0, 10.0}, {1.9,
    3.9}, {0.0, 1.2},
{-10.0, 10.0}, {2.7, 7.5}, {-10.0, 10.0}, {3.1, 20.4}, {0.0, 10.0},
{-1.57, 6.28}, {0.0, 6.28}, {0.001, 0.99}, {0.0, 4.0}, {-5.0, 5.0},
{-3.0, 3.0}, {-4.0, 4.0}, {0.0, 6.0}, {0.0, 6.5}, {-10.0, 10.0} };

double hfunc1(double x);
double hfunc2(double x);
double hfunc3(double x);
double hfunc4(double x);
```

```
double hfunc5(double x);
double hfunc6(double x);
double hfunc7(double x);
double hfunc8(double x);
double hfunc9(double x);
double hfunc10(double x);
double hfunc11(double x);
double hfunc12(double x);
double hfunc13(double x);
double hfunc14(double x);
double hfunc15(double x);
double hfunc16(double x);
double hfunc17(double x);
double hfunc18(double x);
double hfunc19(double x);
double hfunc20(double x);
double hpfunc1(double x);
double hpfunc2(double x);
double hpfunc3(double x);
double hpfunc4(double x);
double hpfunc5(double x);
double hpfunc6(double x);
double hpfunc7(double x);
double hpfunc8(double x);
double hpfunc9(double x);
double hpfunc10(double x);
double hpfunc11(double x);
double hpfunc12(double x);
double hpfunc13(double x);
double hpfunc14(double x);
double hpfunc15(double x);
double hpfunc16(double x);
double hpfunc17(double x);
double hpfunc18(double x);
double hpfunc19(double x);
double hpfunc20(double x);

static double(*pfn[])(double x) = { hfunc1, hfunc2, hfunc3, hfunc4, hfunc5,
    hfunc6, hfunc7, hfunc8, hfunc9, hfunc10,
                                         hfunc11, hfunc12, hfunc13, hfunc14,
                                            hfunc15, hfunc16, hfunc17, hfunc18,
                                            hfunc19, hfunc20,
                                         hpfunc1, hpfunc2, hpfunc3, hpfunc4,
                                            hpfunc5, hpfunc6, hpfunc7, hpfunc8,
                                            hpfunc9, hpfunc10,
                                         hpfunc11, hpfunc12, hpfunc13, hpfunc14,
                                            hpfunc15, hpfunc16, hpfunc17, hpfunc18,
                                            hpfunc19, hpfunc20 };

static std::vector<std::vector<double> > res = { {10},{5.14575},{-0.49139,
    -6.77458, 5.79179},{2.868},{0.966086},
{0.679578},{5.199776},{-0.80032, -7.08351, 5.48286},{17.0392},{7.97867},
{2.09444, 4.18879},{4.71239, 3.14159},{0.70711},{0.22488},{2.41421},
{1.590721},{-3, 3},{2},{5.8728656},{1.195137} };


#endif
```

hansen_functions.cpp

```c
#include "hansen_functions.h"

double hfunc1(double x) {
    return pow(x, 6) / 6.0 - 52.0 / 25.0 * pow(x, 5) + 39.0 / 80.0 * pow(x, 4) +
        71.0 / 10.0 * pow(x, 3) - 79.0 / 20.0 * pow(x, 2) - x + 0.1;
}

double hfunc2(double x) {
    return sin(x) + sin(10 * x / 3);
}

double hfunc3(double x) {
    double res = 0;
    for (int i = 1; i < 6; i++)
        res += i * sin((i + 1) * x + i);
    return -res;
}

double hfunc4(double x) {
    return (-16 * x * x + 24 * x - 5) * exp(-x);
}

double hfunc5(double x) {
    return -(-3 * x + 1.4) * sin(18 * x);
}

double hfunc6(double x) {
    return -(x + sin(x)) * exp(-x * x);
}

double hfunc7(double x) {
    return sin(x) + sin(10 * x / 3) + log(x) - 0.84 * x + 3;
}

double hfunc8(double x) {
    double res = 0;
    for (int i = 1; i < 6; i++)
        res += i * cos((i + 1) * x + i);
    return -res;
}

double hfunc9(double x) {
    return sin(x) + sin(2.0 / 3.0 * x);
}

double hfunc10(double x) {
    return -x * sin(x);
}

double hfunc11(double x) {
    return 2 * cos(x) + cos(2 * x);
}

double hfunc12(double x) {
    return pow(sin(x), 3) + pow(cos(x), 3);
}

double hfunc13(double x) {
    double sgn = 0.0;
```

```
    if (x * x - 1 < 0)
        sgn = -1.0;
    else
        sgn = 1.0;
    return -pow(x * x, 1.0 / 3.0) + sgn * pow(sgn * (x * x - 1.0), 1.0 / 3.0);
}

double hfunc14(double x) {
    return -exp(-x) * sin(2 * acos(-1.0) * x);
}

double hfunc15(double x) {
    return (x * x - 5 * x + 6) / (x * x + 1);
}

double hfunc16(double x) {
    return 2 * (x - 3) * (x - 3) + exp(x * x / 2);
}

double hfunc17(double x) {
    return pow(x, 6) - 15 * pow(x, 4) + 27 * x * x + 250;
}

double hfunc18(double x) {
    if (x <= 3)
        return (x - 2) * (x - 2);
    else
        return 2 * log(x - 2) + 1;
}

double hfunc19(double x) {
    return -x + sin(3 * x) - 1;
}

double hfunc20(double x) {
    return -(x - sin(x)) * exp(-x * x);
}

double hpfunc1(double x) {
    return pow(x, 5) - 10.4 * pow(x, 4) + 1.95 * pow(x, 3) + 21.3 * x * x -
        7.9 * x - 1.0;
}

double hpfunc2(double x) {
    return cos(x) + 10.0 * cos(10.0 * x / 3.0) / 3.0;
}

double hpfunc3(double x) {
    double res = 0.0;
    for (int i = 1; i < 6; i++)
        res += i * (i + 1) * cos((i + 1) * x + i);
    return -res;
}

double hpfunc4(double x) {
    return (16.0 * x * x - 56.0 * x + 29.0) * exp(-x);
}

double hpfunc5(double x) {
    return 3.0 * sin(18.0 * x) - 18.0 * (-3.0 * x + 1.4) * cos(18.0 * x);
```

```
}

double hpfunc6(double x) {
    return (2.0 * x * (x + sin(x)) - cos(x) - 1) * exp(-x * x);
}

double hpfunc7(double x) {
    return cos(x) + 10.0 * cos(10.0 * x / 3.0) / 3.0 + 1 / x - 0.84;
}

double hpfunc8(double x) {
    double res = 0.0;
    for (int i = 1; i < 6; i++)
        res += i * (i + 1) * sin((i + 1) * x + i);
    return res;
}

double hpfunc9(double x) {
    return cos(x) + 2.0 * cos(2.0 * x / 3.0) / 3.0;
}

double hpfunc10(double x) {
    return -sin(x) - x * cos(x);
}

double hpfunc11(double x) {
    return -2.0 * (sin(x) + sin(2.0 * x));
}

double hpfunc12(double x) {
    return 3.0 * cos(x) * sin(x) * (sin(x) - cos(x));
}

double hpfunc13(double x) {
    double st = (1.0 / 3.0);
    if (x == 0.0)
        return 0.0;
    return (2.0 * x / pow((x * x - 1) * (x * x - 1), st) - 2.0 * pow(x, -st)) /
       3.0;
}

double hpfunc14(double x) {
    double pi = acos(-1.0);
    return exp(-x) * (sin(2.0 * pi * x) - 2.0 * pi * cos(2 * pi * x));
}

double hpfunc15(double x) {
    return (2.0 * x * (-x * x + 5.0 * x - 6.0) - (x * x + 1.0) * (5.0 - 2.0 *
       x))
        / ((x * x + 1.0) * (x * x + 1.0));
}

double hpfunc16(double x) {
    return 4.0 * (x - 3.0) + x * exp(x * x / 2.0);
}

double hpfunc17(double x) {
    return 6.0 * pow(x, 5) - 60.0 * x * x * x + 54.0 * x;
}
```

```cpp
double hpfunc18(double x) {
    if (x <= 3)
        return 2.0 * x - 4.0;
    else
        return 2.0 / (x - 2.0);
}

double hpfunc19(double x) {
    return 3.0 * cos(3.0 * x) - 1.0;
}

double hpfunc20(double x) {
    return exp(-x * x) * (2.0 * x * (x - sin(x)) - 1.0 + cos(x));
}
```

test.cpp

```cpp
#include <mpi.h>

#include <iostream>
#include <ctime>

#include "hansen_functions.h"
#include "global_search.h"

using namespace std;

int main(int argc, char* argv[])
{
    int kmax = 1000;
    double precision = 0.01;
    double xmin_par_split, ymin_par_split;
    double xmin_par_oper, ymin_par_oper;
    double xmin_par, ymin_par;
    double xmin_seq, ymin_seq;
    double a, b;
    int grainSize;

    MPI_Init(&argc, &argv);
    int size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    for (size_t i = 0; i < 20; i++) {
        xmin_par_split = numeric_limits<double>::max();
        ymin_par_split = numeric_limits<double>::max();
        xmin_par_oper = numeric_limits<double>::max();
        ymin_par_oper = numeric_limits<double>::max();
        xmin_par = numeric_limits<double>::max();
        ymin_par = numeric_limits<double>::max();
        xmin_seq = numeric_limits<double>::max();
        ymin_seq = numeric_limits<double>::max();
        a = intervals[i][0];
        b = intervals[i][1];

        try {
            double t1_par_split = MPI_Wtime();
            segment_split(pfn[i], a, b, kmax, precision, xmin_par_split,
                ymin_par_split);
```

```
            double t2_par_split = MPI_Wtime();

            double t1_par_oper = MPI_Wtime();
            parallel_operations(pfn[i], a, b, kmax, precision, xmin_par_oper,
                ymin_par_oper);
            double t2_par_oper = MPI_Wtime();

            double t1_par = MPI_Wtime();
            parallel_global_search(pfn[i], a, b, kmax, precision, xmin_par,
                ymin_par);
            double t2_par = MPI_Wtime();


            if (rank == 0) {
                double t1_seq = MPI_Wtime();
                sequential_global_search(pfn[i], a, b, kmax, precision,
                    xmin_seq, ymin_seq);
                double t2_seq = MPI_Wtime();

                cout << "Global min hfunc " << i + 1 << " - " << res[i][0] <<
                    endl;
                cout << "Result parallel split: (" << xmin_par_split << ", " <<
                    ymin_par_split << ")     Time parallel: " << t2_par_split -
                    t1_par_split << endl;
                cout << "Result parallel operations: (" << xmin_par_oper << ", 
                    " << ymin_par_oper << ")     Time parallel: " << t2_par_oper
                    - t1_par_oper << endl;
                cout << "Result parallel: (" << xmin_par << ", " << ymin_par <<
                    ")     Time parallel: " << t2_par - t1_par << endl;
                cout << "Result sequential: (" << xmin_seq << ", " << ymin_seq
                    << ")   Time sequential: " << t2_seq - t1_seq << endl <<
                    endl;
            }
        }
        catch (const char* message) {
            cout << message;
        }

    }
    MPI_Finalize();
}
```

TBB проект:

global_search.h

```
#ifndef __GLOBAL_SEARCH_H__
#define __GLOBAL_SEARCH_H__

void sequential_global_search(double (*fcnPtr)(double), double a, double b, int
    kmax, double precision, double& xmin, double& ymin);

void parallel_global_search(double (*fcnPtr)(double), double a, double b, int
    kmax, double precision, double& xmin, double& ymin);

#endif
```

global_search.cpp

```cpp
#include "tbb/tbb.h"
#define TBB_PREVIEW_CONCURRENT_ORDERED_CONTAINERS 1
#include "tbb/concurrent_map.h"

#include <map>
#include <cmath>
#include <iterator>

#include "global_search.h"

using namespace tbb;

void sequential_global_search(double (*fcnPtr)(double), double a, double b, int
    kmax, double precision, double& xmin, double& ymin)
{
    if (a > b)
        throw "Incorrect bounds: a must be less than b";
    std::map<double, double> x = { {a, fcnPtr(a)}, {b, fcnPtr(b)} };
    int k = 2;
    bool prec = 1;
    while ((k < kmax) && prec) {
        double M = 0;
        for (auto it1 = x.begin(), it2 = ++x.begin(); it2 != x.end(); it1++,
            it2++) {
            M = fmax(M, abs((it2->second - it1->second) / (it2->first -
                it1->first)));
        }
        double r = 2;
        double m = r * M;
        if (M == 0)
            m = 1;
        double R = 0, i1 = 0, i2 = 0;
        for (auto it1 = x.begin(), it2 = ++x.begin(); it2 != x.end(); it1++,
            it2++) {
            double Ri = m * (it2->first - it1->first) + (it2->second -
                it1->second) * (it2->second - it1->second) /
                 (m * (it2->first - it1->first)) - 2 * (it2->second -
                    it1->second);
            if (Ri > R) {
                R = Ri;
                i1 = it1->first;
                i2 = it2->first;
            }
        }
        double new_x = 0.5 * (i2 + i1) - (x[i2] - x[i1]) / (2 * m);
        x[new_x] = fcnPtr(new_x);
        k++;
        prec = i2 - i1 <= precision ? 0 : 1;
    }
    xmin = a;
    ymin = x[a];
    for (const auto& i : x) {
        if (ymin > i.second) {
            ymin = i.second;
            xmin = i.first;
        }
    }
}
```

```
void parallel_global_search(double (*fcnPtr)(double), double a, double b, int
    kmax, double precision, double& xmin, double& ymin)
{
    if (a > b)
        throw "Incorrect␣bounds:␣a␣must␣be␣less␣than␣b";
    concurrent_map<double, double> x = { {a, fcnPtr(a)}, {b, fcnPtr(b)} };
    int k = 2;
    bool prec = 1;
    while ((k < kmax) && prec) {
        double M = 0;
        for (auto it1 = x.begin(), it2 = ++x.begin(); it2 != x.end(); it1++,
            it2++) {
            M = fmax(M, abs((it2->second - it1->second) / (it2->first -
                it1->first)));
        }
        double r = 2;
        double m = r * M;
        if (M == 0)
            m = 1;
        concurrent_map<double, std::pair<double, double>, std::greater<double>>
            R;
        int grainSize = x.size() < 4 ? 1 : x.size() / 4;
        parallel_for(blocked_range<size_t>(0, x.size(), grainSize),
            [&](const blocked_range<size_t>& r) {
                for (auto it1 = x.begin(), it2 = ++x.begin(); it2 != x.end();
                    it1++, it2++) {
                    double Ri = m * (it2->first - it1->first) + (it2->second -
                        it1->second) * (it2->second - it1->second) /
                        (m * (it2->first - it1->first)) - 2 * (it2->second -
                            it1->second);
                    R[Ri] = std::pair<double, double>(it1->first, it2->first);
                }
            });

        int size = R.size() < 4 ? R.size() : 4;
        parallel_for(blocked_range<size_t>(0, size, 1),
            [&](const blocked_range<size_t>& r) {
                auto begin = R.begin();
                auto end = R.begin();
                std::advance(end, size);
                for (auto it = begin; it != end; it++) {
                    prec = it->second.second - it->second.first <= precision ?
                        0 : 1;
                    double new_x = 0.5 * (it->second.second + it->second.first)
                        - (x[it->second.second] - x[it->second.first]) / (2 * m);
                    x[new_x] = fcnPtr(new_x);
                }
            });

        k++;
    }
    xmin = a;
    ymin = x[a];
    for (const auto& i : x) {
        if (ymin > i.second) {
            ymin = i.second;
            xmin = i.first;
        }
    }
};
```

Segment_split.h

```
#ifndef __SEGMENT_SPLIT_H__
#define __SEGMENT_SPLIT_H__
#define NOMINMAX

#include <tbb/tbb.h>

#include <limits>

#include "global_search.h"

using namespace std;
using namespace tbb;

class Segment_split
{
    double (*fcnPtr)(double);
    int kmax;
    double precision;
    double xmin, ymin;
public:
    explicit Segment_split(double (*_fcnPtr)(double), int _kmax, double
        _precision) :
        fcnPtr(_fcnPtr), kmax(_kmax), precision(_precision),
            xmin(numeric_limits<double>::max()),
            ymin(numeric_limits<double>::max()) {};

    Segment_split(const Segment_split& tmp, split) :
        fcnPtr(tmp.fcnPtr), kmax(tmp.kmax), precision(tmp.precision),
            xmin(numeric_limits<double>::max()),
            ymin(numeric_limits<double>::max()) {};

    void operator()(const blocked_range<double>& r) {
        double begin = r.begin(), end = r.end();
        sequential_global_search(fcnPtr, begin, end, kmax, precision, xmin,
            ymin);
    };

    void join(const Segment_split& tmp) {
        if (ymin > tmp.ymin) {
            ymin = tmp.ymin;
            xmin = tmp.xmin;
        }
    };

    void result(double& _xmin, double& _ymin) {
        _xmin = xmin;
        _ymin = ymin;
    };
};

#endif
```

hansen_functions.h

```
#ifndef __HANSEN_FUNCTIONS_H__
#define __HANSEN_FUNCTIONS_H__
```

```
#include <cmath>
#include <vector>

static double intervals[][2] = { {-1.5, 11}, {2.7, 7.5}, {-10.0, 10.0}, {1.9,
    3.9}, {0.0, 1.2},
{-10.0, 10.0}, {2.7, 7.5}, {-10.0, 10.0}, {3.1, 20.4}, {0.0, 10.0},
{-1.57, 6.28}, {0.0, 6.28}, {0.001, 0.99}, {0.0, 4.0}, {-5.0, 5.0},
{-3.0, 3.0}, {-4.0, 4.0}, {0.0, 6.0}, {0.0, 6.5}, {-10.0, 10.0} };

double hfunc1(double x);
double hfunc2(double x);
double hfunc3(double x);
double hfunc4(double x);
double hfunc5(double x);
double hfunc6(double x);
double hfunc7(double x);
double hfunc8(double x);
double hfunc9(double x);
double hfunc10(double x);
double hfunc11(double x);
double hfunc12(double x);
double hfunc13(double x);
double hfunc14(double x);
double hfunc15(double x);
double hfunc16(double x);
double hfunc17(double x);
double hfunc18(double x);
double hfunc19(double x);
double hfunc20(double x);
double hpfunc1(double x);
double hpfunc2(double x);
double hpfunc3(double x);
double hpfunc4(double x);
double hpfunc5(double x);
double hpfunc6(double x);
double hpfunc7(double x);
double hpfunc8(double x);
double hpfunc9(double x);
double hpfunc10(double x);
double hpfunc11(double x);
double hpfunc12(double x);
double hpfunc13(double x);
double hpfunc14(double x);
double hpfunc15(double x);
double hpfunc16(double x);
double hpfunc17(double x);
double hpfunc18(double x);
double hpfunc19(double x);
double hpfunc20(double x);
void matrix_mult();

static double(*pfn[])(double x) = { hfunc1, hfunc2, hfunc3, hfunc4, hfunc5,
    hfunc6, hfunc7, hfunc8, hfunc9, hfunc10,
                                     hfunc11, hfunc12, hfunc13, hfunc14,
                                         hfunc15, hfunc16, hfunc17, hfunc18,
                                         hfunc19, hfunc20,
                                     hpfunc1, hpfunc2, hpfunc3, hpfunc4,
                                         hpfunc5, hpfunc6, hpfunc7, hpfunc8,
                                         hpfunc9, hpfunc10,
```

```
                                     hpfunc11, hpfunc12, hpfunc13, hpfunc14,
                                        hpfunc15, hpfunc16, hpfunc17, hpfunc18,
                                        hpfunc19, hpfunc20};

static std::vector<std::vector<double> > res = { {10},{5.14575},{-0.49139,
    -6.77458, 5.79179},{2.868},{0.966086},
{0.679578},{5.199776},{-0.80032, -7.08351, 5.48286},{17.0392},{7.97867},
{2.09444, 4.18879},{4.71239, 3.14159},{0.70711},{0.22488},{2.41421},
{1.590721},{-3, 3},{2},{5.8728656},{1.195137} };


#endif
```

hansen_functions.cpp

```
#include "hansen_functions.h"

double hfunc1(double x) {
    return pow(x, 6) / 6.0 - 52.0 / 25.0 * pow(x, 5) + 39.0 / 80.0 * pow(x, 4) +
        71.0 / 10.0 * pow(x, 3) - 79.0 / 20.0 * pow(x, 2) - x + 0.1;
}

double hfunc2(double x) {
    return sin(x) + sin(10 * x / 3);
}

double hfunc3(double x) {
    double res = 0;
    for (int i = 1; i < 6; i++)
        res += i * sin((i + 1) * x + i);
    return -res;
}

double hfunc4(double x) {
    return (-16 * x * x + 24 * x - 5) * exp(-x);
}

double hfunc5(double x) {
    return -(-3 * x + 1.4) * sin(18 * x);
}

double hfunc6(double x) {
    return -(x + sin(x)) * exp(-x * x);
}

double hfunc7(double x) {
    return sin(x) + sin(10 * x / 3) + log(x) - 0.84 * x + 3;
}

double hfunc8(double x) {
    double res = 0;
    for (int i = 1; i < 6; i++)
        res += i * cos((i + 1) * x + i);
    return -res;
}

double hfunc9(double x) {
    return sin(x) + sin(2.0 / 3.0 * x);
}
```

```c
double hfunc10(double x) {
    return -x * sin(x);
}

double hfunc11(double x) {
    return 2 * cos(x) + cos(2 * x);
}

double hfunc12(double x) {
    return pow(sin(x), 3) + pow(cos(x), 3);
}

double hfunc13(double x) {
    double sgn = 0.0;
    if (x * x - 1 < 0)
        sgn = -1.0;
    else
        sgn = 1.0;
    return -pow(x * x, 1.0 / 3.0) + sgn * pow(sgn * (x * x - 1.0), 1.0 / 3.0);
}

double hfunc14(double x) {
    return -exp(-x) * sin(2 * acos(-1.0) * x);
}

double hfunc15(double x) {
    return (x * x - 5 * x + 6) / (x * x + 1);
}

double hfunc16(double x) {
    return 2 * (x - 3) * (x - 3) + exp(x * x / 2);
}

double hfunc17(double x) {
    return pow(x, 6) - 15 * pow(x, 4) + 27 * x * x + 250;
}

double hfunc18(double x) {
    if (x <= 3)
        return (x - 2) * (x - 2);
    else
        return 2 * log(x - 2) + 1;
}

double hfunc19(double x) {
    return -x + sin(3 * x) - 1;
}

double hfunc20(double x) {
    return -(x - sin(x)) * exp(-x * x);
}

double hpfunc1(double x) {
    return pow(x, 5) - 10.4 * pow(x, 4) + 1.95 * pow(x, 3) + 21.3 * x * x -
        7.9 * x - 1.0;
}

double hpfunc2(double x) {
    return cos(x) + 10.0 * cos(10.0 * x / 3.0) / 3.0;
```

```
}

double hpfunc3(double x) {
    double res = 0.0;
    for (int i = 1; i < 6; i++)
        res += i * (i + 1) * cos((i + 1) * x + i);
    return -res;
}

double hpfunc4(double x) {
    return (16.0 * x * x - 56.0 * x + 29.0) * exp(-x);
}

double hpfunc5(double x) {
    return 3.0 * sin(18.0 * x) - 18.0 * (-3.0 * x + 1.4) * cos(18.0 * x);
}

double hpfunc6(double x) {
    return (2.0 * x * (x + sin(x)) - cos(x) - 1) * exp(-x * x);
}

double hpfunc7(double x) {
    return cos(x) + 10.0 * cos(10.0 * x / 3.0) / 3.0 + 1 / x - 0.84;
}

double hpfunc8(double x) {
    double res = 0.0;
    for (int i = 1; i < 6; i++)
        res += i * (i + 1) * sin((i + 1) * x + i);
    return res;
}

double hpfunc9(double x) {
    return cos(x) + 2.0 * cos(2.0 * x / 3.0) / 3.0;
}

double hpfunc10(double x) {
    return -sin(x) - x * cos(x);
}

double hpfunc11(double x) {
    return -2.0 * (sin(x) + sin(2.0 * x));
}

double hpfunc12(double x) {
    return 3.0 * cos(x) * sin(x) * (sin(x) - cos(x));
}

double hpfunc13(double x) {
    double st = (1.0 / 3.0);
    if (x == 0.0)
        return 0.0;
    return (2.0 * x / pow((x * x - 1) * (x * x - 1), st) - 2.0 * pow(x, -st)) /
       3.0;
}

double hpfunc14(double x) {
    double pi = acos(-1.0);
    return exp(-x) * (sin(2.0 * pi * x) - 2.0 * pi * cos(2 * pi * x));
}
```

```
double hpfunc15(double x) {
    return (2.0 * x * (-x * x + 5.0 * x - 6.0) - (x * x + 1.0) * (5.0 - 2.0 *
        x))
        / ((x * x + 1.0) * (x * x + 1.0));
}

double hpfunc16(double x) {
    return 4.0 * (x - 3.0) + x * exp(x * x / 2.0);
}

double hpfunc17(double x) {
    return 6.0 * pow(x, 5) - 60.0 * x * x * x + 54.0 * x;
}

double hpfunc18(double x) {
    if (x <= 3)
        return 2.0 * x - 4.0;
    else
        return 2.0 / (x - 2.0);
}

double hpfunc19(double x) {
    return 3.0 * cos(3.0 * x) - 1.0;
}

double hpfunc20(double x) {
    return exp(-x * x) * (2.0 * x * (x - sin(x)) - 1.0 + cos(x));
}
```

test.cpp

```
#include <iostream>
#include <ctime>

#include "Segment_split.h"
#include "hansen_functions.h"

using namespace std;
using namespace tbb;

int main()
{
    int kmax = 1000;
    double precision = 0.01;
    double xmin_par_split, ymin_par_split;
    double xmin_par, ymin_par;
    double xmin_seq, ymin_seq;
    double a, b;
    int grainSize;
    task_scheduler_init init(4);

    for (size_t i = 0; i < 20; i++) {
        xmin_par_split = numeric_limits<double>::max();
        ymin_par_split = numeric_limits<double>::max();
        xmin_par = numeric_limits<double>::max();
        ymin_par = numeric_limits<double>::max();
        xmin_seq = numeric_limits<double>::max();
        ymin_seq = numeric_limits<double>::max();
```

```
        a = intervals[i][0];
        b = intervals[i][1];
        grainSize = (b - a) / 4;
        if (grainSize <= 0)
            grainSize = 1;

        try {
            tick_count t1_par_split = tick_count::now();
            Segment_split s(pfn[i], kmax, precision);
            parallel_reduce(blocked_range<double>(a, b, grainSize), s);
            tick_count t2_par_split = tick_count::now();
            s.result(xmin_par_split, ymin_par_split);

            tick_count t1_par = tick_count::now();
            parallel_global_search(pfn[i], a, b, kmax, precision, xmin_par,
                ymin_par);
            tick_count t2_par = tick_count::now();

            tick_count t1_seq = tick_count::now();
            sequential_global_search(pfn[i], a, b, kmax, precision, xmin_seq,
                ymin_seq);
            tick_count t2_seq = tick_count::now();

            cout << "Global_min_hfunc " << i + 1 << " - " << res[i][0] << endl;
            cout << "Result_parallel_split: (" << xmin_par_split << ", " <<
                ymin_par_split << ")     Time_parallel: " << (t2_par_split -
                t1_par_split).seconds() << endl;
            cout << "Result_parallel: (" << xmin_par << ", " << ymin_par << ")  
                   Time_parallel: " << (t2_par - t1_par).seconds() << endl;
            cout << "Result_sequential: (" << xmin_seq << ", " << ymin_seq <<
                ")   Time_sequential: " << (t2_seq - t1_seq).seconds() << endl
                << endl;
        }
        catch (const char* message) {
            cout << message;
        }

    }
}
```