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

Отчёт по лабораторной работе:

"Умножение разреженных матриц. Элементы типа double. Формат хранения матрицы – столбцовый (CCS)"

Выполнил: студент группы 3822Б1ФИ3_и1
Лаврентьев Алексей

Преподаватель:
доцент кафедры ВВСП, к.т.н
Сысоев А.В.

Нижний Новгород
2025

Введение

Разреженная матрица - матрица, имеющая малое количество ненулевых элементов. Матрица называется разреженной тогда и только тогда, когда при её размерности $N * N$ количество ненулевых элементов не превосходит N. Существует отдельный раздел алгебры, изучающий данные матрицы и их взаимодействие (Sparse algebra), а область применения разреженных матриц достаточно объемна:

1. Постановка и решение задач оптимизации
2. Моделирование сетей
3. Решение дифференциальных уравнений в частных производных

Учитывая частоту применения разреженных матриц на практике, очевидно, что оптимизация операций, проводимых с ними, является важной задачей. Именно этой теме и посвящены выполненные лабораторные работы.

Постановка задачи

Необходимо реализовать алгоритм умножения двух разреженных матриц с элементами типа double и форматом их хранения по столбцам. Решение данной задачи удобно представить в виде последовательности мелких подзадач:

1. Представить исходные матрицы A, B в разреженном виде.
2. Транспонировать одну из исходных матриц (здесь и далее договоримся, что транспонировать будем матрицу A).
3. Умножить полученные матрицы A^T и B с помощью алгоритма, похожего на классическое умножение матриц.
4. Перевести полученную матрицу из разреженного вида в классический.

В рамках курса необходимо реализовать параллельную версию алгоритма умножения двух разреженных матриц. Для реализации параллелизма требуется использование следующих технологий:

- OpenMP (Open Multi Processing)
- TBB (Threading Building Blocks)
- Средства STL (std::thread)
- MPI (Message Passing Interface)

Описание алгоритма

Как уже было обозначенно в разделе введение, разреженные матрицы имеют широкое распространение и хорошо известны во многих областях. Как следствие: существует множество подходов к реализации разреженных матриц. В нашем случае матрица должна храниться по столбцам, а это подразумевает под собой разделение матрицы на 3 вектора: вектор ненулевых элементов, вектор индексов строк, вектор сумм количества значений в каждом столбце исходной матрицы. Чтобы понять идею подхода, обратимся к примеру.

Пусть сущестует некоторая матрица A размерности $4 * 4$.

$$A = \begin{pmatrix} 0 & 0 & 0 & 4 \\ 2 & 0 & 0 & 7 \\ 0 & 1 & 2 & 0 \\ 0 & 5 & 0 & 0 \end{pmatrix}$$

Тогда векторы, её представления в разреженном виде будут выглядеть следующим образом:
$(2, 5, 2, 4, 7)$ - элементы
$(1, 3, 2, 0, 1)$ - индексы строк
$(1, 2, 3, 5)$ - сумма количества элементов по столбцам

В рамках лабораторных работ для хранения разреженных матриц была реализована структура Sparse следующего вида:

```
struct Sparse {
  std::pair<int, int> size;
  std::vector<double> elements;
  std::vector<int> rows;
  std::vector<int> columnsSum;
};
```

Преимущества подобного формата хранения разреженных матриц очевидны, причем, чем больше будет исходная матрица, тем эффективнее будет разреженный формат хранения. Например, для хранения матрицы размерности $200 * 20$, имеющей 100 ненулевых элементов, нам потребуется всего лишь 220 элементов, против 4000 в классическом виде. Сам алгоритм представления матрицы в разреженном виде тривиален и может быть выполнен за 1 проход цикла.

Следующим шагом в работе алгоритма умножения матриц является транспонирование матрицы A. Существует несколько алгоритмов транспонирования матриц, не отличающихся большим разрывом в производительности, поэтому было принято решение воспользоваться наиболее простым в реализации. Его идея заключается в следующем: сделать итерацию по всем ненулевым элементам и сохранить индексы их строк в транспонированном виде, после чего переписать сохраненные значения в нужном порядке, попутно увеличивая счетчик количества элементов по столбцам. Оценка сложности транспонирования составляет $O(M(N+1))$, где N - количество ненулевых элементов, а M - количество столбцов исходной матрицы.

Сам алгоритм умножения матриц достаточно прост, но при этом объемен, так как требует выполнения нескольких условий на каждом шаге (проверка совпадений индексов умножаемых элементов, учет количества элементов в текущем столбце и т д). Грубая оценка сложности алгоритма дает следующую верхнюю границу: $O(N^2 * M^2)$, где зачастую $M < N$.

На последнем этапе происходит преобразование разреженный матрицы к исходному виду. Операция является тривиальной и выполняется за $O(N * M)$.

Описание схемы параллельного алгоритма

Каждая из частей алгоритма умножения разреженных матриц подразумевает перезапись уже существующих данных в новом виде. Проще говоря: мы постоянно сталкиваемся с извлечением и добавлением элементов одного вектора в другой. Из-за этого применение технологий распараллеливания к таким операциям, как приведение к разреженному виду или транспонирование становится невозможным. Мы попросту не можем гарантировать порядок, в котором потоки будут добавлять элементы в векторы, а это имеет ключевое значение.

Таким образом, параллельной обработке может подвергаться лишь часть алгоритма, занимающаяся непосредственно умножением матриц. Идея заключается в том, чтобы в максимально равной степени разделить между потоками итерации внешнего цикла и записывать результат умножения элементов в заранее заготовленные ячейки векторов. Стоит отметить, что задача определения количества памяти, которое нужно зарезервировать под элементы и прочие данные, является далеко не тривиальной и имеет множество подводных камней. В рамках данных лабораторных работ была использована не самая эффективная, но наиболее простая методика - выделение памяти размера N^2, что соответствует результату умножения двух матриц, подходящих под определение "разреженные".

Рассмотрим все вариации параллельного алгоритма умножения разреженных матриц.

OpenMP

OpenMP представляет собой простой, но в то же время мощный инструмент параллельной обработки данных. Главным преимуществом OpenMP является наличие механизма, автоматически разедляющего итерации цикла между потоками. Идея решения выглядит следующим образом:

1. Создание вектора данных с каждого потока. Вектор имеет размер количества существующих потоков, что определяется через ppc::util::GetPPCNumThreads(), а тип вектора представляет собой пару векторов типов double и int.

2. Создание параллельной области с помощью директивы препроцессора #pragma omp parallel и разделение итераций цикла между потоками командой #pragma omp for.

3. На каждом потоке создается своя пара векторов, в котороую "складываются"элементы, получившиеся в результате произведения и индексы их строк. В конце работы цикла пара векторов помещается в вектор общих данных на позицию, определяему функцией omp_get_thread_num().

4. На заключительном шаге происходит слияние данных, полученных на каждом потоке, а также происходит обработка вектора сумм количества элементов.

Благодаря OpenMP можно получить существенный прирост производительности, примерно в 2.5-3 раза в сравненнии с параллельной версией.

TBB

TBB - механизм параллельной обработки данных, предложенный компанией Intel. В отличии от OpenMP, не так прост в использовании и требует более глубокого понимания процесса распределения данных, однако, зачастую, показывает лучшие результаты в плане производительности. На ранней стадии своего существования TBB требовал от программиста использования множества инструментов (operator(), явно прописанный конструктор коприрования и т д), но с появлением в C++ лямбда-выражений применение TBB стало значительно проще. Решение задачи распараллеливания с помощью средств TBB сводится к следуюшим шагам:

1. Создание служебного объекта oneapi::tbb::task_arena размера количества потоков.

2. Вызов у объекта task_arena метода execute() и применение в его рамках функции oneapi::tbb::parallel_for с параметром blocked_range. В конструкторе blocked_range указываются индексы первого и последнего элемента диапазона разбиение, а также grangsize - количественное разделение данных между потоками.

3. Исполнение в рамках parallel_for главного цикла, выполняющего умножения матриц и запись результатов в промежуточный контейнер размерности N^2.

4. Перезапись ненулевых значений элементов и строк в итоговую матрицу, обработка вектора сумм количества элементов.

С использованием средств TBB скорость исполнения параллельного алгоритма в сравнении с последовательным увеличивается в 3 или даже 4 раза!

STL

Как и в большинстве современных языков программирования, в C++ присутствуют встроенные средства распараллеливания. Одним из таких инструментов является класс std::thread, впервые появившийся ещё в C++11. Идея std::thread состоит в создании отдельного потока, в котором будут обрабаываться данные, указанные в конструкторе объекта std::thread. В качестве аргумента конструктор принимает лямбда-выражение, что делает применение std::thread отдаленно схожим с использованием TBB. Шаги работы алгоритма:

1. Создание вектора потоков размера ppc::util::GetPPCNumThreads().
2. Создание лямбда-выражения matrix_multiplicator, в котором происходит умножение элементов на заданном интервале и запись результатов в вектор выделенного размера.
3. Определение количества данных, выдаваемых каждому потоку на исполнение.
4. Запуск цикла по вектору потоков и создание на каждой итерации потока, с указанием в конструкторе лямбда-выражения matrix_multiplicator, начала и конца отрезка итерирования.
5. После завершения цикла необходимо вызвать для каждого потока операцию join(), которая грубо говоря отвечает за завершение потока.
6. Очистить вектор данных от лишних нулевых элементов.

С помощью средств STL можно относительно простым путем применить механизмы распараллеливания и добиться существенного уменьшения времени исполнения программы. Скорость исполнения относительно последовательной версии увеличивается в 3-4 раза.

MPI + STL

MPI - относительно неплохой инструмент для работы в системах с раздельной памятью, который, впрочем, можно применить и в системах с единой памятью. В теории, использование MPI в сочетании со средствами многопоточности может многократно ускорить работу алгоритма, однако не стоит забывать о накладных расходах, которые значительно затормаживают работу алгоритма. Шаги решения задачи:

1. Перегрузка шаблонного метода serialize для структуры Sparse с целью передачи исходных векторов на все процессы с помощью boost::mpi::broadcast.

2. Максимально "честное"разделение обрабатываемых данных между всеми процессами. Распределение вектора разбиения данных между всеми процессами.

3. На каждом процессе вызывается умножение матриц с использованием технологий многопоточной обработки STL.

4. На предыдущем шаге на выходе получается матрица, представляющая собой кусочный ответ. Для максимально быстрой передачи всех кусочных решений на нулевой процесс было принято решение выполнить их слияние в сплошной вектор типа double. После этого получившийся вектор отправляется на нулевой процесс функцией boost::mpi::gatherv.

5. На нулевом процессе происходит коллекционирование данных со всех процессов, после чего, путем достаточной долгой поэтапной обработки ненулевые данные распространяются между векторами результирующей матрицы.

Подобный способ использования технологий MPI может показаться странным и неоптимальным, однако он проявил себя, как более оптимальный, чем передача данных тремя последовательными операциями boost::mpi::gatherv. Итоговое ускорение программы относительно последовательной версии может достигать 1.2 раза.

Статистические данные по экспериментам

Далее будут приведены результаты тестов производительности, полученных, в результате применения каждой технологии. Важно уточнить, что замеры производились на матрицах размерности $700 * 700$, причем заполненных не нулями ровно на $1/6$ часть от их объма. Числа, которым были заполненны матрицы, находились в диапазоне $|500|$.

Результаты тестов будут представлены в одной таблице, скорость выполнения Pipline и Task тестов разделены значком X.

Sequential	0.57X0.494

Технология	1 поток	2 потока	3 потока	4 потока
OpenMP	1.02X0.944	0.573X0.51	0.45X0.36	0.37X0.29
TBB	1.03X0.94	0.6X0.52	0.45X0.39	0.41X0.29
STL	1.03X0.98	0.59X0.46	0.48X0.38	0.4X0.3
MPI + STL(1)	1.18X1.07	0.74X0.63	0.56X0.45	0.48X0.39
MPI + STL(2)	0.77X0.67	0.54X0.45	0.52X0.41	0.5X0.37
MPI + STL(3)	0.7X0.59	0.59X0.501	0.53X0.45	0.49X0.39

Запуски тестов проводились на машине со следующими характеристиками:

- Процессор AMD Ryzen 5 5500U 2.1 Hz.
- ОЗУ 16 Гб.
- Операционная система Windows 11 x64.
- Сборка производилась с использованием Visual Studio 2022.

Выводы из тестов: с увеличением количества потоков, скорость прохождения тестов линейно увеличивается и уже на 3 потоках превосходит sequential версию. Пик производительности достигается на 5-6 потоках, после чего ускорение не наблюдается. В случае с использованием MPI, при грамотном соотношение количества процессов и потоков можно добиться неплохих показателей ускорения (несколько десятков процентов, но меньше 50).

Заключение

Использование механизмов распараллеливания сложных, в плане вычислений, алгоритмов позволяет добиться значительного прироста эффективности, что делает их максимально полезными в повседневной жизни разработчика. С другой стороны, важно помнить о том, что злоупотреблять многопоточностью не стоит, так как в какой-то момент количество ресурсов, затрачиваемых на распараллеливание может превысить эффект от параллельной обработки.

Использованная литература

Список использованных источников:

- Мееров И.Б., Сысоев А.В., при поддержке Intel. Разреженное матричное умножение.
- Мееров И.Б. при участии Лебедева С.А., Пировой А.Ю. Оптимизация структур данных при работе с разреженными матрицами
- Крейг К., Рэндольф Э. Пакет для умножения разреженных матриц.

Приложение

Реализация sequential версии с набором функций, актуальных для каждой версии

```
lavrentiev_a_ccs_seq::Sparse lavrentiev_a_ccs_seq::
CCSSequential::ConvertToSparse(std::pair<int, int> bsize,
    const std::vector<double> &values) {
  auto [size, elements, rows, columns_sum] = Sparse();
  columns_sum.resize(bsize.second);
  for (int i = 0; i < bsize.second; ++i) {
    for (int j = 0; j < bsize.first; ++j) {
      if (values[i + (bsize.second * j)] != 0) {
        elements.emplace_back(values[i + (bsize.second * j)]);
        rows.emplace_back(j);
        columns_sum[i] += 1;
      }
    }
    if (i != bsize.second - 1) {
      columns_sum[i + 1] = columns_sum[i];
    }
  }
  return {.size = bsize, .elements = elements, .rows = rows, .
columnsSum = columns_sum};
}

lavrentiev_a_ccs_seq::Sparse lavrentiev_a_ccs_seq::
CCSSequential::MatMul(const Sparse
&matrix1, const Sparse &matrix2) {
  auto [size, elements, rows, columns_sum] = Sparse();
  columns_sum.resize(matrix2.size.second);
  Sparse new_matrix1 = Transpose(matrix1);
  for (int i = 0; i < static_cast<int>(
  matrix2.columnsSum.size()); ++i) {
    for (int j = 0; j < static_cast<int>(
    new_matrix1.columnsSum.size()); ++j) {
      double sum = 0.0;
      int start_index1 = j != 0 ? new_matrix1.columnsSum[j] -
      GetElementsCount(j, new_matrix1.columnsSum) : 0;
      int start_index2 = i != 0 ? matrix2.columnsSum[i]
      - GetElementsCount(i, matrix2.columnsSum) : 0;
      for (int n = 0; n <
      GetElementsCount(j,new_matrix1.columnsSum); n++) {
        for (int n2 = 0; n2 <
        GetElementsCount(i, matrix2.columnsSum); n2++) {
```

```
            if (new_matrix1.rows[start_index1 + n] ==
            matrix2.rows[start_index2 + n2]) {
              sum += new_matrix1.elements[n + start_index1]
              * matrix2.elements[n2 + start_index2];
            }
          }
        }
        if (sum != 0) {
          elements.emplace_back(sum);
          rows.emplace_back(j);
          columns_sum[i]++;
        }
      }
    }
    for (auto i = 1; i < static_cast<int>
    (columns_sum.size()); ++i) {
      columns_sum[i] = columns_sum[i] + columns_sum[i - 1];
    }
    size.first = matrix2.size.second;
    size.second = matrix2.size.second;

    return {.size = size, .elements = elements, .rows = rows,
    .columnsSum = columns_sum};
}

int lavrentiev_a_ccs_seq::CCSSequential::
GetElementsCount(int index, const std::vector<int>
&columns_sum) {
    if (index == 0) {
      return columns_sum[index];
    }
    return columns_sum[index] - columns_sum[index - 1];
}

std::vector<double> lavrentiev_a_ccs_seq::CCSSequential::
ConvertFromSparse(const Sparse &matrix) {
    std::vector<double> nmatrix(matrix.size.first *
    matrix.size.second);
    int counter = 0;
    for (size_t i = 0; i < matrix.columnsSum.size(); ++i) {
      for (int j = 0; j < GetElementsCount(static_cast<int>(i),
        matrix.columnsSum); ++j) {
        nmatrix[i + (matrix.size.second * matrix.rows[counter])] =
        matrix.elements[counter];
        counter++;
```

```
86      }
87    }
88    return nmatrix;
89  }
90
91  lavrentiev_a_ccs_seq::Sparse lavrentiev_a_ccs_seq::
92  CCSSequential::
93  Transpose(const Sparse &sparse) {
94    auto [size, elements, rows, columns_sum] = Sparse();
95    size.first = sparse.size.second;
96    size.second = sparse.size.first;
97    int need_size = std::max(sparse.size.first,
98    sparse.size.second);
99    std::vector<std::vector<double>> new_elements(need_size);
100   std::vector<std::vector<int>> new_indexes(need_size);
101   int counter = 0;
102   for (int i = 0; i < static_cast<int>(
103   sparse.columnsSum.size()); ++i) {
104     for (int j = 0; j <
105     GetElementsCount(i, sparse.columnsSum); ++j) {
106     new_elements[sparse.rows[counter]].
107     emplace_back(sparse.elements[counter]);
108       new_indexes[sparse.rows[counter]].emplace_back(i);
109       counter++;
110     }
111   }
112   for (int i = 0; i < static_cast<int>
113   (new_elements.size()); ++i) {
114     for (int j = 0; j < static_cast<int>
115     (new_elements[i].size()); ++j) {
116       elements.emplace_back(new_elements[i][j]);
117       rows.emplace_back(new_indexes[i][j]);
118     }
119     if (i > 0) {
120       columns_sum.emplace_back(new_elements[i].size() +
121       columns_sum[i - 1]);
122     } else {
123       columns_sum.emplace_back(new_elements[i].size());
124     }
125   }
126   return {.size = size, .elements = elements, .rows = rows,
127   .columnsSum = columns_sum};
128 }
129
130 bool lavrentiev_a_ccs_seq::CCSSequential::ValidationImpl() {
```

```
131   return task_data->inputs_count[0] *
132   task_data->inputs_count[3] ==
133   task_data->outputs_count[0] &&
134           task_data->inputs_count[0] ==
135           task_data->inputs_count[3] &&
136           task_data->inputs_count[1] ==
137           task_data->inputs_count[2];
138 }
139
140 bool lavrentiev_a_ccs_seq::CCSSequential::PreProcessingImpl() {
141   A_.size = {static_cast<int>(task_data->inputs_count[0]),
142   static_cast<int>(task_data->inputs_count[1])};
143   B_.size = {static_cast<int>(task_data->inputs_count[2]),
144   static_cast<int>(task_data->inputs_count[3])};
145   if (IsEmpty()) {
146     return true;
147   }
148   std::vector<double> am(A_.size.first * A_.size.second);
149   auto *in_ptr = reinterpret_cast<double *>
150   (task_data->inputs[0]);
151   for (int i = 0; i < A_.size.first *
152   A_.size.second; ++i) {
153     am[i] = in_ptr[i];
154   }
155   A_ = ConvertToSparse(A_.size, am);
156   std::vector<double> bm(B_.size.first * B_.size.second);
157   auto *in_ptr2 = reinterpret_cast<double *>
158   (task_data->inputs[1]);
159   for (int i = 0; i < B_.size.first * B_.size.second; ++i) {
160     bm[i] = in_ptr2[i];
161   }
162   B_ = ConvertToSparse(B_.size, bm);
163   return true;
164 }
165
166 bool lavrentiev_a_ccs_seq::CCSSequential::IsEmpty()
167 const {
168   return A_.size.first * A_.size.second == 0 ||
169   B_.size.first * B_.size.second == 0;
170 }
171
172 bool lavrentiev_a_ccs_seq::CCSSequential::RunImpl() {
173   Answer_ = MatMul(A_, B_);
174   return true;
175 }
```

```
176
177 bool lavrentiev_a_ccs_seq::CCSSequential::
178 PostProcessingImpl() {
179   std::vector<double> result = ConvertFromSparse(Answer_);
180   for (auto i = 0; i < static_cast<int>(result.size()); ++i) {
181     reinterpret_cast<double *>(task_data->outputs[0])[i] =
182     result[i];
183   }
184   return true;
185 }
```

Реализация функции умножения матриц в параллельных версиях

```
1  lavrentiev_a_ccs_omp::Sparse lavrentiev_a_ccs_omp
2  ::CCSOMP::MatMul(const Sparse &matrix1,
3  const Sparse &matrix2) {
4    Sparse result_matrix;
5    result_matrix.columnsSum.resize(matrix2.size.second);
6    auto new_matrix1 = Transpose(matrix1);
7    std::vector<std::pair<std::vector<double>,
8    std::vector<int>>> threads_data(
9        std::max(1, ppc::util::GetPPCNumThreads()));
10 #pragma omp parallel
11   {
12     std::pair<std::vector<double>, std::vector<int>>
13     current_thread_data;
14 #pragma omp for
15     for (int i = 0; i < static_cast<int>(
16     matrix2.columnsSum.size()); ++i) {
17       for (int j = 0; j < static_cast<int>
18       (new_matrix1.columnsSum.size()); ++j) {
19         double sum = 0.0;
20         for (int n = 0; n < GetElementsCount(j,
21         new_matrix1.columnsSum); n++) {
22           for (int n2 = 0; n2 < GetElementsCount(i,
23           matrix2.columnsSum); n2++) {
24             if (new_matrix1.rows[CalculateStartIndex(j,
25             new_matrix1.columnsSum) + n] ==
26                 matrix2.rows[CalculateStartIndex
27                 (i, matrix2.columnsSum) + n2]) {
28               sum += new_matrix1.elements
29               [n + CalculateStartIndex(j,
30               new_matrix1.columnsSum)] *
31               matrix2.elements
32             [n2 + CalculateStartIndex(i,
33             matrix2.columnsSum)];
```

```
            }
          }
        }
        if (sum != 0) {
          current_thread_data.first.push_back(sum);
          current_thread_data.second.push_back(j);
          result_matrix.columnsSum[i]++;
        }
      }
    }
    threads_data[omp_get_thread_num()] =
    std::move(current_thread_data);
  }
  for (size_t i = 1; i <
  result_matrix.columnsSum.size(); ++i) {
    result_matrix.columnsSum[i] =
    result_matrix.columnsSum[i] +
    result_matrix.columnsSum[i - 1];
  }
  if (!result_matrix.columnsSum.empty()) {
    result_matrix.elements.resize
    (result_matrix.columnsSum.back());
    result_matrix.rows.resize
    (result_matrix.columnsSum.back());
  }
  int count = 0;
  for (size_t i = 0; i < threads_data.size(); ++i) {
    std::ranges::copy(threads_data[i].first,
    result_matrix.elements.begin() + count);
    std::ranges::copy(threads_data[i].second,
    result_matrix.rows.begin() + count);
    count += static_cast<int>(threads_data[i].first.size());
  }
  result_matrix.size.first = matrix2.size.second;
  result_matrix.size.second = matrix2.size.second;
  return {.size = result_matrix.size,
          .elements = result_matrix.elements,
          .rows = result_matrix.rows,
          .columnsSum = result_matrix.columnsSum};
}

lavrentiev_a_ccs_tbb::Sparse lavrentiev_a_ccs_tbb
::CCSTBB::MatMul(const Sparse &matrix1,
const Sparse &matrix2) {
  oneapi::tbb::task_arena worker(ppc::util::
```

```
  GetPPCNumThreads());
  Sparse imatrix; imatrix.columnsSum.resize(matrix2.size.second);
  imatrix.elements_and_rows.
  resize((matrix2.columnsSum.size() *
  matrix1.columnsSum.size()));
  auto new_matrix1 = Transpose(matrix1);
  auto sum = [&](int i_index, int j_index) {
    double s = 0.0;
    for (int x = 0; x < GetElementsCount(j_index,
    new_matrix1.columnsSum); x++) {
      for (int y = 0; y < GetElementsCount(i_index,
      matrix2.columnsSum); y++) {
        if (new_matrix1.elements_and_rows
        [CalculateStartIndex(j_index, new_matrix1.columnsSum) +
        x].second ==
            matrix2.elements_and_rows
            [CalculateStartIndex(i_index, matrix2.columnsSum)
            + y].second) {
          s += new_matrix1.elements_and_rows[x +
          CalculateStartIndex(j_index,
          new_matrix1.columnsSum)].first *
              matrix2.elements_and_rows
              [y + CalculateStartIndex
              (i_index, matrix2.columnsSum)].first;
        }
      }
    }
    return s;
  };
  worker.execute([&] {
oneapi::tbb::parallel_for(
 oneapi::tbb::blocked_range<int>(0, static_cast<int>
(matrix2.columnsSum.size()),
  matrix2.columnsSum.size() /
ppc::util::GetPPCNumThreads()),
[&](const oneapi::tbb::blocked_range<int> &blocked_range)
{for (int i = blocked_range.begin(); i !=
  blocked_range.end(); ++i) {
for (int j = 0; j < static_cast<int>
(new_matrix1.columnsSum.size()); ++j) {
              double s = sum(i, j);
if (s != 0) {imatrix.elements_and_rows[(i *
                matrix2.size.second) + j] = {s, j};
                imatrix.columnsSum[i]++;
              }
```

```
124             }
125           }
126         });
127   });
128   for (size_t i = 1; i < imatrix.columnsSum.size(); ++i) {
129     imatrix.columnsSum[i] = imatrix.columnsSum[i] +
130     imatrix.columnsSum[i - 1];
131   }
132   imatrix.size.first = matrix2.size.second;
133   imatrix.size.second = matrix2.size.second;
134   std::vector<std::pair<double, int>> new_elements_and_rows;
135   for (size_t i = 0; i <
136   imatrix.elements_and_rows.size(); ++i) {
137     if (imatrix.elements_and_rows[i].first != 0.0) {
138       new_elements_and_rows.emplace_back
139       (imatrix.elements_and_rows[i]);
140     }
141   }
142   return {.size = imatrix.size, .elements_and_rows =
143   new_elements_and_rows, .columnsSum = imatrix.columnsSum};
144 }
145
146 lavrentiev_a_ccs_stl::Sparse lavrentiev_a_ccs_stl::CCSSTL
147 ::MatMul(const Sparse &matrix1, const Sparse &matrix2) {
148   Sparse temporary_matrix;
149   std::vector<std::thread> threads(ppc::util
150   ::GetPPCNumThreads());
151   temporary_matrix.columnsSum.resize
152   (matrix2.size.second);
153   temporary_matrix.elements_and_rows.resize
154   ((matrix2.columnsSum.size() * matrix1.columnsSum.size()) +
155                 std::max(matrix1.columnsSum.size(),
156                 matrix2.columnsSum.size()));
157   auto transposed_matrix = Transpose(matrix1);
158   auto accumulate = [&](int i_index, int j_index) {
159     double sum = 0.0;
160     for (int x = 0; x < GetElementsCount(j_index,
161 transposed_matrix.columnsSum); x++) {
162       for (int y = 0; y < GetElementsCount(i_index,
163       matrix2.columnsSum); y++) {
164         if (transposed_matrix.elements_and_rows
165         [CalculateStartIndex(j_index, transposed_matrix.
    columnsSum)
166         + x]
167         .second == matrix2.elements_and_rows
```

```
168           [CalculateStartIndex
169           (i_index, matrix2.columnsSum)
170           + y].second) {
171             sum += transposed_matrix.elements_and_rows[x +
172             CalculateStartIndex(j_index,
173             transposed_matrix.columnsSum)]
174                         .first *
175           matrix2.elements_and_rows[y +
176           CalculateStartIndex(i_index, matrix2.columnsSum)]
177           .first;
178           }
179         }
180       }
181       return sum;
182     };
183     auto matrix_multiplicator = [&](int begin, int end) {
184       for (int i = begin; i != end; ++i) {
185         for (int j = 0; j < static_cast<int>(
186        transposed_matrix.columnsSum.size()); ++j) {
187           double s = accumulate(i, j);
188           if (s != 0) {
189             temporary_matrix.elements_and_rows
190             [(i * matrix2.size.second) + j] = {s, j};
191             temporary_matrix.columnsSum[i]++;
192           }
193         }
194       }
195     };
196     int thread_data_amount =
197     static_cast<int>(matrix2.columnsSum.size()) /
198     ppc::util::GetPPCNumThreads();
199     for (size_t i = 0; i < threads.size(); ++i) {
200       if (i != threads.size() - 1) {
201         threads[i] =
202         std::thread(matrix_multiplicator, i *
203         thread_data_amount, (i + 1) * thread_data_amount);
204       } else {
205         threads[i] =
206             std::thread(matrix_multiplicator,
207             i * thread_data_amount,
208           ((i + 1) * thread_data_amount) +
209           (matrix2.columnsSum.size() % ppc::util::
210           GetPPCNumThreads()));
211       }
212     }
```

```
213   std::ranges::for_each(threads,
214   [&](std::thread &thread) { thread.join(); });
215   for (size_t i = 1; i <
216   temporary_matrix.columnsSum.size(); ++i) {
217     temporary_matrix.columnsSum[i] =
218     temporary_matrix.columnsSum[i] +
219     temporary_matrix.columnsSum[i - 1];
220   }
221   temporary_matrix.size.first = matrix2.size.second;
222   temporary_matrix.size.second = matrix2.size.second;
223   std::erase_if(temporary_matrix.elements_and_rows,
224   [](auto &current_element) {
225   return current_element.first == 0.0; });
226   return {.size = temporary_matrix.size,
227           .elements_and_rows =
228           temporary_matrix.elements_and_rows,
229           .columnsSum = temporary_matrix.columnsSum};
230 }
231
232 bool lavrentiev_a_ccs_all::CCSALL::RunImpl() {
233   boost::mpi::broadcast(world_, displ_, 0);
234   boost::mpi::broadcast(world_, A_, 0);
235   boost::mpi::broadcast(world_, B_, 0);
236   if (displ_.empty()) {
237     return true;
238   }
239   resize_data_ = static_cast<int>(B_.columnsSum.size() *
240   A_.columnsSum.size());
241   Process_data_ = MatMul(A_, B_,
242   displ_[world_.rank()], displ_[world_.rank() + 1]);
243   CollectSizes();
244   CollectData();
245   if (world_.rank() == 0) {
246     Answer_.columnsSum.clear();
247     Answer_.rows.clear();
248     Answer_.elements.clear();
249     Answer_.elements.reserve(resize_data_);
250     Answer_.rows.reserve(resize_data_);
251     Answer_.columnsSum.reserve(resize_data_);
252     std::vector<double> data_reader((resize_data_ *
253     world_.size() * 2) +
254     std::accumulate(sum_sizes_.begin(),
255     sum_sizes_.end(), 0));
256     std::vector<int> size(world_.size(),
257     resize_data_ * 2);
```

```
258       for (int i = 0; i < world_.size(); ++i) {
259         size[i] += sum_sizes_[i];
260       }
261       boost::mpi::gatherv(world_, sending_data_,
262       data_reader.data(), size, 0);
263       int process_data_num = 0;
264       int sum_by_elements_count = sum_sizes_.front();
265       int past_data = 0;
266       for (int i = 0; i < static_cast<int>(
267       data_reader.size()); ++i) {
268         if (i == (resize_data_ * 2 *
269         (process_data_num + 1)) + sum_by_elements_count) {
270           past_data += (resize_data_ * 2)
271           + sum_sizes_[process_data_num];
272           process_data_num++;
273           sum_by_elements_count +=
274           sum_sizes_[process_data_num];
275         }
276         AddData(data_reader, past_data, i);
277       }
278       for (size_t i = 1; i <
279       Answer_.columnsSum.size(); ++i) {
280         Answer_.columnsSum[i] =
281         Answer_.columnsSum[i] + Answer_.columnsSum[i - 1];
282       }
283       Answer_.size.first = B_.size.second;
284       Answer_.size.second = B_.size.second;
285     } else {
286       boost::mpi::gatherv(world_, sending_data_, 0);
287     }
288     return true;
289   }
```