МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский Нижегородский университет им.
Н.И. Лобачевского» (ННГУ)
Институт информационных технологий, математики и механики

Отчет по курсу "параллельное программиование"
Задача: Умножение плотных матриц, алгоритм Кэннона

Выполнил: студент 3 курса,
Сарафанов М. С.

Работу проверили:
Нестеров А.Ю.
Оболенский А.А.

Лектор: Сысоев А. В.

Нижний Новгород 2025

1 Введение

Матрицы - наиболее удобный и наглядный способ представления больших объемов связанных между собой численных данных. Одной из основных операций над матрицами является их перемножение. Эта операция используется практически везде: от компьютерной графики до расчета траектории полёта самолетов. В связи с этим возникает вопрос: как можно ускорить процесс умножения матриц? Одним из самых очевидных и эффективных способов оптимизации данной операции является параллельное перемножение матриц на нескольких потоках. Примером алгоритма, выполняющего данную задачу, является алгоритм Кэннона.

2 Постановка задачи

В рамках написания лабораторных работ необходимо реализовать алгоритм Кэннона - параллельный алгоритм перемножения матриц. В процессе реализации необходимо поочередно использовать следующие технологии:

- OpenMP
- TBB
- STL
- MPI

После написания всех работ, необходимо провести тесты производительности, вычислить преимущество от использования технологий распараллеливания относительно последовательной версии. Кроме того, необходимо сравнить результаты тестов параллельных версий между собой, выявить наиболее оптимальный вариант.

3 Описание алгоритма

Все мы знакомы с классическим алгоритмом умножения матриц: строки матрицы A последовательно умножаются на столбцы матрицы B, в результате чего на выходе получается матрица C. Пусть матрицы A и B имееют одинаковый размер $n*n$, тогда сложность алгоритма умножения двух матриц очевидно составляет $O(n^3)$. Очевидно, для больших n классическое умножение матриц будет выполняться слишком долго, поэтому были разработаны алгоритмы параллельного умножения, в частности алгоритм Кэннона. Принцип его работы заключается в следующем:

- Все строки исходной матрицы A сдвигаются влево на количество элементов, равное номеру строки. Строки нумеруются с нуля.
- Все столбцы исходной матрицы B сдвигаются вверх на количество элементов, равное номеру столбца. Столбцы нумеруются с нуля.
- Матрицы распределяются между существующими потоками и на каждом потоке происходит последовательный построчный сдвиг (для A), последовательный сдвиг по столбцам (для B) и умножение полученных матриц в скалярном виде.
- Матрицы, полученные в процессе умножения, складываются друг с другом, а в конце происходит суммирование матриц, полученных на каждом потоке.

Результатом данных манипуляций является искомая матрица C. Что касается оценки сложности данного алгоритма, имеем следующее:
$T_{shift} = 2(a + \frac{w*n^2}{p*b})$, где a - латентность, b - пропускная способность, w - размер элемента матрицы.

$T_{calc} = q(\frac{n^2}{p} * c * (\frac{2n}{q} - 1) + \frac{n^2}{p})t$, где n - количество строк и столбцов, а q - количество необходимых итераций.

Очевидно, что $T_{all} = T_{shift} + T_{calc} * (q + 1)$. Из данных формулировок также очевидно, что алгоритм Кэннона применим лишь к квадратным матрицам одинаковых размерностей. В случае не выполнения даннного условия матрицы должны быть приведены к квадратному виду при помощи добавления нулевых элементов.

Использование алгоритма Кэннона для последовательного умножения матриц делает процесс медленнее, чем при использовании классического подхода.

4 Описание параллельных реализаций

Перед тем, как пройтись по особбеностям различных параллельных реализаций, стоит упомянуть общие детали.

1. Для упрощения процесса "лестничного"сдвига матриц по строкам и столбцам было принято решение транспонировать исходную матрицу B. Благодаря такому решению не потребовалось реализовывать 2 сложные функции "лестничного сдвига достаточно и одной.

2. В процессе разработки было использовано несколько подходов для сдвига элементов матриц влево или вверх. Ни один из них не оказался в достаточной степени эффективным и в конце концов было принято решение иммитировать сдвиг элементов матриц. Для этого были реализованы 2 функции, возвращающие индекс элемента, необходимого на данной итерации: GetRowIndex, GetColumnIndex.

3. После выполнения скалярного умножения, получившиеся матрицы необходимо сложить. Для этого в разработанном классе CannonMatrix был перегружен оператор $+=$, который суммирует эллементы двух матриц, причем делает это в транспонированном виде, что позволяет избежать применения обратной операции транспонирования.

Теперь поговорим подробнее о конкретных реализациях.

4.1 OpenMP

После преобразования исходных матриц A, B к требуемому виду, был создан std::vector<CannonMatrix> mul_results, в который записываются результаты умножения матриц на каждом потоке. Цикл имеет диапазон $(0, n)$, где n - количество строк, а создание потоков происходит непосредственно перед с ним помощью директив препроцессора. Сперва указывается # pragma omp parallel, а на следующей строке # pragma omp for. После выполнения цикла происходит итерация по mul_results и к каждому из сохраненных значений применяется оператор $+=$. Внутри самого оператора также происходит разделение процесса сложения между потоками при помощи директив препроцессора.

4.2 TBB

После преобразования исходных матриц A, B к требуемому виду, был создан std::vector<CannonMatrix> mul_results размерности количества потоков, в который записываются результаты умножения матриц на каждом потоке. Кроме того, создается специальный функциональный объект tbb - arena, в котором, в рамках метода execute() происходит распараллеливание алгоритма. Для определения границ итерируемой области каждого потока создается специальный объект blocked_range, в котором через запятую указываются следующие параметры: начальный индекс, конечный индекс, количество данных, обрабатываемое каждым потоком (gransize). В рамках самого метода execute() происходит умножение и суммирование соответствующих частей матрицы. По завершении метода execute(), происходит суммирование матриц, полученных на каждом потоке. Попытки распараллелить данный процесс привели лишь к увеличению времени исполнения и было принято решение оставить суммирование последовательным.

4.3 STL

После преобразования исходных матриц A, B к требуемому виду, был создан std::vector<CannonMatrix> mul_results размерности количества потоков, в который записываются результаты умножения матриц на каждом потоке. Также создается вектор потоков (std::thread), той же размерности и объект структуры ThreadDataAmount, который содержит информацию о разбиении интервала итерации между потоками. Сам цикл, выполняющий итерации, заносится в специальную лямбда функцию multiplicator, что позволяет использовать его в конструкторе класса std::thread. Данный конструктор вызывается в цикле у каждого элемента вектора типа std::thread. Подход к умножению и сложению матриц тот же, что и в TBB версии. После выполнения главного цикла, для каждого из потоков вызывается метод join(), а затем происходит суммирование матриц, полученных на каждом потоке.

4.4 MPI + OMP

Реализация MPI-версии решения потребовала некоторых нововведений. Первым стала перегрузка метода serialize для класса CannonMatrix, что позволило применять к объектам данного класса метод broadcast. Кроме того, потребовалось реализовать специальную функцию, которая будет рассчитывать наиболее оптимальные интервалы итерации для каждого процесса. Для удобного обмена данными была добавлена структура IndexPair, в которой также пришлось перегрузить метод serialize. Что касается непосредственно логики алгоритма умножения матриц, она заключается в следующем: обе матрицы распределяются с нулевого процесса на все остальные, также распространяется вектор смещений. После этого на каждом процессе происходит итерация на заданном интервале, а результаты итераций суммируются в единственную матрицу matrix_sum_on_process. При помощи метода boost::mpi::gatherv полученная матрица отправляется на корневой процесс и записывается в промежуточный вектор. В конце происходит итерация по вектору и суммирование всех элементов с кратными индексами.

5 Результаты экспериментов

Параметры тестов производительности: тесты всех параллельных реализаций выполнялись на квадратных матрицах размерности $250 * 250$, с маленьким диапазоном положительных и отрицательных чисел $[-300, 300]$.

Для наглядности представления, данные о результатах запусков будут представлены в виде таблиц. Первыми указываются результаты pipline тестов, затем task run.

Threads	Sequential	OpenMP	TBB	STL
1	1.03,1.01	1.01,1.0	0.96,0.88	0.75,0.59
2	1.03,1.01	0.626,0.616	0.55,0.392	0.51,0.388
3	1.03,1.01	0.475,0.446	0.52,0.31	0.393,0.217
4	1.03,1.01	0.391,0.382	0.34,0.26	0.294,0.199

Threads	MPI(1)+OMP	MPI(2)+OMP	MPI(3)+OMP	MPI(4)+OMP
1	1.06,0.9	0.685,0.55	0.566,0.55	0.427,0.455
2	0.62,0.42	0.458,0.376	0.395,0.382	0.392,0.369
3	0.548,0.4	0.386,0.351	0.353,0.334	0.347,0.31

Дальнейшие запуски не показали большого прироста производительности.
Запуски производились на компьютере со следующими характеристиками:

- Операционная система macOS
- Процессор Apple M3
- ОЗУ: 8 ГБ
- 512 ГБ памяти ЖД

Вывод: Одновременное использование MPI и средства многопоточной обработки, в частности OpenMP, дало наибольший прирост производительности, чего и следовало ожидать хотя бы из условий применимости алгоритма Кэнона. Кроме того, очевидно, что независимо от технологии, улучшение производительности не будет расти линейно и уже на 7-8 потоках может пойти в обратном направлении. Что касается выбора наиболее эффективной технологии: наилучшие результаты демонстрирует STL, однако TBB отстает не сильно, в отличии от OpenMP.

6 Заключение

Алгоритм Кэннона - крайне полезный инструмент для максимально эффективного решения классической задачи умножения матриц. При правильном использовании, даже самые простые технологии распараллеливания дают ощутимое ускорение в сравнении с последовательной версией алгоритма. Наилучшего результата же можно добиться в случае разумного соотношения технологий многопоточности и средств MPI.

7 Список литературы

- Макаров Д., Шилов Н. Статья на тему реализации алгоритма Кэннона и расчета производительности
- Воеводин В.В. Математические основы параллельных вычислений
- Деммел Дж., Параллельное умножение матриц, Калифорнийский университет в Беркли

8 Приложение

Функции, применяемые независимо от реализации:

```
size_t CanonMatrix::GetRowIndex(size_t index, size_t row_number)
const {
  if (index < sqrt_size_ * row_number) {
    return index;
  }
  auto shift = index - (sqrt_size_ * row_number);
  return (row_number * sqrt_size_) - sqrt_size_ + shift;
}

size_t CanonMatrix::GetColumnIndex(size_t index,
size_t column_index, size_t offset) const {
  if (index + offset < sqrt_size_ * column_index) {
    return index + offset;
  }
  auto shift = index + offset - (sqrt_size_ * column_index);
  return (sqrt_size_ * (column_index - 1)) + shift;
}

void CanonMatrix::StairShift() {
  std::vector<double> new_matrix(matrix_.size());
  std::copy(matrix_.begin(), matrix_.begin() +
  static_cast<int>(sqrt_size_), new_matrix.begin());
  int s_size = static_cast<int>(sqrt_size_);
  for (int i = 1; i < s_size; ++i) {
    std::copy(matrix_.begin() + s_size * i + i,
    matrix_.begin() + s_size * (i + 1), new_matrix.begin()
    + s_size * i);
    for (int j = s_size * i; j < s_size * i + i; ++j) {
      new_matrix[j + s_size - i] = matrix_[j];
    }
  }
  matrix_ = std::move(new_matrix);

CanonMatrix CanonMatrix::MultiplicateMatrix(
const CanonMatrix& canon_matrix, size_t offset) {
```

```
  std::vector<double> c_matrix(size_);
  const auto& b_matrix = canon_matrix.GetMatrix();
  for (size_t i = 0; i < sqrt_size_; ++i) {
    for (size_t j = 0; j < sqrt_size_; ++j) {
      c_matrix[(i * sqrt_size_) + j] =
      matrix_[GetRowIndex((i * sqrt_size_) +
      j + offset, i + 1)] *
     b_matrix[GetColumnIndex((j * sqrt_size_) + i,
     j + 1, offset)];
    }
  }
  return {c_matrix};
}

void CanonMatrix::operator+=(const CanonMatrix& canon_matrix) {
  if (matrix_.empty()) {
    sqrt_size_ = canon_matrix.GetSqrtSize();
    size_ = canon_matrix.GetSize();
    matrix_.resize(size_);
  }
  for (size_t i = 0; i != sqrt_size_; ++i) {
    for (size_t j = 0; j != sqrt_size_; ++j) {
      matrix_[(i * sqrt_size_) + j] +=
      canon_matrix.GetMatrix()[(j * sqrt_size_) + i];
    }
  }
}
void CanonMatrix::Transpose() {
  std::vector<double> new_matrix(size_);
  for (size_t i = 0; i < sqrt_size_; ++i) {
    for (size_t j = 0; j < sqrt_size_; ++j) {
      new_matrix[(i * sqrt_size_) + j] =
      matrix_[(j * sqrt_size_) + i];
    }
  }
  matrix_ = std::move(new_matrix);
}
}
```

Функции RunImpl, соответствующие каждой версии:

```
bool sarafanov_m_canon_mat_mul_seq::CanonMatMulSequential
::RunImpl() {
  std::vector<CanonMatrix> mul_results(a_matrix_.GetSize());
  for (size_t i = 0; i < a_matrix_.GetSize(); ++i) {
    mul_results[i] = a_matrix_.MultiplicateMatrix
    (b_matrix_, i);
  }
  for (auto i = 0; i < static_cast<int>
  (mul_results.size()); ++i) {
    c_matrix_ = c_matrix_ + mul_results[i];
  }
  c_matrix_.Transpose();
  return true;

bool sarafanov_m_canon_mat_mul_omp::CanonMatMulOMP::
RunImpl() {
  c_matrix_.ClearMatrix();
  std::vector<CanonMatrix> mul_results(a_matrix_.GetSize());
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < static_cast<int>
    (a_matrix_.GetSize()); ++i) {
      mul_results[i] =
      a_matrix_.MultiplicateMatrix(b_matrix_, i);
    }
  }
  for (auto &it : mul_results) {
    c_matrix_ += it;
  }
  c_matrix_.Transpose();
  return true;
}

bool sarafanov_m_canon_mat_mul_tbb::CanonMatMulTBB::RunImpl() {
  c_matrix_.ClearMatrix();
```

```
  std::vector<CanonMatrix> mul_results
  (ppc::util::GetPPCNumThreads());
  oneapi::tbb::task_arena arena(ppc::util::GetPPCNumThreads());
  arena.execute([&] {
    oneapi::tbb::parallel_for(
        oneapi::tbb::blocked_range<size_t>
        (0, a_matrix_.GetSize(), a_matrix_.GetSize() /
        ppc::util::GetPPCNumThreads()),
        [&](const oneapi::tbb::
        blocked_range<size_t> &distance) {
          for (auto i = distance.begin();
          i != distance.end(); ++i) {
        mul_results[tbb::this_task_arena::
        current_thread_index()] +=
        a_matrix_.MultiplicateMatrix(b_matrix_, i);
          }
        });
  });
  std::ranges::for_each(mul_results, [&](const auto &vector) {
    if (!vector.IsEmpty()) {
      c_matrix_ += vector;
    }
  });
  return true;
}
}

bool sarafanov_m_canon_mat_mul_stl::CanonMatMulOMP::RunImpl() {
  c_matrix_.ClearMatrix();
  std::vector<CanonMatrix> mul_results
  (ppc::util::GetPPCNumThreads());
  std::vector<std::thread>
  threads(ppc::util::GetPPCNumThreads());
  ThreadDataAmount data_amount
  (ppc::util::GetPPCNumThreads(),
  static_cast<int>(a_matrix_.GetSize()));
  auto multiplicator = [&](int start_index, int end_index,
  size_t thread_index) {
```

```
    for (int i = start_index; i < end_index; ++i) {
      mul_results[thread_index]
      += a_matrix_.MultiplicateMatrix(b_matrix_, i);
    }
  };
  for (size_t i = 0; i < threads.size(); ++i) {
    threads[i] =
        std::thread(multiplicator, i * data_amount.step_size,
            i == threads.size() - 1 ? ((i + 1)
            * data_amount.step_size) +
            data_amount.residual_data_amount
            : (i + 1) * data_amount.step_size,
                    i);
  }
  for (auto &thread : threads) {
    thread.join();
  }
  std::ranges::for_each(mul_results, [&](const auto &vector) {
    if (!vector.IsEmpty()) {
      c_matrix_ += vector;
    }
  });
  return true;
}

bool sarafanov_m_canon_mat_mul_all::CanonMatMulALL::RunImpl() {
  c_matrix_.ClearMatrix();
  boost::mpi::broadcast(world_, a_matrix_, 0);
  boost::mpi::broadcast(world_, b_matrix_, 0);
  boost::mpi::broadcast(world_, indexes_, 0);
  std::vector<CanonMatrix> mul_results
  (ppc::util::GetPPCNumThreads());
#pragma omp parallel
  {
#pragma omp for
    for (int i = indexes_[world_.rank()].first; i <
    indexes_[world_.rank()].second; ++i) {
      mul_results[omp_get_thread_num()] +=
```

```
      a_matrix_.MultiplicateMatrix(b_matrix_, i);
    }
  }
  std::ranges::for_each(mul_results, [&](auto &matrix) {
    if (!matrix.IsEmpty()) {
      matrix_sum_on_process_ += matrix;
    }
  });
  if (matrix_sum_on_process_.IsEmpty()) {
    matrix_sum_on_process_.SetBaseMatrix(std::vector<double>
    (a_matrix_.GetSize()));
  }
  if (world_.rank() == 0) {
    std::vector<double> intermediate_values(a_matrix_.GetSize() *
    world_.size());
    auto sizes = std::vector<int>(world_.size(), static_cast<int>
    (a_matrix_.GetSize()));
    boost::mpi::gatherv(world_, matrix_sum_on_process_.GetMatrix(),
    intermediate_values.data(), sizes, 0);
    std::vector<double> answer(a_matrix_.GetSize());

    for (int i = 0; i < static_cast<int>(answer.size()); ++i) {
      for (int j = 0; j < world_.size(); ++j) {
        answer[i] += intermediate_values[i +
        (j * a_matrix_.GetSize())];

      }
    }
    c_matrix_.SetBaseMatrix(std::move(answer));
  } else {
    boost::mpi::gatherv(world_, matrix_sum_on_process_.GetMatrix(), 0);
  }
  return true;
}
```