МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«**Вятский государственный университет**»

**(«ВятГУ»)**

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

Отчет по лабораторной работе №8

по дисциплине «Высокопроизводительные вычислительные комплексы»

Выполнил студент группы ИВТ-42 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Щесняк Д. С./

Проверил старший преподаватель кафедры ЭВМ\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Вожегов Д.В./

Киров 2017

1. Цель работы

Цель данной лабораторной работы состоит в получении практических навыков выполнения параллельных программ на кластере.

1. Выполнение лабораторной работы

В качестве распараллеливаемой вычислительной задачи в данной лабораторной работе используется вычисление евклидового расстояния между изображениями. Данный алгоритм активно используется в поисковых системах для поиска схожих изображений, а так же используется в таких алгоритмах машинного обучения как К ближайших соседей.

Исходными данными задачи является матрица из изображений, которые представлены в виде вектора длиной 784 (28\*28) и вектор изображения, для которого необходимо выполнить поиск, имеющий размерность 784 (28\*28).

Расстояния между двумя векторами одинаковой размерности вычисляется по следующей формуле:

Результатом работы программы является вектор чисел, который обозначает похожесть изображений c искомым изображением.

1. Исходный код

Исходный код программы представлен на рисунках 1 и 2.

|  |
| --- |
| #include "stdafx.h"  #include <cstdlib>  #include <iostream>  #include "Matrix.h"  #include <cmath>  struct Matrix  {  int lines;  int cols;  double\* vector;  double\*\* items;  };  Matrix::Matrix\* Matrix::empty(int lines, int cols) {  Matrix\* m = new Matrix();  m->lines = lines;  m->cols = cols;  m->vector = new double[lines \* cols];  m->items = new double\*[lines];  for (int l = 0; l < lines; l++) {  m->items[l] = &m->vector[l \* cols];  }  return m;  }  Matrix::Matrix \* Matrix::fromVector(int lines, int cols, double \* vector)  {  Matrix\* m = new Matrix();  m->lines = lines;  m->cols = cols;  m->vector = vector;  m->items = new double\*[lines];  for (int l = 0; l < lines; l++) {  m->items[l] = &m->vector[l \* cols];  }  return m;  }  Matrix::Matrix\* Matrix::createRandom(int lines, int cols) {  Matrix\* m = new Matrix();  m->lines = lines;  m->cols = cols;  //srand(17);  m->vector = new double[lines \* cols];  m->items = new double\*[lines];  for (int l = 0; l < lines; l++) {  m->items[l] = &m->vector[l \* cols];  for (int c = 0; c < cols; c++) {  //m->items[l][c] = (((double) rand() / RAND\_MAX)) \* 100;  m->items[l][c] = rand() % 10;  }  }  return m;  }  void Matrix::freeMatrix(Matrix \* m) {  for (int l = 0; l < m->lines; l++) {  delete m->items[l];  }  delete m;  }  void Matrix::showMatrix(Matrix \* m) {  for (int i = 0; i < m->lines; i++) {  for (int j = 0; j < m->cols; j++) {  std::cout << m->items[i][j] << '\t';  }  std::cout << std::endl;  }  std::cout << std::endl;  }  bool Matrix::cmpMatrix(Matrix \* a, Matrix \* b)  {  if ((a->lines != b->lines) || (a->cols != b->cols)) return false;  for (int i = 0; i < a->lines; i++) {  for (int j = 0; j < a->cols; j++) {  if (a->items[i][j] != b->items[i][j]) return false;  }  }  return true;  }  Matrix::Matrix\* Matrix::distance(Matrix\* a, Matrix\*b) {  if (a->cols != b->cols) return NULL;  Matrix\* m = empty(a->lines, 1);  for (int i = 0; i < a->lines; i++) {  m->items[i][0] = 0;  for (int j = 0; j < a->cols; j++) {  m->items[i][0] += (a->items[i][j] - b->items[i][j]) << 1;  }  m->items[i][0] = sqrt(m->items[i][0]);  }  return m;  } |

Рисунок 1 – Модуль Matrix

|  |
| --- |
| #include "stdafx.h"  #include <cstdlib>  #include <mpi.h>  #include "Matrix.h"  #include <iostream>  #include <ctime>  const int B\_LINES = 1;  const int B\_COLS = 784;  const int A\_LINES = 10000000;  const int A\_COLS = 784;  int main(int argc, char \*\*argv)  {  int numprocs, rank;  MPI\_Init(&argc, &argv);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &numprocs);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  Matrix::Matrix\* b = NULL;  Matrix::Matrix\* a = NULL;  double\* asend = NULL;  double\* buf = new double[B\_LINES\*B\_COLS];  double\* abuf = new double[A\_LINES\*A\_COLS];  double\* resBuf = NULL;  if (rank == 0) {  a = Matrix::createRandom(A\_LINES, A\_COLS);  b = Matrix::createRandom(B\_LINES, B\_COLS);  asend = a->vector;  buf = b->vector;  resBuf = new double[A\_LINES \* B\_COLS];  }  time\_t start = clock();  MPI\_Bcast(buf, B\_LINES \* B\_COLS, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);  b = Matrix::fromVector(B\_LINES, B\_COLS, buf);  MPI\_Scatter(asend, A\_COLS \* A\_LINES / numprocs, MPI\_DOUBLE, abuf, A\_COLS \* A\_LINES / numprocs, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);  a = Matrix::fromVector(A\_LINES / numprocs, A\_COLS, abuf);  Matrix::Matrix \* res = Matrix::distance(a, b);  MPI\_Gather(res->vector, res->lines \* res->cols, MPI\_DOUBLE, resBuf, res->lines \* res->cols, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);  if (rank == 0) {  std::cout << "time = " << (double) (clock() - start) / 1000 << " sec";  }  MPI\_Finalize();  return 0;  } |

Рисунок 2 – Основной код программы

1. Результаты выполнения

В ходе выполнения лабораторной работы, написанная программа запускалась на различном числе ядер и для различного количества чисел. Результаты экспериментов представлены в таблицах 1 и 2. Время рассчитывается как среднее при трех запусках программы. На рисунках 1 и 2 представлены графики зависимости времени выполнения от количества потоков, и зависимость времени выполнения от размера данных.

Таблица 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Количество строк матрице | Время выполнения задачи в секундах при количестве ядер равным N | | | | | | |
| N=2 | N=4 | N=5 | N=8 | N=9 | | N=15 |
| 10000000 | 26.04 | 12.76 | 10.15 | 6.35 | 5.64 | | 3.38 |
| 15000000 | 38.75 | 19.08 | 15.27 | 9.52 | 8.45 | | 5.07 |
| 20000000 | 50.83 | 25.37 | 20.30 | 12.69 | 11.27 | | 6.76 |
| 25000000 | 63.79 | 31.78 | 25.37 | 15.84 | 14.09 | | 8.45 |
|  |  |  |  |  |  | | |
|  | N=16 | N=20 | N=24 | N=27 | N=30 | N=32 | |
| 10000000 | 3.17 | 2.54 | 2.11 | 1.88 | 1.69 | 1.58 | |
| 15000000 | 4.75 | 3.80 | 3.17 | 2.82 | 2.53 | 2.38 | |
| 20000000 | 6.34 | 5.07 | 4.22 | 3.75 | 3.38 | 3.17 | |
| 25000000 | 7.92 | 6.34 | 5.28 | 4.69 | 4.22 | 3.96 | |

Найдем ускорение вычислений при увеличении ядер по следующей формуле:

,

где tNi – время решения при текущем количестве ядер;

tNi+1 – время решения при увеличенном количестве ядер;

a(i,i+1) – ускорение времени решения при текущем количестве ядер относительно времени решения при увеличенном количестве ядер.

Таблица 2

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Количество строк в матрице | a(2,4), % | a(4,5), % | a(5,8), % | | a(8,9), % | | а(9,15), % | | | a(15,16), % |
| 10000000 | 50.99846 | 20.45455 | | 37.43842 | | 11.1811024 | | 40.07092 | | 6.213018 |
| 15000000 | 50.76129 | 19.96855 | | 37.65553 | | 11.2394958 | | 40 | | 6.311637 |
| 20000000 | 50.08853 | 19.98423 | | 37.48768 | | 11.1899133 | | 40.01775 | | 6.213018 |
| 25000000 | 50.18028 | 20.16992 | | 37.56405 | | 11.0479798 | | 40.02839 | | 6.272189 |
|  |  |  | |  | |  | |  | | |
|  | a(16,20), % | a(20,24), % | | a(24,27), % | | a(27,30), % | | a(30,32),% |
| 10000000 | 19.87382 | 16.92913 | | 10.90047 | | 10.10638 | | 6.508876 |
| 15000000 | 20 | 16.57895 | | 11.04101 | | 10.28369 | | 5.928854 |
| 20000000 | 20.03155 | 16.76529 | | 11.13744 | | 9.866667 | | 6.213018 |
| 25000000 | 19.94949 | 16.71924 | | 11.17424 | | 10.02132 | | 6.161137 |

Рисунок 1 – Зависимость времени вычисления от числа потоков

Рисунок 2 – Зависимость времени выполнения от размера данных

1. Вывод

Анализируя представленные в таблицах 1 и 2 данные и рисунки 1 и 2, можно сделать следующие выводы:

При неизменном количестве данных и увеличении числа используемых ядер время решения задачи нелинейно уменьшается:

при неизменном числе используемых ядер и увеличении количества данных время решения увеличивается практически линейно, поскольку каждому ядру достается большая часть задачи и эффект от параллелизма снижается;

если сравнить результаты решения задачи с количеством чисел, равным 20000000 при использовании 16 ядер и результаты решения задачи при 10000000 чисел и количестве ядер, равном 8, можно заметить, что время выполнения практически одинаковое (6.35 и 6.34 секунды соответственно). Исходя из этого, можно сделать вывод, что масштабируемость задачи является хорошей;

с увеличением числа ядер от двух до четырех достигается максимальное ускорение вычислений (при расчетах на четырех ядрах алгоритм работает примерно в два раза быстрее, чем на двух – ускорение в n раз, где n – число ядер). При использовании от 5 до 8 ядер (вычисления внутри одного блейда) ускорение также значительное, но немного меньше, чем в n раз. При использовании данного алгоритма объем пересылаемых данных между процессами небольшой, а внутри одного блейда данные передаются быстро. Этим можно объяснить хорошее распараллеливание алгоритма при числе ядер меньшем восьми;

при решении задачи на 9 и более ядрах происходит уменьшение ускорения параллельного алгоритма. Причем при числе данных график, представленный на рисунке 1, становится практически параллельным оси абсцисс. Это можно объяснить увеличением времени передачи данных между процессами при увеличении числа ядер. Таким образом, при числе ядер больше восьми, ускорение будет меньше, чем n из-за задержек при передаче информации между процессами.