Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

**Отчет по лабораторной работе**

Дисциплина «Параллельные вычисления»

«Разработка программ с использованием библиотек pthread и OpenMP»

Выполнил

студент гр. 13541/2 Е.А. Никитин

Преподаватель И.В. Стручков

«\_\_»\_\_\_\_\_\_\_\_\_2017г.

Санкт-Петербург

2017

Цель работы

Ознакомление с разработкой многопоточных программ, в частности, ознакомление с работой библиотек pthread и OpenMP. Сделать общие выводы по результат проделанной работы.

Постановка задачи

Задание №8 «Расчет определителя матрицы».

Выполнить задачу необходимо сначала последовательной программой, затем с использованием библиотек pthread и OpenMP.

Ход работы

Задание будет выполняться на компьютере с процессором Intel Core i-5 2450M 2.5GHz – это 2-ядерный процессор с поддержкой технологии hyper-threading, т.е. содержит 4 логических потока. Использовался язык С++.

Основными методами расчета определителя матриц являются метод разложения по строке (столбцу) и метод Гаусса (приведение к треугольному виду). Для реализации был выбран метод разложения по строке.

Была написана функция matrixDet, которая рекурсивно рассчитывает определитель квадратной матрицы раскладывая по строке (также используется вспомогательная функция getMatrixWithoutRowAndCol, которая возвращает матрицу на 1 размер меньше – для рассчета миноров). Функции представлена в Листинге 1.

**Листинг 1. Однопоточное решение задачи**

|  |
| --- |
| double matrixDet(double \*\*matrix, int size) {  double det = 0.0;  int degree = 1;    //Условие выхода из рекурсии  if(size == 1) {  return matrix[0][0];  }  //Условие выхода из рекурсии  else if(size == 2) {  return matrix[0][0]\*matrix[1][1] - matrix[0][1]\*matrix[1][0];  }  else {  //Матрица без строки и столбца  double \*\*newMatrix = new double\*[size-1];  for(int i = 0; i < size-1; i++) {  newMatrix[i] = new double[size-1];  }    for(int j = 0; j < size; j++) {  degree = pow((double)-1.0, (double)(j));  getMatrixWithoutRowAndCol(matrix, size, 0, j, newMatrix);  det += (degree \* matrix[0][j] \* matrixDet(newMatrix, size-1));  }  for(int i = 0; i < size-1; i++) {  delete [] newMatrix[i];  }  delete [] newMatrix;  }    return det;  }  void getMatrixWithoutRowAndCol(double \*\*matrix, int size, int row, int col, double \*\*newMatrix) {  int offsetRow = 0; //Смещение индекса строки в матрице  int offsetCol = 0; //Смещение индекса столбца в матрице  int i, j;    //#pragma omp parallel for private(i, j, offsetRow, offsetCol) shared (newMatrix, matrix) //работает, но делает хуже по времени  for( i = 0; i < size-1; i++) {  //Пропустить row-ую строку  if(i == row) {  offsetRow = 1; //Как только встретили строку, которую надо пропустить, делаем смещение для исходной матрицы  }    offsetCol = 0; //Обнулить смещение столбца  for(j = 0; j < size-1; j++) {  //Пропустить col-ый столбец  if(j == col) {  offsetCol = 1; //Встретили нужный столбец, проускаем его смещением  }    newMatrix[i][j] = matrix[i + offsetRow][j + offsetCol];  }  }  } |

Многопоточная реализация задачи с использованием библиотеки pthread

Для многопоточной реализации программа была разбита условно на 2 части. Первая часть – запуск дочерних потоков и передача им параметров. Данное действие запускается функцией start\_decomposition\_threads. В каждом потоке будет запускаться свой рекурсивный поиск определителя с помощью функции matrixDet, реализованной ранее, затем результаты будут сложены в окончательный ответ.

Функция matrixDet запускается с помощью передаваемой в поток специальной функции decomp\_func, параметром которой является указатель на специальный аргумент. Аргумент представляет собой структуру, содержащую следующие данные: указатель на матрицу; указатель на переменную определителя; размер матрицы; начальное и конечное значение итераций цикла – начальный и конечный номер столбцов, которые нужно удалять и считать минор; а также необязательный аргумент – номер потока.

Каждый поток будет отвечать за формирование миноров, соответствующих значениям параметров, переданных им. Суть разбиения такова: каждому потоку дается начальный и конечный столбец, после чего каждый поток циклически убирая один из столбцов своего отрезка начинает подсчет миноров.

В конце достаточно просто сложить полученные результаты, чтобы получить результат.

Код функций и аргумента представлен в листинге 2.

Листинг 2. Многопоточная реализация с использованием библиотеки pthread.

|  |
| --- |
| //аргумент для функции в поток  typedef struct funcArg{  double \*\*matrix;  int size;  double \*det;  int j;  int end;  int thread\_num;  } funcArg\_t;  void \*decomp\_func(void \*args){  funcArg\_t \*arg = (funcArg\_t\*) args;  int size = arg->size;  int degree = 1;  int j = arg->j;  int end = arg->end;  //Матрица без строки и столбца  double \*\*newMatrix = new double\*[size-1];  for(int i = 0; i < size-1; i++) {  newMatrix[i] = new double[size-1];  }  double tmp = 0.0;  for(j; j < end; j++) {  degree = pow((double)-1.0, (double)(j));  getMatrixWithoutRowAndCol(arg->matrix, size, 0, j, newMatrix);  tmp += (degree \* arg->matrix[0][j] \* matrixDet(newMatrix, size-1));  \*arg->det = tmp;  }  for(int i = 0; i < size-1; i++) {  delete [] newMatrix[i];  }  delete [] newMatrix;  return NULL;  }  double start\_decomposition\_threads(double \*\*matrix, int size) {  double det = 0.0;  //детерминант для каждого потока  double dets[NUM\_THREADS];  for (int i=0; i < NUM\_THREADS; i++){  dets[i] = 0.0;  }  //потоки  int status\_addr[NUM\_THREADS];  pthread\_t thread[NUM\_THREADS];  //аргументы для потоков  funcArg\_t args[NUM\_THREADS];  int degree = 1;    if(size == 1) {  return matrix[0][0];  }  else if(size == 2) {  return matrix[0][0]\*matrix[1][1] - matrix[0][1]\*matrix[1][0];  }  else {  //заполнение содержимого аргументов  //cout << "Size > 2 MAIN THREAD START" << endl;  for(int th\_i=0; th\_i<NUM\_THREADS; th\_i++){  args[th\_i].matrix = matrix;  args[th\_i].size = size;  args[th\_i].det = &dets[th\_i];  args[th\_i].thread\_num = th\_i;  if(th\_i==0) args[th\_i].j = 0;  else args[th\_i].j = (size/NUM\_THREADS)\*th\_i;  if(th\_i==NUM\_THREADS-1) args[th\_i].end = size;  else args[th\_i].end = (size/NUM\_THREADS)\*(th\_i+1);  }  //cout << "Threads creating ... " << endl;  //создание потоков  for(int th\_i=0; th\_i < NUM\_THREADS; th\_i++)  pthread\_create(&thread[th\_i], NULL, decomp\_func, (void\*) &args[th\_i]);  //cout << "... Done. Threads executing" << endl;  for(int th\_i=0; th\_i < NUM\_THREADS; th\_i++)  int status = pthread\_join(thread[th\_i], (void\*\*)&status\_addr[th\_i]);    //сумма  //cout << "Add determinant" << endl;  for(int th\_i=0; th\_i < NUM\_THREADS; th\_i++)  det += \*args[th\_i].det;  }  return det;  } |

Многопоточная реализация с использованием библиотеки OpenMP

Данная реализация похожа на реализацию с помощью библиотеки pThread, в ней также программа условно разбивается на 2 части, однако формирование потоков будет создаваться директивами библиотеки OpenMP.

Первую часть запускает функция start\_decomposition\_OMP. За вторую часть ответственна ранее реализованная функция matrixDet, рекурсивно высчитывающая определитель.

Основной цикл будет разбит на потоки с помощью директивы #pragma omp parallel for. В качестве аргументов, описанных как private выступают переменная цикла и degree – степень (-1), в качестве общих аргументов выступают указатель на матрицу, массив со значениями определителя каждого потока и указатели на новую матрицу меньшего размера (у каждого потока своя матрица).

Каждый поток будет использовать свою матрицу и свою переменную-определитель, для безошибочного доступа была использована функция omp\_get\_thread\_num, которая возвращает номер потока; таким образом поток будет обращаться только к ячейке и матрице своего номера.

Код многопоточной реализации с использованием библиотеки OpenMP представлен в листинге 3.

Листинг 3. Многопоточная реализация с использованием библиотеки OpenMP.

|  |
| --- |
| double start\_decomposition\_OMP(double \*\*matrix, int size) {  double det = 0.0;  int degree = 1;    //Условие выхода из рекурсии  if(size == 1) {  return matrix[0][0];  }  //Условие выхода из рекурсии  else if(size == 2) {  return matrix[0][0]\*matrix[1][1] - matrix[0][1]\*matrix[1][0];  }  else {    //3-мерная матрица, для каждого потока своя 2-мерная матрица  double \*\*\*newMatrix = new double\*\*[NUM\_THREADS];  for(int i = 0; i < NUM\_THREADS; i++) {  newMatrix[i] = new double\*[size-1];  }  for(int i = 0; i < NUM\_THREADS; i++) {  for(int j = 0; j < size-1; j++)  newMatrix[i][j] = new double[size-1];  }    //для каждого потока свой детерминант  double det\_m[NUM\_THREADS];  for(int i=0; i<NUM\_THREADS; ++i)  det\_m[i] = 0;    int j;  #pragma omp parallel for private(degree, j) shared(matrix, det\_m, newMatrix)  for(j = 0; j < size; j++) {  int th\_i = omp\_get\_thread\_num(); //номер потока для индекса  degree = pow((double)-1.0, (double)(j));  getMatrixWithoutRowAndCol(matrix, size, 0, j, newMatrix[th\_i]);  det\_m[th\_i] += (degree \* matrix[0][j] \* matrixDet(newMatrix[th\_i], size-1)); }  //подсчет итогового детерминанта  for(int i=0; i<NUM\_THREADS; ++i)  det += det\_m[i];  //удаление 3-мерной матрицы  for(int i = 0; i < NUM\_THREADS; i++) {  for(int j = 0; j < size-1; j++)  delete [] newMatrix[i][j];  }  for(int i=0; i<NUM\_THREADS; ++i)  delete [] newMatrix[i];  delete [] newMatrix;  }    return det;  } |

Расчет вероятностных характеристик

Для расчета вероятностных характеристик был выполнен запуск функций каждого вида по 100 раз. Тестирование проводилось матрице 11 х 11 элементов (небольшой размер матрицы вызван тем, что в отличие от алгоритма Гаусса, рекурсивный алгоритм работает в разы медленнее, например, при увеличении размера квадратной матрицы на 1, время расчёта увеличивается примерно в 10 раз). На основе полученных данных подсчитывается математическое ожидание, дисперсия и доверительных интервал. Данные представлены в таблице 1.

**Таблица 1. Таблица результатов для 100 запусков каждой реализации.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Реализация | Число потоков | Мат. ожидание | Дисперсия | Доверительный интервал 0.95% |
| Single Thread | 1 | 335 | 10.59 | [335.009 – 335.791] |
| pThread | 1 | 340 | 17.6 | [339.747 – 341.053] |
|  | **2** | 207 | 39.06 | [206.359 – 209.241] |
|  | 4 | 219 | 14.9 | [219.431– 220.369] |
|  | 8 | 226 | 38.34 | [225.694 – 228.106] |
| OpenMP | 1 | 338 | 4.34 | [338.529– 339.271] |
|  | 2 | 206 | 37.26 | [205.46 – 207.74] |
|  | **4** | 176 | 7.06 | [175.841 – 176.759] |
|  | 8 | 186 | 34.32 | [184.986 – 187.814] |

Полученные данные показывают, что реализация с помощью библиотек pThread и OpenMP улучшили показатели времени расчета определителя матрицы. Наилучшие результаты библиотека pThread дала при 2 потоках; библиотека OpenMP при 4 потоках. Однако, в дополнение к этому был проведен дополнительный тест на матрице 12 х 12. Число 12 нацело делится на 4, поэтому на 4 потоках результаты работы программ на pThread и OpenMP были практически одинаковы: 20.290 сек и 20.162 сек соответственно. Поэтому можно предположить, что библиотека OpenMP лучше организует распределение вычислений по потокам, когда количество итераций разбиваемого цикла не кратно количеству потоков, соответственно приносит лучший результат.

Вывод

В ходе работы создано три реализации функции нахождения определителя матрицы на языке С++. Данные функции позволяют находить определитель любой матрицы, размер задается параметром, однако данный алгоритм работает очень медленно, увеличивая время расчета примерно в 10 раз при увеличении матрицы на 1.

В ходе работы изучены и использованы библиотеки многопоточного программирования pthread и OpenMP. Реализованные на их основе решения, могут легко изменять количество исполняющих потоков. Однако, не всегда многопоточность позволяет увеличить скорость работы, это зависит от конкретной задачи и количества ядер и логических потоков процессора.

Полученные реализации протестированы на матрице и проведен анализ вероятностных характеристик. В результате экспериментов установлено, что наиболее эффективным оказалось использование библиотеки OpenMP на 4 потоках, которая позволила увеличить скорость работы, причем явное распараллеливание с помощью pThread дало наилучший результат на 2 потоках. Библиотека OpenMP в данном случае дает лучший результат, т.к. при одинаковом количестве потоков показывает такой же или лучший результат в сравнении с библиотекой pThread.