ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

КУРСОВОЙ ПРОЕКТ

по дисциплине “Параллельные вычислительные технологии”

на тему

Разработка параллельной MPI-программы вычисления определителя матрицы методом Гаусса

Выполнил студент Романюта Алексей Андреевич
Ф.И.О.

Группы ИВ – 521

Работу принял _____ доцент д.т.н. М.Г. Курносов
подпись

Защищена _____ Оценка _____

Новосибирск – 2017

СОДЕРЖАНИЕ

Введение

Реализовать параллельный алгоритм нахождения определителя матрицы методом Гаусса для систем с распределенной памятью используя средства стандарта MPI.

Исследовать коэффициент ускорения времени работы программы, используя средства стандарта MPI.

1. Метод Гаусса для нахождения определителя матрицы

Метод Гаусса для нахождения определителя матрицы похож на решение СЛАУ, за исключением того, что отсутствует «Обратный ход» и применим он СТРОГО к квадратным матрицам, т. к. понятие «определитель» присуще только им. Зато «Прямой ход» полностью идентичен методу решения СЛАУ.

Для нахождения определителя матрицы, необходимо так же, как и в методе решения СЛАУ, привести исходную матрицу к «Треугольной», а затем, поскольку у «треугольных» матриц все элементы , стоящие под главной диагональю равны 0, достаточно будет просто перемножить все элементы главной диагонали между собой. В результате получим определитель исходной матрицы [2].

2. Выполнение работы

2.1 Задачи

В данной работе поставленной задачей было написание кода, с использованием стандарта MPI, реализующего алгоритм нахождения определителя матрицы максимально эффективно, а так же, с хорошей масштабируемостью итоговой программы.

Основной проблемой решения поставленной задачи на начальном этапе было понимание работы алгоритмов и необходимых для их корректной работы, условий, а так же, понимание принципов работы коллективных операций обмена библиотеки MPI.

Немаловажным умением, которое приобрелось в результате выполнения данной работы было умение понять математический алгоритм до мельчайших деталей и затем грамотно, и эффективно его реализовать для работы на системах не только с общей, но и распределенной памятью.

2.2 Работа программы

Несколько основных аспектов:

1. Исходная матрица заполняется с помощью генератора псевдослучайных чисел.
2. Исходная матрица хранится в распределенном виде. У каждого процесса находится в памяти по несколько строк матрицы.
3. При расчетах каждый процесс обрабатывает строки с номерами rank + commsize * i, пока не превысит количество строк. Количество обрабатываемых строк определяется вызовом функции get_chunk. Это обеспечивает практически равную загрузку вычислениями всех процессов [1].

2.3 Процесс вычислений

Каждый процесс, в соответствии со своим номером, по порядку строк начинает преобразование исходной матрицы, приводя ее к «треугольной» матрице, процессом выполнения «прямого хода» [1].

$$\begin{matrix} a_{11} & a_{12} & \dots & a_{1(n-1)} & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2(n-1)} & a_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{(n-1)1} & a_{(n-1)2} & \dots & a_{(n-1)(n-1)} & a_{(n-1)n} \\ a_{n1} & a_{n2} & \dots & a_{n(n-1)} & a_{nn} \end{matrix}$$

Рис. 2.1. Исходный вид матрицы

$$\begin{matrix} a_{11} & a_{12} & \dots & a_{1(n-1)} & a_{1n} \\ 0 & a_{22} & \dots & a_{2(n-1)} & a_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & a_{(n-1)(n-1)} & a_{(n-1)n} \\ 0 & 0 & \dots & 0 & a_{nn} \end{matrix}$$

Рис. 2.2. Конечный вид матрицы

$$Det = \prod_{k=1}^{n} a_k$$

Формула 2.3. Определитель матрицы

После чего, вычисляет локально определитель (Для ***K*** из ***M*** строк, которыми владеет процесс выполняется операция перемножения элементов главной диагонали (Каждый процесс знает порядковые номера строк, которыми владеет, следовательно может вычислить позицию элемента главной диагонали), после чего выполняется отправка с помощью MPI_Send локального определителя root процессу (0 по умолчанию), который в свою очередь выполняет редукцию локальных определителей других процессов в свой определитель. Для этого root-процесс так же вычислит свой локальный определитель, но так же домножит его на локальные определители других процесссов, прянятых через MPI_Recv.

Таким образом, итоговый определитель будет храниться в памяти root-процесса [3].

3. Анализ вычислительной сложности алгоритма

Введем следующие обозначения:

- Размер матрицы: ***N***.
- Количество процессов: ***P***.
- Количество строк, которые обрабатывает процесс: ***K*.**

Каждый процесс обрабатывает ***K*** строк. К $= \frac{N}{P}$

Будем считать время выполнения коллективных операций MPI как $O(1)$.

Следовательно, вычислительная сложность *параллельного* алгоритма преобразования матрицы по времени равна:

$$F_1(N, P, K) = O((N-1) * K * N)$$

И так же вычислительная сложность вычисления определителя преобразованной матрицы:

$$F_2(N, P, K) = O(O(K))$$

Исходя из этого можем получить итоговую вычислительную сложность *параллельного* алгоритма по времени:

$$F(N, P, K) = O((N-1) * K * N + K)$$

4. Описание системы, на которой проводились эксперименты

4.1 Характеристики системы

Организация измерений времени работы программы была произведена на учебном вычислительном кластере jet.
Характеристики:

- Количество вычислительных узлов [4]: 18.
- Количество задействованных в вычислениях узлов: 13.
- Соединение между вычислительными узлами: Gigabit Ethernet (1 ГиБ/с).

Характеристики вычислительного узла [4]:

- ЦПУ: 2 x Intel Xeon E5420 2.5 ГГц.
- ОЗУ: 8 Гб DDR3.
- ОС: Fedora 20 x86_64.
- Компилятор: gcc 4.8.3.
- Реализация библиотеки MPI: MPICH 3.2.
- Модель многопоточности: posix.

4.2 Входные данные

Программа на вход получает следующие аргументы:

- Аргумент 1: Размерность матрицы
- Аргумент 2: Номер процесса, который вычислит итоговый результат (root).

5. Результаты экспериментов

Поскольку конфигурация вычислительного узла кластера позволяет использовать 8 GB оперативной памяти, итоговое ускорение было взято относительно не последовательного алгоритма, а параллельного, использующего загрузку памяти по максимуму (30650x30650 элементов типа double), выполненного на подсистеме: 1 выч. узел, 8 выч. ядер .

Таблица 5.1. Результаты экспериментов

Конфигурация подсистемы (Узлы x Процессы)	Время работы разработанного алгоритма, с
1 x 8	35897.736395
2 x 8	20382.150077
3 x 8	12060.333025
4 x 8	9670.462240
5 x 8	8265.015727
6 x 8	6928.367810
7 x 8	5903.236829
8 x 8	6465.189706
9 x 8	4655.926386
10 x 8	4218.389470
11 x 8	3839.606418
12 x 8	3545.239116
13 x 8	4432.845882

Таблица 5.2. Результаты вычисления эффективности работы программы

Отношение конфигураций подсистем к подсистеме, принятой за основу	Полученный коэффициент ускорения времени работы программы
2	1.7
3	2.9
4	3.7
5	4.3
6	5.2
7	6.1
8	5.6
9	7.8
10	8.5
11	9.4
12	10.1
13	8.1

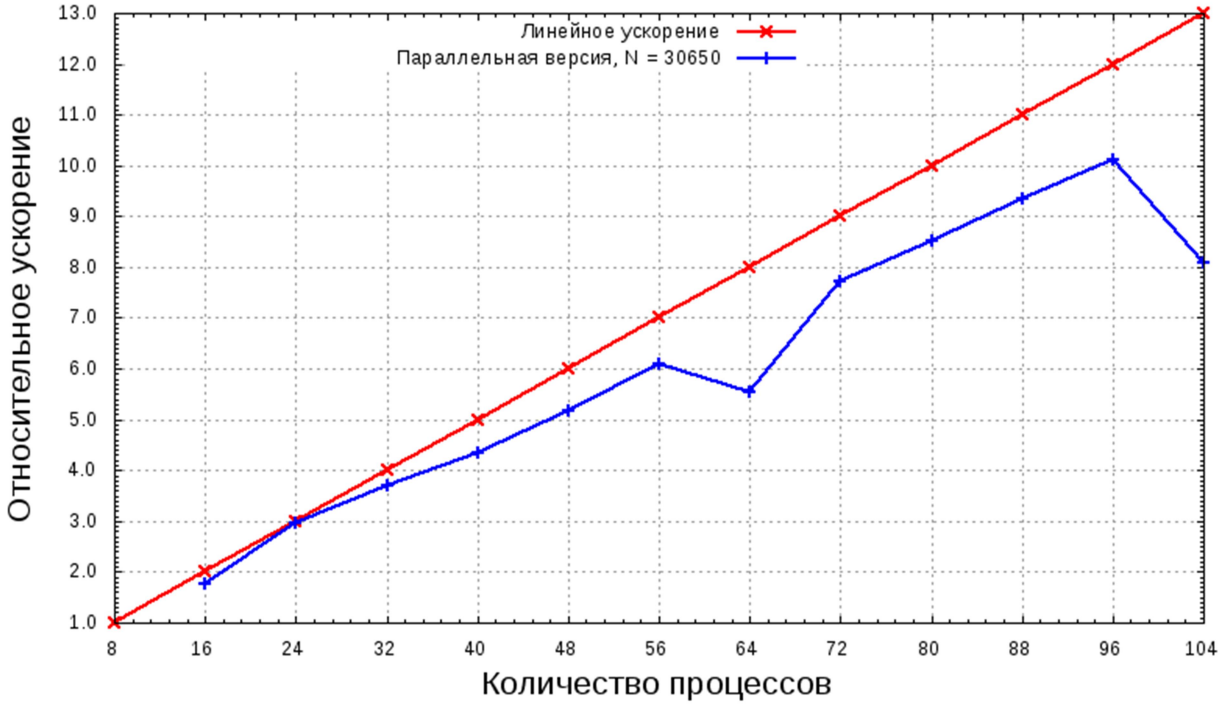

Рис. 5.3. График зависимости времени работы программы в зависимости от количества процессов (Размер матрицы : 30650x3060)

Показан график относительной зависимости коэффициента ускорения работы программы от количества процессов. За точку отсчета принята конфигурация: 1 вычислительный узел , 8 процессов.

Заключение

В результате проделанной работы, был исследован и смоделирован алгоритм нахождения определителя матрицы методом Гаусса для систем с распределенной памятью.

По итогам работы написанной программы на подсистемах различных конфигураций, можно однозначно сделать вывод о хорошей масштабируемости алгоритма, так как ускорение для времени работы программы, полученное в результате сравнения времён работы алгоритма на $\boldsymbol{M}$ процессах с $\boldsymbol{P}$ процессами ($M < P$), близко к линейному (Рис. 5.3)

Список использованной литературы

1. Курс Параллельных Вычислительных Технологий, СибГУТИ, Лекция «Параллельное решние СЛАУ», Курносов М.Г. , 2017г.

2. Ильин В. А., Позняк Э. Г. Линейная алгебра: Учебник для вузов. — 6-е изд., стер. — М.: ФИЗМАТЛИТ, 2004. — 280 с.

3. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1 // University of Tennessee, Knoxville, Tennessee, 2015.

4. Вычислительный кластер D (Jet) [Электронный ресурс]: // ЦПВТ ФГОБУ ВПО «СибГУТИ» 7 декабря 2017. - Электронно текстовые данные. - Режим доступа:
http://cpct.sibsutis.ru/index.php/Main/Jet.

*

Приложение

Исходный код программы

main.c

```
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>
#include <math.h>

int get_chunk(int total, int commsize, int rank)
{
  int n = total;
  int q = n / commsize;
  if (n % commsize)
    q++;
  int r = commsize * q - n;
  /* Compute chunk size for the process */
  int chunk = q;
  if (rank >= commsize - r)
    chunk = q - 1;

  return chunk;
}

int main(int argc, char *argv[])
{
  int n = argc > 1 ? atoi(argv[1]) : 3000;
  int rank, commsize;
  int root = argc > 2 ? atoi(argv[2]) : 0;
  MPI_Init(&argc, &argv);


  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &commsize);

  int nrows = get_chunk(n, commsize, rank);
  int *rows = malloc(sizeof(*rows) * nrows);

  double determinant = 1;
  double *a = malloc(sizeof(*a) * nrows * (n));
  double *tmp = malloc(sizeof(*tmp) * (n));
  for (int i = 0; i < nrows; i++) {
    rows[i] = rank + commsize * i;
```

```
    srand(rows[i] * n);
    for (int j = 0; j < n; j++)
      a[i * n + j] = rand() % 2 + 1;
}
  double t = MPI_Wtime();

  int row = 0;
  for (int i = 0; i < n - 1; i++) {
    if (i == rows[row]) {
    MPI_Bcast(&a[row * n], n, MPI_DOUBLE, rank,
MPI_COMM_WORLD);
    for (int j = 0; j <= n; j++)
      tmp[j] = a[row * n+ j];
    row++;
    } else {
      MPI_Bcast(tmp, n, MPI_DOUBLE, i % commsize,
MPI_COMM_WORLD);
    }
    for (int j = row; j < nrows; j++) {
      double scaling = a[j * n + i] / tmp[i];
      for (int k = i; k < n; k++)
        a[j * n+ k] -= scaling * tmp[k];
    }
  }
  //Determinant
  if (rank == root) {
    double locdet = 1;
    int row, j;
    for (j = rank, row = 0; row < nrows; j+=commsize, ++row) {
        determinant = determinant * a[row*n+ j];
    }

    for (int i = 0; i < commsize; i++){
      if(i == root) continue;
      MPI_Recv(&locdet, 1,  MPI_DOUBLE, i,0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      determinant = determinant * locdet;
    }

  } else {
    double locdet = 1;
    int row, j;
    for (j = rank, row = 0; row < nrows; j+=commsize, ++row) {
      locdet = locdet * a[row * n + j];
    }
    MPI_Send(&locdet, 1,  MPI_DOUBLE, root,0, MPI_COMM_WORLD);
```

```
  }

  t = MPI_Wtime() - t;

  free(tmp);
  free(rows);
  free(a);

  if (rank == root) {
    printf("Gaussian Determimant(MPI) : n %d, procs %d, time
(sec) %.6f\n",
    n, commsize, t);
  }
  MPI_Finalize();
  return 0;
}
```