МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В.Ломоносова

Факультет вычислительной математики и кибернетики

Практикум по курсу

"Суперкомпьютеры и параллельная обработка данных"

Разработка параллельной версии программы для перемножения матриц с использованием ленточного алгоритма

ОТЧЕТ

о выполненном задании

студентки 321 учебной группы факультета ВМК МГУ

Задорожной Юлии Андреевны

Москва
2021

Содержание

1 Постановка задачи

1. Реализовать параллельный алгоритм ленточного перемножения матриц с применением технологий OpenMP и MPI.
2. Исследовать зависимость времени выполнения программы от размера входных данных и числа используемых потоков.
3. Построить графики зависимости времени исполнения от числа потоков для различного объёма входных данных.
4. Сравнить эффективность OpenMP и MPI-версий параллельной программы.

2 Описание алгоритма и код программы

Параллельная версия реализации алгоритма Гаусса представлена ниже:

```c
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <mpi.h>

void prt1a(char *t1, double *v, int n, char *t2);

void print_matrix(double *a);

int N;
double *A;
#define A(i, j) A[(i) * (N + 1) + (j)]
double *X;
int proc_num, myrank;

int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &proc_num);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  int i, j, k;
  N = atoi(argv[1]);

  /* create arrays */
  A = (double *)malloc(N * (N + 1) * sizeof(double));
  X = (double *)malloc(N * sizeof(double));
  if (myrank == 0) printf("GAUSS %dx%d\n---------------------------------\n", N, N);

  srand(12345);
  /* initialize array A*/
  for (i = 0; i <= N - 1; i++)
    for (j = 0; j <= N; j++)
      if (i == j || j == N)
        A(i, j) = 1.f;
      else
        A(i, j) = 0.f;

  double time0 = MPI_Wtime();
  /* elimination */
  for (i = 0; i < N - 1; i++)
  {
    MPI_Bcast(&A(i, i + 1), N - i, MPI_DOUBLE, i % proc_num, MPI_COMM_WORLD);
    for (k = myrank; k <= N - 1; k += proc_num) {
      if (k < i + 1) {
        continue;
      }
      for (j = i + 1; j <= N; j++) {
        A(k, j) = A(k, j) - A(k, i) * A(i, j) / A(i, i);
      }
```

```
    }
  }
  /* reverse substitution */
  X[N - 1] = A(N - 1, N) / A(N - 1, N - 1);
  for (j = N - 2; j >= 0; j--)
  {
    for (k = myrank; k <= j; k += proc_num)
      A(k, N) = A(k, N) - A(k, j + 1) * X[j + 1];
    MPI_Bcast(&A(j, N), 1, MPI_DOUBLE, j % proc_num, MPI_COMM_WORLD);
    X[j] = A(j, N) / A(j, j);
  }
  double time1 = MPI_Wtime();

  if (myrank == 0) {
    printf("Time in seconds=%gs\n", time1 - time0);
    prt1a("X=(", X, N > 9 ? 9 : N, "...)\n");
  }
  free(A);
  free(X);
  return 0;
}

void prt1a(char *t1, double *v, int n, char *t2)
{
  int j;
  printf("%s", t1);
  for (j = 0; j < n; j++)
    printf("%.4g%s", v[j], j % 10 == 9 ? "\n" : ", ");
  printf("%s", t2);
}

void print_matrix(double* a) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N + 1; j++)
            printf("%lf ", A(i, j));
        printf("\n");
    }
    printf("\n");
}
```

Изменения заключаются в работе циклов, цикл с MPIBcast(A(i, i + 1), N - i, MPI DOUBLE, i proc num, MPI COMM WORLD); Процесс с номером m работает по строкам, номера которых при делении на n в остатке дают m. Чтобы посчитать выражение на i-м шаге, нам нужна k-я и i-я строки. Поэтому когда мы в каком-то процессе считываем строку с номером I, мы пересылаем её всем остальным процессам, пока остальные ждут. Второй цикл аналогичен, но он пересылает A(j,n)

Параллельный алгоритм

Из определения метод Гаусса следует, что он реккурентно, итеративно исключает переменные с помощью элементарных преобразований. Для уменьшения времени выполнения программы была использована попытка распараллелить все из имеющихся циклов. Поскольку ни один из циклов не содер- жит зависимостей, это можно было сделать. Для решения возникших сложностей алгоритма из-за цикла был применен алгоритм разбиения матрицы на строки, с последующим вычислением каждой новой строки и всех вместе, а также был изменен порядок обхода матрицы с обхода по столбцам на обход по строкам. При таком подходе достигнутый уровень параллелизма является в большинстве случаев избы- точным. Обычно при проведении практических расчетов такое количество сформированных под- задач превышает число имеющихся процессоров и делает неизбежным этап укрупнения базовых задач. В этом плане может оказаться полезной агрегация вычислений уже на шаге выделения ба- зовых подзадач. Такой подход приводит к снижению общего количества подзадач до величины N.

MPI-версия

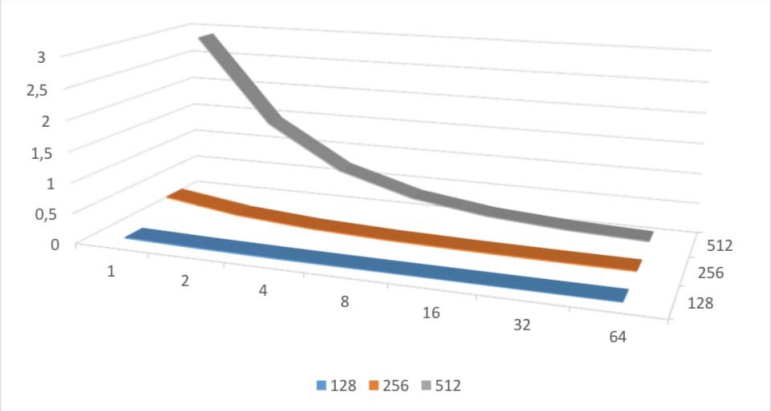
Модификация кода сводится к добавлению нескольких клауз.

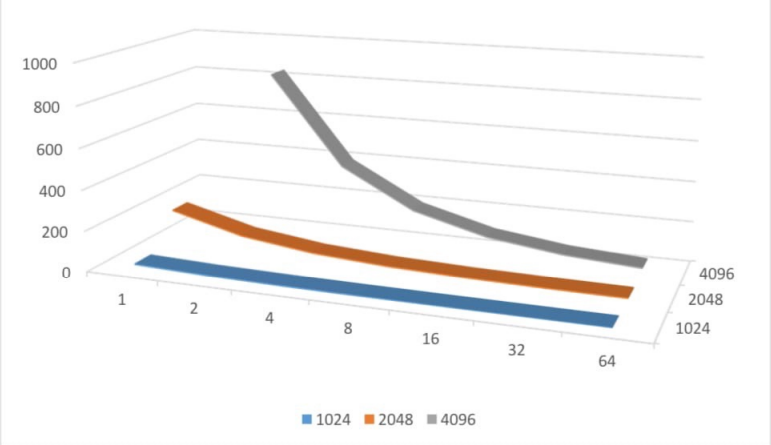
```
    /* initialize array A*/
  for (i = 0; i <= N - 1; i++)
    for (j = 0; j <= N; j++)
      if (i == j || j == N)
        A(i, j) = 1.f;
      else
        A(i, j) = 0.f;

  double time0 = MPI_Wtime();
  /* elimination */
  for (i = 0; i < N - 1; i++)
  {
    MPI_Bcast(&A(i, i + 1), N - i, MPI_DOUBLE, i % proc_num, MPI_COMM_WORLD);
    for (k = myrank; k <= N - 1; k += proc_num) {
      if (k < i + 1) {
        continue;
      }
      for (j = i + 1; j <= N; j++) {
        A(k, j) = A(k, j) - A(k, i) * A(i, j) / A(i, i);
      }
    }
  }
  /* reverse substitution */
  X[N - 1] = A(N - 1, N) / A(N - 1, N - 1);
  for (j = N - 2; j >= 0; j--)
  {
    for (k = myrank; k <= j; k += proc_num)
      A(k, N) = A(k, N) - A(k, j + 1) * X[j + 1];
    MPI_Bcast(&A(j, N), 1, MPI_DOUBLE, j % proc_num, MPI_COMM_WORLD);
    X[j] = A(j, N) / A(j, j);
```

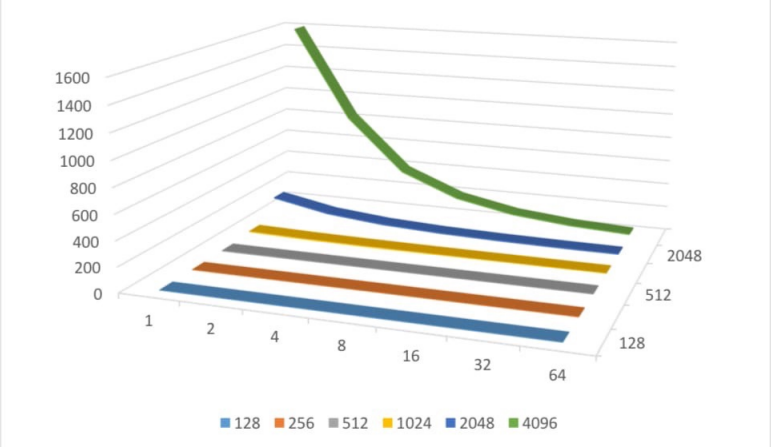
Реализованный алгоритм проверялся на корректность и совпадение по результатам с последовательным алгоритмом.

Результаты былли получены на вычислительном комплексе Polus.

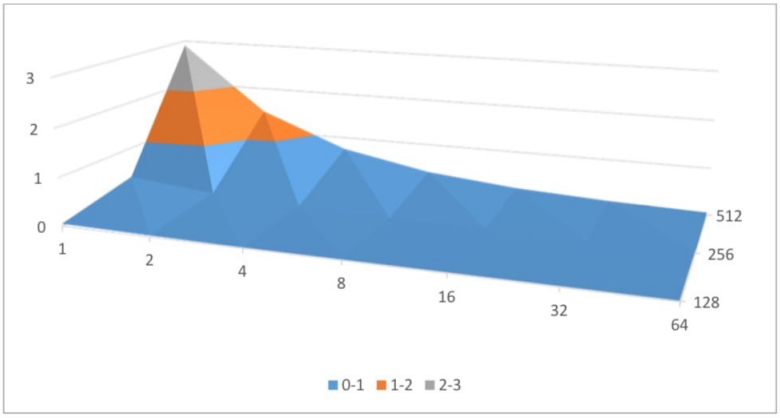
Каждая конфигурация была запущена 5 раз. Ниже приведены усредненные результаты.

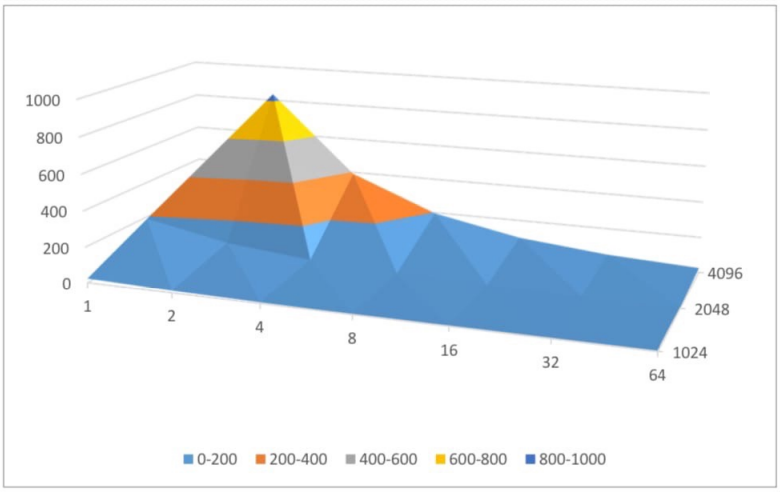

3
2,5
2
1,5
1
0,5
0
1
2
4
8
16
32
64
512
256
128
128
256
512

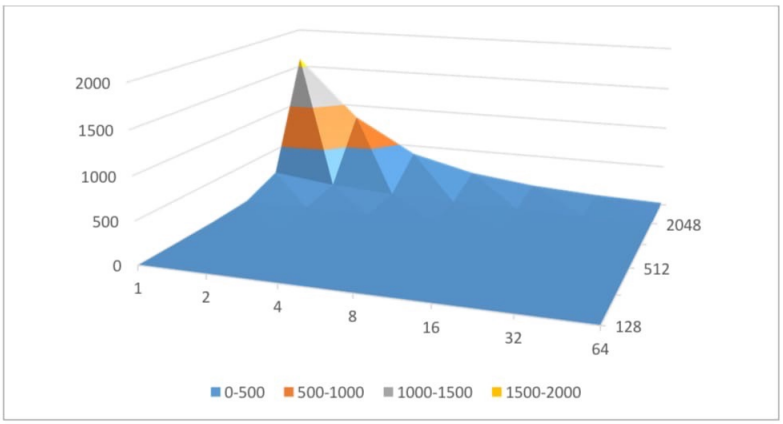

1000
800
600
400
200
0
1
2
4
8
16
32
64
4096
2048
1024
1024
2048
4096


1600
1400
1200
1000
800
600
400
200
0
1
2
4
8
16
32
64
2048
512
128
128
256
512
1024
2048
4096

Процессоры/Размер матрицы	128	256	512	1024	2048	4096
1	0.0459483	0.36366	2.9017	23.2687	188.187	Too Big
2	0.024614	0.18645	1.46295	11.6659	94.2196	836.04
4	0.007153	0.09502	0.736717	5.85155	47.2785	390.345
8	0.0043782	0.0496067	0.374417	2.94583	23.7247	195.466
16	0.0043783	0.0270455	0.194005	1.49364	11.9465	98.0254
32	0.0029669	0.0155146	0.103554	0.768395	6.05319	49.2935
64	0.00239694	0.007006	0.0587355	0.406863	3.10824	24.9343







Получается, что на малых данных MPI получается по эффективности сравнимо с OpenMp, MPI даже эффективнее при большом числе процессоров. Но при возрастании размера матрицы, MPI намного хуже, чем OpenMp. Возможно, это связано с тем, что происходят временные затраты на пересылки сообщений между собой и общая память OpenMP в этом плане гораздо эффективнее. В целом получается, что для большей эффективности можно использовать комбинацию OpenMP и MPI, где MPI работает в случае небольших данных, а OpenMP при больших, чтобы исключить проблему с синхронизацией MPI. Основной вывод же, что параллелизм, как и при работе с OpenMP даёт огромный выигрыш во времени, но потери во времени на больших данных в случае MPI намного больше.

3 Анализ результатов

На одинаковых конфигурациях OpenMP показал результаты лучшие, чем MPI.

Для OMP можно заметить, что при небольших размерах данных с увеличением числа потоков время выполнения программы перестает уменьшаться по причине того, что накладные расходы на создание порции нитей значительно увеличиваются.

Следовательно, создание нитей - основная причина недостаточной масштабируемости программы. Анализируя результаты при различных наборах параметров, можно установить оптимальное число потоков для данного размера данных.

Распараллеливание программы в среднем дало выигрыш по времени в 36 раз при небольших данных и в 63 раза при больших. Как и о жидалось, зависимость число процессоров/ время на определенных данных оказалось линейным. Это подтверждается из графиков и данных, и является теоретическим выводом распараллеливания процессов. При увеличении числа процессоров, время работы уменьшается (на примере графиков и таблицы это неплохо просматривается). Давайте сравним результаты, с результатами OpenMP

Процессоры/Размер матрицы	128	256	512	1024	2048	4096
1	0.003115	0.024327	0.181999	1.55047	12.4399	99.4217
2	0.002	0.01417	0.098851	0.825769	6.29858	50.3307
4	0.00106	0.00887	0.062314	0.407289	3.20299	25.175
8	0.000789	0.006204	0.036908	0.256996	1.62157	12.8999
16	0.000731	0.004142	0.025983	0.209367	1.31767	8.59211
32	0.000719	0.002738	0.020072	0.159401	1.13686	5.85083
64	0.021115	0.002709	0.019805	0.155255	1.13403	5.11938

(Таблица исследований из предыдущей работы)

Отнимем матрицы друг из друга и посмотрим разницу:

Процессоры/Размер матрицы	128	256	512	1024	2048	4096
1	0,0428333	0,339335	2,719501	21,71813	175,7491	Too Big
2	0,0226133	0,1723	1,364079	10,839931	87,92082	785,7143
4	0,0119178	0,08616	0,674402	5,444231	44,07581	365,167
8	0,0063642	0,0434025	0,337511	2,688814	22,10293	182,5681
16	0,0036471	0,0229037	0,168019	1,284253	10,62923	89,43309
32	0,00224794	0,0127767	0,08348	0,608992	4,91631	43,44297
64	-0,018718	0,007354	0,0389303	0,251606	1,9742	19,81472

4 Выводы

Выполнена работа по разработка параллельной версии программы для задачи Gauss. Проанализировано время выполнения программы в зависимости от различных входных данных и числа потоков. Применение технологий OpenMP и MPI позволило значительно ускорить последовательную версию программы.