МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В.Ломоносова

Факультет вычислительной математики и кибернетики

---

# Компьютерный практикум по учебному курсу

# «Распределенные системы»

# Улучшение параллельной версии программы для умножения матрицы на вектор

## ОТЧЕТ

### о выполненном задании

студента 428 учебной группы факультета ВМК МГУ

Тулина Дмитрия Ильича

гор. Москва
2022 год

# Содержание

# Постановка задачи

Ставится задача перемножения матрицы на вектор:

$$A = \begin{vmatrix} a_{11} & a_{12} & a_{1n} \\ a_{21} & a_{22} & a_{2n} \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & a_{nn} \end{vmatrix} B = \begin{vmatrix} b_1 \\ b_2 \\ .. \\ b_n \end{vmatrix} C_i = \sum_{i=1}^{n} A_{in} * B_n C = \begin{vmatrix} c_1 \\ c_2 \\ .. \\ c_n \end{vmatrix} \quad (1)$$

Результатом перемножения матрицы A на вектор B является вектор C, каждый элемент которого есть скалярное произведение соответствующих строк матрицы A на вектор B.
Требуется добавить контрольные точки для продолжения работы программы в случае сбоя, реализовав один из 3-х сценариев работы после сбоя:

1. продолжить работу программы только на “исправных” процессах;
2. вместо процессов, вышедших из строя, создать новые MPI-процессы, которые необходимо использовать для продолжения расчетов;
3. при запуске программы на счет сразу запустить некоторое дополнительное количество MPI-процессов, которые использовать в случае сбоя.

# Алгоритм

## Последовательный алгоритм

```
1 for(int i = 0; i < line; i++)
2 {
3     *(answer+i) = 0;
4     for (j = 0; j < column; j++)
5     {
6         *(answer+i) += (*(matrix + i*column +j))*(*(vector+j));
7     }
8 }
```

Листинг 1: код последовательного алгоритма

Матрично-векторное умножение – это последовательность вычисления скалярных произведений. Поскольку каждое вычисление скалярного произведения векторов длины n требует выполнения n операций умножения и n-1 операций сложения, его трудоемкость порядка O(n). Для выполнения матрично-векторного умножения необходимо выполнить m операций вычисления скалярного произведения, таким образом, алгоритм имеет трудоемкость порядка O(mn).

## Параллельный алгоритм

В MPI модификация сводится к равномерному распределению разных участков матрицы на разные ядра, с помощью MPI_Bcast. В конце полученные отрезки исходного вектора соединяются с помощью MPI_Scatterv.

## Компиляция и запуск программы

Компиляция и запуск программы выполняется в среде docker container.

```
1 FROM abouteiller/mpi-ft-ulfm
2 USER root
3 RUN apk --no-cache add cmake git
4 USER run
```

Листинг 2: описание dockerfile

mpic++ stableversion.cpp mpiexec -n 4 –with-ft ulfm a.out

## Описание программы

Данная программа продолжает работу программы только на “исправных” процессах;

Программа начинает своё выполнение в функции

```
int main(int argc, char **argv)
```

В ней происходит присваивание каждой ветке ее номера, а также дальнейшее выполнение всей программы.
В программе используются следующие вспомогательные функции

- `void verbose_errhandler(MPI_Comm* pcomm, int* perr, ...)`

  Функция взята из примеров, которые ULFM демонстрирует в качестве ознакомления с их приложением. Она была модифицирована, таким образом что добавляет в std::stack<int> failures; все ошибочные процессы. А также изменяет CurrentProcNumbers в соответсвии с оставшимся колчеством живых процессов.

- `void print_vector(int Size, int* Vector)`
  `void print_matrix(int Size, int* Matrix)`

  Функцию отвечающие за печать вектора и матрицы.

- `void RandomDataInitialization(int* Matrix, int* Vector, int Size)`

  Функция осуществялет рандомизацию данных в Matrix и Vector. Для удобства улучшения приложения функция была модфицирована.

$$A = \begin{vmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ \dots & \dots & \dots \\ \dots & \dots & N_{nn} \end{vmatrix} B = \begin{vmatrix} 1 \\ 2 \\ .. \\ N \end{vmatrix} \tag{2}$$

- `void freeMemory(const int* Matrix, const int* Vector, const int* Result,const int`

  Функция осуществялет очистку памяти после завершения работы программы

- `void ProcessInitialization (int* &Matrix, int* &Vector, int* &Result,`
  `int* &ProcRows, int* &ProcResult, int &Size, int &Row`

  Раньше функция просила ввести размерность матрицы, но входе улучшений в прошлом году, ее функционал был сведен к проверке, что размерность матрицы

меньше, чем количество ветвей. И вызову на 0 ветви заполнение матрицы и вектора рандомизированными данными.

- `void dataSharing(int* Matrix, int* ProcRows, int* Vector,int Size, int RowNum)`

  Функция выполняет равномерное распределение данных по веткам при помощи MPI_Scatterv

- `void ParallelResultCalcuation(int* ProcRows, int* Vector,int* ProcResult, int Siz`

  Функция выполняет вычисление своего участка выходного вектора. В улучшенной версии в ветви с номером 2, вызывается сигнал SIGKILL.

- `void RecoveryResultCalculation(int *Result ,int* Matrix, int* Vector,int Size)`

  Функция выполняет востановление потерянных данных при падении ветви. Функция работает до тех пор пока в стеке failures не будет пусто. Из-за отсутвия большого количетсва потоков, было принято решение довычислять все потерянные данные на основной ветке.

- `void ResultReplication(int* ProcResult, int* Result, int Size, int RowNum)`

  Функция отвечающая за соединение всех участков посчитанного вектора. В улучшенной версии пришлось отказаться от MPI_Allgatherv. И все данные собираются при помощи MPI_Send и MPI_Recv на нулевой ветке. Также был добавлен MPI_Barrier, что бы на всех процессах отработал handler ошибки.

## Код программы

```
1 #include <cstdio>
2 #include <cstdlib>
3 #include <ctime>
4 #include <mpi.h>
5 #include <mpi-ext.h>
6 #include <csignal>
7 #include <stack>
8 #include <err.h>
9 #include <climits>
10
11 int ProcNumbers, CurrentProcNumbers;
12 int ProcRank;
13
14 MPI_Comm main_comm = MPI_COMM_WORLD;
15 std::stack<int> failures{};
16 MPI_Errhandler errh;
17
18 void verbose_errhandler(MPI_Comm* pcomm, int* perr, ...) {
19     MPI_Comm comm = *pcomm;
20     int err = *perr;
21     char errstr[MPI_MAX_ERROR_STRING];
22     int i, rank, size, nf, len, eclass;
23     MPI_Group group_c, group_f;
24     int *ranks_gc, *ranks_gf;
25
26     MPI_Error_class(err, &eclass);
27     if( MPIX_ERR_PROC_FAILED != eclass ) {
28         MPI_Abort(comm, err);
29     }
30
31     MPI_Comm_rank(comm, &rank);
32     MPI_Comm_size(comm, &size);
33
34     /* We use a combination of 'ack/get_acked' to obtain the list of
35      * failed processes (as seen by the local rank).
36      */
37     MPIX_Comm_failure_ack(comm);
38     MPIX_Comm_failure_get_acked(comm, &group_f);
39     MPI_Group_size(group_f, &nf);
40     MPI_Error_string(err, errstr, &len);
41     printf("Rank %d / %d: Notified of error %s. %d found dead: { ",
42            rank, size, errstr, nf);
43
44     /* We use 'translate_ranks' to obtain the ranks of failed procs
45      * in the input communicator 'comm'.
46      */
47     ranks_gf = (int*)malloc(nf * sizeof(int));
48     ranks_gc = (int*)malloc(nf * sizeof(int));
49     MPI_Comm_group(comm, &group_c);
50     for(i = 0; i < nf; i++)
51         ranks_gf[i] = i;
52     MPI_Group_translate_ranks(group_f, nf, ranks_gf,
53                               group_c, ranks_gc);
54     for(i = 0; i < nf; i++){
55         printf("%d ", ranks_gc[i]);
```

```
56          failures.push(ranks_gc[i]);
57          CurrentProcNumbers--;
58      }
59      printf("}\n");
60      free(ranks_gf); free(ranks_gc);
61  //    MPIX_Comm_shrink(*pcomm, &main_comm);
62      MPI_Comm_rank(main_comm, &ProcRank);
63      MPI_Comm_size(main_comm, &ProcNumbers);
64  }
65
66  void print_vector(int Size, int* Vector) {
67      for (int i = 0; i < Size; ++i) {
68          printf("Vector[%d] = %d \t", i, Vector[i]);
69      }
70  }
71
72  void print_matrix(int Size, int* Matrix){
73      for (int i = 0; i < Size; ++i) {
74          printf("\n");
75          for (int j = 0; j < Size; ++j) {
76  //            Matrix[i*Size+j] = rand();
77              printf("Matrix[%d][%d] = %d \t", i,j, Matrix[i*Size+j]);
78          }
79      }
80      printf("\n");
81  }
82  void RandomDataInitialization(int* Matrix, int* Vector, int Size){
83      srand(1);
84      for (int i = 0; i < Size; ++i) {
85  //        Vector[i] = rand();
86          Vector[i] = i;
87          for (int j = 0; j < Size; ++j) {
88  //            Matrix[i*Size+j] = rand();
89              Matrix[i*Size+j] = i;
90          }
91      }
92      print_vector(Size, Vector);
93      print_matrix(Size, Matrix);
94  }
95
96  void ProcessInitialization (int* &Matrix, int* &Vector, int* &Result,
97                              int* &ProcRows, int* &ProcResult, int &Size,
        int &RowNum){
98      //Setting the size of the initial matrix and vector
99      int RestRows;
100     if (ProcRank == 0){
101         do {
102             printf("\nChosen objects size = %d\n", Size);
103             if (Size < ProcNumbers) {
104                 printf("Wrong Size, size must be > ProcNumbers\n");
105
106             }
107         } while ((Size < ProcNumbers));
108     }
109     MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
110
111     RestRows = Size;
```

```
112     for (int i=0; i<ProcRank; i++)
113         RestRows = RestRows-RestRows/(ProcNumbers-i);
114     RowNum = RestRows/(ProcNumbers-ProcRank);
115 
116     Vector = new int [Size];
117     Result = new int [Size];
118     ProcRows = new int [RowNum*Size];
119     ProcResult = new int [RowNum];
120 
121     if(ProcRank == 0){
122         Matrix = new int [Size*Size];
123         RandomDataInitialization(Matrix, Vector, Size);
124     }
125 }
126 
127 void freeMemory(const int* Matrix, const int* Vector, const int* Result,
      const int* ProcRows,const int* ProcResult){
128     if (ProcRank == 0)
129         delete [] Matrix;
130     delete [] Vector;
131     delete [] Result;
132     delete [] ProcRows;
133     delete [] ProcResult;
134 }
135 
136 void dataSharing(int* Matrix, int* ProcRows, int* Vector,int Size, int
      RowNum){
137     int *SendNum; // Number of elements sent to the process
138     int *SendInd; // Index of the first data element sent to the process
139     int RestRows=Size; // Number of rows, that have[U+FFFD]t been
      distributed yet
140     MPI_Bcast(Vector,Size,MPI_INT,0,MPI_COMM_WORLD);
141 
142     // Alloc memory for temporary objects
143     SendInd = new int [ProcNumbers];
144     SendNum = new int [ProcNumbers];
145 
146     // Determine the disposition of the matrix rows for current process
147     RowNum = (Size/ProcNumbers);
148     SendNum[0] = RowNum*Size;
149     SendInd[0] = 0;
150     for (int i=1; i<ProcNumbers; i++) {
151         RestRows -= RowNum;
152         RowNum = RestRows/(ProcNumbers-i);
153         SendNum[i] = RowNum*Size;
154         SendInd[i] = SendInd[i-1]+SendNum[i-1];
155     }
156 
157     // Scatter the rows
158     MPI_Scatterv(Matrix , SendNum, SendInd, MPI_INT, ProcRows,SendNum[
      ProcRank], MPI_INT, 0, MPI_COMM_WORLD);
159     //Free the memory
160     delete [] SendNum;
161     delete [] SendInd;
162 }
163 
164 void ParallelResultCalcuation(int* ProcRows, int* Vector,int* ProcResult
```

```
      , int Size, int RowNum){
165     if (ProcRank == 2){
166         raise(SIGKILL);
167     }
168     for (int i = 0; i < RowNum; ++i) {
169         ProcResult[i] = 0;
170         for (int j = 0; j < Size; ++j) {
171             ProcResult[i] += ProcRows[i*Size+j]*Vector[j];
172         }
173     }
174     printf("Rank %d/%d: Completed calculations\n", ProcRank, ProcNumbers
      );
175 }
176 void RecoveryResultCalculation(int *Result ,int* Matrix, int* Vector,int
      Size){
177     int *ReceiveNum;   // Number of elements, that current process sends
178     int *ReceiveInd;  // Index of the first element from current process
179     int RestRows=Size; // Number of rows, that haven[U+FFFD]t been
      distributed yet
180     ReceiveNum = new int [ProcNumbers];
181     ReceiveInd = new int [ProcNumbers];
182     // Determine the disposition of the result vector block
183     ReceiveInd[0] = 0;
184     ReceiveNum[0] = Size/ProcNumbers;
185
186     for (int i=1; i<ProcNumbers; i++) {
187         RestRows -= ReceiveNum[i-1];
188         ReceiveNum[i] = RestRows/(ProcNumbers-i);
189         ReceiveInd[i] = ReceiveInd[i-1]+ReceiveNum[i-1];
190     }
191     while (!failures.empty()){
192         int number_of_proc = failures.top();
193         for (int i = 0; i < ReceiveNum[i]; ++i) {
194             Result[ReceiveInd[number_of_proc]+i] = 0;
195             for (int j = 0; j < Size; ++j) {
196                 printf("Matrix element = %d, Vector element = %d\n",
      Matrix[(ReceiveInd[number_of_proc]+i)*Size+j], Vector[j]);
197                 Result[ReceiveInd[number_of_proc]+i] += Matrix[(
      ReceiveInd[number_of_proc]+i)*Size+j]*Vector[j];
198             }
199         }
200
201         failures.pop();
202     }
203 }
204 void ResultReplication(int* ProcResult, int* Result, int Size, int
      RowNum) {
205     int *ReceiveNum;   // Number of elements, that current process sends
206     int *ReceiveInd;  // Index of the first element from current process
207     int *Tmp_Result;
208     int RestRows=Size; // Number of rows, that haven[U+FFFD]t been
      distributed yet
209
210     // Alloc memory for temporary objects
211     ReceiveNum = new int [ProcNumbers];
212     ReceiveInd = new int [ProcNumbers];
213     Tmp_Result = new int [sizeof(ProcResult)/4];
```

```
214     // Determine the disposition of the result vector block
215     ReceiveInd[0] = 0;
216     ReceiveNum[0] = Size/ProcNumbers;
217 
218     for (int i=1; i<ProcNumbers; i++) {
219         RestRows -= ReceiveNum[i-1];
220         ReceiveNum[i] = RestRows/(ProcNumbers-i);
221         ReceiveInd[i] = ReceiveInd[i-1]+ReceiveNum[i-1];
222     }
223     MPI_Barrier(MPI_COMM_WORLD);
224     for (int i = 0; i < ReceiveNum[ProcRank]; ++i) {
225         printf("Rank %d/%d Result[%d] = %d\n", ProcRank,
    CurrentProcNumbers, i, ProcResult[i]);
226     }
227     // Gather the whole result vector on every processor
228     if (ProcRank == 0){
229         for (int i = 0; i < ReceiveNum[ProcRank]; ++i) {
230             Result[i] = ProcResult[i];
231         }
232         for (int i = 0; i < CurrentProcNumbers-1; ++i) {
233             MPI_Status status;
234             MPI_Recv(Tmp_Result,sizeof(ProcResult)/4,MPI_INT,
    MPI_ANY_SOURCE,
235                      0,MPI_COMM_WORLD, &status);
236             for(int j = 0; j < ReceiveNum[status.MPI_SOURCE]; ++j) {
237                 printf("Proc 0 received message from %d, with %d, will
    start filling from %d\n",
238                        status.MPI_SOURCE, Tmp_Result[j],ReceiveInd[
    status.MPI_SOURCE] + j);
239                 Result[ReceiveInd[status.MPI_SOURCE] + j] = Tmp_Result[j
    ];
240             }
241         }
242     }else{
243         MPI_Send(ProcResult,sizeof(ProcResult)/4,MPI_INT,0,0,
    MPI_COMM_WORLD);
244     }
245 //    int err = MPI_Allgatherv(ProcResult, ReceiveNum[ProcRank], MPI_INT
    , Result, ReceiveNum, ReceiveInd, MPI_INT, MPI_COMM_WORLD);
246 
247     // Free the memory
248     delete [] ReceiveNum;
249     delete [] ReceiveInd;
250 }
251 int main(int argc, char* argv[]) {
252     int* Matrix, *Vector, *Result, Size;
253     int *ProcRows, *ProcResult, RowNum;
254     double Start, Finish, Duration, totalDuration = 0;
255 
256     MPI_Init(&argc, &argv);
257 
258     MPI_Comm_size(MPI_COMM_WORLD, &ProcNumbers);
259     CurrentProcNumbers = ProcNumbers;
260     MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
261     MPI_Comm_create_errhandler(verbose_errhandler, &errh);
262     MPI_Comm_set_errhandler(MPI_COMM_WORLD,
263                             errh);
```

```
264     MPI_Barrier(MPI_COMM_WORLD);
265     if(ProcRank == 0){
266         printf ("Parallel matrix-vector multiplication program\nCurrent
        size is %d\n", ProcNumbers);
267     }
268     for (int Size = 7; Size <= 20; Size+=2000) {
269         totalDuration = 0;
270         ProcessInitialization(Matrix, Vector, Result, ProcRows,
        ProcResult, Size, RowNum);
271 
272         for (int i = 0; i < 1; ++i) {
273 
274             Start = MPI_Wtime();
275             dataSharing(Matrix, ProcRows, Vector, Size, RowNum);
276 //            MPI_Barrier(MPI_COMM_WORLD);
277             ParallelResultCalcuation(ProcRows, Vector, ProcResult, Size,
        RowNum);
278             ResultReplication(ProcResult, Result, Size, RowNum);
279             if (ProcRank == 0) {
280                 RecoveryResultCalculation(Result, Matrix, Vector, Size);
281                 for (int j = 0; j < Size; ++j) {
282                     printf("Result[%d] = %d \n", j, Result[j]);
283                 }
284             }
285             Finish = MPI_Wtime();
286             Duration = Finish - Start;
287             if (ProcRank == 0) {
288                 //printf("Time of execution = %f\n", Duration);
289                 totalDuration += Duration;
290             }
291         }
292         if (ProcRank == 0) {
293             printf("AVG time of execution = %f\n", totalDuration / 20);
294         }
295         freeMemory(Matrix, Vector, Result, ProcRows, ProcResult);
296     }
297     MPI_Finalize();
298 
299     return 0;
300 }
```

Листинг 3: ricart_agrawala.cpp

## Выводы

В ходе работы было реализовано улучшение прошлогодней программы параллельной реализации умножения матрицы на вектор. Была добавлена контрольная точка, с помощью которой выполняется продолжение работы программы в случае сбоя. Был реализован сценарий, при котором программа продолжает работу только на "исправных" процессах.