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 задачи перемножения матриц алгоритмом Фокса с использованием системы MPI.

Отчёт
О выполненном задании

Выполнил:
студент 323 группы
Латыпов Ш. И.

Москва
2021

Содержание

Описание задачи

Задача состояла в том, чтобы реализовать метод Фокса умножения квадратных матриц. Для этого использовались две системы параллельного программирования MPI и OpenMP. Необходимо сравнить версии программы на этих разных средствах и выяснить, какое является более эффективным.

Матрицы A и B делились на блоки размера q, кратного размеру самих матриц. Каждый процесс с координатами (i, j) получал блоки A_{ij}, B_{ij} исходных матриц. Также выполнялось обнуление блока C_{ij}, предназначенного для хранения соответствующего блока результирующего произведения матрицы $C = A \cdot B$.

Затем запускался цикл по m ($m = 0, ..., q-1$), в ходе которого выполнялось три действия:

1. Для каждой строки i ($i = 0, ..., q$) блок A_{ij} одного из процессов пересылался во все процессы этой же строки; при этом индекс j пересылаемого блока определялся по формуле $j = (i + m)\ mod\ q$;

2. Полученный в результате подобной пересылки блок матрицы A и содержащийся в процессе (i, j) блок матрицы B перемножались, и результат прибавлялся к блоку C_{ij};

3. Для каждого столбца j ($j = 0, ..., q$) выполнялась циклическая пересылка блоков матрицы B, содержащихся в каждом процессе (i, j) этого столбца, в направлении убывания номеров строк.

После завершения цикла каждый процесс хранит в себе блок C_{ij}, равный соответствующему блоку произведения $A \cdot B$, и соответственно матрица C будет равна произведернию матриц $A \cdot B$.

Алгоритм программы

Для всех матриц память выделяется динамически, размеры матриц и количество процессов задается пользователем (также проверяется, что входные данные корректны). Элементы матриц A и B принимают значения, соответсвующие их порядковому номеру в матрице (то есть элемент матрицы a_{ij} был соответсвенно раверн $i \cdot matrix\_size + j$). Так как каждый процесс считает один блок матрицы C, то всего процессов в программе $block\_cnt^2$, где $block\_cnt$ - количетсво блоков в строке/столбце матриц A и B ($block\_cnt = matrix\_size \,/\, block\_size$).

```
if (rank == 0) {
  // Выделение памяти для исходных матриц A и B и результирующей матрицы C
  int **matrix_a, **matrix_b, **matrix_c;

  matrix_a = calloc(matrix_size, sizeof(*matrix_a));
  matrix_b = calloc(matrix_size, sizeof(*matrix_b));
  matrix_c = calloc(matrix_size, sizeof(*matrix_c));

  for (int i = 0; i < matrix_size; i++) {
    matrix_a[i] = calloc(matrix_size, sizeof(*matrix_a[i]));
    matrix_b[i] = calloc(matrix_size, sizeof(*matrix_b[i]));
    matrix_c[i] = calloc(matrix_size, sizeof(*matrix_c[i]));
  }

  // Заполнения матриц A и B
  for (int i = 0; i < matrix_size; i++) {
    for (int j = 0; j < matrix_size; j++) {
      matrix_a[i][j] = i * matrix_size + j;
      matrix_b[i][j] = matrix_a[i][j];
    }
  }
```

Рис. 1: Выделение памяти для матриц и заполнение

Алгоритм подразумевает, что в каждом процессе будет дополнительно выделяться память для блоков A_{ij}, B_{ij} и C_{ij} соответсвенно. Гланый процесс делает рассылку этих блоков (кроме C_{ij}) всем остальным процессам. Для этого в программе была написана отдельная функция $get\_block()$, которая в зависимости из входных параметров заполняла блоки значениями из исходных матриц A, B.

```
void get_block(int* tmp, int** m_a, int block_i, int block_j, int block_size) {
  for (int i = 0; i < block_size; i++) {
    for (int j = 0; j < block_size; j++) {
      tmp[i * block_size + j] =
          m_a[block_i * block_size + i][block_j * block_size + j];
    }
  }
}
```

Рис. 2: Получение блока из матрицы

```
// Рассылка блоков Aij и Bij в соответствующие процессы
for (int p = 1; p < number_of_processes; p++) {
  int i = (p - 1) / block_cnt, j = (p - 1) % block_cnt;

  get_block(tmp, matrix_a, i, j, block_size);
  MPI_Send(tmp, block_size * block_size, MPI_INT, p, 0, MPI_COMM_WORLD);

  get_block(tmp, matrix_b, i, j, block_size);
  MPI_Send(tmp, block_size * block_size, MPI_INT, p, 0, MPI_COMM_WORLD);
}
MPI_Barrier(MPI_COMM_WORLD);
```

Рис. 3: Рассылка блоков A_{ij} и B_{ij} из главного процесса

```
} else if (rank != 0) {
  int i = (rank - 1) / block_cnt, j = (rank - 1) % block_cnt;
  int* tmp;    // Для разных переносимых буферов
  int* buf_a;  // Хранит свой блок Aij в виде буфера
  tmp = calloc(block_size * block_size, sizeof(*tmp));
  buf_a = calloc(block_size * block_size, sizeof(*tmp));

  int **get_a, **get_b, **get_c;  // Для хранения блоков Aij, Bij и Cij
  get_a = calloc(block_size, sizeof(*get_a));
  get_b = calloc(block_size, sizeof(*get_b));
  get_c = calloc(block_size, sizeof(*get_c));

  for (int k = 0; k < block_size; k++) {
    get_a[k] = calloc(block_size, sizeof(*get_a[k]));
    get_b[k] = calloc(block_size, sizeof(*get_b[k]));
    get_c[k] = calloc(block_size, sizeof(*get_c[k]));
  }

  // Получение от главного процесса блоков Aij и Bij
  MPI_Recv(buf_a, block_size * block_size, MPI_INT, 0, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  buf_to_matr(buf_a, get_a, block_size);

  MPI_Recv(tmp, block_size * block_size, MPI_INT, 0, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  buf_to_matr(tmp, get_b, block_size);
  MPI_Barrier(MPI_COMM_WORLD);
```

Рис. 4: Начало работы каждого процесса по выделению нужной памяти и получению блоков A_{ij} и B_{ij}

Затем процессы запускают цикл по m и в зависимости от своих координат (i, j) происходит пересылка блока A_{ij} между процессами одной строки. . Умножение блоков реализовано в функции $matrix\_mul()$.

```
int m = 0;  // Цикл по m
for (m = 0; m < block_cnt; m++) {
  int col_num_sended_A = (i + m) % block_cnt;
  // Рассылка блока Aij всем процессам в этой строке
  if (j == col_num_sended_A) {
    for (int k = 0; k < block_cnt; k++) {
      if (k != j) {
        MPI_Send(buf_a, block_size * block_size, MPI_INT,
                 i * block_cnt + k + 1, 0, MPI_COMM_WORLD);
      }
    }
  } else {
    MPI_Recv(tmp, block_size * block_size, MPI_INT,
             i * block_cnt + col_num_sended_A + 1, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    buf_to_matr(tmp, get_a, block_size);
  }

  // Умножение блоков Aij и Bij, запись результата в Cij
  matrix_mul(get_a, get_b, get_c, block_size);
```

Рис. 5: Рассылка блока A_{ij} между процессами одной строки

```
void matrix_mul(int** m_a, int** m_b, int** m_c, int size) {
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      for (int k = 0; k < size; k++) {
        m_c[i][j] += m_a[i][k] * m_b[k][j];
      }
    }
  }
}
```

Рис. 6: Умножение блоков

Последним шагом перед переходом на следующий этап цикла происходила циклическая пересылка блоков B_{ij} в направлении убывания номеров строк. Все процессы по очереди вставали в режим $MPI\_Send()$ и $MPI\_Recv()$ для избежания Deadlock'а процессов.

```
// Циклическая перессылка блока Bij в направление убывания строк.
// Условия if и else необходимы для избежания Deadlock
if (i % 2) {
  MPI_Send(tmp, block_size * block_size, MPI_INT, send_num, 0,
           MPI_COMM_WORLD);
  MPI_Recv(tmp, block_size * block_size, MPI_INT, recv_num, 0,
           MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  buf_to_matr(tmp, get_b, block_size);
} else {
  int* tmp_b = calloc(block_size * block_size, sizeof(*tmp));
  matr_to_buf(tmp_b, get_b, block_size);
  MPI_Recv(tmp_b, block_size * block_size, MPI_INT, recv_num, 0,
           MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  buf_to_matr(tmp_b, get_b, block_size);
  MPI_Send(tmp, block_size * block_size, MPI_INT, send_num, 0,
           MPI_COMM_WORLD);
  free(tmp_b);
}
```

Рис. 7: Циклическая пересылка блоков B_{ij}

Также были написаны вспомогательные функции, которые переделывали свой блок матрицы в одномерный буфер и обратно. Это было сделано для того, чтобы у процессов была возможность пересылать блоки в виде буферов между собой.

```
void buf_to_matr(int* tmp, int** matrix, int block_size) {
  for (int i = 0; i < block_size; i++) {
    for (int j = 0; j < block_size; j++) {
      matrix[i][j] = tmp[i * block_size + j];
    }
  }
}

void matr_to_buf(int* tmp, int** matrix, int block_size) {
  for (int i = 0; i < block_size; i++) {
    for (int j = 0; j < block_size; j++) {
      tmp[i * block_size + j] = matrix[i][j];
    }
  }
}
```

Рис. 8: Перевод буфер в матрицу и обратно

После завершения циклов каждый процесс отправлял свой получившийся блок C_{ij} в главный процесс, тот соответсвенно всё принимал и переносил значения в матрицу C с помощью функции $block\_to\_matrix()$, выводил получившийся результат и время работы на экран, и освобождал всю зарезервированную память.

```
  // Пересылка получившегося блока Cij в главный процесс с номером 0
  MPI_Send(tmp, block_size * block_size, MPI_INT, 0, 0, MPI_COMM_WORLD);

  // Освобождение всей памяти в процессе
  for (int k = 0; k < block_size; k++) {
    free(get_a[k]);
    free(get_b[k]);
    free(get_c[k]);
  }
  free(get_c);
  free(get_a);
  free(get_b);
  free(tmp);
  free(buf_a);
}
```

Рис. 9: Отправка блока C_{ij} в главный процесс

```
for (int p = 1; p < number_of_processes; p++) {
  int i = (p - 1) / block_cnt, j = (p - 1) % block_cnt;

  // Получение получившихся блоков Cij из каждого процесса
  MPI_Recv(tmp, block_size * block_size, MPI_INT, p, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  buf_to_matr(tmp, get_block, block_size);
  // Вставляем этот блок в матрицу C
  block_to_matrix(get_block, matrix_c, i, j, block_size);
}
// Замер времени
double end_time = MPI_Wtime();
printf("Time taken to execute is\n%lf\n", end_time - start_time);

print_m(matrix_c, matrix_size);

// Освобождение памяти
free(tmp);
for (int i = 0; i < block_size; i++) {
  free(get_block[i]);
}
free(get_block);

for (int i = 0; i < matrix_size; i++) {
  free(matrix_a[i]);
  free(matrix_b[i]);
  free(matrix_c[i]);
}
free(matrix_a);
free(matrix_b);
free(matrix_c);
```

Рис. 10: Главный процесс принимает все получившиеся блоки C_{ij}

Также для вывода матриц на экран реализована функция $print\_m()$, которая дополнительно выравнивает вывод на экран в зависимости от кратности и размера чисел в нем

```
void print_m(int** m, int size) {
  char output_format[] = "%*d ";
  int max_elem = m[size - 1][size - 1];
  int x = 1, st = 0;
  while (x < max_elem) {
    x *= 10;
    st++;
  }
  output_format[1] = st + '0';
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      printf(output_format, m[i][j]);
    }
    printf("\n");
  }
  printf("\n");
}
```

Рис. 11: Вывод матрицы на экран

Результаты замеров

В таблице приведены результаты времени работы программы (в секундах) в параллельном режиме в зависимости от введённых данных.

	Количество нитей					
Размер матриц NxN		1	4	16	36	64
	24	0.000186	0.000363	0.000826	0.004706	0.010529
	48	0.001039	0.000669	0.000600	0.005794	0.010590
	72	0.003282	0.001414	0.001241	0.005796	0.012840
	96	0.007388	0.002851	0.003217	0.006414	0.013348
	120	0.014073	0.004433	0.002764	0.007879	0.014419
	144	0.023868	0.009256	0.006904	0.010167	0.014692
	168	0.037469	0.010937	0.009294	0.011050	0.016839
	192	0.055460	0.020127	0.009316	0.011772	0.017506
	216	0.103724	0.028444	0.009512	0.015229	0.019628
	240	0.107568	0.039225	0.019677	0.018955	0.021816

Рис. 12: Таблица времени работы программы

Также представлен 3D график по соответствующим данным.

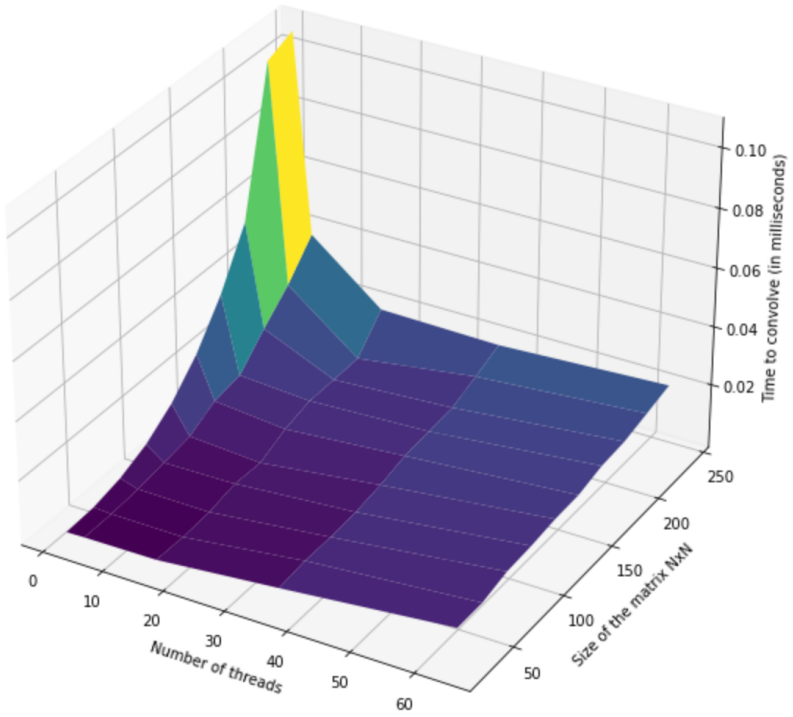

Рис. 13: 3D График MPI версии программы

Результаты замеров предыдущей версии программы, написанной с системой OpenMP

	Количество нитей					
Размер матриц NxN		1	4	16	36	64
	24	0.007407	0.007503	0.007887	0.008966	0.011018
	48	0.007642	0.007675	0.008137	0.009066	0.011048
	72	0.007817	0.007670	0.008078	0.008980	0.011157
	96	0.008231	0.007807	0.008142	0.009051	0.010976
	120	0.008943	0.007984	0.008248	0.009058	0.011256
	144	0.010024	0.008589	0.008209	0.009829	0.011004
	168	0.011440	0.008618	0.008623	0.009618	0.011744
	192	0.013466	0.009187	0.008808	0.009565	0.011685
	216	0.015912	0.009771	0.009006	0.009915	0.012428
	240	0.018880	0.010528	0.010492	0.010087	0.012544

Рис. 14: Таблица OMP

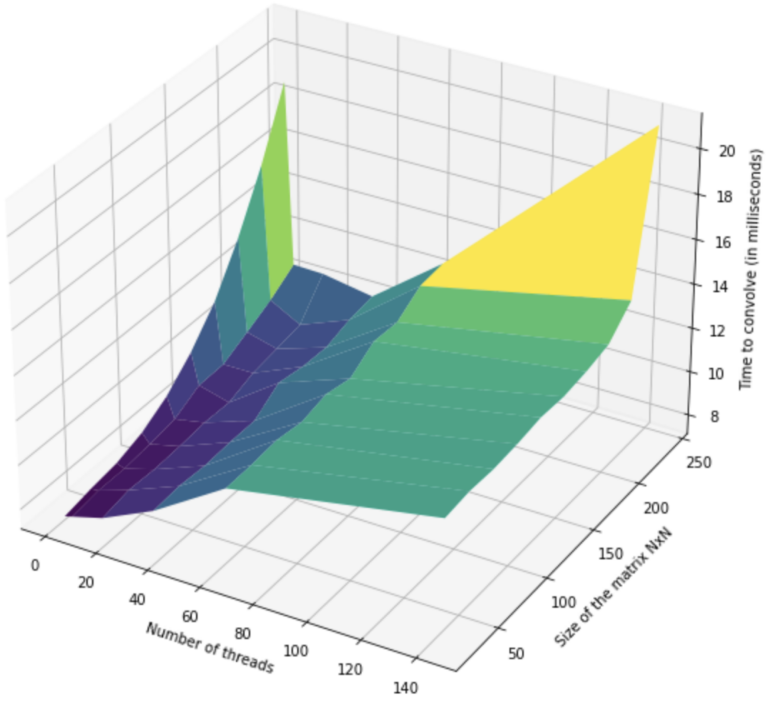

Рис. 15: 3D График OpenMP версии программы

Правильность алгоритма проверялось функцией $matrix\_mul()$. Программа запускалась отдельно с вызовом только этой функции, куда передавались исходные матрицы A и B. Полученная матрица сравнивалась с матрицей, полученной путем распараллелирования программы.

Анализ результатов

В реализации OpenMP:

При малых размерах матрицы распараллелирование не даёт преимущества по времени, а даже наоборот, увеличивает его. Это связано с издержками на планирование доступа к разделяемому ресурсу и с последующими операциями с выделением памяти на блоки для всех нитей и соответсвующего "копирования"данных из исходных матриц.

При рассмотрении матриц большего размера видно, что программа наиболее эффективна при "среднем"количестве нитей в программе (особенно заметно при размере матриц больше 168). Это следует из-за того, что при большом размере блоков (соответсвенно малом количестве нитей) программа долго умножает $A_{ij} \cdot B_{ij}$, но при большом количестве нитей возвращается проблема разделени ресурсов и выделения памяти.

В Реализации MPI:

Распараллелирование программы даёт многократное преимущество (вплоть до X5 уменьшения времени работы) при количестве нитей около 16. При большом же количесве нитей программа наоборот, замедляется, и начинает работать даже медленнее при маленьких размерах матрицы. При большом же размере матриц (примерно при > 168) эффективность сохраняется.

Сравнение OpenMP и MPI:

При малых размерах матрицы (до 72) MPI является более эффективной при любом количестве нитей. С увеличением же размеров, наоборот, технология OpenMP сохраняет время работы примерно на том же уровне, в отличии от MPI, которая с увеличением размера матриц начинает линейно терять эффективность.

Выводы

Выполнена работа по разработке параллельной версии программы для задачи перемножения матриц алгоритмом Фокса. Изучены средства параллельных алгоритмов систем OpenMP и MPI.

Также проанализировано время работы программы при разных входных параметрах на суперкомпьютере МГУ. Сделан вывод о том, что при правильном расчёте количества нитей для матриц разного размера можно заметить прирост производительности по времени на системах, рассчитаных для работы с многопоточными вычислениями. При этом для матриц разного размера и при использовании разного количества нитей обе системы OpenMP и MPI показывают отличный прирост производительности, но OpenMP всё же является более эффективной для большего количества случаев.