###### МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

###### ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

###### НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

###### Факультет информационных технологий

**Кафедра параллельных вычислений**

ОТЧЕТ

О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ

ПАРАЛЛЕЛЬНАЯ РЕАЛИЗЦАИЯ РЕШЕНИЯ СИСТЕМЫ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИЙ УРАВНЕНИЙ С ПОМОЩЬЮ MPI

Студентки 2 курса, группы 21205

**Евдокимовой Дари Евгеньевны**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Кандидат технических наук, доцент

А.Ю. Власенко

Новосибирск 2022

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc129538248)

[ЗАДАНИЕ 3](#_Toc129538249)

[ОПИСАНИЕ РАБОТЫ 4](#_Toc129538250)

[ЗАКЛЮЧЕНИЕ 6](#_Toc129538251)

[Приложение 1. Листинг последовательной программы 7](#_Toc129538252)

[Приложение 2. Проверка корректности работы последовательной программы 13](#_Toc129538253)

[Приложение 3. Скрипт для запуска последовательной программы 14](#_Toc129538254)

[Приложение 4. Листинг параллельной программы с использованием функций MPI 15](#_Toc129538255)

[Приложение 4. Проверка корректности работы параллельной программы 24](#_Toc129538256)

[Приложение 5. Скрипт для работы параллельной программы 25](#_Toc129538257)

[Приложение 6. Результаты работы последовательной и параллельной на 1, 2, 4, 8, 16, 24 процессах 26](#_Toc129538258)

[Приложение 7. Графики времени, ускорения и эффективности 27](#_Toc129538259)

[Приложение 8. Скрипт для профилирования параллельной программы 29](#_Toc129538260)

[Приложение 9. Скрины из traceanalyzer 30](#_Toc129538261)

# ЦЕЛЬ

1. Написание решения СЛАУ итерационным методом с помощью последовательной и параллельных программ.
2. Ознакомление и работа с кластером НГУ.
3. Изучение профилирования.
4. Сравнение времени работы, ускорения и эффективности последовательной и параллельной программ.

# ЗАДАНИЕ

1. Написать 2 программы (последовательную и параллельную с  
   использованием MPI) на языке C/C++, которые реализуют итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b в соответствии с выбранным вариантом. Здесь A– матрица размером N×N, x и b – векторы длины N.   
   Тип элементов – double.
2. Параллельную программу реализовать с тем условием, что матрица A и вектор b инициализируются на каком-либо одном процессе, а затем матрица A «разрезается» по строкам на близкие по размеру, возможно не одинаковые, части, а вектор b раздается каждому процессу.  
   Уделить внимание тому, чтобы при запуске программы на различном

числе MPI-процессов решалась одна и та же задача (исходные данные

заполнялись одинаковым образом).

1. Замерить время работы последовательной программы и параллельных на 2, 4, 8, 16, 24 процессах.
2. Построить графики времени, ускорения и эффективности распараллеливания от числа используемых ядер.  
   Исходные данные, параметры N и ε подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.
3. Выполнить профилирование программы с помощью jumpshot или ITAC (Intel Trace Analyzer and Collecter) при использовании 16-и/или 24-х ядер. На основании полученных результатов сделать вывод.
4. Составить отчет, содержащий исходные коды разработанных программ и построенные графики.

# ОПИСАНИЕ РАБОТЫ

Выбранный метод – метод минимальных невязок.

1. Был создан файл *nompi.cpp*, в котором была реализована последовательная программа СЛАУ методом минимальных невязок и добавлено время измерения программы с помощью функции clock\_gettime().Полный компилируемый листинг программы см. Приложение 1.
2. Корректность работы программы была осуществлена со следующими входными параметрами: размер матрицы равен 10.

Элементы главной диагонали A равны 2.0, остальные равны 1.0. Формируется вектор u, элементы которого заполняются произвольными значениями, например, . Элементы вектора b получаются путем умножения матрицы A на вектор u. В этом случае решением системы будет вектор, равный вектору u. Начальные значения элементов вектора x равны 0.

1. Результаты проверки корректности см. Приложение 2. Команда для компиляции:

*g++ -o nompi nompi.cpp*

*./nompi 10*

1. Далее началась работа на кластере.   
   Размер матрицы, при котором время программы составляет не менее 30сек: 2000.
2. Скрипт (run\_nompi.sh) для запуска программы nompi.cpp смотреть в Приложении 3.
3. Итак, для последовательной программы время работы составляет 51,00640 секунд, размер матрицы равен 2000.
4. Был создан файл withmpi.cpp, в котором реализован итерационный алгоритм решения СЛАУ методом минимальных невязок при помощи MPI-функций. Полный компилируемый листинг программы см. в Приложении 4.
5. Результаты проверки (на нескольких процессах) корректности см. Приложение 5. Команда для компиляции (не на кластере!):

*mpicxx -oversubscribe -o withmpi withmpi.cpp*

*./withmpi 10*

1. Затем началась работа на кластере. Был написан скрипт run\_withmpi.sh (см. Приложение 6) для параллельной программы.
2. Было измерено время работы программы на 1, 2, 4, 8, 16, 24 процессах. Скрипт представлен в Приложении 5.
3. Результаты измерения последовательной и параллельной программ на 1, 2, 4, 8, 16, 24 процессах представлен в Приложении 6.
4. Полученные результаты измерения времени представлены в Таблице 1.

Таблица1. Результаты измерений времени

|  |  |  |
| --- | --- | --- |
| Количество процессов | Время работы последовательной  программы, с | Время работы параллельной программы, с |
| 1 | 51,00640 | 6,26128 |
| 2 | 2,98342 |
| 4 | 2,16029 |
| 8 | 1,84773 |
| 16 | 0,99935 |
| 24 | 0,57067 |

1. На основании полученных данных рассчитаем эффективность и ускорение. Графики результатов представлены в Приложении 7.
2. Далее надо было зайти на кластер с флагом -X (для графического интерфейса), чтобы провести профилирование программы на 16 процессах. Скрипт для запуска профилирования смотреть в Приложении 8.
3. Скрины из itac traceanalyzer смотреть в Приложении 9.

# ЗАКЛЮЧЕНИЕ

В ходе работы мы смогли «распараллелить» программу для решения СЛАУ посредством использования функций MPI. Оказалось, что увеличение числа процессов может ускорить время работы программы.

Из результатов анализа профилирования видно, что львиная доля времени тратится на Bcast, т. к. на каждой итерации вектор X передавался всем процессам; тау, логическая переменная конца завершения цикла и количество итераций — это хоть и переменные (не такие большие, как массив (т. е. вектор X)), но на передачу их значений всем процессам тратится достаточно много времени.

Так же много времени забирает функция Allgatherv, поскольку в методе минимальных невязок 2 умножения матрицы на вектор, т. е. мы должны с каждого процесса собрать данные.

Меньше всего времени тратится на Allreduce, поскольку используется для редукции только дважды.

По результатам графиков времени, ускорения и эффективности можно сделать вывод о том, что при увеличении числа процессов увеличиваются так же и затраты на передачу между ними, что негативно сказывается на показателях работы программы.

# Приложение 1. Листинг последовательной программы

#include <cmath>

#include <cstdlib>

#include <iostream>

void printMatrix(double \*matrix, const size\_t sizeInput) {

  for (size\_t i = 0; i < sizeInput; ++i) {

    for (size\_t j = 0; j < sizeInput; ++j) {

      std::cout << matrix[j + i \* sizeInput] << "\t";

    }

    std::cout << std::endl;

  }

  std::cout << std::endl;

}

void printVector(const double \*vector, const size\_t sizeInput) {

  for (size\_t i = 0; i < sizeInput; ++i) {

    std::cout << std::fixed << vector[i] << " ";

  }

  std::cout << std::endl;

}

double \*fillRandomMatrix(const size\_t sizeInput) {

  double \*matrixA = new double[sizeInput \* sizeInput];

  srand(0);

  for (size\_t i = 0; i < sizeInput; ++i) {

    for (size\_t j = 0; j < sizeInput; ++j) {

      if (i == j) {

        matrixA[i \* sizeInput + j] = 9999;

      } else {

        matrixA[i \* sizeInput + j] = rand() % 500 + 150.15;

      }

    }

  }

  return matrixA;

}

double \*fillConstantMatrix(const size\_t sizeInput) {

  double \*matrixA = new double[sizeInput \* sizeInput];

  for (size\_t i = 0; i < sizeInput; ++i) {

    for (size\_t j = 0; j < sizeInput; ++j) {

      if (i == j) {

        matrixA[i \* sizeInput + j] = 2.0;

      } else {

        matrixA[i \* sizeInput + j] = 1.0;

      }

    }

  }

  return matrixA;

}

double \*fillRandomVector(const size\_t sizeInput) {

  srand(0);

  double \*vector = new double[sizeInput];

  for (size\_t i = 0; i < sizeInput; ++i) {

    vector[i] = rand() % 500;

  }

  return vector;

}

double \*fillConstantVector(const size\_t sizeInput) {

  double \*vector = new double[sizeInput];

  for (size\_t i = 0; i < sizeInput; ++i) {

    vector[i] = sizeInput + 1.0;

  }

  return vector;

}

void multimplyMatrixOnVector(const double \*matrix,

                             const double \*vector, double \*res,

                             const size\_t sizeInput) {

  for (size\_t i = 0; i < sizeInput; ++i) {

    res[i] = 0;

    for (size\_t j = 0; j < sizeInput; ++j) {

      res[i] += matrix[i \* sizeInput + j] \* vector[j];

    }

  }

}

double \*fillVectorU(const size\_t sizeInput) {

  double \*vector = new double[sizeInput];

  for (size\_t i = 0; i < sizeInput; ++i) {

    vector[i] = sin(2 \* 3.1415 \* i / sizeInput);

  }

  return vector;

}

void fillVectorB(double \*b, const double \*fullMatrixA,

                 size\_t sizeInput) {

  double \*vectorU = fillVectorU((int)sizeInput);

  // std::cout << "vector U is: " << std::endl;

  // printVector(vectorU, sizeInput);

  multimplyMatrixOnVector(fullMatrixA, vectorU, b, sizeInput);

  delete[] vectorU;

}

double countScalarMult(const double \*vector1, const double \*vector2,

                       const size\_t sizeInput) {

  double res = 0;

  for (size\_t i = 0; i < sizeInput; ++i) {

    res += vector1[i] \* vector2[i];

  }

  return res;

}

double countVectorLength(const size\_t sizeInput,

                         const double \*vector) {

  return sqrt(countScalarMult(vector, vector, sizeInput));

}

void countVectorMultNumber(const double \*vector, double scalar,

                           double \*res, const size\_t sizeInput) {

  for (size\_t i = 0; i < sizeInput; ++i) {

    res[i] = scalar \* vector[i];

  }

}

void substructVectors(const double \*vector1, const double \*vector2,

                      double \*res, const size\_t sizeInput) {

  for (size\_t j = 0; j < sizeInput; ++j) {

    res[j] = vector1[j] - vector2[j];

  }

}

void copyVector(const double \*src, double \*dst,

                const size\_t sizeInput) {

  for (size\_t i = 0; i < sizeInput; i++) {

    dst[i] = src[i];

  }

}

int main(int argc, char \*argv[]) {

  if(argc != 2) {

    std::cout << "Bad input! Enter matrix size" << std::endl;

  }

  const size\_t sizeInput = atoi(argv[1]);

  const double accuracy = 1e-10;

  const double epsilon = 1e-10;

  const size\_t maxIterationCounts = 50000;

  const size\_t maxConvergenceCount = 5;

  size\_t iterationCounts = 0;

  int convergenceCount = 0;

  double critCurrentEnd = 1;

  double prevCritEnd = 1;

  double prevPrevCritEnd = 1;

  bool isEndOfAlgo = false;

  struct timespec endt, startt;

  clock\_gettime(CLOCK\_MONOTONIC\_RAW, &startt);

  double \*Atmp = new double[sizeInput];

  double \*y = new double[sizeInput];

  double \*tauY = new double[sizeInput];

  double \*xCurr = new double[sizeInput];

  double \*xNext = new double[sizeInput];

  ///\* for testing data with RANDOM values ========= \*/

  double \*matrixA = fillRandomMatrix(sizeInput);

  // printMatrix(matrixA, sizeInput);

  double \*b = fillRandomVector(sizeInput);

  // std::cout << "vector b is: " << std::endl;

  // printVector(b, sizeInput);

  std::fill(xCurr, xCurr + sizeInput, 0);

  // std::cout << "vector X at start is: " << std::endl;

  // printVector(xCurr, sizeInput);

  ///\* for testing data with RANDOM values ========= \*/

  ///\* for testing when vector b uses sin ========= \*/

  // double \*matrixA = fillConstantMatrix(sizeInput);

  // // printMatrix(matrixA, sizeInput);

  // double \*b = new double[sizeInput];

  // fillVectorB(b, matrixA, sizeInput);

  // // std::cout << "vector b is" << std::endl;

  // // printVector(b, sizeInput);

  // std::fill(xCurr, xCurr + sizeInput, 0);

  // // std::cout << "vector X at start is: " << std::endl;

  // // printVector(xCurr, sizeInput);

  ///\* for testing when vector b uses sin ========= \*/

  double bNorm = countVectorLength(sizeInput, b);

  while (1) {

    multimplyMatrixOnVector(matrixA, xCurr, Atmp,

                            sizeInput);      // A \* x\_n

    substructVectors(Atmp, b, y, sizeInput); // y\_n = A \* x\_n - b

    double yNorm =

        countVectorLength(sizeInput, y); // || A \* x\_n - b ||

    std::fill(Atmp, Atmp + sizeInput, 0);

    multimplyMatrixOnVector(matrixA, y, Atmp, sizeInput); // A \* y\_n

    double numeratorTau =

        countScalarMult(y, Atmp, sizeInput); // (y\_n, A \* y\_n)

    double denominatorTau =

        countScalarMult(Atmp, Atmp, sizeInput); // (A \* y\_n, A \* y\_n)

    double tau = numeratorTau / denominatorTau;

    countVectorMultNumber(y, tau, tauY, sizeInput); // tau \* y

    substructVectors(xCurr, tauY, xNext,

                     sizeInput); // x\_n+1 = x\_n - tau \* y

    double critCurrentEnd = yNorm / bNorm;

    if (iterationCounts > maxIterationCounts) {

      std::cout << "Too many iterations. Change init values"

                << std::endl;

      delete[] xCurr;

      delete[] xNext;

      delete[] Atmp;

      delete[] y;

      delete[] tauY;

      delete[] matrixA;

      delete[] b;

      return 0;

    }

    if (critCurrentEnd < epsilon && critCurrentEnd < prevCritEnd &&

        critCurrentEnd < prevPrevCritEnd) {

      isEndOfAlgo = true;

      break;

    }

    copyVector(xNext, xCurr, sizeInput);

    prevPrevCritEnd = prevCritEnd;

    prevCritEnd = critCurrentEnd;

    iterationCounts++;

    std::cout << "iteration " << iterationCounts

              << " ended ===" << std::endl;

  }

  clock\_gettime(CLOCK\_MONOTONIC\_RAW, &endt);

  if (isEndOfAlgo) {

    std::cout << "Iteration amount in total: " << iterationCounts

              << std::endl;

    std::cout << "Time taken: "

              << endt.tv\_sec - startt.tv\_sec +

                     accuracy \* (endt.tv\_nsec - startt.tv\_nsec)

              << " sec" << std::endl;

    // std::cout << "Solution (vector X is): " << std::endl;

    // printVector(xNext, sizeInput);

  } else {

    std::cout << "There are no solutions =( Change input \n";

  }

  delete[] xCurr;

  delete[] xNext;

  delete[] Atmp;

  delete[] y;

  delete[] tauY;

  delete[] matrixA;

  delete[] b;

  return 0;

}

# Приложение 2. Проверка корректности работы последовательной программы

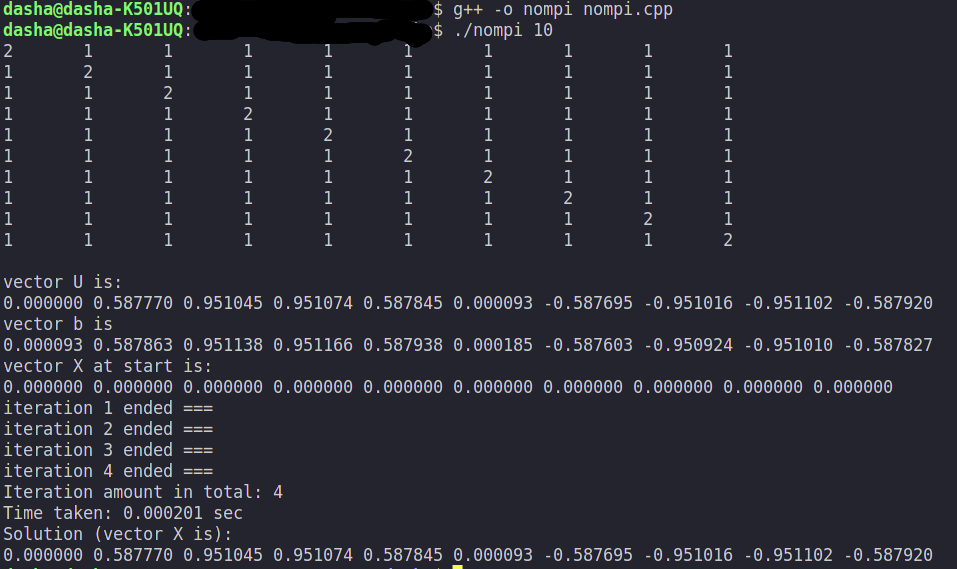


Рис. 1. Проверка корректности работы последовательной программы

# Приложение 3. Скрипт для запуска последовательной программы

1. Файл run\_nompi.sh:
2. #!/bin/sh
3. #PBS -q bl2x220g7q
4. #PBS -l walltime=00:02:00
5. #PBS -l select=1:ncpus=12:mem=2000m
6. #PBS -N task\_nompi
7. cd $PBS\_O\_WORKDIR
8. g++ -o nompi nompi.cpp
9. ./nompi 2000

# Приложение 4. Листинг параллельной программы с использованием функций MPI

1. #include <cmath>
2. #include <iostream>
3. #include <mpi.h>
4. void printMatrix(double \*matrix, const size\_t sizeInput) {
5. std::cout << "Full matrix A is: " << std::endl;
6. for (size\_t i = 0; i < sizeInput; ++i) {
7. for (size\_t j = 0; j < sizeInput; ++j) {
8. std::cout << matrix[j + i \* sizeInput] << "\t";
9. }
10. std::cout << std::endl;
11. }
12. std::cout << std::endl;
13. }
14. void printVector(const double \*vector, const size\_t sizeInput) {
15. for (size\_t i = 0; i < sizeInput; ++i) {
16. std::cout << std::fixed << vector[i] << " ";
17. }
18. std::cout << std::endl;
19. }
20. void fillRandomMatrix(double \*fullMatrA, size\_t sizeInput) {
21. srand(0);
22. for (size\_t i = 0; i < sizeInput; i++) {
23. for (size\_t j = 0; j < sizeInput; j++) {
24. fullMatrA[i \* sizeInput + j] =
25. (i == j) ? 9999 : rand() % 500 + 150.15;
26. }
27. }
28. }
29. void fillConstantMatrix(double \*fullMatrA, size\_t sizeInput) {
30. for (size\_t i = 0; i < sizeInput; i++) {
31. for (size\_t j = 0; j < sizeInput; j++) {
32. fullMatrA[i \* sizeInput + j] = (i == j) ? 2.0 : 1.0;
33. }
34. }
35. }
36. void fillRandomVector(double \*vector, size\_t sizeInput) {
37. srand(0);
38. for (size\_t i = 0; i < sizeInput; i++) {
39. vector[i] = rand() % 500;
40. }
41. }
42. void fillConstantVector(double \*vector, size\_t sizeInput) {
43. for (size\_t i = 0; i < sizeInput; i++) {
44. vector[i] = sizeInput + 1;
45. }
46. }
47. void multimplyMatrixOnVector(const double \*matrix,
48. const double \*vector, double \*res,
49. size\_t matrixRowsNumber,
50. size\_t matrixColumnsNumber) {
51. for (size\_t i = 0; i < matrixRowsNumber; i++) {
52. res[i] = 0;
53. for (size\_t j = 0; j < matrixColumnsNumber; j++) {
54. res[i] += matrix[i \* matrixColumnsNumber + j] \* vector[j];
55. }
56. }
57. }
58. double \*fillVectorU(const int N) {
59. double \*u = new double[N];
60. for (int i = 0; i < N; i++) {
61. u[i] = sin(2 \* M\_PI \* (i) / N);
62. }
63. return u;
64. }
65. void fillVectorB(double \*b, const double \*fullMatrixA,
66. size\_t sizeInput) {
67. double \*u = fillVectorU((int)sizeInput);
68. std::cout << "vector u is: " << std::endl;
69. printVector(u, sizeInput);
70. multimplyMatrixOnVector(fullMatrixA, u, b, sizeInput, sizeInput);
71. delete[] u;
72. }
73. double countScalarMult(const double \*vector1, const double \*vector2,
74. const size\_t size) {
75. double res = 0;
76. for (size\_t i = 0; i < size; i++) {
77. res += vector1[i] \* vector2[i];
78. }
79. return res;
80. }
81. double countVectorLength(const double \*vector, const size\_t size) {
82. return sqrt(countScalarMult(vector, vector, size));
83. }
84. void countVectorMultNumber(const double \*vector, double scalar,
85. double \*res, const size\_t size) {
86. for (size\_t i = 0; i < size; ++i) {
87. res[i] = scalar \* vector[i];
88. }
89. }
90. void subtractVectors(const double \*vector1, const double \*vector2,
91. double \*res, const size\_t firstVectorSize,
92. const int firstVectorElementsOffset,
93. const int secondVectorElementsOffset) {
94. for (size\_t i = 0; i < firstVectorSize; i++) {
95. res[i] = vector1[firstVectorElementsOffset + i] -
96. vector2[secondVectorElementsOffset + i];
97. }
98. }
99. void copyVectors(const double \*src, double \*dst, const size\_t size) {
100. for (size\_t i = 0; i < size; i++) {
101. dst[i] = src[i];
102. }
103. }
104. int \*countElemsNumInEachProc(size\_t amountOfProcs, size\_t sizeInput) {
105. int \*elemsNum = new int[amountOfProcs];
106. int basicRowsCount = sizeInput / amountOfProcs;
107. int restRowsCount = sizeInput % amountOfProcs;
108. for (size\_t i = 0; i < amountOfProcs; i++) {
109. elemsNum[i] = basicRowsCount \* sizeInput;
110. if (restRowsCount > 0) {
111. elemsNum[i] += sizeInput;
112. --restRowsCount;
113. }
114. }
115. return elemsNum;
116. }
117. int \*countRowsInEachProcess(const int \*elementsNumberArr,
118. int amountOfProcs, size\_t sizeInput) {
119. int \*rowsNum = new int[amountOfProcs];
120. for (size\_t i = 0; i < amountOfProcs; i++) {
121. rowsNum[i] = elementsNumberArr[i] / sizeInput;
122. }
123. return rowsNum;
124. }
125. int \*createElemsOffsetArr(const int \*elemsNum, int amountOfProcs) {
126. int \*elementsOffsetArray = new int[amountOfProcs];
127. int elementsOffset = 0;
128. for (size\_t i = 0; i < amountOfProcs; i++) {
129. elementsOffsetArray[i] = elementsOffset;
130. elementsOffset += elemsNum[i];
131. }
132. return elementsOffsetArray;
133. }
134. int \*createRowsOffsetArr(const int \*elementsOffsetArray,
135. int amountOfProcs, size\_t sizeInput) {
136. int \*rowsOffsetArray = new int[amountOfProcs];
137. for (size\_t i = 0; i < amountOfProcs; i++) {
138. rowsOffsetArray[i] = elementsOffsetArray[i] / sizeInput;
139. }
140. return rowsOffsetArray;
141. }
142. int main(int argc, char \*\*argv) {
143. if(argc != 2) {
144. std::cout << "Bad input! Enter matrix size" << std::endl;
145. }
146. const size\_t sizeInput = atoi(argv[1]);
147. const double epsilon = 10e-10;
148. double critCurrentEnd = 1;
149. double prevCritEnd = 1;
150. double prevPrevCritEnd = 1;
151. double tau;
152. bool isEndOfAlgo = false;
153. size\_t iterationCounts = 0;
154. const size\_t maxIterationCounts = 50000;
155. int convergenceCount = 0;
156. const size\_t maxConvergenceCount = 5;
157. MPI\_Init(&argc, &argv);
158. double startTime = MPI\_Wtime();
159. int amountOfProcs, rankOfCurrProc;
160. MPI\_Comm\_size(MPI\_COMM\_WORLD, &amountOfProcs);
161. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rankOfCurrProc);
162. double \*fullMatrA = new double[sizeInput \* sizeInput];
163. double \*b = new double[sizeInput];
164. double \*xCurr = new double[sizeInput];
165. ///\* for testing when vector b uses sin ========= \*/
166. // if (rankOfCurrProc == 0) {
167. //   fillConstantMatrix(fullMatrA, sizeInput);
168. //   // printMatrix(fullMatrA, sizeInput);
169. //   fillVectorB(b, fullMatrA, sizeInput);
170. //   std::fill(xCurr, xCurr + sizeInput, 0);
171. //   // std::cout << "vector X at start is: " << std::endl;
172. //   // printVector(xCurr, sizeInput);
173. // }
174. ///\* for testing when vector b uses sin ========= \*/
175. ///\* for testing data with RANDOM values ========= \*/
176. if (rankOfCurrProc == 0) {
177. fillRandomMatrix(fullMatrA, sizeInput);
178. // printMatrix(fullMatrA, sizeInput);
179. fillRandomVector(b, sizeInput);
180. // std::cout << "vector b is: " << std::endl;
181. // printVector(b, sizeInput);
182. std::fill(xCurr, xCurr + sizeInput, 0);
183. // std::cout << "vector X at start is: " << std::endl;
184. // printVector(xCurr, sizeInput);
185. }
186. ///\* for testing data with RANDOM values ========= \*/
187. double bNorm;
188. if (rankOfCurrProc == 0) {
189. bNorm = sqrt(countScalarMult(b, b, sizeInput));
190. }
191. // sends a message from root to all group process, including itself
192. MPI\_Bcast(b, sizeInput, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
193. // amount of elems, handling each process
194. int \*elemsNum = countElemsNumInEachProc(amountOfProcs, sizeInput);
195. // amount of rows, which each process handles
196. int \*rowsNum =
197. countRowsInEachProcess(elemsNum, amountOfProcs, sizeInput);
198. // count, from which element data sends to each process
199. int \*elementsOffsetArray =
200. createElemsOffsetArr(elemsNum, amountOfProcs);
201. // count, from which row data sends to each process
202. int \*rowsOffsetArray = createRowsOffsetArr(
203. elementsOffsetArray, amountOfProcs, sizeInput);
205. double \*partMatrA = new double[elemsNum[rankOfCurrProc]];
206. /\*\*\* MPI\_Scatterv - рапределяет блоки данных по всем процессам
207. \* @param sendbuf [in] - address of send buffer (significant only at
208. \* root)
209. \* @param sendcounts [in] - array (=group size)
210. \* specifying the number of elements to send to each processor
211. \* @param displs [in] - array (=group size).
212. \* Entry i specifies the displacement (relative to sendbuf from
213. \* which to take the outgoing data to process i)
214. \* i-oe значение посылает данные в i-й блок
215. \* @param sendtype - type of sending data
216. \* @param recvbuf [out] - address of reciever buffer's starting
217. \* @param recvcounts - amount of elems that we recieve
218. \* @param recvtype - type of receiving data
219. \* @param root - number of provess-reciever
220. \* @param Comm - communicator
221. \*/
222. MPI\_Scatterv(fullMatrA, elemsNum, elementsOffsetArray, MPI\_DOUBLE,
223. partMatrA, elemsNum[rankOfCurrProc], MPI\_DOUBLE, 0,
224. MPI\_COMM\_WORLD);
225. double \*partMultResultVector\_Ax =
226. new double[rowsNum[rankOfCurrProc]];
227. double \*partMultResultVector\_Ay =
228. new double[rowsNum[rankOfCurrProc]];
229. double \*partVectorY = new double[rowsNum[rankOfCurrProc]];
230. double \*partNextX = new double[rowsNum[rankOfCurrProc]];
231. double \*partMultVectorByScalar\_TauY =
232. new double[rowsNum[rankOfCurrProc]];
233. double \*vectorY = new double[sizeInput];
234. double \*xNext = new double[sizeInput];
235. while (1) {
236. MPI\_Bcast(xCurr, sizeInput, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
237. multimplyMatrixOnVector(partMatrA, xCurr, partMultResultVector\_Ax,
238. rowsNum[rankOfCurrProc], sizeInput);
239. subtractVectors(partMultResultVector\_Ax, b, partVectorY,
240. rowsNum[rankOfCurrProc], 0,
241. rowsOffsetArray[rankOfCurrProc]);
242. /\*\*\* MPI\_Allgatherv - собирает блоки с разным числом элементов от
243. каждого процесса
244. \* @param sendbuf [in] - starting address of send buffer
245. \* @param sendcounts [in] - amount of elems in send buffer
246. \* @param sendtype - data type of send buffer elements
247. \* @param recvbuf [out] - address of receive buffer
248. \* @param recvcounts - array (=group size) containing the number of
249. elements that are to be received from each process
250. \* @param displs - array (=group size). Entry i
251. specifies the displacement (relative to recvbuf) at which to place
252. the incoming data from process i
253. \* @param recvtype - data type of recieve buffer elements
254. \* @param Comm - communicator
255. \*/
256. // y\_n = A \* x\_n - b
257. MPI\_Allgatherv(partVectorY, rowsNum[rankOfCurrProc], MPI\_DOUBLE,
258. vectorY, rowsNum, rowsOffsetArray, MPI\_DOUBLE,
259. MPI\_COMM\_WORLD);
260. // A \* y\_n
261. multimplyMatrixOnVector(partMatrA, vectorY,
262. partMultResultVector\_Ay,
263. rowsNum[rankOfCurrProc], sizeInput);
264. // || A \* x\_n - b ||
265. double yNorm = 0;
266. if (rankOfCurrProc == 0) {
267. double squareMetricVectorY =
268. countScalarMult(vectorY, vectorY, sizeInput);
269. yNorm = sqrt(squareMetricVectorY);
270. }
271. // start counting tau
272. double numeratorTau, denominatorTau;
273. double partScalarProduct\_YAy =
274. countScalarMult(partVectorY, partMultResultVector\_Ay,
275. rowsNum[rankOfCurrProc]);
276. // (y\_n, A \* y\_n)
277. MPI\_Reduce(&partScalarProduct\_YAy, &numeratorTau, 1, MPI\_DOUBLE,
278. MPI\_SUM, 0, MPI\_COMM\_WORLD);
279. double partScalarProduct\_AyAy = countScalarMult(
280. partMultResultVector\_Ay, partMultResultVector\_Ay,
281. rowsNum[rankOfCurrProc]);
282. // (A \* y\_n, A \* y\_n)
283. MPI\_Reduce(&partScalarProduct\_AyAy, &denominatorTau, 1,
284. MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);
285. if (rankOfCurrProc == 0) {
286. tau = numeratorTau / denominatorTau;
287. }
288. MPI\_Bcast(&tau, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
289. countVectorMultNumber(partVectorY, tau,
290. partMultVectorByScalar\_TauY,
291. rowsNum[rankOfCurrProc]);
292. subtractVectors(xCurr, partMultVectorByScalar\_TauY, partNextX,
293. rowsNum[rankOfCurrProc],
294. rowsOffsetArray[rankOfCurrProc], 0);
295. // x\_n+1 = x\_n - tau \* y
296. MPI\_Allgatherv(partNextX, rowsNum[rankOfCurrProc], MPI\_DOUBLE,
297. xNext, rowsNum, rowsOffsetArray, MPI\_DOUBLE,
298. MPI\_COMM\_WORLD);
299. critCurrentEnd = yNorm / bNorm;
300. if (rankOfCurrProc == 0) {
301. if (critCurrentEnd < epsilon && critCurrentEnd < prevCritEnd &&
302. critCurrentEnd < prevPrevCritEnd) {
303. isEndOfAlgo = true;
304. }
305. if (iterationCounts > maxIterationCounts) {
306. isEndOfAlgo = true;
307. }
308. }
309. MPI\_Bcast(&isEndOfAlgo, 1, MPI\_C\_BOOL, 0, MPI\_COMM\_WORLD);
310. if (isEndOfAlgo) {
311. break;
312. }
313. copyVectors(xNext, xCurr, sizeInput);
314. prevPrevCritEnd = prevCritEnd;
315. prevCritEnd = critCurrentEnd;
316. if (rankOfCurrProc == 0) {
317. iterationCounts++;
318. }
319. MPI\_Bcast(&iterationCounts, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
320. if (rankOfCurrProc == 0) {
321. std::cout << "iteration " << iterationCounts
322. << " ended ====" << std::endl;
323. }
324. }
325. double endTime = MPI\_Wtime();
326. if (rankOfCurrProc == 0) {
327. if (iterationCounts < maxIterationCounts) {
328. std::cout << "Iteration amount in total: " << iterationCounts
329. << std::endl;
330. std::cout << "Time taken: " << endTime - startTime << " sec"
331. << std::endl;
332. // std::cout << "Solution (vector X is): " << std::endl;
333. // printVector(xCurr, sizeInput);
334. }
335. else {
336. std::cout << "There are no solutions =( Change input \n";
337. }
338. delete[] fullMatrA;
339. }
340. delete[] partMatrA;
341. delete[] partMultVectorByScalar\_TauY;
342. delete[] partVectorY;
343. delete[] partNextX;
344. delete[] partMultResultVector\_Ax;
345. delete[] partMultResultVector\_Ay;
346. delete[] b;
347. delete[] xCurr;
348. delete[] xNext;
349. delete[] vectorY;
350. delete[] rowsOffsetArray;
351. delete[] elementsOffsetArray;
352. delete[] rowsNum;
353. delete[] elemsNum;
354. MPI\_Finalize();
355. return 0;
356. }

# Приложение 4. Проверка корректности работы параллельной программы

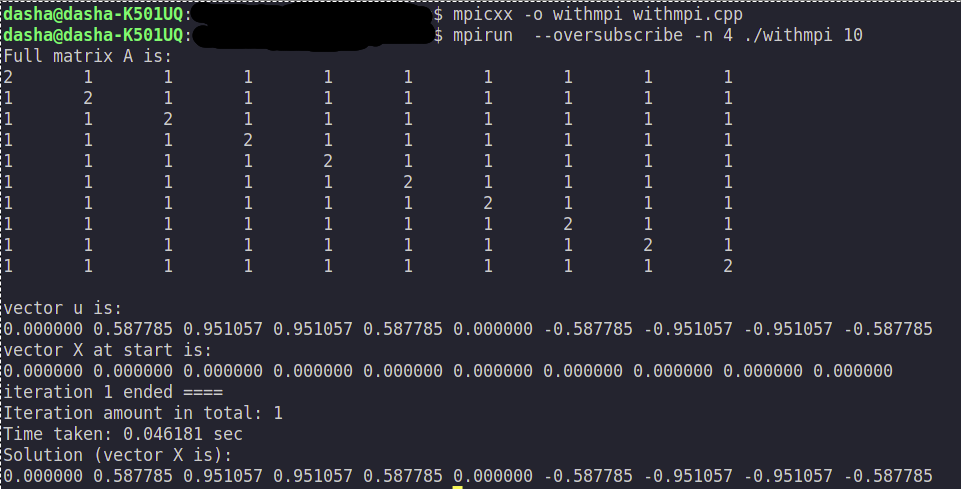


Рис. 1. Результат на 4 процессах

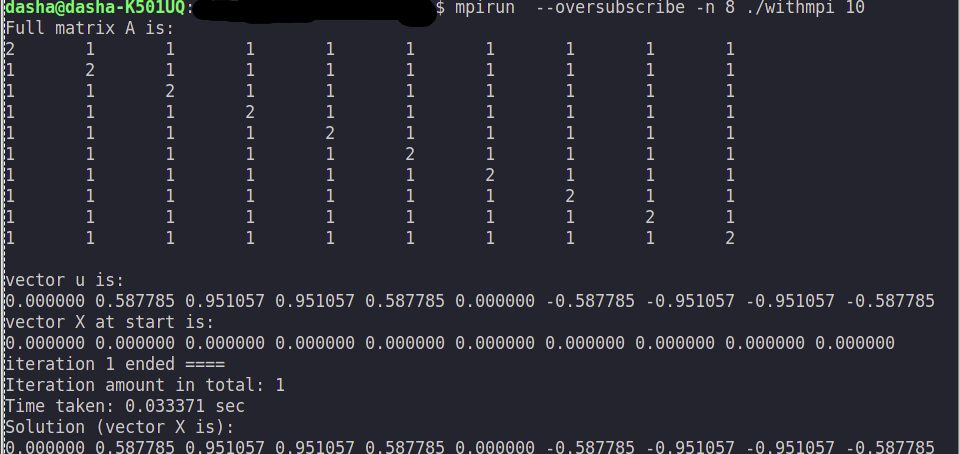


Рис. 2. Результат на 8 процессах

# Приложение 5. Скрипт для работы параллельной программы

Файл run\_withmpi.sh

#!/bin/bash

#PBS -l walltime=00:02:00

#PBS -l select=2:ncpus=1:mpiprocs=1:mem=10000m,place=scatter:exclhost

#PBS -m n

#PBS -N task\_withmpi

cd $PBS\_O\_WORKDIR

MPI\_NP=$(wc -l $PBS\_NODEFILE | awk '{ print $1 }')

echo "Number of MPI process: $MPI\_NP"

#echo 'File $PBS\_NODEFILE:'

#cat  $PBS\_NODEFILE

mpiicpc -o withmpi withmpi.cpp -std=c++14

mpirun -hostfile $PBS\_NODEFILE -np $MPI\_NP ./withmpi 2000

\*Примечание.

Для измерения времени на разных процессах: 1, 2, 4, 8, 16, 24 изменяется количество узлов и процессов. Т.е. строка с выбором узлов и процессов будет выглядеть так:

Для 1го процесса:

#PBS -l select=1:ncpus=1:mpiprocs=1:mem=10000m,place=scatter:exclhost

Для 2х процессов:

#PBS -l select=2:ncpus=1:mpiprocs=1:mem=10000m,place=scatter:exclhost

Для 4х процессов:

#PBS -l select=2:ncpus=2:mpiprocs=2:mem=10000m,place=scatter:exclhost

Для 8х процессов:

#PBS -l select=2:ncpus=4:mpiprocs=4:mem=10000m,place=scatter:exclhost

Для 12х процессов:

#PBS -l select=2:ncpus=8:mpiprocs=8:mem=10000m,place=scatter:exclhost

Для 24х процессов:

#PBS -l select=2:ncpus=12:mpiprocs=12:mem=10000m,place=scatter:exclhost

# Приложение 6. Результаты работы последовательной и параллельной на 1, 2, 4, 8, 16, 24 процессах

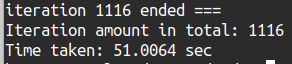


Рис. 1. Время работы последовательной программы

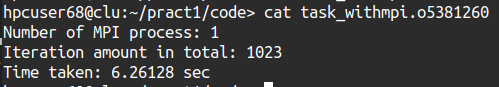


Рис. 2. Время работы параллельной программы на 1м процессе

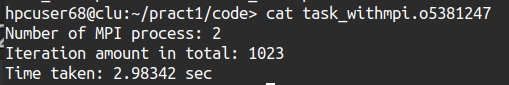


Рис. 3. Время работы параллельной программы на 2х процессах

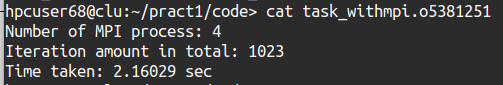


Рис. 4. Время работы параллельной программы на 4х процессах

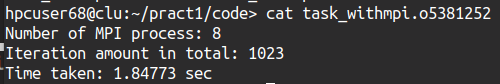


Рис. 5. Время работы параллельной программы на 8и процессах

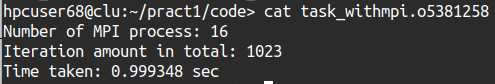


Рис. 6. Время работы параллельной программы на 16и процессах

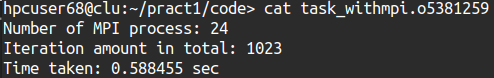
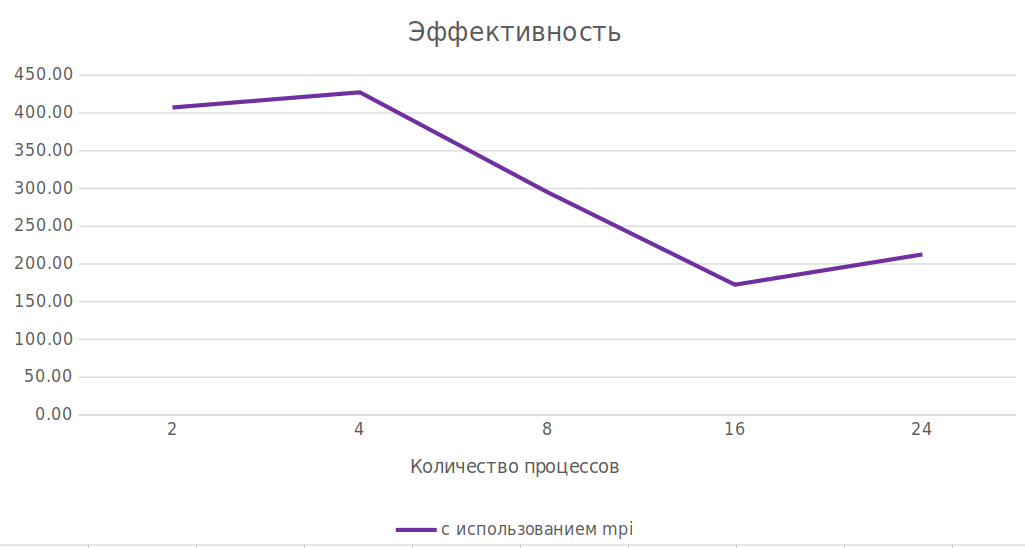


Рис. 7. Время работы параллельной программы на 24и процессах

# Приложение 7. Графики времени, ускорения и эффективности

1. Рис. 1. График зависимости от времени
2. Рис. 2. График ускорения
4. Рис. 3. График эффективности

# Приложение 8. Скрипт для профилирования параллельной программы

Файл run\_alanysis.sh

#!/bin/bash

#PBS -l walltime=00:02:00

#PBS -l select=2:ncpus=8:mpiprocs=8:mem=10000m,place=scatter:exclhost

#PBS -m n

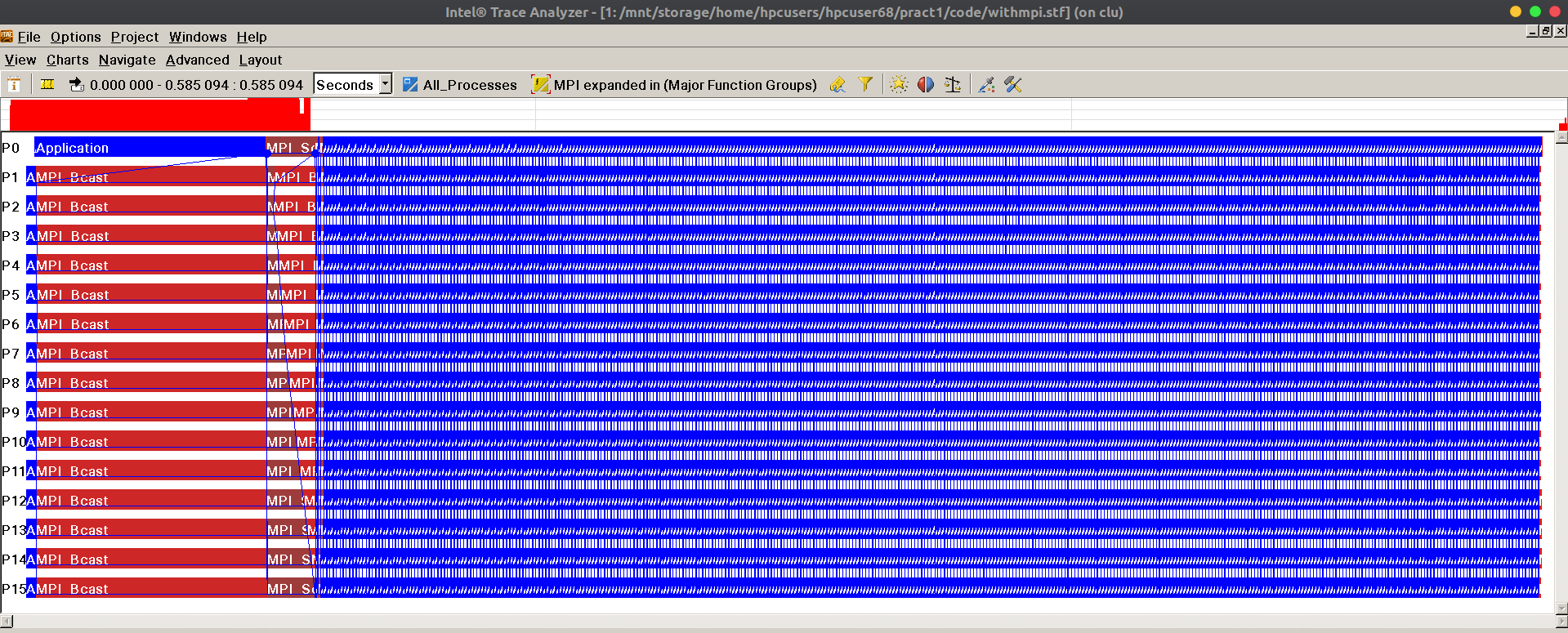
cd $PBS\_O\_WORKDIR

MPI\_NP=$(wc -l $PBS\_NODEFILE | awk '{ print $1 }')

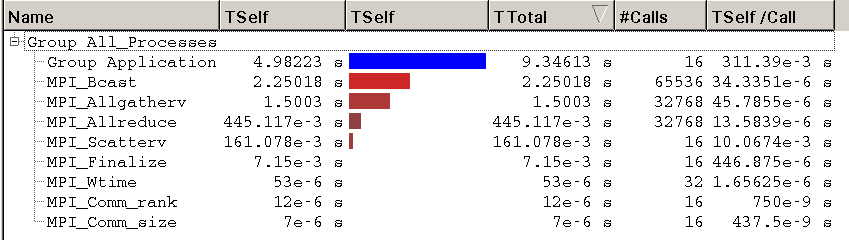
echo "Number of MPI process: $MPI\_NP"

mpirun -trace -machinefile $PBS\_NODEFILE -np $MPI\_NP -perhost 2 ./withmpi 2000

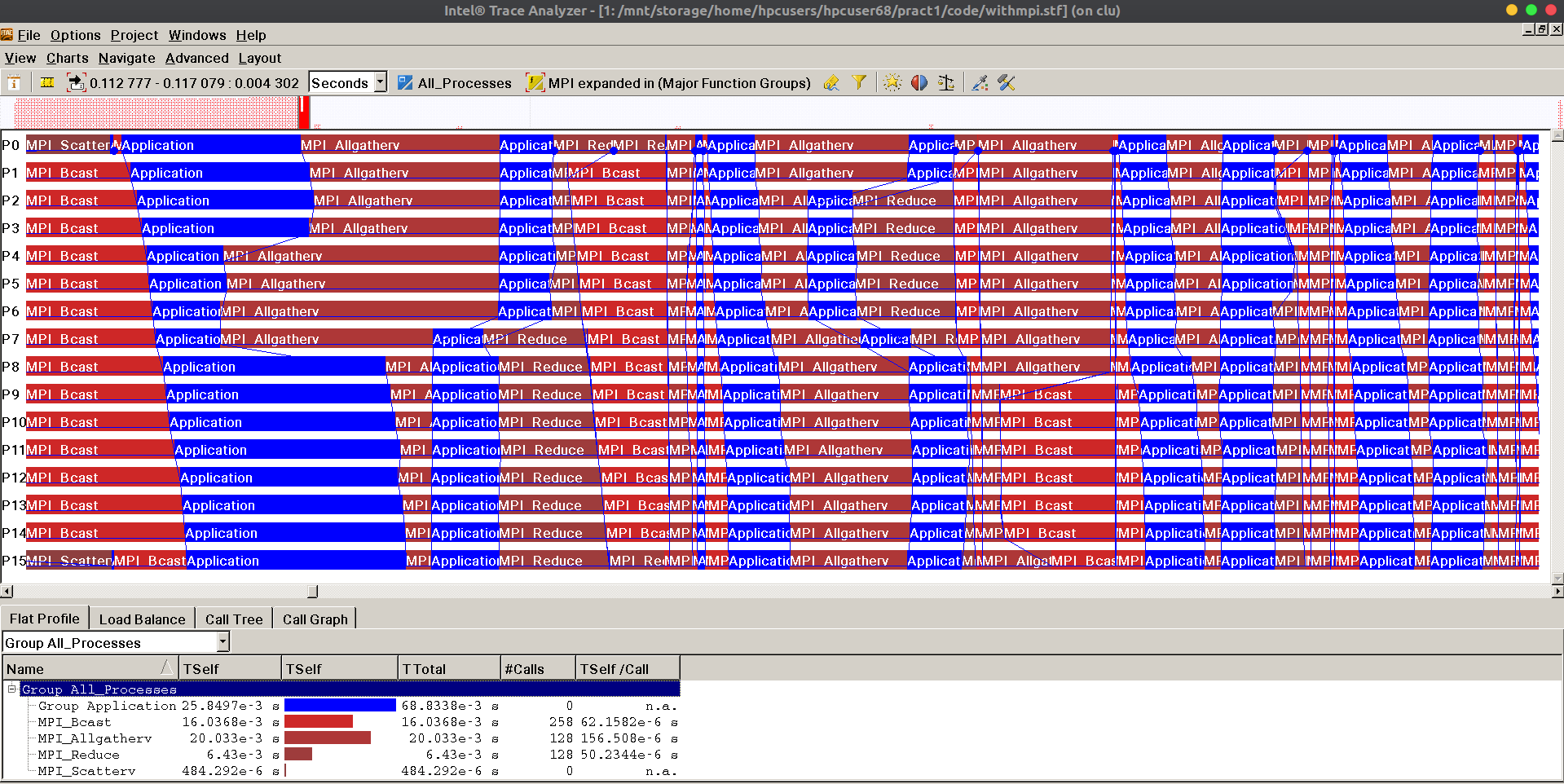
# Приложение 9. Скрины из traceanalyzer



1. Рис. 1.



1. Рис. 2.



1. Рис. 3.