МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского»
(ННГУ)

ОТЧЕТ ПО ПРАКТИКЕ №1

ТЕМА: ПАРАЛЛЕЛЬНАЯ СОРТИРОВКА ШЕЛЛА

Выполнил работу:
студент группы
381603-3 ПМиИ
Перов Дмитрий

Проверил:
Нестеров Александр Юрьевич

Нижний Новгород 2018

Содержание[1]

[1] Выберите пункт и нажмите на ссылку для быстрого перехода

Введение

В настоящее время развитие вычислительных систем испытывает кризис программного обеспечения. Он связан с невозможностью дальнейшего развития аппаратного обеспечения. Адекватного ответа на возникший кризис до сих пор не найдено, но одним из способов его преодоления является разработка параллельных программ. Для их разработки существует два принципиально разных подхода. Первый из них заключается в создании новых алгоритмов, которые будучи не эффективными на стандартных последовательных архитектурах позволяют получить при решении возникающих задач значительной эффективности на архитектурах параллельных. Второй подход — это попытка адаптации хорошо исследованных последовательных алгоритмов для новых архитектур. Именно второму подходу посвящена данная работа.

ПУНКТ №1. Постановка учебной задачи

В рамках данной работы ставится задача адаптации сортировки массива алгоритмом Шелла на кластере с использованием сортировки слиянием:

- Реализация разбиения массива на подмассивы.
- Отсортировать подмассивы алгоритмом Шелла одновременно на узлах кластера.
- Объединить подмассивы.
- Отсортировать получившийся массив сортировкой слиянием.
- Сравнить время работы параллельного и линейного варианта реализации алгоритма.

ПУНКТ №2. Руководство пользователя

Пример работы программы:

```
D:\VisualStudioCommunity2017>mpiexec -n 4 D:\ITMM\ParallelProgrammin
g\MPC\modules\task_3\Perov_Dima_task3_ShellMergeSort\Debug\Perov_Dim
a_task3_ShellMergeSort.exe 10

*LINE*
Size: 10
Vector: 0.0837898, -0.214286, 1.73737, 0.594595, -0.307368, 10.9946,
 -1.01, -1.22222, -0.358124, 0.759944,

ShellSortTime: 1.92007e-06

Size: 10
Vector: -1.22222, -1.01, -0.358124, -0.307368, -0.214286, 0.0837898,
 0.594595, 0.759944, 1.73737, 10.9946,

*PARALLEL*
ShellMergeSortTime: 0.00618005

Size: 10
Vector: -1.22222, -1.01, -0.358124, -0.307368, -0.214286, 0.0837898,
 0.594595, 0.759944, 1.73737, 10.9946,

Result: plBuffer

SPEED-UP: 0.000310688
```

```
D:\VisualStudioCommunity2017>mpiexec -n 4 D:\ITMM\Paralle
lProgramming\MPC\modules\task_3\Perov_Dima_task3_ShellMer
geSort\Debug\Perov_Dima_task3_ShellMergeSort.exe 10000000


*LINE*
ShellSortTime: 12.6903

*PARALLEL*
ShellMergeSortTime: 6.26971

Result: plBuffer

SPEED-UP: 2.02407
```

- mpiexec -n 4
  Запуск программы на 4 узлах кластера.
- При размере вектора меньше или равным 10 будет выводиться весь вектор.
- Также выводятся данные о скорости сортировки массива в линейном и в параллельном варианте алгоритма.
- Выдаётся имя буфера, куда сохраняется отсортированный вектор.

Пункт №3 Руководство программиста

Описание алгоритмов

Алгоритм сортировки Шелла

Идея метода заключается в сравнение разделенных на группы элементов последовательности, находящихся друг от друга на некотором расстоянии. Изначально это расстояние равно **d** или **mySize/2**, где **mySize** — общее число элементов. На первом шаге каждая группа включает в себя два элемента расположенных друг от друга на расстоянии **mySize/2**; они сравниваются между собой, и, в случае необходимости, меняются местами. На последующих шагах также происходят проверка и обмен, но расстояние **d** сокращается на **d/2**, и количество групп, соответственно, уменьшается, а количество элементов в группе увеличивается. Постепенно расстояние между элементами уменьшается, и на **d=1** проход по массиву происходит в последний раз.

Оценка сложности алгоритма:

В лучшем случае *O(n(log(n)));*

В худшем случае *O(n^2);*

Алгоритм сортировки Слиянием

Алгоритм использует принцип «разделяй и властвуй»: задача разбивается на подзадачи меньшего размера, которые решаются по отдельности, после чего их решения комбинируются для получения решения исходной задачи. Конкретно процедуру сортировки слиянием можно описать следующим образом:

- Если в рассматриваемом массиве один элемент, то он уже отсортирован — алгоритм завершает работу.
- Иначе массив разбивается на две части, которые сортируются рекурсивно.
- После сортировки двух частей массива к ним применяется процедура слияния, которая по двум отсортированным частям получает исходный отсортированный массив.

Оценка сложности алгоритма:

В лучшем случае $O(n(log(n)));$

В худшем случае $O(n(log(n)))$;

Вычислительный эксперимент

Результаты экспериментов запуска программы на 2 вычислительных узлах:

Количество элементов	10^4	10^5	10^6	10^7
Среднее ускорение	0.91	0.999	1.25	1.382

Результаты экспериментов запуска программы на 4 вычислительных узлах:

Количество элементов	10^4	10^5	10^6	10^7
Среднее ускорение	0.77	1.18	1.7	2.03

Результаты экспериментов запуска программы на 8 вычислительных узлах:

Количество элементов	10^4	10^5	10^6	10^7
Среднее ускорение	0.58	1.103	1.85	2.26

Результаты экспериментов запуска программы на 16 вычислительных узлах:

Количество элементов	10^4	10^5	10^6	5*10^6
Среднее ускорение	0.23	0.835	1.83	2.1

Использование MPI функций для реализации параллельного алгоритма Шелла.

MPI_Scatter() разбивает сообщение из буфера посылки процесса root на равные части размером sendcount и посылает i-ю часть в буфер приема процесса с номером i (в том числе и самому себе).

Функция MPI_Gather() производит сборку блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером root. Длина блоков предполагается одинаковой. Объединение происходит в порядке увеличения номеров процессов-отправителей. То есть данные, посланные процессом i из своего буфера sendbuf, помещаются в i-ю порцию буфера recvbuf процесса root. Длина массива, в который собираются данные, должна быть достаточной для их размещения.

Заключение

Была разработана параллельная версия алгоритма Шелла, которая эффективней линейной версии в два раза на достаточно больших данных. Также был проиллюстрирован в вычислительном эксперименте закон Амдала: на 16 вычислительных узлах последовательная часть программы, а именно передача данных процессам, обрабатывается достаточно много времени, из-за чего видно замедление работы. Из экспериментов можно сделать вывод, что оптимально будет запускать программу на 8 узлах при достаточно больших данных.

Код программы

```
// copyright : (C) by diper1998
#include <mpi.h>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <string>

void GetVectorInfo(const double *myVector, size_t mySize) {
  std::cout << std::endl << "Size: " << mySize << std::endl;
  std::cout << "Vector: ";
  for (unsigned i = 0; i < mySize; i++) {
    std::cout << myVector[i] << ", ";
  }
  std::cout << std::endl;
}

int Test(const double *firstVector, const double *secondVector, size_t mySize) {
  for (unsigned i = 0; i < mySize; i++) {
    if (firstVector[i] != secondVector[i]) {
      return 0;
    }
  }
  return 1;
}

void SortShell(double *myVector, size_t mySize) {
  unsigned i, j, k;
  double t;
  for (k = mySize / 2; k > 0; k /= 2)
    for (i = k; i < mySize; i++) {
      t = myVector[i];
      for (j = i; j >= k; j -= k) {
        if (t < myVector[j - k])
          myVector[j] = myVector[j - k];
        else
          break;
      }
      myVector[j] = t;
    }
}

double *SortMerge(double *myVector, double *myBuffer, unsigned borderLeft,
                  unsigned borderRight) {
  if (borderLeft == borderRight) {
    myBuffer[borderLeft] = myVector[borderLeft];
    return myBuffer;
  }

  unsigned borderMiddle = (borderLeft + borderRight) / 2;

  // divide and rule
  double *bufferLeft = SortMerge(myVector, myBuffer, borderLeft, borderMiddle);
  double *bufferRight =
```

```
    SortMerge(myVector, myBuffer, borderMiddle + 1, borderRight);

  // merge of two sorted halves
  double *target = bufferLeft == myVector ? myBuffer : myVector;

  unsigned currentLeft = borderLeft, currentRight = borderMiddle + 1;
  for (unsigned i = borderLeft; i <= borderRight; i++) {
    if (currentLeft <= borderMiddle && currentRight <= borderRight) {
      if (bufferLeft[currentLeft] < bufferRight[currentRight]) {
        target[i] = bufferLeft[currentLeft];
        currentLeft++;
      } else {
        target[i] = bufferRight[currentRight];
        currentRight++;
      }
    } else if (currentLeft <= borderMiddle) {
      target[i] = bufferLeft[currentLeft];
      currentLeft++;
    } else {
      target[i] = bufferRight[currentRight];
      currentRight++;
    }
  }
  return target;
}

int main(int argc, char *argv[]) {
  srand(static_cast<int>(time(0)));

  double *plVector;
  double *lnVector;
  size_t vectorSize;

  std::string commandArgument;
  commandArgument = argv[1];
  vectorSize = atoi(commandArgument.c_str());

  plVector = new double[vectorSize];
  lnVector = new double[vectorSize];

  // for parallel block

  double plStartTime = 0;
  double plEndTime = 0;
  double plWorkTime = 0;

  int flag;
  int myId, numProcs;

  // for line block

  double lnStartTime = 0;
  double lnEndTime = 0;
  double lnWorkTime = 0;
```

```
// Initialize the MPI environment

MPI_Init(&argc, &argv);
MPI_Initialized(&flag);  // check
if (!flag) {
  std::cout << "Error: MPI_Init";
}

// Description of the communicator
// communicator manages groups of parallel processes
// determining the number of processes in a group
MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
// determining the rank of a process in a group
MPI_Comm_rank(MPI_COMM_WORLD, &myId);

double *plBlock;
size_t blockSize = vectorSize / numProcs;
plBlock = new double[blockSize];

double *plBuffer;
plBuffer = new double[vectorSize];

if (myId == 0) {
  for (unsigned i = 0; i < vectorSize; i++) {
    lnVector[i] = static_cast<double>(std::rand() % 50) /
                (static_cast<double>(std::rand() % 100) + 1) -
                 static_cast<double>(std::rand() % 50) /
                (static_cast<double>(std::rand() % 100) + 1);
    plVector[i] = lnVector[i];
  }

  // SHELL
  std::cout << std::endl << "*LINE*";

  if (vectorSize <= 10) GetVectorInfo(lnVector, vectorSize);

  lnStartTime = MPI_Wtime();
  SortShell(lnVector, vectorSize);
  lnEndTime = MPI_Wtime();
  lnWorkTime = lnEndTime - lnStartTime;

  std::cout << std::endl << "ShellSortTime: " << lnWorkTime << std::endl;

  if (vectorSize <= 10) GetVectorInfo(lnVector, vectorSize);
  // END SHELL
}

MPI_Barrier(MPI_COMM_WORLD);

plStartTime = MPI_Wtime();

MPI_Scatter(plVector, blockSize, MPI_DOUBLE, plBlock, blockSize, MPI_DOUBLE,
          0, MPI_COMM_WORLD);

SortShell(plBlock, blockSize);

MPI_Barrier(MPI_COMM_WORLD);
```

```
  MPI_Gather(plBlock, blockSize, MPI_DOUBLE, plVector, blockSize, MPI_DOUBLE, 0,
          MPI_COMM_WORLD);

  if (myId == 0) {
    SortMerge(plVector, plBuffer, 0, vectorSize - 1);
    plEndTime = MPI_Wtime();
    plWorkTime = plEndTime - plStartTime;

    std::cout << std::endl << "*PARALLEL*";
    std::cout << std::endl << "ShellMergeSortTime: " << plWorkTime << std::endl;

    if (Test(lnVector, plVector, vectorSize)) {
      if (vectorSize <= 10) GetVectorInfo(plVector, vectorSize);
      std::cout << std::endl << "Result: plVector" << std::endl;
    }

    if (Test(lnVector, plBuffer, vectorSize)) {
      if (vectorSize <= 10) GetVectorInfo(plBuffer, vectorSize);
      std::cout << std::endl << "Result: plBuffer" << std::endl;
    }

    std::cout << std::endl
            << "SPEED-UP: " << lnWorkTime / plWorkTime << std::endl;
  }

  MPI_Finalize();

  delete[] plVector;
  delete[] lnVector;
  delete[] plBuffer;
  delete[] plBlock;

  return 0;
}
```