Лабораторные работы по курсу "Параллельное и распределённое программирование"

Игорь Комолых, Сергей Лущик

30 мая 2018 г.

1 Умножение матриц

Эта лабораторная работа заключалась в сравнении последовательной и параллельной реализации алгоритмов умножения матриц, а так же в сравнении времени работы программы при разных способах обхода массива.

В результате выполнения работы были получены:

- описанный класс Matrix
- bash и sbatch файлы запуска программы на персональных компьютерах и кластере САФУ
- python-скрипт для построения графиков на основе полученных данных.

Результатом выполнения программы является строка, в которой через запятую указаны количество используемых потоков, размерность квадратной матрицы, время работы. При запуске bash или sbatch сценариев, происходит формирование CSV-файла, по данным которого в дальнейшем можно строить графики ускорения, эффективности и времени работы (см. Рис. 1 и 2). Программы собирались и запускались на вычислительном кластере САФУ.

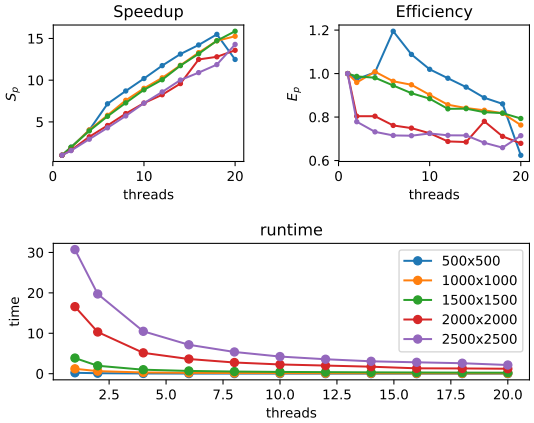

Рис. 1: графики до смены порядка обхода матриц.

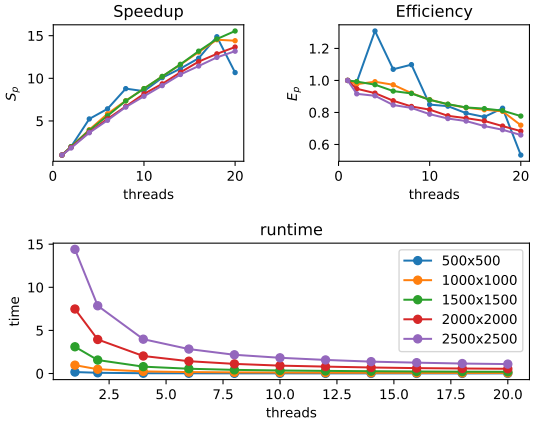

Рис. 2: графики после смены порядка обхода матриц.

По рис. 1 и 2 видно, что после изменения порядка обхода матрицы с привычного “строки-столбцы” на “столбцы-строки” (см. Листинг 1) время работы программы может сокращаться в 2 и более раза, в некоторых случаях удавалось достичь ускорения в 4-5 раз.

Данный пример демонстрирует особенности устройства кэша процессора и оперативной памяти. При обращении к какой-либо ячейке памяти, в кэш вместе с ней загружаются и несколько соседних ячеек. При обращении в порядке

“строки-столбцы” два элемента, над которыми производятся операции в смежных итерациях алгоритма, в памяти будут находиться на расстоянии, равном размеру строки матрицы. Если же обходить массивы в порядке “столбцы-строки”, смежные итерации будут оперировать элементами одной строки матрицы, элементы которой располагаются в памяти друг за другом.

```
  //
  // строки-столбцы
  //
        #pragma omp parallel for shared(result, first, second)
        for (size_t i = 0; i < result.rows(); ++i)
          for (size_t j = 0; j < result.cols(); ++j) {
            result(i, j) = 0;
            for (size_t k = 0; k < result.rows(); ++k)
              result(i, j) += first(i, k) * second(k, j);
          }

    //
    // столбцы-строки
    //

        #pragma omp parallel for shared(result, first, second)
        for (size_t j = 0; j < result.cols(); ++j)
          for (size_t i = 0; i < result.rows(); ++i) {
            result(i, j) = 0;
            for (size_t k = 0; k < result.rows(); ++k)
              result(i, j) += first(i, k) * second(j, k);
          }
```

Листинг 1: Два способа обхода матрицы

2 Задача Дирихле для уравнения Пуассона

С использованием класса матриц, полученного в ходе выполнения первой лабораторной были реализованы последовательный и параллельный алгоритмы решения задачи Дирихле для уравнения Пуассона, написаны python-скрипты для построения графиков поверхностей по полученным данным. Результат тестового запуска алгоритма для уравнения $f(x,y) = 4$ с краевыми условиями $g(x,y) = (x - 0.5)^2 + (x - 0.5)^2$ представлен на рис. 3.

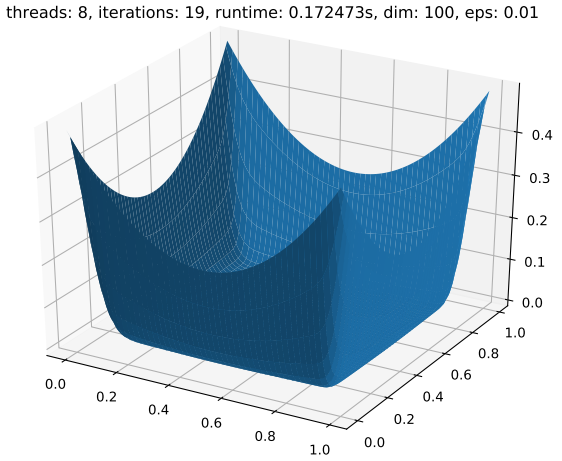

Рис. 3: График решения задачи Дирихле для уравнения Пуассона.

Для того чтобы гарантировать получение точно таких же решений, как и в непараллельном алгоритме Гаусса-Зейделя, параллельный алгоритм был построен по волновой схеме. Для вычисления значения текущего элемента $U_{i,j}$ алгоритм Гаусса-Зейделя использует два ранее вычесленных элемента $U_{i-1,j}$ и $U_{i,j-1}$, Вначале, таким условиям удовлетворяет только элемент $U_{1,1}$, однако, после его подсчета становится доступна следующая диагональ $U_{2,1} - U_{1,2}$. Получаем, что выполнение одной итерации можно разбить на последовательность шагов, на каждом из которых вычисляются узлы, расположенные на одной из диагоналей исходной сетки.

```
DirichletResult solveDirichlet(size_t N,  double eps) {
  auto startTime = std::chrono::steady_clock::now();

  Matrix u_mat(N+2,  N+2);
  Matrix f_mat(N,  N);
  double h = 1.0 / (N + 1);

  // Заполнение матриц u_mat, f_mat начальными значениями

  double max, u0, d;
  size_t i = 0, j = 0, iterations = 0;
  std::vector<double> mx(N+1);
  do {
    iterations++;
    // нарастание волны (k - длина фронта волны)
    for (size_t k = 1; k < N+1; k++) {
      mx[k] = 0;
      #pragma omp parallel for shared(u_mat, k, mx) private(i, j, u0, d) schedule(static, 1)
      for (i = 1; i < k+1; i++) {
        j = k + 1 - i;
        u0 = u_mat(i, j);
        u_mat(i, j) = 0.25 * (u_mat(i-1, j) + u_mat(i+1, j) + u_mat(i, j-1) + u_mat(i, j+1) - h*h*f_mat(i-1,
              j-1));
        d = std::fabs(u_mat(i, j) - u0);
        if (d > mx[i]) mx[i] = d;
      }
    }
    for (size_t k = N-1; k > 0; k--) {
      #pragma omp parallel for shared(u_mat, k, mx) private(i, j, u0, d) schedule(static, 1)
      for (i = N-k+1; i < N+1; i++){
        j = 2*N - k - i + 1;
        u0 = u_mat(i, j);
        u_mat(i, j) = 0.25 * (u_mat(i-1, j) + u_mat(i+1, j) + u_mat(i, j-1) + u_mat(i, j+1) - h*h*f_mat(i-1,
              j-1));
        d = std::fabs(u_mat(i, j) - u0);
        if (d > mx[i]) mx[i] = d;
      }
    }
    max = 0;
    for (i = 1; i < N+1; i++) {
      if (mx[i] > max) max = mx[i];
    }
  } while (max > eps);
  \end{minipage}

  auto runtime = std::chrono::steady_clock::now();
  auto runtimeDuration = std::chrono::duration_cast<std::chrono::duration<double>>(runtime - startTime);
  DirichletResult result(omp_get_max_threads(), u_mat, iterations, runtimeDuration.count(), eps);
  return result;
}
```

Листинг 2: solveDirichlet(size_t N, double eps)

Реализация алгоритма, представеного на листинге 2, была скомпилирована и запущена на вычислительном кластере САФУ. При замере времени работы программы на различном числе ядер, были получены графики, изображенные на рисунке 4.

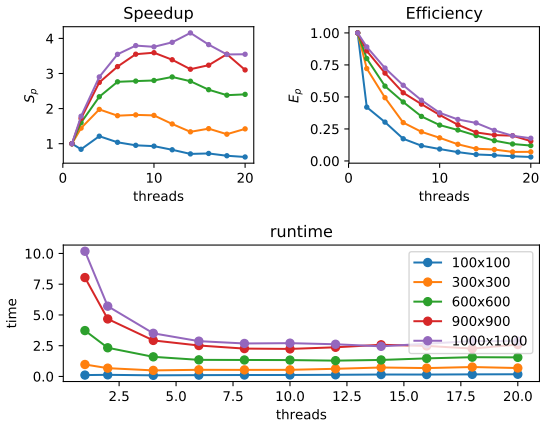

Рис. 4: графики ускорения, эффективности и времени работы волнового алгоритма.

Сравнение результатов этой и предыдущей лабораторной работы наглядно демонстрирует, что все алгоритмы обладают разной эффективностью и по-разному переносят распараллеливание. Замечено так же, что при манипуляции с флагами оптимизации при различных версиях компилятора g++ могут приводить к разным и иногда непредсказуемым результатам: так, например, при сборке проекта под платформу macOS компилятором g++ версии 8.1.0 с использованием ключа оптимизации -O2 приводило к тому, что при запуске программы на одном ядре время выполнения составляло условные 2 сек., при запуске на двух ядрах — 4 сек, на 4 — 8, и т.д. При этом, при использовании компилятора версии 5.4.0, поставляющегося в составе семейства дистрибутивов на базе Ubuntu 16.04, данная проблема не воиспроизводилась.