**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ**

**РОССИЙСКОЙ ФЕДЕРАЦИИ**

**БРЯНСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**Кафедра «Информатика и программное обеспечение»**

**ЛАБОРАТОРНАЯ РАБОТА № 5**

**ПО ДИСЦИПЛИНЕ**

**«Архитектура вычислительных систем»**

**Тема: ТЕХНОЛОГИЯ OPENmp: РЕКУРСИВНЫЙ ПАРАЛЛЕЛИЗМ   
И ПАРАДИГМА «рАЗДЕЛЯЙ И ВЛАСТВУЙ». пАРАЛЛЕЛЬНЫЕ СЕКЦИИ. УПРАВЛЕНИЕ ВЛОЖЕННЫМ ПАРАЛЛЕЛИЗМОМ**

Выполнил студент гр. О-21-ИВТ-1-по-Б

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Шпаков В.В.

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_2024 г.

Преподаватель

\_\_\_\_\_\_\_\_\_\_\_ к.ф.-м.н., доц. Дмитроченко О.Н.

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_2024 г.

**Брянск 2024**

**СОДЕРЖАНИЕ**

[1. Цель работы 3](#_Toc155916589)

[2. Теоретические сведения 4](#_Toc155916590)

[2.1. Рекурсивное программирование 4](#_Toc155916591)

[2.2. Парадигма «разделяй и властвуй» 4](#_Toc155916592)

[2.3. Описание параллельных секций 6](#_Toc155916593)

[2.4. Пример использования параллельных секций 7](#_Toc155916594)

[2.5. Вложенные параллельные области 8](#_Toc155916595)

[3. Задания 9](#_Toc155916596)

[3.1. Задание 1 9](#_Toc155916597)

[3.2. Задание 2 11](#_Toc155916598)

[3.3. Задание 3 14](#_Toc155916599)

[3.4. Анализ 2 и 3 программы 16](#_Toc155916600)

# Цель работы

Целью лабораторной работы является:

построение рекурсивных алгоритмов с разбиением решаемой задачи на несколько независимых подзадач с последующим комбинированием результатов для исполнения на многопроцессорных системах;

изучение особенностей алгоритмов, построенных на основании парадигмы «разделяй и властвуй», и этапов разделения задачи и комбинации результатов решения независимых подзадач;

приобретение практических навыков использования параллельных секций для одновременного исполнения подзадач и управления вложенным параллелизмом.

# Теоретические сведения

## Рекурсивное программирование

Использование в программах, основанных на технологии OpenMP, циклических (итеративных) конструкций предполагает распараллеливание по данным, при котором обеспечивается одновременное (параллельное) выполнение одних и тех же вычислительных действий над различными обрабатываемыми данными (например, вычисление площадей трапеций с различными длинами оснований).

Также часто для решения поставленной задачи необходимо выполнить различные процедуры обработки данных, при этом процедуры или полностью не зависят друг от друга, или же слабо связаны друг с другом. Распространенным является случай рекурсивного параллелизма, когда в программе есть одна или несколько рекурсивных процедур, и их вызовы независимы, т.е. каждая из них работает над своей частью общих данных. Такие рекурсивные процедуры можно выполнить параллельно, т.е. распараллелить по задачам, – рекурсивное распараллеливание.

Рекурсивное распараллеливание используется для решения комбинаторных задач – сортировка, планирование (задача коммивояжера) и игры (шахматы и другие).

## Парадигма «разделяй и властвуй»

«Разделяй и властвуй» – значимая парадмгма разработки алгоритмов, заключающаяся в [рекурсивном](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BA%D1%83%D1%80%D1%81%D0%B8%D1%8F) разбиении решаемой задачи на две или более подзадачи того же типа, но меньшего размера, и комбинировании их решений для получения ответа к исходной задаче. Разбиения выполняются до тех пор, пока все подзадачи не окажутся элементарными Применение параллельного программирования в численном интегрировании.

Распараллеливание алгоритма, построенного на парадигме «разделяй и властвуй», становится элементарным: на вычислители назначаются обработка отдельных ветвей решения задачи. Это позволяет легко распределять работу над одним и тем же набором данных между несколькими ядрами.

Рассмотрим ряд алгоритмов, использующих стратегию «разделяй и властвуй» и дающих прямую возможность реализовать ее так, чтобы задействовать преимущества многоядерной системы.

Сумма двух векторов

Так как элементы векторов независимы друг от друга, можно разделить общую задачу определения элементов результирующего вектора

, ,

на несколько независимых подзадач. Так, в двухпроцессорной вычислительной системе задачу можно разделить на две подзадачи для их параллельного выполнения и последующей комбинации результатов:

, ,

, .

Сортировка слиянием

Алгоритм сортировки слиянием следующий:

1. Сортируемый подмассив разбивается на две части примерно одинакового размера. Если размер подмассива равен единице, то происходит возврат из рекурсии.
2. Каждая из получившихся частей сортируется отдельно, путем двойного рекурсивного перехода к первому шагу алгоритма. Каждый рекурсивный вызов осуществляется с одним из подмассивов, полученных на первом этапе алгоритма.
3. Два упорядоченных по возрастанию массива половинного размера соединяются в один. Для этого выполняем процедуру:

а) в цикле берём меньший из двух первых элементов подмассивов и записываем его в результирующий массив;

б) счетчики номеров элементов результирующего массива и подмассива, из которого был взят элемент, увеличиваем на единицу;

в) когда один из подмассивов закончился, добавляем все оставшиеся элементы второго подмассива в результирующий массив.

Отметим, что, как только массив разделен на два подмассива, они становятся независимыми. Это показано пунктирными линиями на рис. 3. Таким образом, подмассив можно отдать процессору, чтобы тот манипулировал им произвольно, безо всякой блокировки отдельных элементов.

Быстрая сортировка (quicksort)

Алгоритм основывается на последовательном разделении сортируемого набора данных на блоки меньшего размера таким образом, что все значения одного из этих блоков не превышают значений другого блока. Шаги алгоритма:

1. Выбирается опорный элемент массива (обозначим ). С точки зрения корректности алгоритма за опорный можно выбрать любой элемент. В примере постоянно выбирается элемент на средней позиции, например на первом шаге с индексом .
2. Массив разбивается на два части сравнением остальных элементов с опорным. Все элементы, меньше опорного элемента, должны оказаться слева от него, а все элементы, больше опорного, – справа от него. Эта операция выполняется проходом массива слева и справа и обменом местами элементов, которые находятся не в своей половине массива. Так, в примере на первом проходе будет осуществлен обмен пар элементов 0 и 6, 1 и 8, 3 и 4.
3. Алгоритм вызывается рекурсивно дважды для подмассивов, лежащих слева и справа от опорного. Для второго прохода в примере выбираются подмассивы из интервала индексов и .
4. Алгоритм продолжает выполняться, пока не появятся подмассивы, состоящие из одного или двух элементов. Первый возвращается в исходном виде, во втором, при необходимости, сортировка сводится к перестановке двух элементов.

Разбиение в процессе сортировки задачи на более мелкие подзадачи дает возможность ускорения последовательного алгоритма на многопроцессорных системах. Для этого главный поток на первом проходе выбирает ведущий элемент и сортирует остальные элементы относительно него. На втором проходе два полученных независимых подмассива могут одновременно сортироваться уже двумя потоками. Третий проход позволяет использовать в процессе сортировки уже четыре потока и когда все процессоры получат свою часть, ускорение алгоритма будет максимальным.

## Описание параллельных секций

Для поддержки организации параллельных вычислений независимых подзадач в OpenMP применяется директива *sections.* Блок, описанный с помощью директивы *sections,* размещается в параллельном регионе и позволяет выделять параллельно выполняемые программные секции. Директива *sections* имеет вид:

#pragma omp sections [опция[[,] опция]...]

Эта директива определяет набор независимых секций кода, каждая из которых выполняется своей нитью. В конце блока происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки.

Директива *section* предполагает использование опций:

*private(список)* – задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено;

*reduction(оператор:список)* – задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; над локальными копиями переменных после завершения выполнения секции всеми нитями выполняется заданный оператор.

Директива *section* задаёт участок кода внутри секции *sections* для выполнения одной нитью.

#pragma omp section

Какие именно нити будут задействованы для выполнения какой секции, не специфицируется. Если количество нитей больше количества секций, то часть нитей для выполнения данной секции не будет задействована. Если количество нитей меньше количества секций, то некоторым (или всем) нитям достанется более одной секции.

Как и в конструкции *#pragma omp parallel for*, необходимо убедиться в независимости секций друг от друга, чтобы они могли выполняться параллельно. Если в секциях изменяются общие ресурсы без синхронизации доступа к ним, результат может оказаться непредсказуемым.

## Пример использования параллельных секций

В параллельном регионе расположены три секции, которые будут распределены между тремя нитями из общего пула из семи нитей, что определено опцией *num\_threads*. В секциях потоки выведут свой номер, а после достижения конца блока sections, все нити напечатают одинаковое сообщение со своим номером.

Функции, создающие параллельные секции, могут вызываться рекурсивно. Рекурсивные вызовы поддерживаются и параллельными регионами. Если создание вложенных секций разрешено, то при рекурсивных вызовах будут создаваться все новые и новые потоки. Возможно, это не то, что необходимо программисту, так как это может привести к созданию большого числа потоков. Чтобы ограничить число потоков, в программе можно запретить вложение. Тогда приложение будет рекурсивно вызывать функцию, содержащую директиву *sections*, но создание вложенных секций происходить не будет.

## Вложенные параллельные области

Параллельные области могут быть вложенными. По умолчанию вложенная параллельная область выполняется одной нитью. Это управляется установкой переменной среды *OMP\_NESTED*. Установить значение переменной среды можно в настройках проекта Microsoft Visual Studio или в командном интерпретаторе при помощи следующей команды:

SET OMP\_NESTED=true

Изменить значение переменной *OMP\_NESTED* можно также с помощью вызова функции

void omp\_set\_nested(int nested)

Функция разрешает или запрещает вложенный параллелизм. В качестве значения параметра задаётся 0 или 1. Если вложенный параллелизм разрешён, то каждая нить, в которой встретится описание параллельной области, породит для её выполнения новую группу нитей. Сама породившая нить станет в новой группе нитью-мастером.

Рассмотрим пример, иллюстрирующий использование вложенных параллельных областей. Вызов функции *omp\_set\_nested()* перед первой частью разрешает использование вложенных параллельных областей. Каждая нить внешней параллельной области породит новые нити, каждая из которых напечатает свой номер вместе с номером породившей нити.

Далее вызов *omp\_set\_nested()*запрещает использование вложенных параллельных областей. Во второй части вложенная параллельная область будет выполняться без порождения новых нитей, что и видно по выводимым строкам на экан.

# Задания

Разработать параллельные программы с использованием директивы *sections*:

1. определяющую сумму двух векторов, разделением общей задачи определения результирующего вектора на две независимые подзадачи, и последующей комбинации результатов;

осуществляющую сортировку слиянием массива с заданным пользователем размером *N* из случайных элементов;

осуществляющую быструю сортировку массива с заданным пользователем размером *N* из случайных элементов.

Алгоритм второго и третьего задания должен реализовывать функцию разбиение массива на две части с последующим рекурсивным вызовом функции для каждой из двух получившихся частей. Оценить затраченное на расчеты процессорное время:

* при последовательном исполнении алгоритма;
* параллельном исполнении алгоритма с запрещенным вложенным параллелизмом;
* параллельном исполнении алгоритма с разрешенным вложенным параллелизмом

Проанализировать полученные результаты.

## Задание 1

*Листинг 1*

|  |
| --- |
| #include <omp.h>  #include <iostream>  #include <vector>  #include <ctime>  #include <clocale>  #include <iomanip>  using namespace std;  void sumVectors(const vector<int>& a, const vector<int>& b, vector<int>& c, int start, int end) {  for (int i = start; i < end; i++) {  c[i] = a[i] + b[i];  }  }  int main() {  setlocale(LC\_ALL, "ru");  const int N = 100;  srand(time(NULL));  vector<int> a(N), b(N), c(N);  double time\_start, time\_end;    for (int j = 0; j < N; j++) {  a[j] = rand();  b[j] = rand();  }  #pragma omp parallel sections num\_threads(3)  {    #pragma omp section  {  time\_start = omp\_get\_wtime();  sumVectors(a, b, c, 0, N / 3);  }    #pragma omp section  {  sumVectors(a, b, c, N / 3, 2 \* N / 3);  }    #pragma omp section  {  sumVectors(a, b, c, 2 \* N / 3, N);  time\_end = omp\_get\_wtime();  }  }  cout << "Вектор A:" << endl;  for (int j = 0; j < N; j++) {  cout << " " << a[j] << " ";  }  cout << endl << endl;  cout << "Вектор B:" << endl;  for (int j = 0; j < N; j++) {  cout << " " << b[j] << " ";  }  cout << endl << endl;  cout << "Вектор C (результат суммы):" << endl;  for (int j = 0; j < N; j++) {  cout << " " << c[j] << " ";  }  cout << endl << endl;  cout << "Параллельный режим. Время: " << fixed << setprecision(6) << time\_end - time\_start << " секунд" << endl;  return 0;  } |

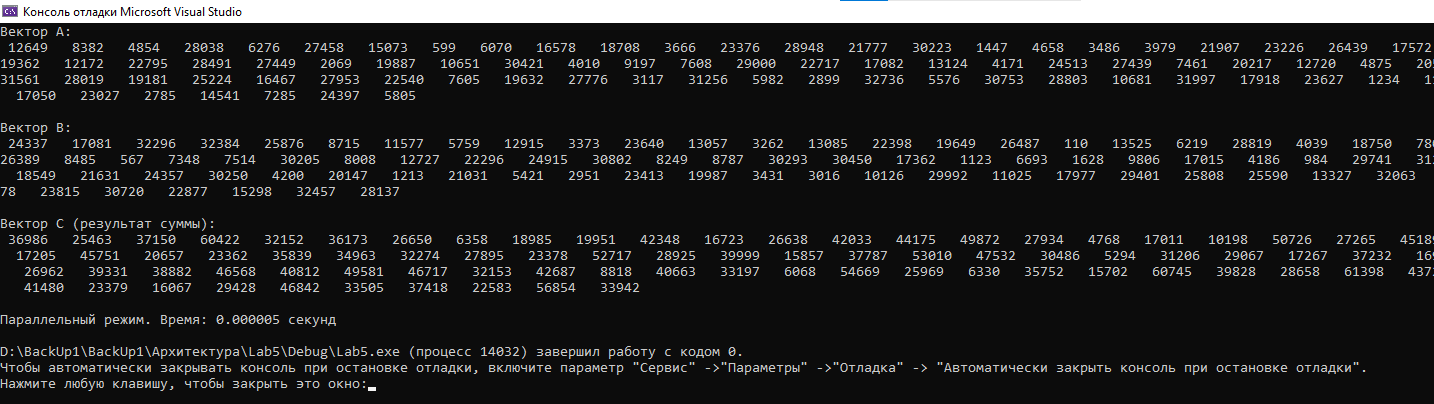


Рис. 1. Результат выполнения

## Задание 2

*Листинг 2*

|  |
| --- |
| #include <iostream>  #include <omp.h>  #include <cstdlib>  #include <ctime>  #include <clocale>  using namespace std;  void merge(int arr[], int l, int m, int r) {  int i, j, k;  int n1 = m - l + 1;  int n2 = r - m;  int\* L = new int[n1];  int\* R = new int[n2];  for (i = 0; i < n1; i++)  L[i] = arr[l + i];  for (j = 0; j < n2; j++)  R[j] = arr[m + 1 + j];  i = 0;  j = 0;  k = l;  while (i < n1 && j < n2) {  if (L[i] <= R[j]) {  arr[k] = L[i];  i++;  }  else {  arr[k] = R[j];  j++;  }  k++;  }  while (i < n1) {  arr[k] = L[i];  i++;  k++;  }  while (j < n2) {  arr[k] = R[j];  j++;  k++;  }  delete[] L;  delete[] R;  }  void mergeSortNested(int arr[], int l, int r) {  if (l < r) {  int m = l + (r - l) / 2;  #pragma omp parallel sections num\_threads(2)  {  #pragma omp section  mergeSortNested(arr, l, m);  #pragma omp section  mergeSortNested(arr, m + 1, r);  }  merge(arr, l, m, r);  }  }  void mergeSortNoNested(int arr[], int l, int r) {  if (l < r) {  int m = l + (r - l) / 2;  #pragma omp single  {  mergeSortNoNested(arr, l, m);  mergeSortNoNested(arr, m + 1, r);  }  merge(arr, l, m, r);  }  }  int main() {  srand(time(0));  setlocale(LC\_ALL, "ru");  int N;  cout << "Введите размер массива: ";  cin >> N;  int\* arrNested = new int[N];  int\* arrNoNested = new int[N];  for (int i = 0; i < N; i++) {  arrNested[i] = rand() % 100;  arrNoNested[i] = arrNested[i];  }  cout << "Неотсортированный массив: ";  for (int i = 0; i < N; i++) {  cout << arrNested[i] << " ";  }  cout << endl << endl;  double start, end;  start = omp\_get\_wtime();  mergeSortNested(arrNested, 0, N - 1);  end = omp\_get\_wtime();  cout << "Отсортированный массив (вложенный параллелизм): ";  for (int i = 0; i < N; i++) {  cout << arrNested[i] << " ";  }  cout << endl;  cout << "Время последовательного выполнения: " << end - start << " секунд" << endl;  start = omp\_get\_wtime();  mergeSortNested(arrNested, 0, N - 1);  end = omp\_get\_wtime();  cout << "Время параллельного выполнения (вложенный параллелизм): " << end - start << " секунд" << endl;  start = omp\_get\_wtime();  mergeSortNoNested(arrNoNested, 0, N - 1);  end = omp\_get\_wtime();  cout << "Отсортированный массив (без вложенного параллелизма): ";  for (int i = 0; i < N; i++) {  cout << arrNoNested[i] << " ";  }  cout << endl;  cout << "Время параллельного выполнения (без вложенного параллелизма): " << end - start << " секунд" << endl;  delete[] arrNested;  delete[] arrNoNested;  return 0;  } |

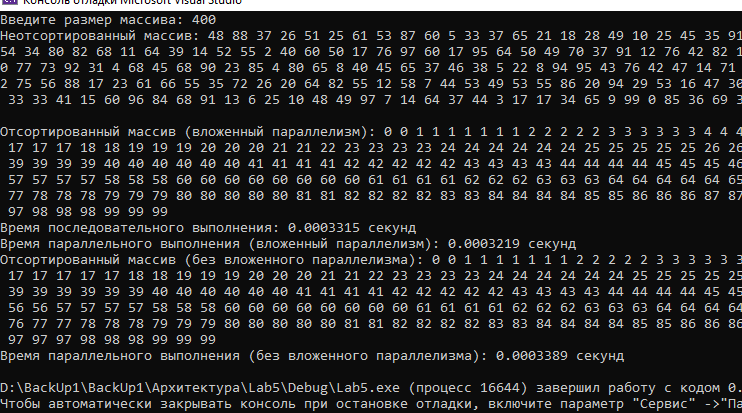


Рис. 2. Результат выполнения

## Задание 3

*Листинг 3*

|  |
| --- |
| #include <iostream>  #include <omp.h>  #include <cstdlib>  #include <ctime>  #include <clocale>  using namespace std;  int partition(int arr[], int low, int high) {  int pivot = arr[high];  int i = (low - 1);  for (int j = low; j < high; j++) {  if (arr[j] <= pivot) {  i++;  swap(arr[i], arr[j]);  }  }  swap(arr[i + 1], arr[high]);  return (i + 1);  }  void quickSort(int arr[], int low, int high) {  if (low < high) {  int pi = partition(arr, low, high);  #pragma omp parallel sections  {  #pragma omp section  quickSort(arr, low, pi - 1);  #pragma omp section  quickSort(arr, pi + 1, high);  }  }  }  int main() {  srand(time(0));  setlocale(LC\_ALL, "ru");  int N;  cout << "Введите размер массива: ";  cin >> N;  int\* arr = new int[N];  for (int i = 0; i < N; i++) {  arr[i] = rand() % 100;  }  cout << "Неотсортированный массив: ";  for (int i = 0; i < N; i++) {  cout << arr[i] << " ";  }  cout << endl << endl;  double start, end;  start = omp\_get\_wtime();  quickSort(arr, 0, N - 1);  end = omp\_get\_wtime();  cout << "Отсортированный массив: ";  for (int i = 0; i < N; i++) {  cout << arr[i] << " ";  }  cout << endl << endl;  cout << "Время последовательного выполнения: " << end - start << " секунд" << endl;  omp\_set\_nested(0);  start = omp\_get\_wtime();  quickSort(arr, 0, N - 1);  end = omp\_get\_wtime();  cout << "Время параллельного выполнения с запрещенным вложенным параллелизмом: " << end - start << " секунд" << endl;  omp\_set\_nested(1);  start = omp\_get\_wtime();  quickSort(arr, 0, N - 1);  end = omp\_get\_wtime();  cout << "Время параллельного выполнения с разрешенным вложенным параллелизмом: " << end - start << " секунд" << endl;  delete[] arr;  return 0;  } |

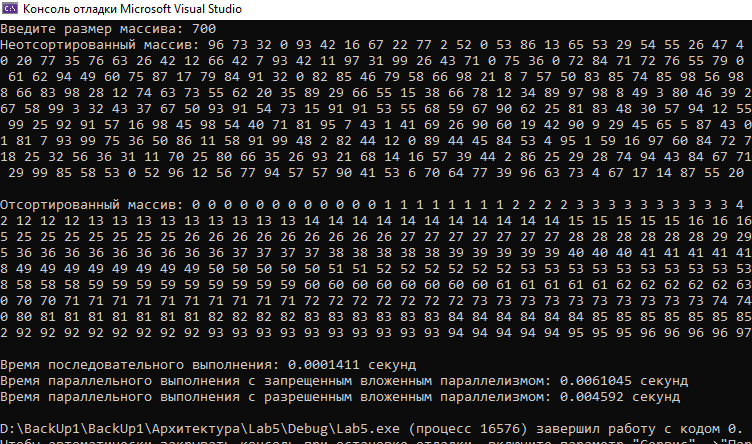


Рис. 3. Результат выполнения

## Анализ 2 и 3 программы

Сортировка слиянием:

Параллельный режим с запрещенным вложенным параллелизмом: В данном случае, запрет вложенного параллелизма привел к лучшей эффективности. Это может быть связано с тем, что в данном алгоритме создание дополнительных потоков не приносит существенной выгоды из-за относительно простой структуры сортировки слиянием.

Последовательный режим: Последовательное выполнение оказалось вторым по быстроте. Это неудивительно, так как на небольших размерах данных, создание дополнительных потоков может вызвать большие накладные расходы.

Параллельный режим с разрешенным вложенным параллелизмом: В данном случае, разрешение вложенного параллелизма не привело к значительному улучшению времени выполнения. Возможно, из-за дополнительных накладных расходов на управление потоками.

Быстрая сортировка:

Последовательный режим: В данном случае, последовательное выполнение оказалось самым быстрым. Вероятно, структура быстрой сортировки, которая хорошо подходит для последовательного выполнения, делает создание дополнительных потоков менее эффективным на небольших объемах данных.

Параллельный режим с запрещенным вложенным параллелизмом: Запрет вложенного параллелизма привел к улучшению времени выполнения, возможно, из-за уменьшения накладных расходов на управление потоками.

Параллельный режим с разрешенным вложенным параллелизмом: Разрешение вложенного параллелизма не принесло существенных выигрышей, что может быть связано с характером быстрой сортировки, где наличие большего числа потоков не всегда приносит выигрыш в производительности.

Выводы: Результаты подчеркивают важность тестирования и адаптации параллельных алгоритмов к конкретным характеристикам задачи и объему данных. На небольших объемах данных, последовательное выполнение может быть эффективнее, и внимательный выбор стратегии параллелизма необходим для достижения наилучших результатов.