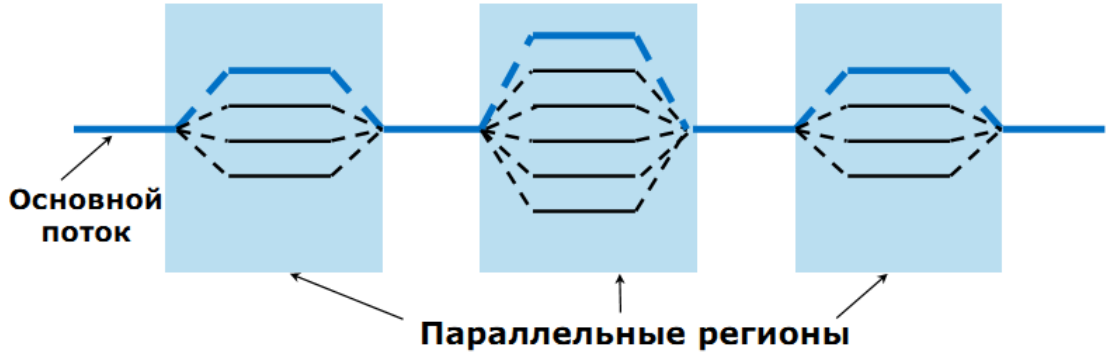
Лабораторная работа №5. Технология OpenMP

5.1 Цель лабораторной работы

Получить навыки разработки параллельных алгоритмов с использованием технологии OpenMP.

5.2 Теоретический материал

OpenMP (Open Multi-Processing) – API, предназначенное для программирования многопоточных приложений для систем с общей памятью (для языков C, C++, Fortran).



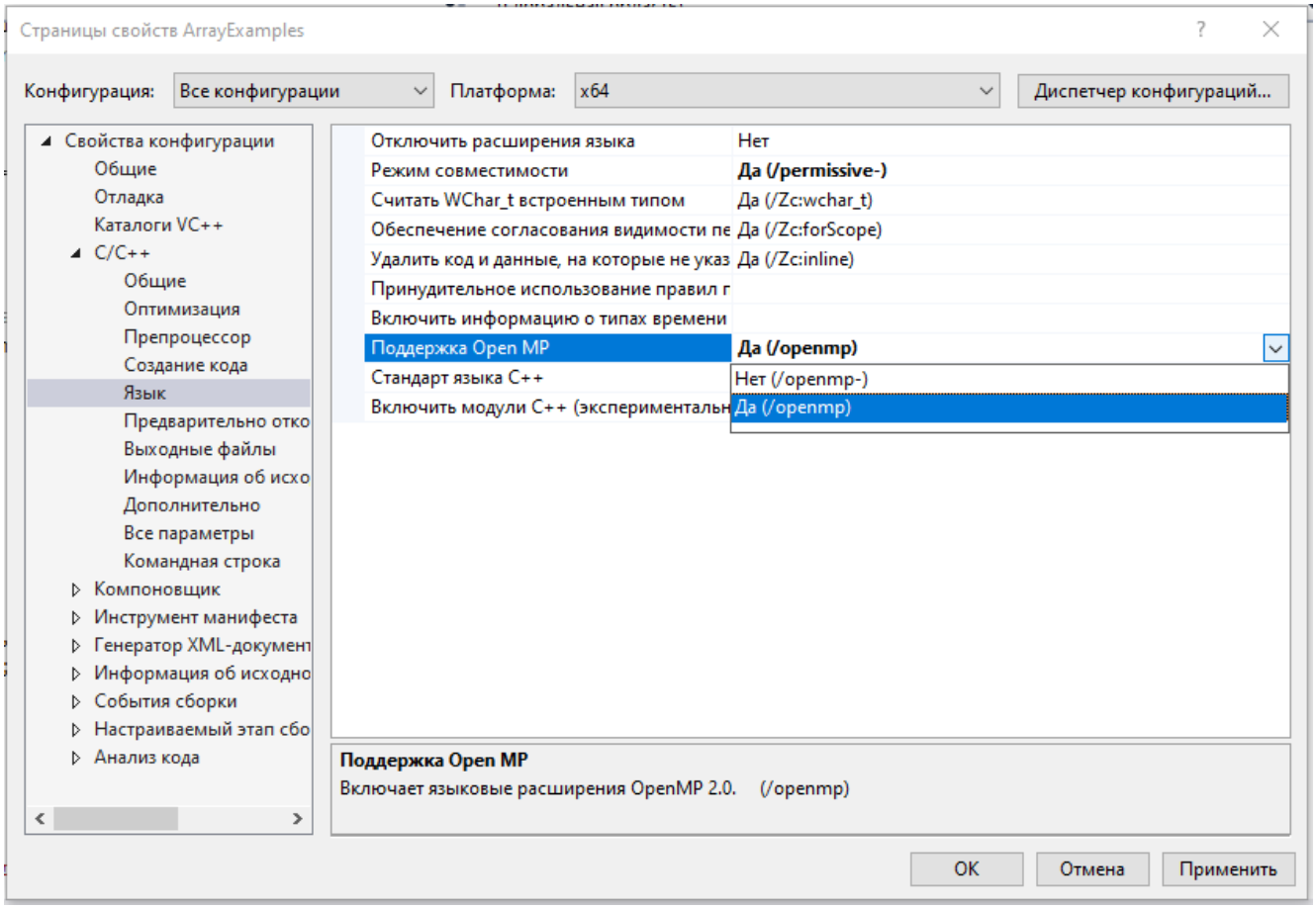
Описание стандартов можно посмотреть здесь.

OpenMP – набор параметров директивы `#pragma` и вспомогательных функций. Все директивы начинаются с `#pragma omp`. Каждая директива может иметь несколько дополнительных атрибутов. Все функции OpenMP начинаются с `omp_`. Для использования функций OpenMP необходимо подключить библиотеку `<omp.h>`.

5.2.1 Компиляция решения с поддержкой OpenMP

Компиляция программы с поддержкой OpenMP должна осуществляться с ключом ***–openmp*** (компилятор Intel) или ***–fopenmp*** (gcc).

В Visual Studio включить поддержку OpenMP можно в свойствах проекта: **Свойства проекта –> C/C++ –> Язык –> Поддержка OpenMP–>Да(/openmp)**.



Важно: при переносе проекта проверяйте, что поддержка OpenMP включена! В Visual Studio 2019 нужно будет отключить режим совместимости.

5.2.2 Проверка поддержки OpenMP

При включении поддержки OpenMP определяется константа `_OPENMP`. Проверить поддерживается ли OpenMP можно при помощи следующего фрагмента кода:

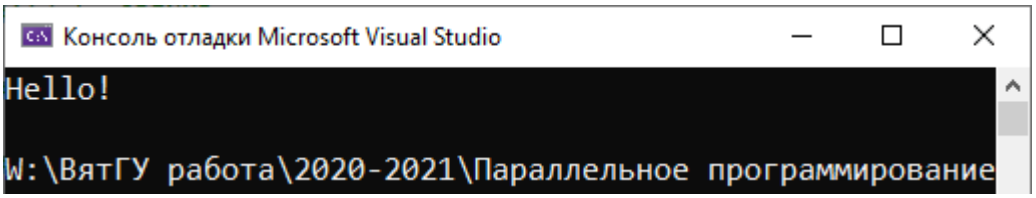
```
int main()
{
#ifdef _OPENMP
   printf("OpenMP is supported! %d\n", _OPENMP);
#else
   printf("OpenMP is not supported!\n");
#endif
}
```

При отключенном OpenMP директивы будут проигнорированы. Поэтому при запуске следующего фрагмента кода

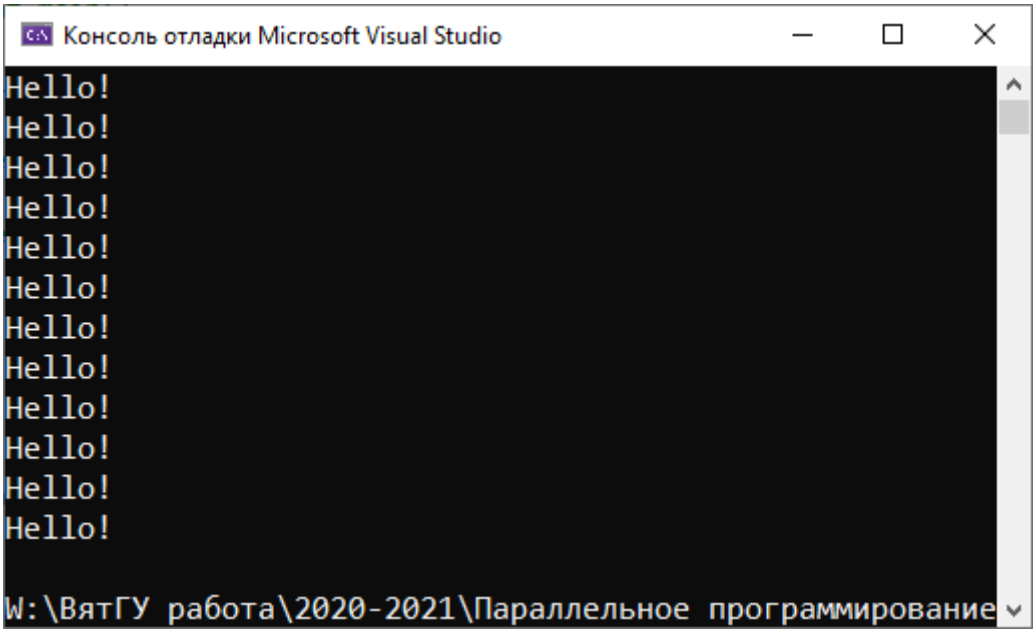
```
int main()
{
```

```
    #pragma omp parallel
    {
       cout << "Hello!\n";
    }
    return 0;
}
```

строка “Hello” будет выведена один раз.



Если включить поддержку OpenMP, то на системе с 12 потоками получим



5.2.3 Определение времени работы параллельной OpenMP-программы

Стандарт OpenMP включает определение специальных функций для измерения времени. Получение текущего момента времени выполнения программы обеспечивается при помощи функции `omp_get_wtime()`, результат вызова которой есть количество секунд, прошедших от некоторого определенного момента времени в прошлом.

Возможная схема применения функции `omp_get_wtime` может состоять в следующем:

```
double t1, t2, dt;
```

```
t1 = omp_get_wtime();
 …
t2 = omp_get_wtime ();
dt = t2 – t1;
```

Точность измерения времени также может зависеть от среды выполнения параллельной программы. Для определения текущего значения точности может быть использована функция `omp_get_wtick()`, позволяющая определить время в секундах между двумя последовательными показателями времени аппаратного таймера используемой компьютерной системы.

5.2.4 Директива parallel

Директива `parallel` определяет параллельную область. Когда основной поток доходит до параллельной области создается группа потоков. Код параллельной области дублируется между параллельно исполняемыми потоками. Если к параллельной области относится несколько операторов, то они заключаются в фигурные скобки.

```
#pragma omp parallel
{
   // Код внутри блока выполняется параллельно
   std::cout << "Hello!\n";
}
```

В конце области обеспечивается синхронизация потоков и все потоки завершаются.

Параметры (clauses) директивы parallel

- ✓ `if (условие)` – условие: группа потоков создается только в случае, если данное условие истинно;
- ✓ `num_threads(число)` – количество создаваемых потоков. Если данный параметр не указан, то значение берется из переменной окружения `OMP_NUM_THREADS`;
- ✓ `private (список_локальных_переменных)` – список локальных переменных потока, для этих переменных создаются локальные копии в локальном адресном пространстве каждого потока, начальное значение для таких переменных не определено;

- ✓ `firstprivate (список_локальных_переменных)` – список локальных переменных потока, для этих переменных создаются локальные копии в локальном адресном пространстве каждого потока, начальное значение для таких переменных соответствует значению глобальной переменной с тем же именем;
- ✓ `shared (список_разделяемых_переменных)` – список разделяемых переменных потока, использование данного параметра позволяет отследить совместное использование потоками разных переменных, по умолчанию все переменные, созданные до параллельной области, считаются разделяемыми;
- ✓ `reduction (оператор: список)` – операция редукции, которая заключается в обработке значений локальных копий переменной и получении одного глобального значения данной переменной, данный параметр подробно будет рассмотрен в подразделе 5.2.9.

Данные параметры могут быть применены не только к директиве parallel, но и ко многим другим директивам OpenMP.

Некоторые функции могут быть вызваны как из последовательной, так и из параллельной области. В этом случае возникает необходимость определить, выполняется ли фрагмент в рамках параллельного кода. Для этого можно использовать функцию `omp_in_parallel()`, которая возвращает `true`, если вызвана из параллельной области, и `false`, если вызвана из последовательной.

5.2.5 Определение и установка количества параллельных потоков

Функции `omp_get_thread_num()` и `omp_get_num_threads()` позволяют узнать индекс текущего потока и общее количество запущенных потоков. Поток с индексом 0 – главный поток. Именно он остается после завершения параллельной области.

Пример. Каждый поток печатает приветствие, главный поток также выводит общее количество потоков в параллельной области.

```
int nthreads, tid;
// Создание параллельной области
#pragma omp parallel private(tid)
```

```
{
   // печать номера потока
   tid = omp_get_thread_num();
   printf("Hello World from thread = %d\n", tid);
   // Печать количества потоков – только master
   if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
   }
} // Завершение параллельной области
```

В случае, если нужно задать необходимое количество потоков для всей программы, а не для конкретной параллельной области, можно использовать функцию `omp_set_num_threads(int  num_threads)`, где `num_threads` – требуемое количество потоков в параллельной области.

В случае, если требуемое количество потоков не задано в программе, значение берется из переменной окружения `OMP_NUM_THREADS`.

При необходимости определить оптимальное количество потоков можно использовать функцию `omp_get_num_procs()`, которая возвращает число процессоров, доступных приложению.

Функция `omp_get_max_threads()` возвращает максимально допустимое количество потоков.

Пример. Каждый поток вычисляет свою часть суммы чисел от 1 до *N*.

```
long long Sum(int);

int main()
{
   cout << "Start program!\n";
   long long totalSum = 0, localSum = 0;
   int N = 1000000000;
   // Создание параллельной области
   #pragma omp parallel firstprivate(localSum)
                        shared(totalSum)
   {
      localSum = Sum(N);
      cout << "LocalSum: " << localSum << endl;
      totalSum = totalSum + localSum; //так делать нельзя!
```

```
    } // Завершение параллельной области
    cout << "Total sum = " << totalSum << endl;
}

long long Sum(int max)
{
    int step = 1;
    //если функция запущена из параллельной области
    if (omp_in_parallel())
        //получаем число параллельно выполняемых потоков
        step = omp_get_num_threads();
    long long s = 0;
    // получаем номер текущего потока
    int tid = omp_get_thread_num();
    // считаем нужную сумму, каждый поток суммирует числа,
    //имеющие остаток tid + 1 при делении на step
    for (int i = tid + 1; i <= max; i = i + step)
        s = s + i;
    return s;
}
```

5.2.6 Директива for

Директива `for` позволяет распределить между параллельно выполняемыми потоками итерации цикла, который должен следовать непосредственно после директивы. Потоки не создаются.

Директива `for` накладывает ограничения на структуру соответствующего цикла, который должен иметь каноническую форму:

```
for (idx=start; idx { <  } end; { idx++          })
                    { <= }      { ++idx         }
                    { >= }      { idx--         }
                    { >  }      { --idx         }
                                { idx += inc    }
                                { idx -= inc    }
                                { idx = idx + inc }
                                { idx = inc + idx }
                                { idx = idx - inc }
```

Переменная цикла (`idx`) не должна модифицироваться внутри тела оператора цикла. Если переменная цикла не определена как `lastprivate`, то после выполнения цикла её значение не определено.

Правильность программы не должна зависеть от того, какой поток выполняет конкретную итерацию.

Параметры (clauses) директивы `for`

- ✓ `private (список_локальных_переменных)`;
- ✓ `firstprivate (список_локальных_переменных)`;
- ✓ `lastprivate (список_локальных_переменных)` – глобальной переменной после параллельного региона присваивается значение из потока, который бы последним исполнялся последовательно;
- ✓ `reduction (оператор: список)`;
- ✓ `ordered` – используется совместно с директивой ordered, позволяет задать фрагмент тела цикла, который все потоки должны выполнять в исходном порядке;
- ✓ `schedule (вид [, размер_порции])` – определяет способ распределения итераций цикла между потоками. Размер порции должен быть инвариантным относительно цикла положительным целочисленным значением. Вид планирования может быть один из следующих:
  - `static` – итерации делятся на блоки по chunk итераций и статически разделяются между потоками; если параметр chunk не определен, итерации делятся между потоками равномерно и непрерывно;
  - `dynamic` – распределение итерационных блоков осуществляется динамически (по умолчанию chunk = 1);
  - `guided` – размер итерационного блока уменьшается экспоненциально при каждом распределении; chunk определяет минимальный размер блока (по умолчанию chunk = 1);

- `auto` – решение по назначению размера порций делегировано компилятору и определяется во время выполнения, параметр chunk не указывается;
- `runtime` – правило распределения определяется переменной `OMP_SCHEDULE`, при использовании runtime параметр chunk не задается;

✓ `nowait` – используется в случае, если в конце не требуется синхронизация, при отсутствии данного параметра в конце параллельного цикла происходит неявная барьерная синхронизация параллельно работающих потоков: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки.

В случае, если в параллельной области находится только один цикл, итерации которого необходимо раздать параллельно выполняемым потокам, директивы parallel и for можно объединить

```
#pragma omp parallel for [clauses]
```

Пример. Суммируются два массива размера *N*. Замеряется время суммирования

```
long long totalSum = 0;
int N;
N = 100000000; //10^8
double start, end, diff;
//Создадим три массива размера N
int *v1 = new int[N],
    *v2 = new int[N],
    *vres = new int[N];
//Проинициализируем созданные массивы
for (int i = 0; i < N; ++i)
{
   v1[i] = 1;   //все единицы
   v2[i] = i;   //числа от 0 до N-1
   vres[i] = 0; //обнулим
}
int i;
start = omp_get_wtime(); //начинаем замерять время
//Создаем параллельный регион
```

```
//переменная цикла - разделяемая
#pragma omp parallel shared(i)
{
   //Раздадим итерации цикла последовательными порциями
   #pragma omp for schedule(static)
   for (i = 0; i < N; ++i) //цикл
      vres[i] = v1[i] + v2[i]; //итерации независимы
}
end = omp_get_wtime(); //заканчиваем отсчет времени
diff = end - start;   //время вычислений
cout << diff << endl; //выводим на консоль

int err = 0; //Проверим правильность работы программы
for (int i = 0; i < N; ++i)
   if (vres[i] != i + 1) //если элемент неверный
   {
      err++; //увеличим число ошибок
      cout << i << ": " << vres[i]; //выведем его
   }
if (err) //если была хотя бы одна ошибка
//то выведем сообщение об ошибках
   cout << "Wrong program, num errors: " << err << endl;
else cout << "Good" << endl; //иначе выведем “Good”
```

5.2.7 Директива sections

Директива `sections` позволяет реализовать параллелизм по задачам. Потоки не создаются.

Синтаксис:

```
#pragma omp sections
{
#pragma omp section
    операторы
#pragma omp section
    операторы
}
```

Параметры (clauses) директивы `sections`

- ✓ `private (список_локальных_переменных);`
- ✓ `firstprivate (список_локальных_переменных);`
- ✓ `lastprivate (список_локальных_переменных);`
- ✓ `reduction (оператор: список);`
- ✓ `nowait`.

В случае, если в параллельной области находится только один цикл, итерации которого необходимо раздать параллельно выполняемым потокам, директивы parallel и for можно объединить

```
#pragma omp parallel sections [clauses]
```

Пример. В параллельном регионе создаются две секции. Поток, выполняющий секцию выводит соответствующее сообщение.

```
string message;
int tid;
#pragma omp parallel num_threads(5) private(message, tid)
{
   tid = omp_get_thread_num();
   message = "Hello, tid = " + to_string(tid) + "\n";
   cout << message;
   message.clear();
   #pragma omp sections
   {
      #pragma omp section
      {
         message = "Section 1, tid = " + to_string(tid) +
"\n";
         cout << message;
      }
      #pragma omp section
      {
         message = "Section 2, tid = " + to_string(tid) +
"\n";
         cout << message;
      }
   }
}
```

5.2.8 Вложенный параллелизм

Вложенный параллелизм позволяет создавать новые параллельные регионы внутри исходных. Ниже приведен пример использования вложенных параллельных регионов. Также вложенный параллелизм может потребоваться при распараллеливании рекурсивных алгоритмов.

```
int main()
{
   omp_set_nested(1);
   #pragma omp parallel num_threads(2)
   {
      #pragma omp sections
      {
         #pragma omp section
         {
            #pragma omp parallel num_threads(5)
            cout << "Hello\n";
         }
         #pragma omp section
         {
            #pragma omp parallel num_threads(7)
               cout << "Bye\n";
         }
      }
   }
   return 0;
}
```

5.2.9 Операция редукции

Параметр `reduction` определяет список переменных, для которых выполняется операция редукции. Перед выполнением параллельной области для каждого потока создаются копии этих переменных. Потоки формируют значения в своих локальных переменных, при завершении параллельной области над всеми локальными значениями выполняются необходимые операции редукции, результаты которых запоминаются в исходных (глобальных) переменных.

Синтаксис:

```
reduction (оператор: список_переменных)
```

Переменная, по которой проводится редукция, должна быть скалярной, в выражении она может присутствовать только один раз. Список операторов ограничен следующими: `+, -, *, &, ^, |, &&, ||, max, min`.

Пример. Вычислим сумму элементов массива.

```
int N = 100000000; //10^8
long long sum = 0;
int *arr = new int[N];
for (int i = 0; i < N; ++i)
   arr[i] = 2 * i + 1;
#pragma omp parallel reduction(+: sum)
{
   #pragma omp for schedule(dynamic, 1000)
   for (int i = 0; i < N; ++i)
      sum = sum + arr[i];
}
delete[]arr;
```

5.2.10 Организация взаимоисключения при использовании общих переменных

Наиболее простой способ синхронизации потоков – использование атомарных операций.

Директива `#pragma omp atomic` определяет переменную, доступ к которой (чтение/запись) должен быть выполнен как неделимая операция. Список допустимых операций приведен ниже:

```
+, -, *, /, &, ^, |, >>, <<
```

В случае, если нужно организовать более сложное взаимоисключение потоков или операция недопустимая, то можно использовать директиву

```
#pragma omp critical [name],
```

которая определяет фрагмент кода, который должен выполняться только одним потоком в каждый текущий момент времени (критическая секция).

Пример. Вычислим сумму элементов массива.

```
int N = 100000000; //10^8
long long sum = 0;
int *arr = new int[N];
for (int i = 0; i < N; ++i)
   arr[i] = 2 * i + 1;
#pragma omp parallel shared(sum) private(local_sum)
{
   #pragma omp for schedule(dynamic, 1000)
   for (int i = 0; i < N; ++i)
      local_sum = local_sum + arr[i];
   #pragma omp critical
      sum = sum + local_sum;
}
delete[]arr;
```

Для взаимоисключения потоков также можно использовать замки, которые работают как мьютексы.

5.3. Задание на лабораторную работу

0. Проверить, поддерживается ли OpenMP. Если нет, то включить поддержку в параметрах проекта.

1. Скопировать фрагмент кода в программу. Запустить на выполнение.

```
#include <omp.h>
int main () {
  int nthreads, tid;
  // Создание параллельной области
#pragma omp parallel private(tid)
  {
    // печать номера потока
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    // Печать количества потоков – только master
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
```

```
    } // Завершение параллельной области
}
```

Ответить на вопросы:

а) Зачем нужна директива `parallel`?

б) Сколько потоков было запущено? Почему?

в) Сколько потоков одновременно работают с переменной `tid`? Почему?

г) Поток с каким `tid` останется после завершения параллельной области?

Запросить требуемое количество потоков у пользователя. Задать количество потоков для параллельной области.

2. Написать программу, задающую работу двух потоков. Первый поток в цикле выводит последовательно числа от 1 до N, а второй – N раз выводит слово «HELLO». Число N задаётся пользователем.

3. Написать параллельную программу, находящую поэлементное произведение двух массивов размера N. Задать параметр `schedule`, попробовать разные аргументы.

4. Написать параллельную программу, вычисляющую максимальное значение среди элементов вектора, используя директивы `critical`.

5. Написать программу, задающую работу $M + K$ потоков. Первые M потоков вычисляют сумму от 1 до N, а оставшиеся K потоков вычисляют длину N-мерного вектора. Число N задаётся пользователем.

Указание. Использовать секции и вложенный параллелизм.

6. Написать параллельную программу, которая каждый элемент вектора размера N заменяет на его наибольший простой делитель. Число N задается пользователем. Элементы вектора – случайные натуральные числа из диапазона $[10^5, 10^6]$.

Замерить среднее время выполнения программы для $N = 2 \cdot 10^7$, $5 \cdot 10^7$ и 10^8 на 1, 2, 4 и 8 потоках. Вычислить среднее ускорение для 2, 4 и 8 потоков. Построить диаграмму зависимости ускорения от числа потоков для каждого размера вектора (3 графика на одной диаграмме).

7. Написать параллельную программу, выполняющую умножение двух матриц размера $N \times N$. Разработать программы с использованием распараллеливания циклов разного уровня вложенности.

Замерить среднее время выполнения программ для $N = 500, 1000, 2000$ на 1, 2, 4 и 8 потоках. Сравнить полученные результаты. Оцените величину накладных расходов на создание и завершение потоков.

Для оптимального варианта вычислить среднее ускорение на 2, 4 и 8 потоках. Построить диаграмму зависимости ускорения от числа потоков для каждого размера матриц (3 графика на одной диаграмме).

8. Написать параллельную программу, выполняющую поиск максимального значения среди минимальных элементов строк матрицы размера $N \times N$.

Обосновать выбор средств и методов для распараллеливания.

Замерить среднее время выполнения программ для $N = 500, 1000, 2000$ на 1, 2, 4 и 8 потоках. Вычислить среднее ускорение на 2, 4 и 8 потоках.

5.4. Результаты лабораторной работы

Результаты лабораторной работы представляются в виде отчета по лабораторной работе. В отчет включается титульный лист, цель работы, задание на лабораторную работу, описание и обоснование правильности алгоритма, листинг с комментариями, скриншоты, доказывающие правильность работы программы, полученные результаты и выводы по лабораторной работе.

Пример оформления титульного листа приведен на следующей странице.

Отчет оформляется в электронном виде и высылается на e-mail vbyzov.vyatsu@gmail.com (в теме или тексте письма, а также в названии документа с отчетом должны фигурировать ФИ студента, его группа, номер лабораторной работы).

Лабораторная работа считается зачтенной после её устной защиты у преподавателя.