###### **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 № 1**

Определение времени работы прикладных программ

студента 2 курса, группы 23201

Сорокина Матвея Павловича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А.С. Матвеев

Новосибирск 2024

**СОДЕРЖАНИЕ**

[**ЦЕЛЬ** 3](#_Toc176299562)

[**ЗАДАНИЕ** 3](#_Toc176299563)

[**ОПИСАНИЕ РАБОТЫ** 4](#_Toc176299564)

[**ЗАКЛЮЧЕНИЕ** 8](#_Toc176299566)

[**ПРИЛОЖЕНИЯ** 9](#_Toc176299567)

# **ЦЕЛЬ**

Изучение методов измерения времени работы программы, оптимизации этих измерений, анализ влияния различных уровней оптимизации компилятора *GCC* на время выполнения программы.

# **ЗАДАНИЕ**

1. Написать программу на языке *C* или *C++*, которая реализует выбранный алгоритм из задания.
2. Проверить правильность работы программы на нескольких тестовых наборах входных данных.
3. Выбрать значение параметра *N* таким, чтобы время работы программы было порядка 15 секунд.
4. По приведенной методике определить время работы подпрограммы тестовой программы с относительной погрешностью не более 1%.
5. Составить отчет по лабораторной работе.

# **ОПИСАНИЕ РАБОТЫ**

1. Реализовано задание 2 - алгоритм вычисления числа Пи метом Монте-Карло на языке *C++*.

Также предоставлен *bash-скрипт* ***compile\_and\_run.sh***, компилирующий и запускающий программу с конкретными знаениями N, записывающий время вычисления числа Пи с различными уровнями оптимизации компилятора *GCC* в файл *report.csv* (см. Приложение 3), после чего запускает ***create\_table.sh***, визуализирующий информацию из *report.csv* в терминале (см. Приложение 4).

Время замерялось перед началом и после окончания работы функции, вычисляющей число Пи. Разность этих двух значений дает общее время выполнения функции. Для проверки точности измерений, код программы запускается несколько раз.

1. Для измерения времени функции в разделе «СПОСОБЫ ПОЛУЧЕНИЯ ПОКАЗАНИЙ НЕКОТОРЫХ ТАЙМЕРОВ» были выбраны пункты **4.2** и **4.3**
2. Код программы с использованием библиотечной функции ***clock\_gettime*** из библиотеки *time.h* предоставлен (см. Приложение 1).

|  |
| --- |
| int clock\_gettime(clockid\_t *clockid*, struct timespec \**tp*); |

Функция ***clock\_gettime*** с параметром *CLOCK\_MONOTONIC\_RAW* сохраняет значение системного таймера в переменную структуры *timespec*. В приложении 1 мы видим что используются две переменные данной структуры: start и end.   
Структура состоит из двух полей: *tv\_sec* и *tv\_nsec*. В связи с этим, вычисление переменной taken\_time происходит по следующей формуле:

|  |
| --- |
| double taken\_time = (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / 1e9; |

# Обратившись к сайту <https://man7.org/linux/man-pages/man3/clock_gettime.3.html>, стоит заметить, чтовсего существует 11 таймеров, поддерживаемых *Linux*, которые мы можем указать в параметре *clock\_id.* Ниже представлены описания несколько таймеров:

* ***CLOCK\_REALTIME:*** таймер, измеряющий ”реальное” время. Значение может быть измененно вручную пользователем, с помощью команды *clock\_settime()* или во время синхронизации времени через *NTP*.

**NTP (Network Time Protocol)** - сетевой протокол, используемый для синхронизации времени на компьютерах и серверах через интернет. Он поддерживает точное время на устройствах, основываясь на специальных серверных источниках точного времени.

* ***CLOCK\_MONOTONIC***: отсчитывает монотонное время, которое начинается с какого-то неопределенного момента (в Linux — количество секунд, прошедших с запуска системы). Монотонное время не подвержено изменениям системного времени, не будет изменено в случае изменения системного времени вручную, но может подвергаться мелким корректировкам, связанным с синхронизацией системы, например, через *NTP.* Также не учитывается время, когда процесс был приостановлен (например, с помощью *CTRL-Z*).

Важно: если пользователь вручную изменит системное время через настройки операционной системы, то системное время (например, возвращаемое функцией *CLOCK\_REALTIME*) будет изменено, но время, отсчитываемое с помощью *CLOCK\_MONOTONIC*, не изменится.

* ***CLOCK\_MONOTONIC\_COARSE***: более быстрая, но менее точная версия *CLOCK\_MONOTONIC*. Меньшее время выполнения достигается с помощью использования *vDSO*.

**vDSO (virtual dynamic shared object)** - механизм ядра для экспорта набора подпрограмм пространства ядра в приложения пространства пользователя.

Некоторые системные вызовы ядра, которые используются в пользовательском коде настолько часто, что это сильно влияет на общую производительность, так как идут временные затраты на переключение контекста из пользовательского режима в режим ядра. vDSO предоставляет оптимизированный путь. Предоставляется доступ к разделяемой области памяти, где ядро уже записало значения времени. Программа считывает эти данные из памяти напрямую, без переключения в режим ядра.

Использовалась информация с сайта: https://man7.org/linux/man-pages/man7/vdso.7.html

* ***CLOCK\_BOOTTIME***: аналогичен *CLOCK\_MONOTONIC*, но позволяет учитывать время, когда процесс был приостановлен.
* ***CLOCK\_MONOTONIC\_RAW:*** монотонное время, но в отличие от **C**LOCK\_MONOTONIC, оно представляет "сырое" аппаратное время, которое не подвергается никаким корректировкам, включая ***NTP***.

1. Код программы с использованием библиотечной функции ***times*** предоставлен (см. Приложение 2).

Функция ***times*** возвращает количество тиков (ticks), затраченных процессом на выполнение программы.

Тик - минимальная единица времени, которые процессор использует для работы над процессом.

Функция ***sysconf(\_SC\_CLK\_TCK),*** возвращает количество тиков в секунду. В моем коде это переменная ***clocks\_per\_sec:***

|  |
| --- |
| long clocks\_per\_sec = sysconf(\_SC\_CLK\_TCK); |

В моем коде время выполнения вычисляется как разница между значениями *tms\_utime* до и после выполнения функции, вычисляющей число Пи, деленная на количество тиктов в секунду:

|  |
| --- |
| double taken\_time = static\_cast<double>(end.tms\_utime - start.tms\_utime) / static\_cast<double>(clocks\_per\_sec); |

Обратившись к сайту <https://man7.org/linux/man-pages/man2/times.2.html>, можем понять, как устроена структура ***tms****.*

A white text with black text

Description automatically generated with medium confidence

Функция ***times()*** заполняет структуру *tms* значениями времени, которое процесс провел в различных состояниях:

* tms\_utime: тики процессора или же *CPU ticks*, потраченных процессором на выполнение инструкций пользователя (инструкций программы, не относящихся к системным вызовам).
* tms\_stime: время (в тиках), которое процессор тратит на выполнение кода ядра операционной системы, который выполняет системные вызовы (system calls).
* tms\_cutime: количество тиков, потраченных на выполнение инструкций пользователями (в пространстве пользователя) всех дочерних процессов данного процесса.

Cодержит сумму значений полей *tms\_utime* и *tms\_cutime* для всех ожидающих завершения дочерних процессов.

* tms\_cstime: количество тиков, потраченных на выполнение системных вызовов (в режиме ядра) всеми дочерними процессами данного процесса.

Cодержит сумму значений полей *tms\_stime* и *tms\_cstime* для всех ожидающих завершения дочерних процессов.

В моей программе используются только поля *tms\_utime* (пользовательский режим) для измерения времени выполнения задачи.

Итого, код в приложении 2 работает следующим образом:

* захват начального времени (в тиках) до вызова функции ***PiCalculation***.
* захват конечного времени (в тиках) после вызова функции.
* вычисление разницы между этими значениями, чтобы получить количество тиков, потраченных на выполнение функции.
* деление количества тиков на значение ***clocks\_per\_sec***, чтобы получить время выполнения в секундах.

# **ЗАКЛЮЧЕНИЕ**

В ходе данной лабораторной работы мы познакомились с различными методами измерения работы программ и научились пользоваться ими на практике.

# **ПРИЛОЖЕНИЯ**

**Приложение 1:Исходный код программы, с использованием библиотечной функции clock\_gettime**

|  |
| --- |
| #include <iostream>  #include <time.h>  #include <cstdlib> // for atoi and atof  #include <cmath> // for pow  long double PiCalculation(long long n) {      long long m = 0;      srand(time(NULL));      for (long long i = 0; i < n; i++) {          // double x = static\_cast<double>(rand()) / RAND\_MAX // диапозон [0;1]          // RAND\_MAX \* 2.0 // увеличиваем диапозон до [0; 2]          // RAND\_MAX \* 2.0 - 1.0 // сдвигаем диапозон до [-1; 1]          double x = static\_cast<double>(rand()) / RAND\_MAX \* 2.0 - 1.0;          double y = static\_cast<double>(rand()) / RAND\_MAX \* 2.0 - 1.0;          if (x \* x + y \* y <= 1.0) {              m++;          }      }      return 4.0 \* m / n;  }  int main(int argc, char \*argv[]) {      if (argc != 2) {          std::cerr << "Invalid number of arguments" << std::endl;          return 0;      }      struct timespec start, end;      long long n = atoll(argv[1]);      std::cout << "N = " << n << std::endl;      int runs = 5;      double time\_total = 0;      for (long long i = 0; i < runs; i++) {          clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);          double pi = PiCalculation(n);          clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end);          double taken\_time = (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / 1e9;          std::cout << "pi = " << pi << std::endl;          // std::cout << "Run #" << i + 1 << " took " << taken\_time << " seconds to complete" << std::endl;          std::cout << "Run time: " << taken\_time << "\n" << std::endl;          time\_total += taken\_time;      }      std::cout << "Average time: " << time\_total / runs << " seconds" << std::endl;      return 0;  } |

**Приложение 2:Исходный код программы, с использованием библиотечной функции times**

|  |
| --- |
| #include <iostream>  #include <unistd.h> // for sysconf  #include <cstdlib> // for atoi and atof  #include <cmath> // for pow  long double PiCalculation(long long n) {      long long m = 0;      srand(time(NULL));      for (long long i = 0; i < n; i++) {          double x = static\_cast<double>(rand()) / RAND\_MAX \* 2.0 - 1.0;          double y = static\_cast<double>(rand()) / RAND\_MAX \* 2.0 - 1.0;          if (x \* x + y \* y <= 1.0) {              m++;          }      }      return 4.0 \* m / n;  }  int main(int argc, char \*argv[]) {      if (argc != 2) {          std::cerr << "Invalid number of arguments" << std::endl;          return 0;      }        struct tms start, end;      long clocks\_per\_sec = sysconf(\_SC\_CLK\_TCK);      long long n = atoll(argv[1]);      std::cout << "N = " << n << std::endl;      int runs = 5;      double time\_total = 0;      for (long long i = 0; i < runs; i++) {          times(&start);          double pi = PiCalculation(n);          times(&end);          double taken\_time = static\_cast<double>(end.tms\_utime - start.tms\_utime) / static\_cast<double>(clocks\_per\_sec);          std::cout << "pi = " << pi << std::endl;          std::cout << "Run time: " << taken\_time << "\n" << std::endl;          time\_total += taken\_time;      }      std::cout << "Average time: " << time\_total / runs << " seconds" << std::endl;      return 0;  } |

**Приложение 3: bash-скрипт для компиляции и запуска программы, записи результата в файл report.csv**

|  |
| --- |
| #!/bin/bash  input="PiCalculation.cpp"  optimization\_levels=("-O0" "-O1" "-O2" "-O3" "-Os")  n=650000000  echo "Optimixation level, N value, Time taken (seconds)" > report.csv  for i in "${optimization\_levels[@]}"; do      g++ -o pi $input $i -std=c++11      # output=$(./pi $n | grep "Average time:" | awk '{print $3}')      output=$(./pi $n)      # время выполнения каждой итерации      echo "$output" | grep "Run time:" | awk -v opt="$i" -v N="$n" '{print opt ", " N ", " $3}' >> report.csv      # среднее время      avg\_time=$(echo "$output" | grep "Average time:" | awk '{print $3}')      echo "$i, $n, $avg\_time" >> report.csv  done  echo "Successfully generated report.csv"  ./create\_table.sh |

**Приложение 4: bash-скрипт для визуализации результатов временных измерений, записанных в report.csv**

|  |
| --- |
| #!/bin/bash  if [ ! -f report.csv ]; then      echo "Файл report.csv не найден!"      exit 1  fi  # заголовок таблицы  echo -e "Оптимизация\t N\t\t Время выполнения (сек)"  prev=""  # -F',': запятая в качестве разделителя столбцов  # NR>1: пропускаем первую строку  awk -F',' 'NR>1 {      if (prev != $1 && NR > 2) {          print ""; # перевод на новую строку      }      printf "%s\t\t%s\t%s\n", $1, $2, $3;      prev = $1;  }' report.csv |

**Приложение 5: результат работы bash-скрипта *compile\_and\_run.sh***

A screenshot of a computer

Description automatically generated

**Приложение 6: результаты измерений с использованием *bash-скрипта* *compile\_and\_run.sh* в *report.csv***

**A screenshot of a computer

Description automatically generated**

**Приложение 7: результат работы скомпилированной программы *PiCalculation.cpp* не используя *bash-скрипты***

A screenshot of a computer

Description automatically generated

**Приложение 8: код для создания графика зависимости времени от оптимизации компиляции на языке *Python***

|  |
| --- |
| import matplotlib.pyplot as plt  optimization\_levels = ['-O0', '-O1', '-O2', '-O3', '-Os']  times\_o0 = [14.423, 14.2111, 14.1823, 14.1113, 14.1713, 14.2198]  times\_o1 = [13.9702, 13.9063, 13.9273, 13.9084, 13.8982, 13.9221]  times\_o2 = [13.9923, 13.8732, 13.8724, 13.8527, 13.8729, 13.8927]  times\_o3 = [13.9162, 13.8699, 13.8782, 13.8897, 13.9842, 13.9077]  times\_os = [13.7664, 13.8066, 14.025, 13.2114, 13.7947, 13.9208]  # последних значения времени (средние) для каждого уровня оптимизации  last\_times = [      times\_o0[-1],      times\_o1[-1],      times\_o2[-1],      times\_o3[-1],      times\_os[-1]  ]  plt.figure(figsize=(12, 6))  plt.plot(optimization\_levels, last\_times, 'o-', color='skyblue')  plt.xlabel('Optimization Level')  plt.ylabel('Last Time Taken (seconds)')  plt.title('Last Execution Time vs Optimization Levels')  plt.grid(True)  plt.show() |

**Приложение 9: график зависимости времени от оптимизации компиляции**

***A graph with a line going up

Description automatically generated***