Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа программной инженерии

КУРСОВАЯ РАБОТА

по дисциплине: «Алгоритмы и структуры данных»

Выполнила
студентка гр. в5130904/30022 Г.М.Феллер

Руководитель
старший преподаватель С.А.Федоров

«_____»_____ 2024 г.

Санкт-Петербург
2024

Содержание

Задачи курсовой работы

Дан список владельцев телефонов в виде:

```
ФАМИЛИЯ    ТЕЛЕФОН
15 симв.   10 симв.
```

Пример входного файла:

```
Петров          9111634576
Фёдоров         9111635687
```

Отсортировать этот список в порядке убывания номеров телефонов, используя метод вставок.

Пример выходного файла:

```
Петров          9111634576
Фёдоров         9111635687
```

Введение

Цель курсовой работы - выбор структуры данных для решения поставленной задачи на современных микроархитектурах.
Для достижения поставленной цели нехободимо выполнить следующие задачи:

1. Реализовать задание с использованием массивов строк.
2. Реализовать задание с использованием массивов символов.
3. Реализовать задание с использованием массивов структур.
4. Реализовать задание с использованием структур массивов.
5. Реализовать задание с использованием массивов структур или структур массивов (на выбор) и с использованием хвостовой рекурсии при обработке данных.
6. Реализовать задание с использованием динамического списка.
7. Провести анализ на регулярный доступ к памяти.
8. Провести анализ на векторизацию кода.
9. Провести сравнительный анализ реализаций.

Глава 1. Реализация и анализ применения различных структур данных

§1 Массив строк

Для хранения телефонных номеров используется одномерный массив целых чисел. Так как телефонный номер могут быть любым целочисленным 10-ти символьным числом, нужно учитывать, что значения большинства номеров будут превышать максимальное значение, которое может быть представлено 32-битным целочисленным типом `integer(kind=4)`, что составляет `2,147,483,647`. Поэтому, для корректного хранения, необходимо использовать 64-битный целочисленный тип `integer(kind=8)`, который иммет гораздо больший диапазон. Данный параметр задан в модуле `environment.f90`

```
integer, parameter      :: I_ = INT64
```

Листинг 1: Разновидность типа для целочисленных переменных

Для первого варианта реализации курсового проекта хранение данных происходит с использованием массива строк для хранения фамилий и массива целочисленных чисел для хранения телефонных номеров.

```
character(OWNER_LEN, kind=CH_) :: Owners(PHONE_AMOUNT)
integer(I_)                    :: Phones(PHONE_AMOUNT)
```

Листинг 2: Инициализация массивов для хранения исходных данных

Основные операторы ввода и вывода данных:

```
! Чтение списка телефонных номеров
open (file=input_file, encoding=E_, newunit=In)
   format = '(A15, 1x, I10)'
   read(In, format, iostat=IO) (Owners(i), Phones(i), i = 1, PHONE_AMOUNT)
   call Handle_IO_status(IO, "reading phonebook list")
close (In)

! Вывод списка
open(file=output_file, encoding=E_, position=position, newunit=Out)
   write(Out, '(/a)') list_name
   format = '(A15, 1x, I10)'
   write(Out, format, iostat=io) (Owners(i), Phones(i), i = 1,
      ↪ PHONE_AMOUNT)
   call Handle_IO_status(IO, "writing phonebook")
close(Out)
```

Листинг 3: Ввод и вывод данных

Сортировка вставками (Insertion Sort) - это алгоритм сортировки, на каждом шаге которого массив постепенно перебирается слева направо. При этом каждый последующий элемент размещается таким образом, чтобы он оказался между двумя ближайшими элементами с минимальным и максимальным значением. Вычислительня сложность - $O(n^2)$.

Цикл `do concurrent` предполагает, что итерации могут выполняться параллельно, что позволяет значительно ускорить выполнение программы на многопроцессорных системах.

Внутренний цикл `do while` выполняется до тех пор пока индекс j не станет меньше 1 или значение `Phones(j)` больше либо равно значению `Phones(i)`. Это нужно для нахождения позиции, куда должен быть вставлен текущий элемент `Phones(i)`.

После того, как найдена подходящая позиция, происходит сдвиг элементов массивов `Owners` и `Phones`. Функци `cshift` (циклический сдвиг) сдвигает элементы на 1 вправо, освобождая место для вставки текущего элемента. За счет использования данной функции возможна потенциальная векторизация.

```
do concurrent(i = 2:PHONE_AMOUNT)
   j = i - 1
   do while (j >= 1 .and. Phones(j) < Phones(i))
         j = j - 1
   end do
   Owners(j+1:i) = cshift(Owners(j+1:i), -1)
```

```
   Phones(j+1:i) = cshift(Phones(j+1:i), -1)
end do
```

Листинг 4: Сортировка методом вставок

При компиляции выполняется векторизация за счет использования функции `cshift`.

```
src/phonebook_process.f90:22:53: optimized: basic block part vectorized
    ↪ using 16 byte vectors
src/phonebook_process.f90:22:53: optimized: basic block part vectorized
    ↪ using 16 byte vectors
src/phonebook_process.f90:22:53: optimized: basic block part vectorized
    ↪ using 16 byte vectors
src/phonebook_process.f90:23:53: optimized: basic block part vectorized
    ↪ using 16 byte vectors
src/phonebook_process.f90:23:53: optimized: basic block part vectorized
    ↪ using 16 byte vectors
src/phonebook_process.f90:23:53: optimized: basic block part vectorized
    ↪ using 16 byte vectors
```

Листинг 5: Векторизация при компиляции

§2 Массив символов

В данной реализации решения, для хранения фамилий были использованы массивы символов. В одном случае решение реализовано без регулярного доступа к памяти. Во второй реализации решения регулярный доступ к памяти обеспечен (благодаря этому при получении данных строка будет сплошной).

При этом подход к хранению массива целочисленных значений телефонных номеров не менялся.

```
! Массивы фамилий и телефонов. Array(i, j), i - строка, j -столбец
character(kind=CH_)      :: Owners(PHONE_AMOUNT, OWNER_LEN) = ""
integer(I_)              :: Phones(PHONE_AMOUNT)
```

Листинг 6: Инициализация массивов с нерегулярным доступом

```
! Массивы фамилий и телефонов. Array(j, i): j - столбец, i - строка
character(kind=CH_) :: Owners(OWNER_LEN, PHONE_AMOUNT) = ""
integer(I_)         :: Phones(PHONE_AMOUNT)
```

Листинг 7: Инициализация массивов с регулярным доступом

В дальнейших листингах решения с использованием массива символов будет представлен код для варианта с регулярным доступом к памяти.

Основные операторы ввода вывода данных.

```
! Чтение списка телефонных номеров.
open (file=Input_File, encoding=E_, newunit=In)
   format = '(' // OWNER_LEN // 'A1, 1x, I10)'
   ! Храним по столбцам (j, i)
   read (In, format, iostat=IO) (Owners(:, i), Phones(i), i = 1,
      ↪ PHONE_AMOUNT)
   call Handle_IO_status(IO, "reading phone list")
close (In)

! Вывод списка.
open (file=output_file, encoding=E_, position=position, newunit=Out)
   write (out, '(/A)') List_name
   format = '(' // OWNER_LEN // 'A1, 1x, I10)'
   write (Out, format, iostat=IO) (Owners(:, i), Phones(i), i = 1,
      ↪ PHONE_AMOUNT)
   call Handle_IO_status(IO, "writing " // List_name)
close (Out)
```

Листинг 8: Ввод и вывод данных

Функция сортировки аналогична представленной в листинге 4. Меняется только вставка фамилий на нужную позицию с учетом того, что данные хранятся в двумерном массиве символов.

В данной реализации также будет обеспечена потенциальная векторизация за счет использования функции `cshift` (при компиляции векторизация выполняется). Так как сравнение и обход элементов выполняется только на массиве `Phones`, а в массиве `Owners` выполняется только вставка элемента на нужную позицию, векторизация за счет регулярного доступа не может быть гарантирована.

```
Owners(:, j+1:i) = cshift(Owners(:, j+1:i), -1, dim=2)
```

Листинг 9: Сортировка массива символов

§3 Массив структур

В этом проекте в качестве структуры данных использовались массивы структур для данных о фамилиях владельцев и телефонных номерах.

```
type phonebook
   character(OWNER_LEN, kind=CH_) :: Owners = ""
   integer(I_)                    :: Phones = 0
end type phonebook
```

Листинг 10: Структура для хранения данных

При чтении из файла, вначале входные данные записываются в создаваемый двоичный файл. После считываются в производную структуру данных.

```
! Создание двоичного файла
open (file=Input_File, encoding=E_, newunit=In)
recl = OWNER_LEN * CH_ + I_
open (file=Data_File, form='unformatted', newunit=Out, access='direct',
   ↪ recl=recl)
   format = '(A15, 1x, I10)'
   do i = 1, PHONE_AMOUNT
      read (In, format, iostat=IO) phone
      call Handle_IO_status(IO, "reading formatted phonebook list, line "
         ↪ // i)

      write (Out, iostat=IO, rec=i) phone
      call Handle_IO_status(IO, "creating unformatted file with phonebook
         ↪ list, record " // i)
   end do
close (In)
close (Out)

! Чтение списка: фамилии и телефонные номера.
recl = (OWNER_LEN * CH_ + I_) * PHONE_AMOUNT
open (file=Data_File, form='unformatted', newunit=In, access='direct',
   ↪ recl=recl)
   read (In, iostat=IO, rec=1) Phone_list
   call Handle_IO_status(IO, "reading unformatted class list")
close (In)

! Вывод списка.
open (file=Output_File, encoding=E_, position=Position, newunit=Out)
   write (out, '(/a)') List_name
   format = '(A15, 1x, I10)'
   write (Out, format, iostat=IO) Phone_list
   call Handle_IO_status(IO, "writing " // List_name)
close (Out)
```

Листинг 11: Ввод и вывод данных. Создание двоичного файла.

Из-за отличая структуры данных, меняется код сортировки списка телефонных номеров - не нужно отдельно сортировать фамилии и телефонные номера, выполняется сортировка массива структур. В

данной реализации также будет обеспечена потенциальная векторизация за счет использования функции `cshift` (векторизация выполняется при компиляции программы).

```
do i = 2, PHONE_AMOUNT
   j = i - 1
   do while (j >= 1 .and. Phone_list(j)%Phones < Phone_list(i)%Phones)
         j = j - 1
   end do
   Phone_list(j + 1:i) = cshift(Phone_list(j+1:i), -1, dim=1)
end do
```

Листинг 12: Сортировка массива структур

§4 Структура массивов

В данном проекте вместо массива структур используется структура массивов.

```
type phonebook
   character(OWNER_LEN, kind=CH_), allocatable :: owner(:)
   integer(I_), allocatable                    :: phone(:)
end type phonebook
```

Листинг 13: Структура массивов фамилий и номеров телефонов

При чтении из файла, входные данные вначале записываются в создаваемый двоичный файл. После, они считываются оттуда в производную структуру данных:

```
! Создание неформатированного файла данных.
open (file=input_file, encoding=E_, newunit=In)
   format = '(a15, 1x, i10)'
   read (In, format, iostat=IO) (tmp_owner(i), tmp_phone(i), i = 1,
      ↪ PHONE_AMOUNT)
   call Handle_IO_status(IO, "reading formatted group list, line " // i)
close (In)

open (file=data_file, form='unformatted', newunit=Out, access='stream')
   write (Out, iostat=IO) tmp_owner, tmp_phone
   call Handle_IO_status(IO, "creating unformatted file with group list,
      ↪ record " // i)
close (Out)

! Чтение списка.
open (file=Data_File, form='unformatted', newunit=In, access='stream')
   allocate(phonepage%owner(PHONE_AMOUNT), phonepage%phone(PHONE_AMOUNT))
   read (In, iostat=IO) phonepage%owner, phonepage%phone
   call Handle_IO_status(IO, "reading unformatted class list")
close (In)

! Вывод списка.
open (file=Data_File, form='unformatted', newunit=In, access='stream')
   allocate(phonepage%owner(PHONE_AMOUNT), phonepage%phone(PHONE_AMOUNT))
   read (In, iostat=IO) phonepage%owner, phonepage%phone
   call Handle_IO_status(IO, "reading unformatted class list")
close (In)
```

Листинг 14: Ввод и вывод данных. Создание двоичного файла

Сортировка массивов выполняется аналогично реализации в §1. В данной реализации также обеспечена потенциальная векторизация за счет использования функции `cshift` (векторизация выполняется при компиляции программы).

```
do concurrent (i = 2:PHONE_AMOUNT)
   j = i - 1
   do while (j >= 1 .and. phonepage%phone(j) < phonepage%phone(i))
```

```
      j = j - 1
      if (j == 0) exit
   end do
   phonepage%owner(j+1:i) = cshift(phonepage%owner(j+1:i), -1)
   phonepage%phone(j+1:i) = cshift(phonepage%phone(j+1:i), -1)
end do
```

Листинг 15: Сортировка структуры массивов

§5 Хвостовая рекурсия

В данном проекте структура данных – структура массивов, обработка которой производится посредством хвостовой рекурсии. Хвостовая рекурсия требует, чтобы рекурсивный вызов являлся последней операцией в функции или подпрограмме.

```
type phonebook
   character(OWNER_LEN, kind=CH_), allocatable :: owner(:)
   integer(I_), allocatable                    :: phone(:)
end type phonebook
```

Листинг 16: Структура для хранения данных

Ввод и вывод данных аналогичен реализации представленной в листинге 14.

В функции `Sort_phone_list` рекурсивный вызов `call Sort_phone_list(Phone_List, i+1)` является последним действием перед выходом из подпрограммы. Использование хвостовой рекурсии позволяет компилятору оптимизировать код, преобразовывая рекурсивные вызовы в итерации и сокращая использование стека. В данной реализации также обеспечена потенциальная векторизация за счет использования функции `cshift` (векторизация выполняется при компиляции программы).

```
pure recursive subroutine Sort_phone_list(phonepage, i)
   type(phonebook), intent(inout) :: phonepage
   integer :: j
   integer, intent(in) :: i

   j = i - 1

   do while (j >= 1 .and. (phonepage%phone(j) < phonepage%phone(i)))
      j = j - 1
   end do

   phonepage%owner(j+1:i) = cshift(phonepage%owner(j+1:i), -1)
   phonepage%phone(j+1:i) = cshift(phonepage%phone(j+1:i), -1)

   if (i <= PHONE_AMOUNT) &
      call Sort_phone_list(phonepage, i+1)

end subroutine Sort_phone_list
```

Листинг 17: Реализация сортировки с использованием хвостовой рекурсии

§6 Динамический список

В первом варианте решение реализовано с использованием динамического однонаправленного списка (`pointer`). Второй вариант решения - с использованием рекурсивно размещаемого типа (`allocatable`).

```
type phonebook
   character(OWNER_LEN, kind=CH_) :: owner = ""
   integer(I_)                    :: phone = 0
   type(phonebook), pointer       :: next  => Null()
end type phonebook
```

Листинг 18: Динамический список, объявленный через pointer

```
type phonebook
   character(OWNER_LEN, kind=CH_) :: owner = ""
   integer(I_)                    :: phone = 0
   type(phonebook), allocatable   :: next
end type phonebook
```

Листинг 19: Динамический список, объявленный через allocatable

Ввод и вывод данных.

```
! Чтение списка фамилии и телефоны.
function Read_phone_list(Input_File) result(Phone_List)
   type(phonebook), pointer   :: Phone_List
   character(*), intent(in)   :: Input_File
   integer  In

   open (file=Input_File, encoding=E_, newunit=In)
      Phone_List => Read_phone(In)
   close (In)
end function Read_phone_list
! Чтение следующего значения.
recursive function Read_phone(In) result(Phone_List)
   type(phonebook), pointer  :: Phone_List
   integer, intent(in)       :: In
   character(:), allocatable :: format
   integer  IO

   allocate (Phone_List)
   format = '(A15, 1x, i10)'
   read (In, format, iostat=IO) Phone_List%owner, Phone_List%phone
   call Handle_IO_status(IO, "reading line from file")
   if (IO == 0) then
      Phone_List%next => Read_phone(In)
   else
      deallocate (Phone_List)
   end if
end function Read_phone

! Вывод списка.
subroutine Output_phone_list(Output_File, Phone_List, List_Name, Position)
   character(*), intent(in)    :: Output_File, Position, List_Name
   type(phonebook), intent(in) :: Phone_List
   integer                     :: Out

   open (file=Output_File, encoding=E_, position=Position, newunit=Out)
      write (out, '(/a)') List_Name
      call Output_phones(Out, Phone_List)
   close (Out)
end subroutine Output_phone_list
recursive subroutine Output_phones(Out, phonepage)
   type(phonebook), intent(in) :: phonepage
   integer, intent(in)         :: Out
   character(:), allocatable   :: format
   integer                     :: IO

   format = '(A15, 1x, i10)'
   write (Out, format, iostat=IO) phonepage%owner, phonepage%phone
   call Handle_IO_status(IO, "writing owners&phones")
   if (Associated(phonepage%next)) &
      call Output_phones(Out, phonepage%next)
end subroutine Output_phones
```

Листинг 20: Чтение и вывод данных (динамический однонаправленный список)

```
! Чтение списка фамилии и телефоны.
function Read_phone_list(Input_File) result(Phone_List)
   type(phonebook), allocatable :: Phone_List
   character(*), intent(in)     :: Input_File
   integer  In

   open (file=Input_File, encoding=E_, newunit=In)
      call Read_phone(In, Phone_List)
   close (In)
end function Read_phone_list

! Чтение следующей записи.
recursive subroutine Read_phone(In, phonepage)
   type(phonebook), allocatable :: phonepage
   integer, intent(in)          :: In
   character(:), allocatable    :: format
   integer  IO

   allocate (phonepage)
   format = '(A15, 1x, i10)'
   read (In, format, iostat=IO) phonepage%owner, phonepage%phone
   call Handle_IO_status(IO, "reading line from file")
   if (IO == 0) then
      call Read_phone(In, phonepage%next)
   else
      deallocate (phonepage)
   end if
end subroutine Read_phone

! Вывод списка.
subroutine Output_phone_list(Output_File, Phone_List, List_Name, Position)
   character(*), intent(in)     :: Output_File, Position, List_Name
   type(phonebook), allocatable :: Phone_List
   integer  :: Out

   open (file=Output_File, encoding=E_, position=Position, newunit=Out)
      write (out, '(/a)') List_Name
      call Output_phones(Out, Phone_List)
   close (Out)
end subroutine Output_phone_list

recursive subroutine Output_phones(Out, phonepage)
   type(phonebook), allocatable :: phonepage
   character(:), allocatable    :: format
   integer, intent(in)          :: Out
   integer                      :: IO

   format = '(A15, 1x, i10)'
   if (allocated(phonepage)) then
      write (Out, format, iostat=IO) phonepage%owner, phonepage%phone
      call Handle_IO_status(IO, "writing owners&phones")
      call Output_phones(Out, phonepage%next)
   end if
end subroutine Output_phones
```

Листинг 21: Чтение и вывод данных (рекурсивно размещаемый тип)

Доступ к памяти производится по ссылкам, поэтому векторизация не задействуется. Сортировка вставками реализована с помощью хвостовой рекурсии.

Для первого варианта реализации сортировка выполняется с использованием поля `next`.

```
pure recursive subroutine Sort_phone_list(Phone_list, Sorted_list)
   type(phonebook), pointer :: Phone_list, temp, Sorted_list
```

```
3
4     if (associated(Phone_list)) then
5        ! Указатель на текущий элемент сохраняется во временной переменной.
6        temp => Phone_list
7        ! Перемещаем указатель на следующий элемент списка.
8        Phone_list => Phone_list%next
9        ! Обнуляем указатель на следующий элемент временной переменной.
10       temp%next => null()
11       ! Вызываем подпрограмму вставки для вставки временного элемента в от
             ↪ сортированный список.
12       call Insertion(Sorted_list, temp)
13       ! Рекурсивно вызываем эту же функцию для оставшихся элементов списка
             ↪ .
14       call Sort_phone_list(Phone_list, Sorted_list)
15    end if
16 end subroutine Sort_phone_list
17
18 pure recursive subroutine Insertion(Sorted_list, temp)
19    type(phonebook), pointer :: Sorted_list, temp
20
21    if (.not. associated(Sorted_list) .or. Sorted_list%phone < temp%phone)
          ↪ then
22       ! Временный элемент становится первым в списке.
23       temp%next => Sorted_list
24       Sorted_list => temp
25    else
26       ! Рекурсивно вызываем эту же функцию для следующего элемента списка.
27       call Insertion(Sorted_list%next, temp)
28    end if
29 end subroutine Insertion
```

Листинг 22: Сортировка методом вставок

Во втором варианте реализации, обмен элементов массива происходит с помощью встроенной команды `move_alloc`. Оператор `move_alloc` используется для передачи аллоцированной памяти и связей между узлами списка, минимизируя необходимость дополнительного копирования данных. Этот метод обеспечивает эффективную перестановку узлов без потери данных и сохранения структуры списка.

```
1  pure recursive subroutine Sort_phone_list(Phone_list, Sorted_list)
2     type(phonebook), allocatable, intent(inout) :: Phone_list, Sorted_list
3     type(phonebook), allocatable :: temp
4
5     if (Allocated(Sorted_list%next)) then
6        if (Sorted_list%phone < Sorted_list%next%phone) then
7           call move_alloc(Sorted_list%next, temp)
8           call move_alloc(temp%next, Sorted_list%next)
9
10          call Insertion(Phone_list, temp)
11          call Sort_phone_list(Phone_List, Sorted_list)
12
13       else
14          call Sort_phone_list(Phone_List, Sorted_list%next)
15       end if
16    end if
17 end subroutine Sort_phone_list
18
19 pure recursive subroutine Insertion(Sorted_list, temp)
20    type(phonebook), allocatable, intent(inout) :: Sorted_list, temp
21
22    if (Sorted_list%phone < temp%phone) then
23       ! Временный элемент становится первым в списке.
24       call move_alloc(Sorted_list, temp%next)
25       call move_alloc(temp, Sorted_list)
```

```
26      else
27          ! Рекурсивно вызываем эту же функцию для следующего элемента списка.
28          call Insertion(Sorted_list%next, temp)
29      end if
30
31  end subroutine Insertion
```

Листинг 23: Сортировка телефонных номеров методом вставок с помощью move_alloc

Глава 2. Сравнение реализаций

Ниже приведена сравнительная таблица реализаций по критериям: сплошные данные, регулярный доступ, векторизация, потенциальная векторизация, а также по показателям: время работы при обработке данных в секундах, сложность участка кода по количеству строк и эффективность участка кода по обработке данных возведенная в 10^6.

Таблица 1: Анализ набора средств

Набор средств	1	2a	2b	3	4	5	6a	6b
Сплошные данные	+	-	+	+	+	+	-	-
Регулярный доступ	+	-	+	+	+	+	-	-
Векторизация	+	+	+	+	+	+	-	-
Потенциальная векторизация	+	+	+	+	+	+	-	-
Объем данных (кол-во записей)	90000	90000	90000	90000	90000	37400*	13500*	13500*
Время работы при обработке данных	12,041	10,576	12,728	8,722	12,943	1,368	0,299	0,306
Сложность кода по обработке данных	8	8	8	7	9	8	13	16
Эффективность при обработке данных	10381	14576	9821	16379	8585	12913	257268	204248

*Максимальное число возможных обработанных данных до получения ошибки `Segmentation fault (core dumped)`

Заключение

В ходе работы было разработано 6 проектов для сортировки методом вставки списка телефонных номеров по убыванию и выполнены следующие задачи:

1. Реализовано задание с использованием массивов строк.
2. Реализовано задание с использованием массивов символов.
3. Реализовано задание с использованием массивов структур.
4. Реализовано задание с использованием структур массивов.
5. Реализовано задание с использованием массивов структур или структур массивов (на выбор) и с использованием хвостовой рекурсии при обработке данных.
6. Реализовано задание с использованием динамического списка.
7. Проведен анализ на регулярный доступ к памяти.
8. Проведен анализ на векторизацию кода.
9. Проведен сравнительный анализ реализаций.

Использование различных структур данных при которых данные по разному размещаются в оперативной памяти, влияет на эффективность при обработке данных. Исходя из данных сравнительной таблицы, наиболее оптимальной структурой данных для решения вышеупомянутой задачи на современной микроархитектуре Firestorm (Apple M1) является структура массивов.