КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ
Кафедра системного анализа и информационных технологий

Е.Л.СТОЛОВ, Р.Х.ЛАТЫПОВ, Р.Г.РУБЦОВА,
Б.Г.МУБАРАКОВ

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

Учебно-методическое пособие

Казань – 2019

УДК 004 (076.5)
ББК З 973.2-018

Принято на заседании кафедры системного анализа и информационных технологий
Протокол № 3 от 13 ноября2019 г.

Принято на заседании учебно-методической комиссии Института Вычислительной математики и информационных технологий
Протокол № 3 от 21 ноября 2019 г.

РЕЦЕНЗЕНТ:
доктор физико-математических наук,
профессор кафедры системного анализа и информационных технологий КФУ
Ш.Т.Ишмухаметов

Столов Е.Л., Латыпов Р.Х., Рубцова Р.Г., Мубараков Б.Г.
Параллельные вычисления: Учебно-методическое пособие / Е.Л.Столов, Р.Х.Латыпов, Р.Г.Рубцова, Б.Г.Мубараков – Казань: Казанский ун-т, 2019. - 55 с.

В данном курсе рассматривается простейшая вычислительная сеть на основе кластера (cluster), составленного из персональных компьютеров образующих локальную Ethernet сеть. В частности, кластер может содержать только один компьютер. Рассматриваются приемы программирования с помощью библиотек MPI и PVM под управлением ОС LINUX. Приводятся примеры решения задач, относящихся к различным областям знания, с помощью указанных средств. Поскольку опыт работы в среде ОС LINUX не предполагается, приводится сводка основных команд с краткими пояснениями.

Содержание

Программа курса **«Параллельные вычисления»**

Курс включает 8 лекций (каждая лекция рассчитана на 4 аудиторных часа) и 6 лабораторных занятий.

Содержание лекций

Определение параллельных вычислений. Аппаратура, поддерживающая параллельные вычисления, и ее производительность. Вычислительный кластер.

Примеры задач, для которых существуют эффективные параллельные алгоритмы: взаимодействие множества частиц, задача поиска, задача сортировки, взлом шифра, фильтрация сигнала с помощью нескольких фильтров, распространение тепла в пластине, поиск кратчайшего пути, решение задач с помощью статистического моделирования.

Операционная система LINUX, поддерживающая параллельные вычисления. Основные пользовательские команды.

Интерфейс MPI. Point-to-point команды. Понятие блокирующей и неблокирующей команды. Групповые операции: MPI_Bcast, MPI_Reduce.

Интерфейс PVM. Принципиальное отличие – динамическое порождение процессов. Формат команды pvm_spawn. Point-to-point команды. Понятие блокирующей и неблокирующей команды. Групповые операции.

Другие вычислительные системы, поддерживающие параллельные вычисления (OpenMP, Condor, grid-вычисления).

ЛЕКЦИЯ 1

1.1. Основные определения

Параллельный компьютер это множество процессоров, решающих совместно вычислительную задачу. До недавнего времени такой компьютер рассматривался как разновидность экзотики, однако в последнее время он привлекает значительный интерес, поскольку появились задачи, требующие как больших вычислительных ресурсов, так и обширной оперативной памяти. В основном он находит применение в научных проектах: ядерные исследование, расшифровка генома. В последнее время применяется и в коммерческих организациях: обработки больших массивов (Data mining), разработка лекарств, проектирование автомобилей и самолетов, создание анимационных фильмов. Последние достижения в этой области можно найти на сайте http://www.top500.org/. Единица измерения - количество операций с плавающей точкой в секунду (flops).

Megaflops 10^6 flops

Gigaflops 10^9 flops

Teraflops 10^{12} flops

Petaflops 10^{15} flops

Тактовая частота процессоров при известных технологиях достигла пределов. Это связано как с размерами (паразитные емкости), так и с проблемой отвода тепла. Альтернативный подход стал возможен в связи с созданием быстрых сетей передачи данных. Аппаратура с пропускной способностью 1000 Mbits per second является обычной, что позволяет создавать вычислительные кластеры (cluster). Для этого сравнительно дешевые однопроцессорные компьютеры или компьютеры с малым числом процессоров объединяют в вычислительную сеть, используя стандартные протоколы. В этом случае говорят о распределенных (distributed) вычислениях. При этом различают вычисления с помощью кластера или сети (grid). В последнем случае компьютеры могут быть распределены по всему миру. При решении таких задач речь идет не о скорости решения, а о размере задачи. При этом появляются дополнительные проблемы, связанные с надежностью и безопасностью. Часть компьютеров может быть выключена на какое-то время, а затем вычисления возобновляются. С другой стороны, эффективное использование вычислительной сети возможно лишь на основе специальных алгоритмов, в которых предполагается использование произвольного числа процессоров (scalability) .

С точки зрения внутренней архитектуры различают компьютеры *SIMD* (single instruction multiple data), когда несколько процессоров выполняют одну и туже программу,

но каждый из них использует свои данные, и MIMD (multiple instruction multiple data), когда каждый процессор решает свою задачу. Как будет показано ниже, вычислительный кластер может эмулировать каждый из этих типов компьютеров.

1.2. Структура кластера

В простейшем случае вычислительный кластер имеет следующую структуру (рисунок 1):

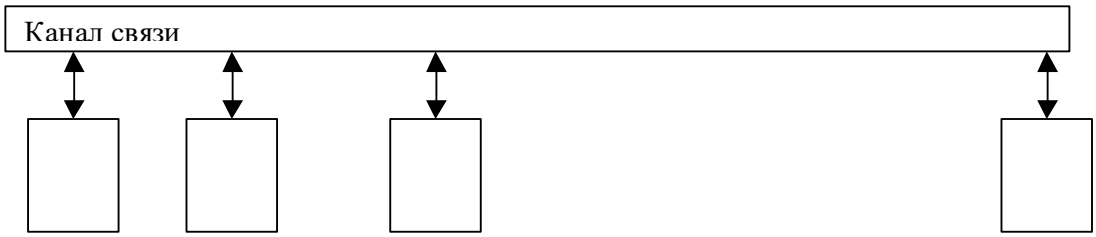

Рисунок 1. Структура вычислительного кластера

Несколько компьютеров связаны через общий канал связи и обмен идет с помощью стандартного протокола TCP/IP. В случае, когда компьютеры находятся на малом расстоянии друг от друга, возможен обмен данных с помощью специальных switch`ей, обеспечивающих большую скорость. Но в любом случае, эта скорость уступает скорости обмена между процессором и памятью внутри компьютера, поэтому при разработке алгоритма решения нужно стараться свести к минимуму обмен информацией между компьютерами.

1.3. Элементы параллельной программы

По определению, параллельная программа состоит из отдельных подзадач (task) для выполнения каждой из которых выделяется отдельный процесс. Различные процессы могут выполняться как на одном и том же процессоре, так и на разных. Таким образом, число одновременно запущенных процессов может не совпадать с числом процессоров в системе. Как правило, создается главный процесс (ГП) (mater), под управлением которого происходит решение задачи. Программа включает процедуры порождения подзадач (slave), завершение подзадач и обмен данными между отдельными подзадачами. Обмен данными может быть синхронизирован ГП, а может производиться асинхронно по мере готовности, поэтому параллельная программа включает средства для выполнения

указанных действий. Синхронизация с помощью ГП применяется, когда разные процессы обращаются к одной и той же оперативной памяти. Обмен между процессами, живущими на разных компьютерах, производится, как правило, по асинхронной схеме, но при этом есть возможность производить синхронизацию всех процессов в определенных точках программы. Все указанные функции реализуются с помощью специальных библиотек, подключаемых к стандартным библиотекам компилятора. В данном курсе рассматриваются программы, написанные на языке С. Существуют также библиотеки, поддерживающие программы на языках С++ и FORTRAN.

Как уже отмечалось выше, параллельная программа должна работать на вычислительной сети с произвольным количеством процессоров. Это свойство называется шкалированием. Критическими параметрами являются время обмена данными между процессорами и время доступа к оперативной памяти. Простое увеличение числа процессоров может не дать эффекта, если упомянутые времена велики. Результат работы программы не зависит от числа процессоров, однако, последовательность выполнения отдельных подзадач будет зависеть от параметров кластера. Рассмотрим простейший пример умножения матрицы на вектор (1).

$$y_i = \sum_{j=0}^{n} a_{i,j} x_j, \quad i = 1,\ldots,m \qquad (1)$$

Очевидно, что вычисление каждого из значений y_i можно реализовать в виде отдельной подзадачи. Если вычисление производится с помощью кластера, и при этом каждый из процессов печатает найденный результат сразу же после вычисления, то предсказать заранее последовательность печатаемых значений нельзя. Если же печать осуществляет ГП, то можно получить все найденные значения в естественном порядке.

Этот пример иллюстрирует различные стратегии организации вычислений. В реальной ситуации значение m превосходит число физических процессоров. ГП может порождать очередную подзадачу по мере освобождения процессора, а может запустить сразу все процессы, предоставив операционной системе распределять процессы между процессорами. Эта задача известна как задача балансировки кластера.

Продолжим рассмотрение предыдущего примера. Предположим, что кластер состоит из 3 частей, как показано на рисунке 2.

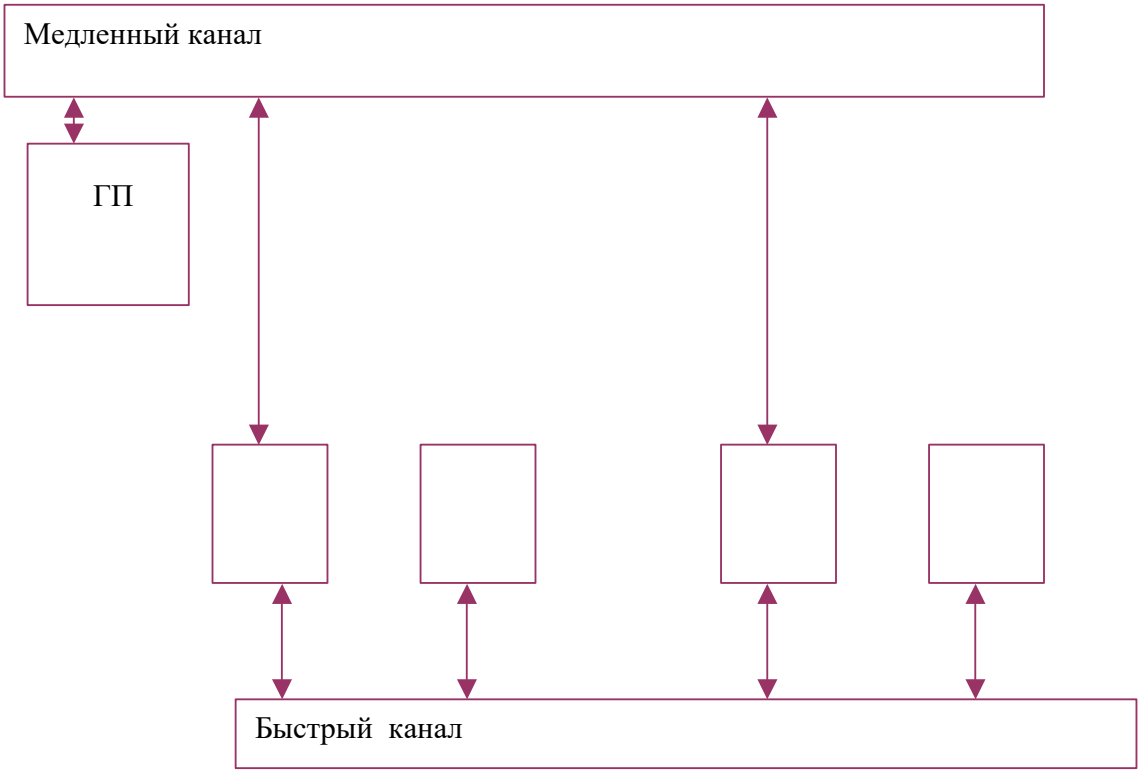

Рисунок 2. Кластер

Компьютер, на котором живет ГП, соединен через медленный канал с одним из компьютеров в подкластере. В свою очередь, каждый из этих компьютеров соединен через быстрый канал с остальными компьютерами подкластера. В этом случае ГП порождает процессы для вычисления одного значения y_i в каждом из подкластеров на компьютере, с которым он имеет соединение. В свою очередь, каждый из этих компьютеров порождает процессы внутри подкластера для завершения вычислений. Например, один из компьютеров вычисляет (2):

$$y_{i,0} = \sum_{j=0}^{[n/2]} a_{i,j} x_j \qquad (2)$$

а второй подсчитывает (3):

$$y_{i,1} = \sum_{j=[n/2]+1}^{n} a_{i,j} x_j \qquad (3)$$

и передает результат вычислений первому. Выбор стратегии вычислений определяет программист, хотя возможна ситуация, когда такая стратегия определяется средствами ОС.

Еще одна проблема – балансировка процессоров кластера с тем, чтобы загрузка каждого из них была примерно одинаковой. Из сделанных замечаний следует, что при

переходе на параллельный компьютер, как правило, необходимо менять код. Для переносимости программы используют интерфейсы независимые от платформы.

ЛЕКЦИЯ 2

2.1. Примеры параллельных программ

Существует много элегантных примеров параллельных алгоритмов, предполагающих активный обмен данными между задачами. Кластерная вычислительная сеть не подходит для эффективной реализации таких алгоритмов, однако, она может быть использована для отладки соответствующих программ. В тех случаях, когда требуется создание специализированных вычислительных устройств для реализации программы, кластер является удобной моделью. С другой стороны, имеются многочисленные задачи, для решения которых кластер является очень эффективным. Некоторые из таких задач будут описаны ниже.

2.2. Вычисление конечных разностей

Предположим, что функция задана своими значениями в точках с постоянным шагом *h:*

$$y_0 = f(x_0), y_1 = f(x_0 + h), \ldots, y_N = f(x_0 + Nh).$$

Для построения интерполяционного многочлена требуется найти конечные разности этой функции. Разности первого порядка определяются формулой:

$$y_{i,1} = y_i - y_{i-1}, \quad i = 1, \ldots, N$$

Разности порядка k вычисляются согласно определению:

$$y_{i,k} = y_{i,k-1} - y_{i-1,k-1}, \quad i = k, \ldots, N$$

Схема вычислений выглядит так (см. рисунок 3)

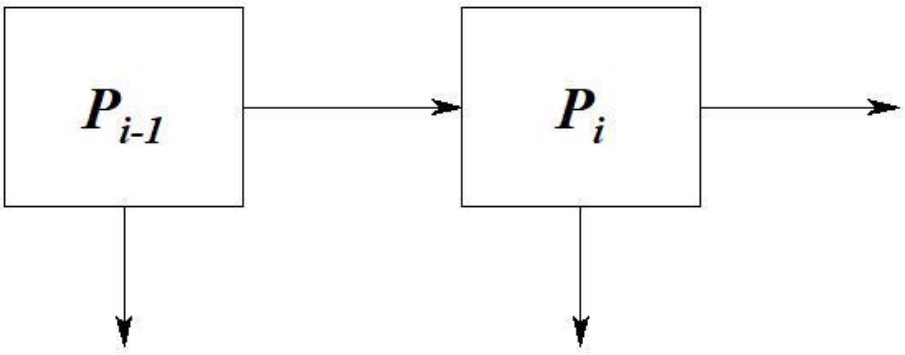

Рисунок 3. Схема вычислений

На первом шаге запускается N процессов $P_i, i = 0, \ldots, N$ и процесс с номером i получает значение y_i. Затем в цикле:

```
For k=1,N
      For i=k,N
```

Используя загруженное значение и значение, полученное от процесса с номером $i-1$, процесс с номером i подсчитывает $y_{i,k}$, сохраняет найденное значение и выводит его. Процесс с номером $k-1$ завершается.

```
      End For
End For
```

2.3. Взаимодействие множества частиц

Имеется N частиц, движущихся на плоскости внутри области, ограниченной прямоугольником. При столкновении с границей, частица отражается от нее согласно закону – угол падения равен углу отражения. Радиус частиц считается малым, они имеют одинаковую массу, поэтому при столкновении двух частиц происходит обмен скоростями (см. рис. 4)

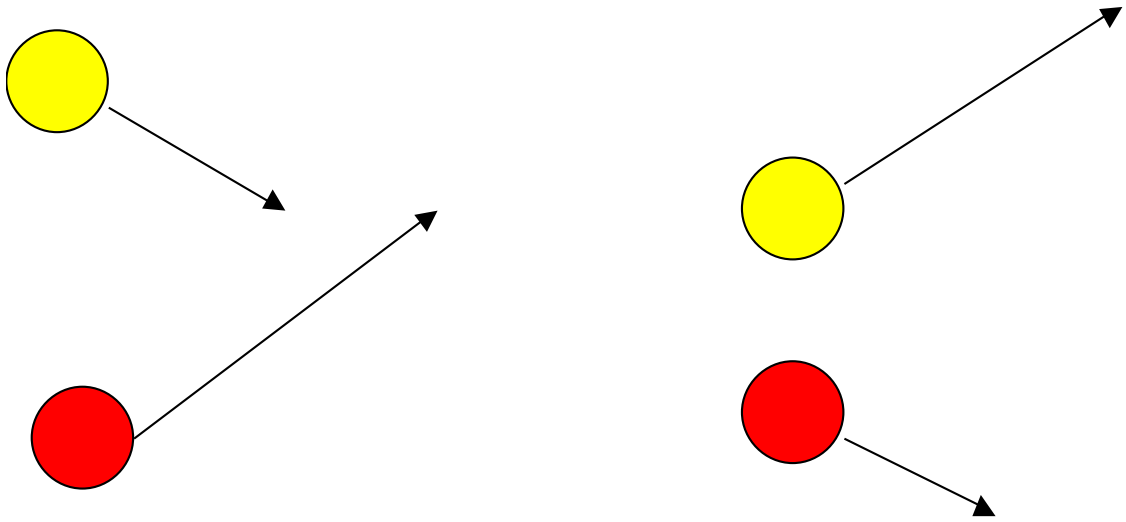

Рисунок 4. Столкновение двух частиц

В начальный момент времени задаются координаты каждой частицы и ее скорость. Выбираем шаг дискретизации h. Если в момент времени t частица имела координаты x_t, y_t, скорость v_t и не соприкасалась с границей или другими частицами, то в момент времени $t+h$ ее координаты определяются вектором

$$(x_{t+h}, y_{t+h}) = (x_t, y_t) + hv_t. \qquad (4)$$

Если же в момент времени t частица касалась границы или соприкасалась с другой частицей, ее скорость меняется, а положение в следующий момент времени определяется формулой (4) но с новым значением вектора скорости.

Данный пример интересен тем, что возможны два подхода к реализации программы. Первый подход заключается в том, что число запущенных процессов равняется количеству частиц, и каждый процесс управляет лишь одной частицей. В этом случае в каждый момент времени ГП должен знать положение каждой частицы и определять, какие из них соприкасаются, вычислять новые значения скоростей и предавать их соответствующим процессам. Роль остальных процессов сводится к вычислению координат согласно (1) и в передаче новых значений в ГП. Ясно, что такая реализация имеет смысл лишь в том случае, когда кластер содержит один мощный компьютер, где живет ГП, а остальные компьютеры имеют значительно меньшую мощность. Преимущество данного подхода состоит в простоте программирования.

Другой подход заключается в том, что вся прямоугольная область разбивается на части, и поведением частиц в каждой части управляет один процесс. В случае выхода какой-либо частицы за пределы области управления процесс передает информацию о ней в ГП, либо связывается с процессом, управляющим частицами в соседней области. Программирование данного подхода несколько сложнее, однако, сеть используется более эффективно.

2.4. Задача поиска

В предыдущем примере использовался прием наиболее характерный для кластерных вычислений: область данных задачи разбивается на части, для обработки каждой части создается специальный процесс, а окончательный результат выдается ГП на основе данных, полученных от остальных процессов. В качестве примера рассмотрим следующую задачу.

В одномерном массиве требуется найти нужное значение. Решение заключается в разбиении массива на части, каждая из которых передается одному процессу. После решения задачи одним из процессов это решение передается в ГП, который останавливает остальные процессы. На этом примере видно, что ГП не только может запускать процессы, но и завершать их в случае необходимости.

2.5. Задача сортировки - слияния

Другим интересным приложением является задача о сортировке одномерного массива. И в этом случае первоначальные данные разбиваются на части, а каждый процесс сортирует свою часть данных. После этого отсортированные части передаются в

ГП, который производит окончательную сортировку. Интересно отметить, что каждый из процессов, обрабатывающих свою часть данных, может использовать тот же алгоритм. Сортировка, осуществляемая ГП, называется слиянием. Имеются два отсортированных массива $(a_0,\ldots,a_M)$ $(b_0,\ldots,b_N)$, $a_i \le a_{i+1}, b_j \le b_{j+1}$. Требуется построить отсортированный массив $(c_0,\ldots,c_{M+N+1})$, содержащий оба предыдущих. Наибольшее распространение нашли два алгоритма слияния.

Если длины массивов примерно равны, то результирующий массив строится следующим образом.

Полагаем $c_0 = \min(a_0, b_0)$

Если $c_0 = a_0$ удаляем элемент a_0 из первого массива, иначе удаляем b_0 из второго

Для выбора элемента c_1 применяем тот же алгоритм к обновленным массивам.

Процесс продолжается до тех пор, пока один из массивов не будет исчерпан. После этого добавляют в конец отсортированного массива остаток второго.

Если $N << M$, применяется другой алгоритм.

Делением пополам, находим положение элемента b_0 в первом массиве и вставляем его так, чтобы сохранить упорядоченность.

В новом массиве среди элементов, удовлетворяющих неравенству $a_i > b_0$, находим положение b_1 и вставляем его.

Процесс продолжается до тех пор, пока не будут исчерпаны все элементы второго массива.

2.6. Взлом шифра

С ростом возможностей вычислительной техники наибольшее распространение для взлома шифров получает подход, основанный на переборе. При этом перебор может относиться как к подбору ключа, так и к подбору схемы шифрования в рамках определенной модели. Рассмотрим простейший пример. На рисунке 5 изображен обычный двоичный регистр сдвига с обратными связями, заданными многочленом $f(x) = a_0 x^n \oplus a_1 x^{n-1} \oplus \cdots \oplus a_n$. Если в момент времени t регистр находится в состоянии $y_{t+n},\ldots,y_t$, то в следующий момент времени он перейдет в состояние $y_{t+n+1},\ldots,y_{t+1}$

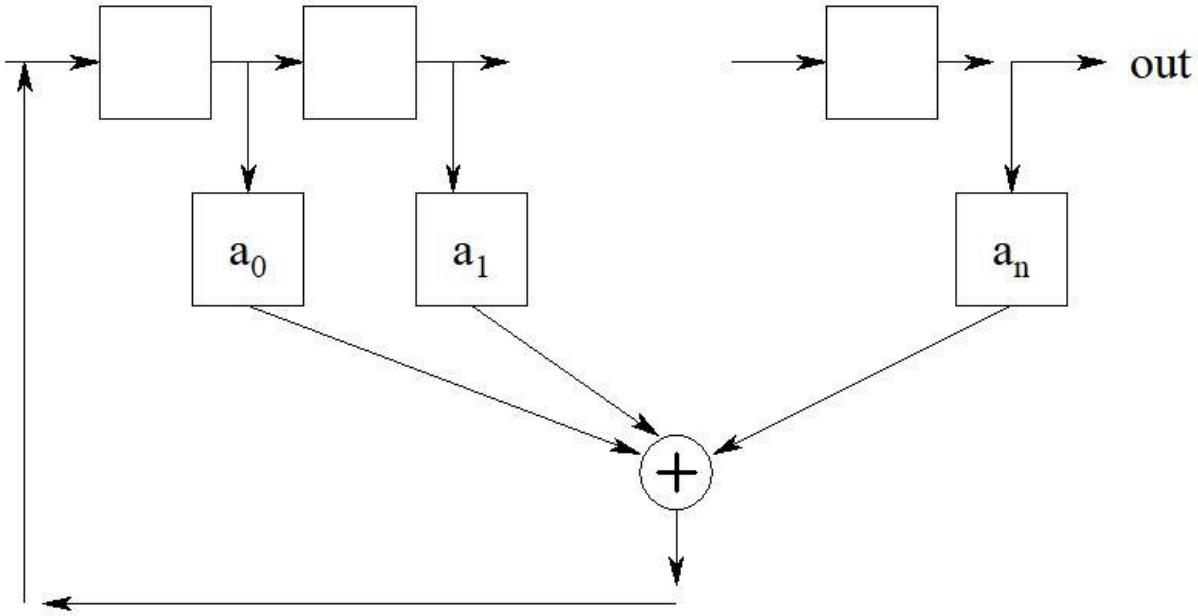

Рисунок 5. Двоичный регистр сдвига с обратными связями

При этом $y_{t+n+1} = a_0 y_{t+n} \oplus \cdots \oplus y_t a_n$

Схема шифрования приведена на рисунке 6. В регистр устанавливается начальное состояние (ключ) и задаются обратные связи.

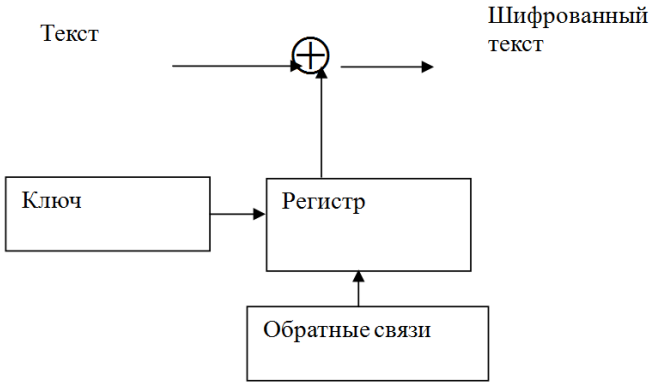

Рисунок 6. Схема шифрования

Исходный текст побитно складывается по модулю 2 с битами, сгенерированными регистром. Для расшифровки текста надо установить генератор в то же состояние и повторить процедуру аналогичную шифрованию.

Процесс взлома заключается в полном переборе возможных начальных ключей и обратных связей. Если регистр имеет $n+1$ разряд, то всего имеется 2^{2n+2} возможностей. Эти задачи распределяются между разными процессами. Проблема заключается в автоматическом определении момента успешной расшифровки.

2.7. Разложение числа на простые множители

Составные элементом алгоритма шифрования является представление некоторого большого целого нечетного числа в виде произведения двух простых множителей. Для решения этой задачи предложены многочисленные алгоритмы, один из которых, метод Ферма излагается ниже.

Предположим, что целое нечетное $N = AB$, где сомножители в правой части тоже целые. Эти сомножители можно представить в виде $A = P - Q, B = P + Q$. Поскольку числа A, B - нечетные, числа P, Q - целые. Имеем $N = P^2 - Q^2$, или

$$Q^2 = P^2 - N \qquad (5)$$

Последняя формула лежит в основе вычислений. Очевидно, что $P > \sqrt{N}$. С другой стороны, $P + Q < N$. Поскольку $P > Q$, $2Q < N$, поэтому $P^2 < N + N^2/4$. Это дает границы изменения P. Для каждого значения P проверяем, является ли выражение (5) полным квадратом. Однако, если некоторое число является полным квадратом, то и его остаток по любому модулю является полным квадратом. В LINUX включена библиотека ssl, позволяющая работать с числами большой длины. Каждое такое число представляется как последовательность байтов, а последний байт есть остаток от деления этого числа на 256. Все квадраты по модулю 256 вычисляются заранее, поэтому проверку (5) производят лишь в том случае, если последний байт является полным квадратом. На самом деле, такую проверку можно производить по любому модулю. Параллельная реализация алгоритма заключается в распределении всего интервала изменения P между различными процессами.

ЛЕКЦИЯ 3

3.1. Метод Монте-Карло

Этот метод использует статистическое моделирование. В качестве примера рассмотрим подсчет площади и объема сложных фигур. Вычисления основаны на следующем утверждении. Пусть имеются две фигуры $F_1 \subset F_2$ и случайная величина ξ равномерно распределенная в F_2. Тогда $\frac{P_1}{P_2} = \frac{|F_1|}{|F_2|}$, где P_j вероятность $\xi \in F_j$. Эта вероятность оценивается с помощью моделирования: подсчитывается относительная частота попадания случайной величины во внутрь фигуры. При параллельном моделировании каждый процесс проводит моделирование независимо, после чего главный процесс подводит окончательные расчеты. Пусть ς случайная величина равномерно распределенная на отрезке $[0,1]$. В этом случае вектор $(\varsigma_1, \varsigma_2)$ равномерно распределен в квадрате, вектор $((\sqrt{\varsigma_1}\cos(2\pi\varsigma_2), \sqrt{\varsigma_1}\sin(2\pi\varsigma_2))$ равномерно распределен в единичном круге. Пусть $\vartheta = 2\varsigma_1 - 1$, $\lambda = \arccos(\vartheta)$. Тогда вектор $(\sin(\lambda)\cos(2\pi\varsigma_2), \sin(\lambda)\sin(2\pi\varsigma_2), \vartheta)$ равномерно распределен на единичной сфере.

3.2. Цифровая фильтрация сигнала

Следующий пример иллюстрирует два подхода к реализации цифрового фильтра. Известно, что цифровой фильтр с передаточной функцией:

$F(w) = \frac{f(w)}{g(w)}$, где $f(w), g(w)$ - вещественные многочлены, может быть реализован в виде некоторой схемы, состоящей из фильтров с конечным временем отклика и рекурсивных фильтров первого и второго порядка.

3.3. Среда разработки Linux

Ниже приведены для справки команды, наиболее часто используемые в процессе разработки программ. Приводимые сведения ни в коем случае не могут заменить специальную литературу, а предназначены только для указания ссылок.

On-line информация о команде выводится командой:

`man имя_команды` или `info имя_команды`

Более подробное описание какой-либо подсистемы представлено в сборниках HowTo. Наконец, некоторые фундаментальные подсистемы (компилятор, отладчик, различные редакторы) подробно описаны разработчиками и доступны в Internet.

Последние версии Linux имеют как графический, так и текстовый интерфейс. Ниже рассматривается работа только в текстовом режиме.

Для подключения к Linux машине из Windows в режиме «выполнить» набираем команду:

`telnet имя Linux машины` или `ee IP адрес`

После этого надо ввести имя пользователя и пароль.

3.4. Ведение собственной директории

Каждому пользователю отводится своя директория. Одновременно могут работать много пользователей, число которых, на практике, ограничено ресурсами Linux машины. Файловая система имеет обычную иерархическую структуру, корнем которой является /. Например, пользователь с именем Stud может иметь личную директорию /home/Stud (обратите внимание на отличие от Windows в направлении значка slash). Команда pwd печатает полное имя директории, в которой находится пользователь (текущая директория). Внутри своей директории пользователь может создавать свое дерево файлов. Команда mkdir имя_директории создает новую директорию внутри текущей. Команда cd имя_директории делает новую директорию текущей. Имя, состоящее из одной точки и слеша ./, является синонимом текущей директории. Например, пусть в директории /home/Stud находится файл с именем TEXT, и эта директория является текущей. В этом случае имена TEXT и ./TEXT ссылаются на один и тот же файл. Еще одно отличие от Windows заключается в том, что в именах файлов верхнее или нижнее положение регистра является существенным. Содержимое директории печатается командой ls. У этой команды много опций, перечисление которых можно увидеть, введя команду man ls. Отметим наиболее важные. Команда ls –l печатает имена файлов со всеми атрибутами, команда ls –a печатает скрытые файлы. В отличие от Windows, где вся информация о настройках программ, выбранных пользователем, содержится в реестре, в Linux эта информация помещается в директорию пользователя, но имена файлов или директорий начинаются с точки. Это и есть скрытые файлы, которые можно увидеть, воспользовавшись опцией команды ls. Для удаления файла используется команда rm, для удаления директории добавляется опция –r. Например, если в директории /home/Stud находится директория temp, то для удаления последней в директории /home/Stud выполняется команда rm –r temp. Для удаления всех файлов директории выполняется команда rm *. Эта команда не удалит поддиректории и скрытые файлы. При удалении файлов, как правило, делается запрос на правомерность удаления каждого файла. Чтобы избежать этого запроса, пользуются опцией –f в команде удаления. Можно объединить

несколько команд таким образом, чтобы выход одной стал входом для другой. (программный канал). В этом случае команды разделяются вертикальной чертой. Например, чтобы узнать, находится ли в данной директории файл, в имени которого имеется подстрока tem, достаточно выполнить команду:

```
ls | grep tem.
```

О свойствах исключительно полезной команды grep можно узнать, введя команду man grep. Для копирования файла используют команду cp старое_имя новое_имя, для переименования файла вводят команду mv старое_имя новое_имя. При работе в LINUX большое значение имеют переменные окружения (environment), Эти переменные устанавливаются различными способами, простейший из которых заключается в модификации файла .cshrc в корне каталога пользователя (скрытый файл).

3.5. Просмотр и редактирование файлов

С точки зрения Linux нет различия между исполняемыми, текстовыми или ресурсными файлами. Эти различия проявляются только в описании методов доступа к этим файлам. Эти методы видны если печатается команда:

```
ls  -l имя_файла
```

Для изменения методов доступа используется команда chmod.

Для просмотра файла можно пользоваться любой из утилит или одним из многочисленных текстовых редакторов. Простейшей является утилита more. Чтобы просмотреть содержимое файла, вводим команду more имя_файла. Эта же утилита используется для последовательного просмотра содержимого директории:

```
ls| more.
```

Более мощная утилита, позволяющая кроме просмотра осуществлять и поиск, носит имя less. Утилита cat позволяет кроме просмотра осуществлять ввод и слияние файлов. Кроме того, существуют многочисленные текстовые редакторы от простейшего pico (обычный экранный ввод) до очень сложного, с развитым языком программирования редактора emacs. Очень кратко остановимся на описании редактора vi, или vim. Этот редактор ориентирован на разработчика программ, хотя и представляется на первых порах не очень удобным. Редактор предполагает работу в текстовом режиме, хотя у него есть графический клон gvim, который позволяет работать с мышью. Ниже будет приведен самый необходимый минимум команд. Редактор снабжен прекрасным on-line справочником, по которому можно ознакомиться со всеми возможностями.

Основная особенность редактора заключается в том, что он может находиться в двух режимах: командном и экранном. После загрузки редактора командой vi для входа в

экранный режим нажимаем клавишу ‘i’. После этого можно вводить текст стандартным образом. Для выхода из экранного режима нажимаем клавишу ‘esc’, и редактор переходит в командный режим. Для сохранения содержимого экрана в файле набираем команду:

```
:w имя_файла
```

Отметим, что двоеточие является элементом команды. Для загрузки файла в редактор используется команда:

```
:e имя_файла.
```

Редактор содержит несколько буферов, поэтому возможно редактирования сразу нескольких файлов. Имеется буфер обмена. Для удаления участка текста вводим команду:

```
:n1,n2d.
```

Здесь n1 и n2 первая и последняя строки участка текста, подлежащего удалению. Весь фрагмент помещается в буфер обмена. Для вставки содержимого буфера обмена после позиции курсора нажимаем на клавишу ‘p’. Современный стиль программирования предполагает использование длинных идентификаторов переменных и функций. Для упрощения работы с такими идентификаторами редактор имеет две полезные команды автодополнения. В экранном режиме печатается начало идентификатора, после чего нажимаем Ctrl P или Ctrl N в зависимости от того, находится нужный идентификатор перед позицией курсора или после нее. В результате начальное имя идентификатора дополняется до полного. Имеются также команды поиска и замены. Кроме того, vi может обрабатывать команды из командного файла для обработки всего файла, однако в последнее время для этих целей используют специальные средства вроде Perl или Python. Не выходя из редактора, можно выполнить команду интерпретатора, напечатав:

```
:!команда
```

Для выхода из редактора вводят команду:

```
:q   или :q!,
```

если сделанные изменения не нужно сохранять. Утилита ctags создает специальный файл меток, содержимым которого пользуется редактор. С помощью этих меток можно быстро переходить от одного фрагмента программы к другому.

3.6. Компиляция и запуск программы

Для компиляции программы, написанной на языке С, используется команда:

```
gcc  -o имя_исполняемого_файла имя1.c
```

Здесь имя1.c – имя файла, содержащего текст программы. Если имеется несколько модулей или используются специальные библиотеки, то формат команды

модифицируется. Например, если программа состоит из двух модулей и использует математические функции, команда имеет вид:

```
gcc  -o имя_исполняемого_файла имя1.c имя2.c –lm
```

Команда на выполнение исполняемого файла имеет вид:

```
./имя_исполняемого_файла
```

Обратим внимание на точку в начале команды. Для того, чтобы эта команда выполнилась необходимо также, чтобы в описании доступа к файлу предусматривалась возможность исполнения (см. man chmod).

3.7. Отладка программы

Для возможности отладки программы необходимо, чтобы она была скомпилирована с опцией –g:

```
gcc -g-o имя_исполняемого_файла имя1.c
```

После этого программу можно запустить в режиме отладчика:

```
gdb имя_исполняемого_файла
```

О командах отладчика можно узнать, введя команду:

```
info gdb.
```

Приведем лишь простейшие из них.

После запуска отладчика надо установить точки прерывания. Для установки прерывания при входе в программу набираем команду:

```
break main (Можно заменить break на b)
```

Команда l позволяет просмотреть текущий фрагмент исходного кода. Любое прерывание можно установить командой `b num_line` . Значение номера строки присутствует и в редакторе Vi. (ctrl-g).

Для запуска программы набираем `run.`

Для перехода к следующей команде после прерывания набираем n или s в зависимости от перехода к следующей строке или входа в подпрограмму.

Значение переменной получаем, набрав команду: `p name_of_identifier.`

Для выхода из отладчика набираем q.

3.8. MPI - Описание системы и практическое руководство

При написании данного раздела автор пользовался книгой *MPI: The Complete Reference*. by Marc Snir , and others. MIT Press (Книга доступна в Интернете).

MPI – Message Passing Interface является стандартом для реализации вычислений на основе параллельной вычислительной системы. Он был сформирован к 1994. Существуют реализации стандарта как открытые, например LAM, LAM, так и коммерческие. Мы

будем рассматривать реализацию LAM под управлением LINUX. Этот стандарт поддерживает различные языки программирования, однако, мы будем иметь дело только с программированием на основе языка С. Полное рассмотрение стандарта не является целью данных лекций. Описываются наиболее часто используемые функции и контекст, в котором они применяются.

3.9. Обзор системы

С точки зрения системного программирования, установка LAM сводится к установке дополнительных библиотек и специальных опций компилятора. В результате для программиста становятся доступными функции интерфейса MPI. Запуск программы осуществляется с помощью специального скрипта. Перед началом работы запускается скрипт lamboot. Цель этой команды включить в кластер имеющиеся компьютеры и установить среду окружения для пользователя. После окончания работы следует выполнить команду `.lamhalt`. При установке LAM системе сообщаются все доступные процессоры кластера. Эти процессоры могут быть частью одного компьютера или принадлежать разным компьютерам кластера. Один из параметров скрипта указывает, сколько процессов должно быть запущено. Система сама решает, каким образом процессы будут распределены среди физически доступных процессоров. Все процессы исполняют одну и ту же программу, однако внутри программы можно определить номер процесса, исполняющего данную программу. Процессы обмениваются данными, используя номер процесса в качестве параметра.

Параллельное исполнение начинается после создания среды выполнения: запускаются процессы в количестве, указанном при запуске (MPI_Init). После этого каждый процесс узнает свой индивидуальный ранг (MPI_Comm_rank) и другие параметры, если это необходимо. Обмен между процессами осуществляется командами MPI_Send, MPI_Recv и MPI_Bcast. Последняя применяется, если информация передается всем процессам одновременно. Перед окончанием каждый процесс должен выполнить команду MPI_Finalize

3.10. Пример программы

Для того, чтобы проиллюстрировать сказанное выше, рассмотрим пример MPI программы. Программа требует, чтобы было запущено, по крайней мере, два процесса. Все процессы печатают свои имена. Процессы с номером 0 сообщаем всем свое имя и процесс с номером 1 его модифицирует.

```
1. #include "mpi.h"
2. #include <stdio.h>
3. #include <string.h>
```

```
4. int main( int argc, char *argv[])
5. {
6. int  n, myid, numprocs;
7. int  namelen;
8. char processor_name[MPI_MAX_PROCESSOR_NAME];

9. MPI_Init(&argc,&argv);
10. MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
11. MPI_Comm_rank(MPI_COMM_WORLD,&myid);
12. MPI_Get_processor_name(processor_name,&namelen);

13. fprintf(stderr,"Process %d on %s\n",myid, processor_name);

14. if(numprocs<2){

15.
16. fprintf(stderr,"The program needs at least 2 processes");
17. MPI_Abort(MPI_COMM_WORLD,1);
18. }
19. if (myid == 0){
20. strcpy(processor_name,"I'm zero");
21. fprintf(stderr,"%s",processor_name);
22.
23. n=strlen(processor_name);
24. MPI_Send(&n, 1, MPI_INT, 1,0, MPI_COMM_WORLD);
25. MPI_Send(processor_name,n+1,MPI_CHAR,1,1,MPI_COMM_WORLD);
26. }
27. if(myid==1){
28. MPI_Status sts;
29. MPI_Recv(&n, 1, MPI_INT, 0,0, MPI_COMM_WORLD,&sts);
30. MPI_Recv(processor_name,n+1,MPI_CHAR,0,1,MPI_COMM_WORLD,&sts);
31.
32. strcat(processor_name,"+1\n");
33. fprintf(stderr,"%s",processor_name);
34. }
35. MPI_Finalize();

36. return 0;
37. }
```

Процесс с номером 0 передает фразу «I'm zero» процессу с номером 1, последний модифицирует эту фразу и печатает ее. Рассмотрим более подробно содержимое листинга. Строка 1 всегда присутствует в MPI программе, поскольку файл "mpi.h" содержит объявление основных функций MPI. Переменные, объявленные в строках 6-8, являются локальными в каждом процессе. Это означает, что их инициализация в одном процессе не влияет на содержимое в других процессах. Строка 9 присутствует в каждой MPI программе. Только после вызова этой функции возможно обращение к другим MPI функциям. Строка 10 позволяет каждому процессу узнать общее число процессов, участвующих в вычислениях. В строке 11 каждый процесс узнает свой номер. В строке 12 каждый процесс узнает имя компьютера, на котором он выполняется, и длину этого имени. Строка 17 завершает аварийно выполнение всех программ, если общее число процессов меньше 2. В строках 19-26 процесс с номером 0 формирует в массиве

processor_name фразу ,"I'm zero" и вычисляет ее длину. Затем процесс 0 посылает информацию процессу с номерам 0, а процесс с номером 1 ее модифицирует. Строка 35 должна присутствовать в каждой MPI программе для нормального завершения.

В данном описании остался нераскрытым смысл некоторых параметров. Это будет сделано позже.

Из сказанного выше вытекает, что MPI интерфейс лучше всего подходит для SIMD программ, хотя использование переменной с номером данного процесса позволяет модифицировать выполнение программы каждым процессом. Заметим также, что общее число процессов определяется в скрипте вызова и не может быть изменено в процессе выполнения. Впрочем, возможно, что в следующих версиях будет предусмотрена опция динамического вызова.

3.11. Обмен данными между процессами (point to point communications)

Как правило, в процессе параллельных вычислений происходит обмен данными между двумя процессами. При этом один процесс посылает данные, а другой их получает. Для этой цели предназначена пара команд: MPI_Send и MPI_Recv. Рассмотрим формат этих команд.

```
int MPI_Send(void* buff, int count, MPI_Datatype dtype, int dest,
int tag , MPI_Comm com);
```

Эта команда предполагает, что пересылаемые данные находятся в некотором буфере, поэтому указывается адрес этого буфера. Это относится и к ситуации, когда пересылается только одно число, в этом случае в buff помещается его адрес. Параметр dtype указывает тип пересылаемых данных. Укажем некоторые из них: MPI_INT, MPI_CHAR, MPI_FLOAT, MPI_DOUBLE. Они соответствуют обычным объявлениям переменных в C, а параметр count указывает количество элементов указанного типа в буфере, подлежащих пересылке. Возможно и определение пользователем своих типов данных, но при этом возникают проблемы с непрерывным расположением данных в памяти. Простой пример такого определения будет рассмотрен позже. Возможно определение пользовательского типа и в случае, когда данные не составляют непрерывный массив, но мы не будем останавливаться на путях преодоления указанных трудностей.

Параметр com определяет домен процессов. Стартовый скрипт задает общее число процессов. Множество этих процессов составляют домен с фиксированным именем MPI_COMM_WORLD. В то же время, пользователь может создать свой собственный домен, поместив в него лишь часть процессов, стартовавших в начале. В любом домене

каждый процесс получает свой номер, называемый рангом. Параметр dest определяет ранг приемника в домене com. Предполагается, что и посылающий процесс также принадлежит тому же домену. Процесс может одновременно принадлежать нескольким доменам. Каждое сообщение, посылаемое командой MPI_Send, помечается tag'ом, который должен быть неотрицательным числом.

```
int MPI_Recv(void* buff, int count, MPI_Datatype dtype, int
src, int tag, MPI_Comm com, MPI_Status *stat);
```

Выполнение данной команды приостанавливается до тех пор, пока не придет сообщение от источника. Параметры buff, dtype, com имеют то же назначение, что и в предыдущей команде, count – не превышает значение соответствующего параметра в команде MPI_Send. Команда начинает выполняться лишь после того, как пришло сообщение от процесса с номером src с соответствующим tag'ом. Параметр src, ранг источника в домене может быть равен MPI_ANY_SOURCE, и тогда конкретное значение ранга игнорируется. Точно так же параметр tag может равняться MPI_ANY_TAG, и тогда игнорируется значение tag'а. После выполнения команды MPI_Recv системный приемный буфер очищается, и остается лишь та часть сообщения, которая была прочитана в буфер команды MPI_Recv. Параметр stat определяет фактический статус принятого сообщения. Дело в том, что команда начинает выполняться как только придет какое-либо сообщение с нужными параметрами. Структура stat содержит поля с информацией о фактическом статусе полученного сообщения, а также фактические значения источника и tag. Для получения фактической длины полученного сообщения используется функция MPI_Get_count(&stat,dtype,&count), которая вызывается после MPI_Recv. На практике это делается в тех случаях, когда имеются сомнения в надежности канала между источником и приемником.

Команды MPI_Send и MPI_Recv являются примерами блокирующих (blocking) команд. Это означает, что после вызова этих команд работа программы продолжится лишь при выполнении некоторых условий. Понятие «блокирующий» не стандартизируется и зависит от реализации. Система может содержать специальные системные буферы, где собираются сообщения, предназначенные для отправки, и принятые сообщения, однако, наличие таких буферов не является обязательным. Для команды MPI_Send разблокирование может означать как помещение сообщения в системный буфер и его последующая очистка, так и получение сообщения командой MPI_Recv. Хотя в принципе допустимо требование подтверждения о приемке сообщения, однако это существенно замедлит выполнение. Решение проблемы заключается в

создании аналогов команд посылки и приема, где постулируется отсутствие блокирования.

Использование блокирующих команд может привести к зависанию программы (deadlock). Рассмотрим следующий фрагмент:

```
if(rank==0){
MPI_Recv(buff,count,dtype,1,tag,com);
MPI_Send(buff,count,dtype,1,tag,com);
}
if(rank==1){
MPI_Recv(buff,count,dtype,0,tag,com);
MPI_Send(buff,count,dtype,0,tag,com);
}
```

Процесс 0 ожидает сообщение от процесса 1 и только после этого может послать сообщение процессу 1, а процесс 1 ждет сообщения от процесса 0. Здесь зависание неизбежно. Более тонкая ситуация возникает в следующем фрагменте:

```
if(rank==0){
MPI_Send(buff,count,dtype,1,tag,com);
MPI_Recv(buff,count,dtype,1,tag,com);
}
if(rank==1){
MPI_Send(buff,count,dtype,0,tag,com);
MPI_Recv(buff,count,dtype,0,tag,com);
}
```

В этом случае все зависит от реализации блокирующий функций. Если команда MPI_Send блокирует работу программы до тех пор, пока не получить подтверждение от команды MPI_Recv зависание будет и в этом случае. Ниже будет показано, каким образом избежать подобных коллизий. Следующий фрагмент работает без проблем:

```
if(rank==0){
MPI_Send(buff,count,dtype,1,tag,com);
MPI_Recv(buff,count,dtype,1,tag,com);
}
if(rank==1){
MPI_Recv(buff,count,dtype,0,tag,com);
MPI_Send(buff,count,dtype,0,tag,com);
}
```

Необходимость подобных пересылок возникает достаточно часто. Примером служит обмен данными в пограничных областях в задаче о распространении тепла, рассмотренной выше. Для совершения указанных действий может использоваться специальная команда:

```
int MPI_Sendrecv(void* sendbuff, int sendcount, MPI_Datatype
senddtype, int dest, int sendtag, void* recvbuff, int recvcount,
MPI_Datatype recvdtype, int source, int recvtag, MPI_Comm com,
MPI_Status *stat).
```

Это блокирующая команда, но ее использование исключает зависание. С другой стороны, сообщения, посланные MPI_Send, читаются этой командой, а посланные с помощью MPI_Sendrecv сообщения читаются командой MPI_Recv.

ЛЕКЦИЯ 4

4.1. Неблокирующие операции (Nonblocking communications)

Использование неблокирующих операций предполагает наличие системных буферов для отправки и принятия сообщений. В противном случае теряется преимущество от использования таких операций, поскольку каждый раз нужно получать подтверждение о завершении операции. Впрочем, пользователь может создавать собственных буфер, но описание необходимых для этого операций выходит за пределы лекций.

Функция:

```
int MPI_Isend(void* buff, int count, MPI_Datatype dtype, int
dest,  int tag , MPI_Comm com,
MPI_Request * rqst),
```

также предназначена для посылки сообщений. Она инициирует начало передачи, но в отличие от MPI_Send , здесь нет никакой остановки для ожидания результата. Аналогично, функция:

```
int MPI_Irecv(void* buff, int count, MPI_Datatype dtype, int src,
int tag, MPI_Comm com, MPI_Request* rqst)
```

инициирует процесс получения сообщения, не ожидая результата.

Статус отправленного или принятого сообщения определяется с помощью структуры rqst, порождаемой операцией, и содержание структуры меняется по мере выполнения операции, однако, чтобы извлечь этот статус используют функции MPI_Wait(MPI_Request*, MPI_Status* sts) или MPI_Test(MPI_Request*, int* flag, MPI_Status* sts). Первая функция переводит программу в режим ожидания до завершения операции, а вторая позволяет определить текущий статус. Завершение MPI_Isend заключается в очистке системного буфера, а завершение MPI_Irecv означает получение сообщения. Функция MPI_Test возвращает значение flag==true, если операция завершена. Отметим, что эта функция может вызываться многократно с одними и теми же параметрами, чтобы определить момент завершения. Например:

```
MPI_Request rqst;
MPI_Status sts;
int flag=0;
MPI_Irecv( buff,  count, dtype,  src,  tag, com,  &rqst)
while(!flag){
…….
MPI_Test(&rqst,&flag,&sts);
}
```

Внутри цикла выполняются операции, а после завершения приема сообщения происходит выход из цикла. В этом случае можно уточнить принятую информацию,

используя функцию MPI_Get_count (MPI_Status*, MPI_Datatype, int* count). С ее помощью можно подсчитать число принятых элементов. На практике, нет особого смысла использовать MPI_Isend вместе с MPI_Wait, поскольку это равносильно обычному блокирующему вызову. Исключение составляет случай, когда предполагается, что порядок приема сообщений не совпадает с порядком посылки. В этом случае использование блокирующей функции посылки может привести к зависанию, если реализована функция подтверждения приема сообщения. Использование MPI_Irecv вместе с MPI_Test имеет смысл, когда у процесса имеется работа до получения очередного сообщения (например, обработка предыдущего). Рассмотрим реализацию алгоритма цифровой фильтрации, когда цифровой фильтр представлен в виде суммы параллельных элементарных фильтров. В терминах передаточной функции:

$$F(w) = \sum_{k=0}^{N-1} F_k(w).$$

Предположим, что число процессоров P значительно меньше, чем N. Запускаем P процессов, и каждый из запущенных процессов реализует один из элементарных фильтров.

Главная программа (один из процессов) содержит список всех элементарных фильтров. Производит первоначальное распределение фильтров между процессами, включая себя.

Остальные процессы по окончании фильтрации сообщают об этом главной программе.

Главная программа выставляет запрос:

```
MPI_Irecv( buff, count, dtype, MPI_ANY_SOURCE, tag, com, rqst).
```

После этого, проводя фильтрацию, периодически опрашивает с помощью функции: MPI_Test(&rqst,&flag,&sts) состояние запроса. После получения значения true для flag анализирует содержимое sts, определяет, какой из процессов завершил работу и назначает ему новую работу, если список фильтров еще не исчерпан.

Недостаток неблокирующих операций состоит в том, что интенсивное использование таких операций без соответствующих операций MPI_Wait может привести к переполнению системных буферов. Частично, этот недостаток может быть компенсирован использованием функции MPI_Waitany.

На рисунке 7 представлена схема последовательного соединения фильтров, а на рисунке 8 – схема параллельного соединения. Отметим, что здесь значок $\oplus$ означает обычное суммирование, а не суммирование по модулю 2.

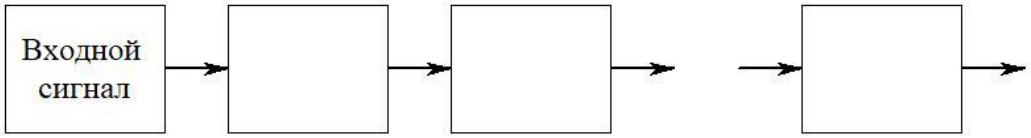

Рисунок 7. Схема последовательного соединения фильтров

Возможны и комбинации параллельного и последовательного соединения. Мы здесь не затрагиваем вопроса, каким образом находится декомпозиция произвольного фильтра в структурную схему того или иного вида, а также недостатки или преимущества той или иной реализации.

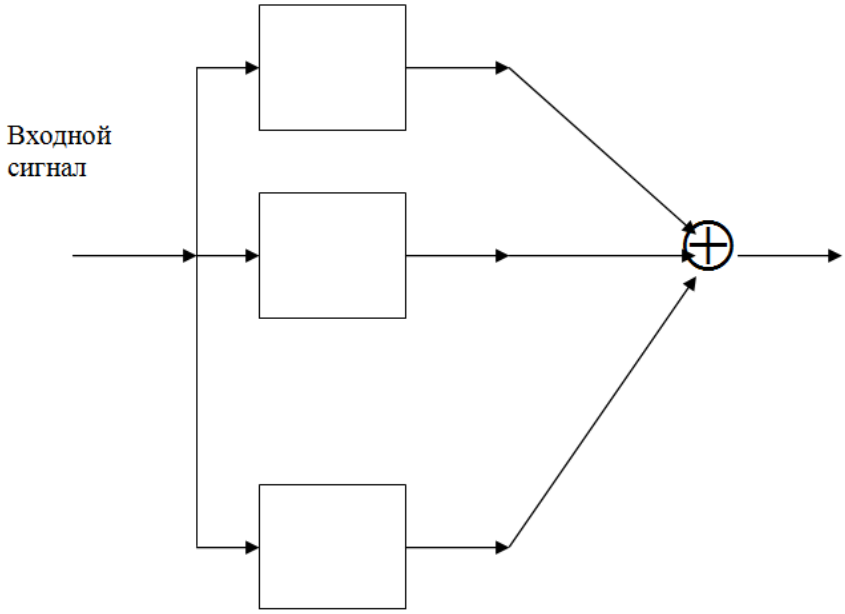

Рисунок 8. Схема последовательного соединения фильтров

Очевидно, что такая реализация фильтра укладывается на параллельную вычислительную сеть, достаточно создать для каждого элементарного фильтра свой процесс. Отметим, что в этом случае взаимодействие ГП с остальными сведено к минимуму, достаточно взаимодействия процессов друг с другом. Процессы должны обмениваться информацией о завершении очередного этапа обработки и о готовности приступить к следующему этапу.

4.2. Быстрое преобразование Фурье

Быстрое преобразование Фурье (БПФ) Fast Fourier Transform (FFT) играет исключительно важную роль в цифровой обработке сигналов. Если функция задана своими значениями в N точках : $f(\frac{k}{N}), k = 0,\ldots,N-1$, то ее дискретное преобразование Фурье определяется формулой:

$$F(m) = \frac{1}{N}\sum_{k=0}^{N-1} \exp(-2\pi ikm/N) f(\frac{k}{N}),\ m = 0,\ldots,N-1.$$

Простейшая схема распараллеливания заключается в распределении вычисления различных значений $F(m)$ между разными процессорами. Если N обладает специальными арифметическими свойствами, например $N = 2^n$, существует схема вычисления (БПФ), требующая меньшее количество арифметических операций. В то же время, БПФ требует интенсивного обмена данными между различными процессорами, поэтому практическое применение находит параллельная реализация БПФ, когда все процессоры находятся внутри одного чипа.

4.3. Распространение тепла в пластине

Рассмотрим задачу о распространении тепла в пластине. В основе решения лежит уравнение Лапласа:

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0$$

На границе пластины температура известна. Требуется определить температуру в остальных точках. Задача решается сеточным методом. Вся область покрывается сеткой с постоянным шагом. Для простоты будем считать, что величина шага равна единице. Первая производная по x заменяется разностью $U(x+1,y) - U(x,y)$, а вторая производная аппроксимируется разностью:

$$(U(x+1,y) - U(x,y)) - (U(x,y) - U(x-1,y)) = -2U(x,y) + U(x+1,y) + U(x-1,y) \quad (6)$$

Проведя аналогичные вычисления для переменной y, получим сеточный аналог уравнения Лапласа в виде:

$$U(x,y) = \frac{1}{4}(U(x+1,y) + U(x-1,y) + U(x,y+1) + U(x,y-1)) \quad (6)$$

Решение задачи (6) производится с помощью итераций. Для этого полагается начальное значение функции $U_0(x,y) = 0$ во внутренних точках пластины, а на границе значения U_0 известны. После этого используется рекуррентная формула:

$$U_{n+1}(x,y) = \frac{1}{4}(U_n(x+1,y) + U_n(x-1,y) + U_n(x,y+1) + U_n(x,y-1))$$

Вычисления производят до тех пор, пока не будет выполнено условие:

$$\max|U_{n+1}(x,y) - U_n(x,y)| < \delta,$$

где значение δ выбрано заранее.

Коллективные операции затрагивают все процессы, входящие в группу и являются блокирующими для процессов группы. Они не конфликтуют с обычными point to point операциями (операции, инициированные коллективными функциями, не могут взаимодействовать с point to point операциями).

Функция MPI_Barrier (MPI_Comm com), присутствующая во всех процессах группы com синхронизирует выполнение отдельных процессов. Продолжение вычислений возможно лишь после того, как все процессы группы выполнят эту команду.

Функция MPI_Bcast (void* buff, int count, MPI_Datatype dtype, int root, MPI_Comm com) передает данные из буфера buff в процессе с рангом root остальным процессам группы. Типичная ошибка при использовании этой функции заключается в отсутствии предварительной инициализации переменной count в остальных процессах группы. Если это значение не известно на этапе программирования, оно должно передаваться той же командой, где в качестве буфера используется &count.

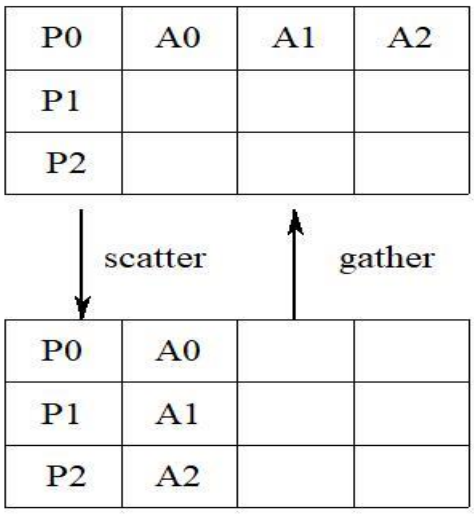

Рисунок 9. Команды MPI_Gather и MPI_Scatter

Смысл команд MPI_Gather и MPI_Scatter представлен на рисунке 9. Первая из них имеет следующее объявление.

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype
sdtype,void* recvbuf,
int recvcount,MPI_Datatype rdtype, int root, MPI_Comm com).
```

В случае, изображенном на рисунке, три процесса содержат 3 массива одного размера и одного типа. Один из них выбирается в качестве root. В результате выполнения команды все три массива помещаются в один массив в процессе root в порядке соответствующем рангам процессов. Рассмотрим следующую программу:

```
#include "mpi.h"
#include <stdio.h>
```

```
int main( int argc, char *argv[])
{
    int   myid, numprocs, i;
    int  sendbuf[2];
    int *recvbuf;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    if(numprocs<2){
     fprintf(stderr,"The program needs at least 2 processes");
     MPI_Abort(MPI_COMM_WORLD,1);
    }
    sendbuf[0]=myid;
    sendbuf[1]=myid+1;
    if (myid == 1){
    recvbuf=(int*)malloc(numprocs*sizeof(sendbuf));
    }
 MPI_Gather(sendbuf,2,MPI_INT,recvbuf,2,MPI_INT,1,MPI_COMM_WORLD);
 if(myid==1){
 for(i=0;i<2*numprocs;i++)
 fprintf(stderr,"%d ",recvbuf[i]);
 }

    MPI_Finalize();

    return 0;
}
```

Каждый процесс помещает в свой sendbuf значения myid и myid+1, после чего все они собираются в массиве recvbuf в процессе с рангом 1. Выделение памяти для recvbuf производится только в процессе с номером 1, но этот аргумент присутствует во всех процессах. Обратим внимание, что значение recvcount равно размеру массива, посылаемому каждым процессом, а не суммарной размерности. Суммарная размерность использована в операторе malloc. В этой связи, на первый взгляд являются излишними указания параметров rctype и recvcount. На самом деле такое определение делает использование функции более гибким. В предыдущем примере можно ввести свой тип переменных int[2]. Используя этот тип, полагаем recvcount=1. Функция MPI_Scatter имеет тот же набор аргументов и выполняет операцию обратную к предыдущей.

Последовательное выполнение MPI_Gather и MBI_Bcast можно заменить одной командой MPI_Allgather.

4.4. Оператор свертывания (Reduce)

Предположим, что имеется N процессов, каждый из которых подсчитывает некоторое значение a_i, после чего надо найти, например, $\sum a_i$. Для выполнения указанных действий предназначена специальная команда

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype
dtype, MPI_Op oper, int root, MPI_Comm com).
```

К числам, находящимся в массивах sendbuf каждого из процессов, поэлементно применяется одна и та же операция oper. Результат записывается в recvbuf процесса root. Например, (рисунок 10).

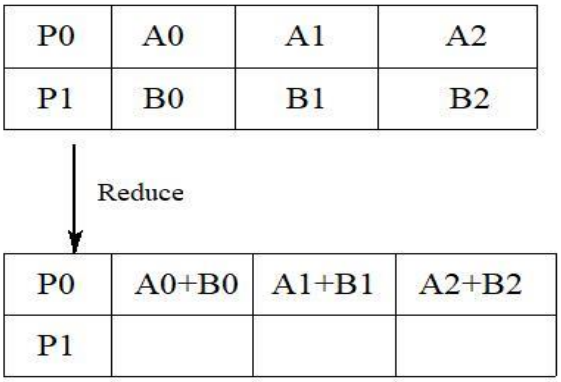

Рисунок 10. Команда MPI - Reduce

Процессы 0 и 1 содержат буферы размера 3. Выберем в качестве root процесс 0, а в качестве oper=MPI_SUM. Тогда поэлементная сумма обоих массивов будет помещена в recvbuf процесса 0. Существует много встроенных операций. Перечислим некоторые из них: MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD - произведение, MPI_LAND, MPI_LOR – логические AND и OR. Две последние операции применимы к целым числам. Существует возможность и определения операции пользователем. Ниже будет рассмотрен соответствующий пример.

4.5. Пользовательское определение типа и операции в команде MPI_Reduce

Операция, вводимая пользователем обязательно должна быть ассоциативной, поскольку в противном случае результат выполнения операции не определен. Ассоциативными являются сложение и умножение чисел, которые в то же время будут и коммутативными. Известно, что умножение квадратных матриц одного размера является ассоциативной операцией, но вообще говоря, некоммутативной. Покажем, каким образом можно определить тип и операцию, чтобы в результате операции MPI_Reduce получить произведение матриц. Согласно определению, эта команда производит операции над элементами массива, поэтому первая задача заключается в построении массива, элементами которого будут матрицы. Для этого надо создать пользовательский MPI_Datatype для определения матриц. Задача упрощается, поскольку при стандартном

определении матрица занимает непрерывный сегмент оперативной памяти. Рассмотрим пример программы, определяющей новый тип для работы с матрицами:

```
1.    #include "mpi.h"
2.    #include <stdio.h>
3.    #define MATR_SIZE 3
4.    int main( int argc, char *argv[])
5.    {
6.    int   myid, numprocs, i,j;
7.    float  sendrecvbuf[2][2];
8.    int *recvbuf;
9.          MPI_Init(&argc,&argv);
10.   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
11.   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
12.   MPI_Datatype    myMatr;
13.   MPI_Type_contiguous(MATR_SIZE*MATR_SIZE,MPI_FLOAT,&myMatr);
14.   MPI_Type_commit(&myMatr);
15.   if(numprocs<2){
16.   fprintf(stderr,"The program needs at least 2 processes");
17.   MPI_Abort(MPI_COMM_WORLD,1);
18.   }
19.   if(myid==0){
20.   for(i=0;i!= MATR_SIZE;i++)
21.   for( j=0;j!= MATR_SIZE;j++)
22.   sendrecvbuf[i][j]=(float)(i-j);
23.   MPI_Send(sendrecvbuf,1,myMatr,1,0,MPI_COMM_WORLD);
24.   }
25.   if(myid==1){
26.   MPI_Status sts;
27.   MPI_Recv(sendrecvbuf,1,myMatr,0,0,MPI_COMM_WORLD,&sts);
28.   for(i=0;i!= MATR_SIZE;i++)
29.   for( j=0;j!= MATR_SIZE;j++)
30.   fprintf(stderr,"%f ",sendrecvbuf[i][j]);
31.   }
32.   MPI_Finalize();
33.   return 0;
34.   }
```

В программе, представленной выше в строке 12 объявляется пользовательский тип myMatr. В строке 13 определяется, что данные этого типа занимают непрерывный сегмент памяти, состоящий из MPI_FLOAT, причем эта область имеет длину MATR_SIZE*MATR_SIZE. После выполнения команды в строке 14 этот тип данных становится доступным функциям интерфейса. Команды 23 и 27 работают с этими данными, как со встроенными. Обратим внимание, что счетчик в командах MPI_Send и MPI_Recv равен 1, однако работа с отосланными данным производится как с обычной матрицей. При желании, мы можем определить массив, элементами которого будут квадратные матрицы.

Перейдем теперь к следующему шагу: определению пользовательской операции. Прежде всего, надо определить функцию, которая будет осуществлять нужную операцию.

Она имеет следующий прототип:

typedef void MPI_User_function(void* invec, void* inoutvec, int* len, MPI_Datatype dtype).

Предполагается, что имеются два массива invec и inoutvec одинаковой длины len, каждый элемент этих массивов содержит указатели на элементы типа dtype. Для каждого $0 \le i < len$ функция извлекает аргументы с помощью указателей invec[i], inoutvec[i] и помещает результат по адресу inoutvec[i] . Далее, надо определить операцию на основе построенной функции. Для этого сначала эту операцию объявляют, а затем определяют с помощью функции MPI_Op_create.

```
     #include "mpi.h"
     #include <stdio.h>
     #define MATR_SIZE  3
     typedef float matr[MATR_SIZE][MATR_SIZE];
     void matrProd(matr*in, matr* inout){
      int i,j,k;
      matr temp;
      for(i=0;i!=MATR_SIZE;i++)
            for(j=0;j!=MATR_SIZE;j++){
                  float sum=0;
                  for(k=0;k!=MATR_SIZE;k++)
                        sum +=((*in)[i][k])*((*inout)[k][j]);
                  temp[i][j]=sum;
            }
      for(i=0;i!=MATR_SIZE;i++)Групповые   взаимодействия   (Collective
     communications)
            for(j=0;j!=MATR_SIZE;j++)
                  (*inout)[i][j]=temp[i][j];
}

void fun(void* in,void* inout,int* len,MPI_Datatype* ex){

      int i;

      for(i=0;i<*len;i++){
            matr* matIn=(matr*)in;
            matr* matOut=(matr*)inout;
            matrProd(matIn,matOut);
            in++;
            inout++;
      }
}

int main( int argc, char *argv[])
{
    int   myid, numprocs, i,j;
    matr  sendbuf,recvbuf;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Datatype  myMatr;
    MPI_Op              myOp;
    MPI_Type_contiguous(MATR_SIZE*MATR_SIZE,MPI_FLOAT,&myMatr);
    MPI_Type_commit(&myMatr);
    MPI_Op_create(fun,0,&myOp);
    if(numprocs<2){
          fprintf(stderr,"The program needs at least 2 processes");
          MPI_Abort(MPI_COMM_WORLD,1);
```

```
    }
    for(i=0;i!=MATR_SIZE;i++)
          for( j=0;j!=MATR_SIZE;j++)
                sendbuf[i][j]=(float)(i-j+myid);
    MPI_Reduce(sendbuf,recvbuf,1,myMatr,myOp,0,MPI_COMM_WORLD);


    if(myid==0){
          for(i=0;i!=MATR_SIZE;i++)
          for( j=0;j!=MATR_SIZE;j++)
                fprintf(stderr,"%f ",recvbuf[i][j]);
    }

    MPI_Finalize();

    return 0;
}
```

4.6. Поиск кратчайшего пути в графе

Известная задача о коммивояжёре формулируется следующим образом. Имеется n городов, известно расстояние между каждой парой этих городов. Требуется обойти все города, не заходя в один город дважды, выбрав кратчайший путь. Различают задачи с возвращением в точку исхода или без него. Эта задача является разновидностью задачи о поиске кратчайшего пути на графе. Идея параллельной реализации алгоритма выглядит следующим образом. Имеется начальная точка S, из которой исходят k дуг, идущие в вершины $A_1,\ldots,A_k$. Запускают k процессов, каждый из которых решает аналогичную задачу, имея в качестве источника точку $A_i, i=1,\ldots,k$. Завершив решение, каждый процесс передает результат master процессу, который и решает вопрос об оптимальном выборе. В частности, для задачи о комивояжере с возвращением в качестве S можно брать любую вершину, после чего возникают $(n-1)!$ вариантов обхода. Задача естественным образом укладывается на сеть, содержащую $n-1$ процессов.

4.7. Цифровая обработка изображений, “водяные знаки”

Обработка изображений, как правило, требует больших ресурсов, с другой стороны, естественное разбиение изображения на части позволяет использовать параллельные вычисления. Мы остановимся лишь на одном аспекте проблемы весьма популярном в последнее время. Имеется в виду создание и обнаружение «водяных знаков» (watermarks). Речь идет о добавлении в рисунок специальной информации, не искажающей сам рисунок, но невидимой при обычном просмотре, либо построение специальной функции, порождающей для данного рисунка вполне определенное значение. При этом преследуются различные цели: подтверждение авторского права на рисунок, проверка попыток изменения изображения, идентификация изображения, скрытая передача информации и другие. Такая же задача решается и для аудио продукции.

Добавленная информация получила название watermark по аналогии с обычными водяными знаками, а соответствующая отрасль знаний стала частью методов защиты информации. Вместе с появлением водяных знаков возникла задача их обнаружения и удаления, а у разработчиков методов watermarking появилась проблема защиты от указанных действий.

Рассмотрим только проблему автоматической идентификации изображения. В автоматическом режиме требуется определить наличие искомого изображения среди представленного списка. Основная проблема заключается в том, что в результате применения процедур сжатия изображения или удаления из него небольшого фрагмента его восприятие человеком меняется мало, в то же время с точки зрения цифрового представления происходят кардинальные изменения в файле. Таким образом, задача идентификации должна исходить из содержимого изображения, инвариантного по отношению указанных преобразований.

В настоящее время предложены многочисленные методы watermarking. Мы остановимся лишь на двух из них. Первый заключается в добавлении в текстурную часть изображения периодического сигнала, который обнаруживается с помощью подсчета корреляции. Недостатком этого метода является неустойчивость по отношению к изменению размера изображения. Наиболее перспективной является некоторая процедура выделения особых точек изображения. Это могут быть наиболее яркие точки, точки границ объектов и т.п. Другими словами, удаление или изменение положения этих точек существенно исказит исходное изображение. Строится граф, вершинами которого являются указанные точки, а дуги помечаются расстояниями между этими точками в оригинале. После предъявления списка изображений, для каждого из них строится аналогичный граф, после чего производится сравнение с оригиналом. Процедуры поиска особых точек и процедуры сравнения графов укладываются на параллельную вычислительную сеть.

ЛАБОРАТОРНЫЕ ЗАНЯТИЯ

Лабораторная работа 1. Среда разработки Linux

Данная работа предназначена для ознакомления студента с основными пользовательскими командами ОС Linux. Для более подробного описания команд приведенных ниже студенту следует обратиться к любой книге по Linux либо воспользоваться системным описанием в Интернете (Linux documentation project).

1.1. Вход в систему

Предполагается, что рабочие станции работают в среде Windows или LINUX, имеется Linux сервер, к которому есть доступ через локальную сеть, и у пользователя имеется пароль.

1. Установить латинскую клавиатуру.
2. Выполнить команду Пуск\выполнить\telnet адрес_сервера в Windows .
3. Ввести login, password.

В LINUX:

1. Войти в систему с локальным паролем для данной машины.
2. Ввести команду telnet адрес_сервера.
3. Ввести login, password.

Для изменения пароля ввести команду passwd старый пароль и затем ввести новый пароль. Если пароль не удалось заменить, надо увеличить его длину.

После успешного входа в систему пользователь попадает в собственный каталог, и интерпретатор (shell) ОС готов воспринимать команды. Для каждой приводимой ниже команды имеется on-line справочник.

1.2. Работа со справочником

Для доступа к справочнику достаточно набрать команду man имя_команды. Точно так же происходит запрос о формате функции в языке С. Поскольку некоторые команды shell и имена функций совпадают, команды справочника разбиты на секции. Для доступа к нужной секции набирается команда man номер_секции имя_команды. Если имя команды точно не известно, но известно, какое либо слово в ее описании, набирают команду man –k слово. Эта опция работает, если предварительно была создана база справочника. Для наиболее сложных команд информация получается командой info имя.

1.3. Работа с каталогом (директорией) и файлами

Имя текущего каталога отображается командой `pwd`. Стандартный каталог пользователя, по умолчанию, имеет вид /home/имя_пользователя, хотя это не обязательно. При входе в систему пользователь попадает в свой каталог. Внутри каталога пользователь может создавать свой каталог командой `mkdir` имя.

Упражнение 1.1

Создать каталог. С помощью справочника научиться пользоваться командой `cd`. Выяснить смысл обозначений ./ и ../. Автодополнение имени с помощь клавиши Tab.

Каталог просматривается командой ls. Файл можно создать любым текстовым редактором. Редактор рассмотрим позже. Простейший способ создать файл заключается в вводе команды: `cat>`имя, после чего можно вводить текст. Для выхода из команды набираем ctrl+D. Если имя файла начинается с точки, то это скрытый файл.

Упражнение 1.2

Создать файл внутри директории. Создать новую директорию. Перейти в нее и создать там еще один файл и скрытый файл. Вернуться в рабочую директорию. Ознакомиться с опциями команды `ls` . Что получится при выполнении команды с опциями –a –l –d -R их комбинацией? Опробовать смысл * в имени файла.

Параметры доступа к файлу высвечиваются при использовании опции –l в команде `ls`. Значение r означает возможность читать, w – возможность писать, x – возможность исполнять, d – признак директории. Эти три тройки символов относятся соответственно к пользователю, группе пользователя, остальным пользователям

Упражнение 1.3

Ознакомиться с командами `chmod, chown` для смены параметров доступа к созданным файлам. Выполнить изменение

Для манипуляций с файлами и каталогами предназначены команды: `rm` - удаление, `cp` - копирование, `mv` - переименование.

Упражнение 1.4

Ознакомиться с опциями команд `rm, cp,mv`. Каким образом копируются , удаляются и переименовываются библиотеки? Как копировать только те файлы, которые были изменены после последнего копирования?

Для поиска файлов по атрибутам в дереве каталогов предназначена команда `find`. Для поиска файла по его содержанию предназначена команда `grep`. Эти команды имеют довольно сложный формат, с которым следует ознакомиться по какому-нибудь руководству. Простейший способ работы с этими командами следующий. Для поиска файла по имени заходим в узел дерева каталогов, под которым предположительно находится файл, и набираем команду `find -name` имя. Если имя точно не известно, поиск производим так: `find -name` "часть_имени*". Для поиска файла по содержимому в простейшем случае поступаем так. Заходим в каталог, где предположительно находится файл, и выполняем команду: `grep` "что_ищем" *.

Упражнение 1.5

Найти собственные файлы, начав с каталога /. Что означает опция –R в команде grep? Найти собственные файлы по содержимому, начав с каталога /.

1.4. Просмотр и редактирование файлов

С точки зрения Linux нет различия между исполняемыми, текстовыми или ресурсными файлами. Для просмотра файла можно пользоваться любой из утилит cat, more, less. С помощью утилиты cat можно вводить файл, объединять текстовые файлы.

Упражнение 1.6

Ознакомиться с опциями cat. Создать два текстовых файла, объединить их в один. Просмотреть содержимое с помощью каждой из упомянутых утилит.

Более сложные манипуляции с текстовыми файлами производят с помощью одного из многочисленных текстовых редакторов от простейшего pico (обычный экранный ввод) до очень сложного, с развитым языком программирования редактора emacs. Очень кратко остановимся на описании редактора vi, или vim. Этот редактор ориентирован на разработчика программ, хотя и представляется на первых порах не очень удобным. Редактор предполагает работу в текстовом режиме, хотя у него есть графический клон gvim, который позволяет работать с мышью. Ниже будет приведен самый необходимый минимум команд. Редактор снабжен прекрасным on-line справочником, по которому можно ознакомиться со всеми возможностями.

Основная особенность редактора заключается в том, что он может находиться в двух режимах: командном и экранном. После загрузки редактора командой vi для входа в экранный режим нажимаем клавишу ‘i’. После этого можно вводить текст стандартным образом. Для выхода из экранного режима нажимаем клавишу ‘esc’, и редактор переходит в командный режим. Для сохранения содержимого экрана в файле набираем команду:

`:w` имя_файла

Отметим, что двоеточие является элементом команды. Для загрузки файла в редактор используется команда

`:e` имя_файла.

Редактор содержит несколько буферов, поэтому возможно редактирования сразу нескольких файлов. Имеется буфер обмена. Для удаления участка текста вводим команду:

`:n1,n2d.`

Здесь n1 и n2 первая и последняя строки участка текста, подлежащего удалению. Весь фрагмент помещается в буфер обмена. Для вставки содержимого буфера обмена после позиции курсора нажимаем на клавишу ‘p’. Современный стиль программирования предполагает использование длинных идентификаторов переменных и функций. Для упрощения работы с такими идентификаторами редактор имеет две полезные команды автодополнения. В экранном режиме печатается начало идентификатора, после чего нажимаем Ctrl P или Ctrl N в зависимости от того, находится нужный идентификатор перед позицией курсора или после нее. В результате начальное имя идентификатора дополняется до полного. Имеются также команды поиска и замены. Кроме того, vi может обрабатывать команды из командного файла для обработки всего файла, однако в последнее время для этих целей используют специальные средства вроде Perl или Python. Не выходя из редактора, можно выполнить команду интерпретатора, напечатав:

`:!команда`

Для выхода из редактора вводят команду:

`:q` или `:q!`, если сделанные изменения не нужно сохранять. Утилита ctags создает специальный файл меток, содержимым которого пользуется редактор. С помощью этих меток можно быстро переходить от одного фрагмента программы к другому.

Упражнение 1.7

Опробовать упомянутые команды редактора vi. Нажав F1 в режиме редактора, ознакомиться с другими возможностями редактора.

Лабораторная работа 2 (продолжение работы 1). Переменные окружения. Команды пользователя. Редактор Vi

2.1. Переменные окружения

Все сказанное ниже относится к случаю, когда SHELL=tcsh.

Каждый пользователь имеет набор переменных окружения, сгенерированных при установке системы. Значения этих переменных можно увидеть, набрав команду env. Используя файл .cshrc можно поменять значения этих переменных, а также ввести новые переменные. Новая переменная вводится командой setenv ИМЯ Значение. Значения из .cshrc активизируются при входе в систему, а также при выполнении команды source .cshrc. Значение любой переменной можно увидеть, выполнив команду echo $имя_переменной.

Упражнение 2.1

Просмотреть опции команды env. Выполните команду env. С помощью текстового редактора ввести новую переменную в .cshrc, активизировать файл и просмотреть значение введенной переменной.

2.2. Команды пользователя

Пользователь может создавать собственные команды, комбинируя стандартные команды либо вводя в них новые параметры. Для этого в .cshrc вставляется команда alias ИМЯ_КОМАНДЫ фактическая команда. Если фактическая команда содержит параметры, то фактическая команда вместе с параметрами иногда заключается в апострофы (все зависит от версии shell).

Упражнение 2.2

Создать файл командой echo “Some”>some. Скопировать его, выполнив команду cp some some1. Выполнить команду /bin/rm some, после чего выполнить команду rm some1.

Открыть с помощью текстового редактора файл .cshrc и объяснить различие в выполнении команд. Снова создать файл some и выполнить команду env –i rm some. Объяснить результат.

2.3. Компиляция и запуск программы

В простейшем случае компиляция программы, написанной на C, и создание исполняемого файла осуществляется одной командой:

```
gcc  -o имя_исполняемого_файла имя_исходного_файла
```

При успешном завершении операции запуск программы выполняется командой:

```
./имя_исполняемого_файла
```

То же самое можно выполнить, не выходя из Vi. Это делается командой:

```
:!команда
```

Упражнение 8

Напишите программу «`Hello, world`» и выполните ее.

2.4. Утилита make

Эта утилита предназначена для создания проектов. Ее особенность заключается в выполнении лишь тех операций, которые необходимы (если файл не менялся со времени предыдущего обновления, его компиляция не производится). В данной лабораторной работе рассматриваются лишь простейшие команды на неформальном уровне. Для более полного ознакомления следует обратиться к On-line документации (info make) или специальным книгам.

Утилита вызывается командой make.[options] [name_of_goal] Ее работа основана на обработке специального файла. Такой файл должен быть единственным в каждой директории. По умолчанию, он называется Makefile или makefile. Возможно использование файла с произвольным именем, но в этом случае вызов осуществляется командой `make –f имя_файла`.

2.5. Структура Makefile’a

Формат файла должен удовлетворять специальным требованиям. Для его создания больше всего подходит редактор Vi. Основная идея обработки заключается в описании цели и команд, с помощью которых цель достигается. Указывается зависимость цели от используемых файлов, хотя этот список может быть пустым. Целей может быть несколько. Нужная цель указываются в явной форме при вызове (make имя_цели), либо по умолчанию используется первая цель в списке целей. Цель описывается в следующей форме:

Имя_цели: список файлов, от которых зависит целевой файл.

Команда необходимая для создания цели.

Команда всегда начинается с символа TAB !. Пример описания цели:

```
hello:
     gcc –o hello hello.c
```

Здесь список зависимостей пустой. Если проект содержит два файла, hello.c, mdata. и пустой файл My.h

Файл hello.c

```
#include <stdio.h>
#include “My.h”
char* mdate();
int main(){
printf(“\n Hello on %s\n”,mdate());
return 0;
}
```

Файл mdata.c

```
#include <time.h>
char* mdate(){
time_t tm;
time(&tm);
return asctime(localtime(&tm));
}
```

Для создания проекта используют makefile:

```
hello: hello.o mdate.o
gcc –o hello hello.o mdate.o
%o:%.c
gcc –c $<
```

Строки 2, 4 начинаются с символа TAB. Строки 3, 4 определяют правило для создания объектных файлов. Здесь использована автоматическая переменная $<, заменяющая первый элемент в строке зависимостей.

Следующий шаг заключается во введении переменных и использовании других автоматический переменных:

```
PROG1=hello
PROG2=mdate
Hello: $(PROG1).o $(PROG2).o
     gcc –o $@ $^
%c:%o
     gcc –c $<
$(PROG1).o: My.h
clean
     rm Hello
     rm *.o
```

В этом примере использованы еще две автоматические переменные: $@ - имя цели, $^ список зависимостей. Для того, чтобы в явной форме указать зависимость от My.h, помещаем дополнительную строку. Команда make clean удаляет все созданные файлы, сохраняя только исходные тексты.

Замечание Типичная ошибка при использовании Makefile вставка пробелов после имени программ. Используйте vi для удаления пробелов.

Лабораторная работа 3. Продвинутые команды редактора Vi. Компиляция и выполнение простейшей программы.

3.1. Продвинутые команды редактора Vi

В процессе компиляции при обнаружении ошибок компилятор указывает номер строки программы, содержащей ошибку. Для перехода к нужной строке в командном режиме набираем номер_строки shift G. Для перехода в начало файла набираем 1shift G, для перехода в конец файла shift G (hift G это нажатие клавиши G на верхнем регистре). Для удаления блока строк ставим курсор на первую строку блока и определяем номер строки (он высвечивается в нижней строке экрана). Ставим курсор на последнюю строку блока и набираем команду :dномер_первой_стоки_блока,. Для вставки удаленного блока ставим курсор на нужную строку и вводим p shift p (в зависимости от того, где должен помещаться блок, после строки или перед ней).

Упражнение 3.1

Введите текст и переместите блок указанным образом.

Редактор запоминает перемещение по тексту. Командами ctrl O ctrl I перемещаемся по строкам редактирования программы.

Упражнение 3.2

Опробуйте указанные команды на каком либо текстовом файле.

Редактор позволяет просматривать бинарные файлы. Для этого открываем бинарный файл, как обычный и выполняем команду

В предыдущем примере команда начиналась с ‘!’ . Это означает, что фактически выполнялась команда shell’a. Таким образом, не выходя из редактора можно выполнить любую команду.

Упражнение 3.3

В командном режиме редактора наберите команду :!echo “Hello everybody”>hello.txt. Затем наберите команду :e hello.txt. После этого :!rm hello.txt.

3.2. Компиляция и выполнение простейшей программы

Компиляция и связывание производится одной командой gcc. Эта команда имеет огромное количество опций, но настроена таким образом, что для обычной работы достаточно опций, установленных по умолчанию, и очень ограниченное число опций

устанавливается пользователем. Стандартная команда для создания исполняемого файла выглядит так:

```
gcc –o name_exec name1.c –lname_lib
```

Здесь name_exec имя исполняемого файла, name1.c имя файла с текстом. Опция –lname_lib добавляется в случае, если нужна специальная библиотека. В частности, если нужна библиотека математических функция, добавляется –lm в командную строку. Если нужна только компиляция, используется команда:

```
gcc –c  name1.c
```

В списке С файлов могут присутствовать несколько файлов, но только один из них содержит функцию main().

Упражнение 3.4

Ниже приведена простая программа, содержащая ошибку. Исправьте ошибку, затем откомпилируйте ее и вызовите со своим именем.

```
/*
   MyFirst.c
   For compilation print
   gcc -o Myfirst MyFirst.c
   then
   ./MyFirst Name
*/

#include <stdio.h>
int main(int argc, char argv){
      if(argc<2)
            printf("\n Hello world\n");
      else
            printf("\n Hello %s",argv[1]);
      return 0;
}
```

Замечание. Программа дана в виде рисунка и не годится для непосредственного копирования.

Лабораторная работа № 4. Программа распределенного поиска (MPI)

4.1. Программа распределенного поиска (MPI)

Имеется программа CreateRand, которая создает файл предписанной длины. Вызов программы осуществляется командой ./CreateRand NUM, где NUM определяет длину файла. Текст программы находится на файле CreateRand.c. Программа создает файл с именем Rand.bin, содержащей случайные числа в бесформатной форме.

Основная программа находится в файле secdMPI.c .

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
int FindVal(int* buff,int val,int len){
     int retVal=-1,
         i;
     for(i=0;i<len;i++)
          if(val==buff[i]){
               retVal=i;
               break;
          }
     return retVal;
}
int main(int argc,char *argv[])
{
    int    fileLen, myid, numprocs,i,
           lengthPart,     // part for single process
           actualLength,   // actual length
           offset=0;  // start position for reading
    int         val,       // what we are looking for
           position;  // where is the value
    FILE*  fpIn;
    int*   buff;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
//    All process open source file
    fpIn=fopen("Rand.bin","r");
//     Zero process calculates length of file and length of parts and
sends beginning for each part
    if(myid==0){
         int off;
         printf("\n Enter value\n");
         scanf("%d",&val);
         fseek(fpIn,0,SEEK_END);
         fileLen=ftell(fpIn)/4;
         fseek(fpIn,0,SEEK_SET);
         lengthPart=fileLen/numprocs;
         for(i=1;i<numprocs;i++){
              off=4*i*lengthPart;
              MPI_Send(&off,1,MPI_INT,i,0,MPI_COMM_WORLD);
         }

    }
```

```
    MPI_Bcast(&val,1,MPI_INT,0,MPI_COMM_WORLD);
    MPI_Bcast(&lengthPart,1,MPI_INT,0,MPI_COMM_WORLD);
// Each slave allocates memory and reads its part of the file
    buff=(int*)malloc(4*lengthPart);
    if(myid ==0){
         MPI_Status  sts;
         actualLength=fread(buff,4,lengthPart,fpIn);
         position=FindVal(buff,val,actualLength);
         if(position!=-1){
              printf("\n Found position %d \n",position);
              MPI_Abort(MPI_COMM_WORLD,0);
         }
         else{
              for(i=1;i<numprocs;i++){

MPI_Recv(&position,1,MPI_INT,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&sts);
                   if(position!=-1){
                    printf("\n Found position %d \n",position);
                    MPI_Abort(MPI_COMM_WORLD,0);
                    }
              }
              printf("\n Can't find the value\n");
         }
    }
    else{

         MPI_Status sts;
         MPI_Recv(&offset,1,MPI_INT,0,0,MPI_COMM_WORLD,&sts);
         fseek(fpIn,offset,SEEK_SET);
         actualLength=fread(buff,4,lengthPart,fpIn);
         position=FindVal(buff,val,actualLength);
         if(position!=-1)
              position +=offset/4;
         MPI_Send(&position,1,MPI_INT,0,1,MPI_COMM_WORLD);
    }
    fclose(fpIn);
         free(buff);
    MPI_Finalize();
    return 0;
}
```

4.2. Структура программы

Программа содержит два модуля: int FindVal(buff,val,len), которая ищет число val в буфере buff длины len. Программа возвращает позицию числа в буфере, либо –1, если поиск не удался.

Основной модуль использует интерфейс MPI.

Все процессы открывают файл Rand.bin.

Нулевой процесс читает файл, находит его длину в байтах, переводит в длину в целых числах, определяет длину и смещение для каждого процесса с положительным рангом и рассылает эту информацию. Все процессы считывают свою часть файла и производят поиск.

Нулевой процесс читает свою часть файла. Если поиск успешен, он прекращает выполнение остальных процессов. В противном случае он ожидает сообщений от остальных процессов и печатает окончательный результат.

Задание

- Изучить текст программы CreateRand.c.
- Переписать программу *CreateRand.c* так, чтобы генерировались числа только из заданного интервала.
- Изучить текст программы secdMPI.c
- Переделать программу secdMPI.c так, чтобы был реализован алгоритм:

Нулевой процесс читает файл.

Определяет размеры частей и пересылает их другим процессам.

Каждый процесс печатает результат поиска. Если поиск положительный, то он завершает остальные процессы, в противном случае печатает свой номер и сообщение о неудаче.

Лабораторная работа 5. Работа с утилитой make

Утилита make предназначена для работы с большими проектами или для компиляции программ, требующих нестандартных опций. Особенностью утилиты является то, что она выполняет минимум необходимой работы: если файл входит в проект, но не менялся с момента последней компиляции, объектный файл не будет создаваться заново. Вся информация об утилите доступна с помощью команды info make.

Упражнение 5.1.

Создайте два файла, как указано ниже,

```
// File mn.c
void prn(char* str);
char str[]="It is me";
int main(){
prn(str);
return 0;
}

// File prn.c
#include <stdio.h>
void prn(char* str){
printf("\n%s\n",str);
}
```

Поместите их в одну директорию, создайте исполняемый файл с именем mn командой:

`gcc –o mn mn.c prn.c` и выполните его.

Упражнение 5.2

Создайте файл с именем makefile `mn`:

```
	gcc –o mn mn.c prn.c
clean:
	rm mn
```

Создайте исполняемый файл командой `make`. После этого выполните команду `make clean`. Обратите внимание на то, что все команды начинаются с символа tab.

Упражнение 5.3

Создайте файл с именем makefile:

```
src = mn.c prn.c
mn: $(src)
	gcc –o mn $(src)
```

Упражнение 5.4

Создайте файл с именем makefile:

```
obj = mn.o prn.o
mn: $(obj)
      gcc –o mn $(obj)
```

Объясните разницу в работе утилиты с этими файлами.

Переменные в makefile используют также для обозначения библиотек, подключаемых для создания исполняемого файла.

Упражнение 5.5

Создайте файл с именем makefile:

```
mn: mn.c prn.c
      gcc –o $@ $^
```

Объясните смысл автоматических переменных

Упражнение 5.6

Создайте файл с именем makefile

```
LIB = m
mn: mn.c
      gcc  -o mn –l$(LIB)
```

Напишите программу mn.c, подсчитывающую sin(1/k), k=1,..,10 и откомпилируйте ее с помощью данного файла.

Программа должна содержать строку `#include <math.h>`.

Лабораторная работа 6. Работа с длинными числами

В дистрибутив LINUX включена библиотека ssl для работы с числами произвольной длины. Для ознакомления с функциями библиотеки наберите `man bn`, после чего будут получены ссылки на дальнейший материал.

Упражнение 6.1

Наберите программу

```
#include <stdio.h>
#include <openssl/bn.h>
char str1[]="21";
char str2[]="112";
int main(){
 BIGNUM* a=BN_new();
 BIGNUM* b=BN_new();
 BIGNUM* c=BN_new();
 BN_dec2bn(&a,str1);
 BN_dec2bn(&b,str2);
 BN_add(c,a,b);
 printf("\n Result %s",BN_bn2dec(c));
 BN_print_fp(stdout,c);
 return 0;
}
```

Упражнение 6.2

Создайте makefiel, в котором предусмотрено подключение библиотеки ssl.

Откомпилируйте с его помощью предыдущую программу и запустите ее. Объясните полученный результат.

Упражнение 6.3

Напишите программу для перемножения двух длинных чисел и для деления с остатком и проверьте результат ее работы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Параллельное программирование на OpenMP. [Электронный ресурс]. — URL: http://ccfit.nsu.ru/arom/data/openmp.pdf (Дата обращения:10.09.2019).

2. Message Passing Interface (MPI). [Электронный ресурс]. — 2019 - URL: https://computing.llnl.gov/tutorials/mpi/ (Дата обращения:01.10.2019).

3. MPI Tutorial [Электронный ресурс]. — 2019 - URL: https://mpitutorial.com/tutorials/ (Дата обращения:01.10.2019).

4. Интернет-Университет Суперкомпьютерных Технологий: Параллельное программирование. [Электронный ресурс]. — 2019 - URL: https://www.intuit.ru/studies/courses/1110/153/info (Дата обращения:01.10.2019).