МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования
«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ»

Кафедра автоматизированных систем управления (АСУ)

УТВЕРЖДАЮ
Зав. кафедрой АСУ, профессор
А.М. Кориков

АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ КОМПЛЕКСОВ

Тема 3. Архитектуры вычислительных комплексов

Учебно-методическое пособие

для студентов уровня основной образовательной программы: **магистратура**
направление подготовки: **09.04.01 - Информатика и вычислительная техника**

Разработчик
доцент кафедры АСУ

В.Г. Резник

2017

Резник В.Г.

Архитектура вычислительных комплексов. Тема 3. Архитектуры вычислительных комплексов. Учебно-методическое пособие. – Томск, ТУСУР, 2017. – 64 с.

Учебно-методическое пособие предназначено для изучения темы №3 по дисциплине «Архитектура вычислительных комплексов» для студентов уровня основной образовательной программы магистратура направления подготовки: 09.04.01 «Информатика и вычислительная техника».

Оглавление

Введение

Данное методическое пособие содержит учебный материал по третьей теме дисциплины «*Архитектура вычислительных комплексов*», - сокращенно АВК.

Изложенный материал является обязательной частью процесса обучения магистранта по направлению подготовки 09.04.01 «Информатика и вычислительная техника» и содержит как теоретическую часть, так и методические указания по выполнению двух лабораторных лабораторных работ.

Последовательность и тематическая направленность учебного материала данного пособия предполагает, что магистрант успешно освоил теоретический материал по первой и второй темам дисциплины, а также выполнил первые шесть лабораторных работ.

Изложенный материал разбит на ряд разделов, последовательность которых определяет сам порядок процесса обучения.

В первом разделе представлен торетический материал по объявленной теме «Архитектура вычислительных комплексов», который рассчитан на 2 академических часа и двух лабораторных работ по 8 академических часов каждая, с помошью которых закрепляется теоретический материал.

Второй и третий разделы содержат методические описания самих лабораторных работ, выполняемых в рамках данной темы, пронумерованных в общем порядке и озаглавленных:

- лабораторная работа №7 - «*Технология OpenMP*»;
- лабораторная работа №8 - «*Технология MPI*».

Замечание

Общая технология проведения всех работ, предполагает, что каждый студент должен иметь flash-память не менее 2 Гб для сохранения личного материала и запуска УПК АСУ.

1 Тема 3. Архитектуры вычислительных комплексов

К концу 60-х годов, накопилось множество разработок с различными архитектурными решениями в области разработки новых вычислительных систем.
Соответственно, назрела необходимость в их классификации.

Гарвардская архитектура — архитектура ЭВМ, разработаная *Говардом Эйкеном* **в конце 1930-х** годов в Гарвардском университете.
Отличительными признаками этой архитектуры являются:

- *хранилище инструкций* и *хранилище данных* представляют собой разные физические устройства;
- *канал инструкций* и *канал данных* так же физически разделены.

Архитектура фон Неймана — широко известный *принцип совместного хранения* команд и данных *в общей памяти компьютера*.
Основы учения о такой архитектуре вычислительных машин заложил *фон Нейман* ***в 1944 году***, когда подключился к созданию первого в мире лампового компьютера ЭНИАК.

В результате ряда проведенных исследований, очень популярной стала ***Классификация ВК*** по соотношению *потока команд* и *потока данных*.

1.1 Основы многопоточной (мультитредовой) архитектуры

Научное сообщество и сообщество разработчиков ЭВМ признало классификацию, предложенную ***в 1970 годах*** *Г. Флинном*.
Классификационным признаком этой группировки вычислительных систем является *соотношение* между *потоком команд* и *потоком данных*.

По этому признаку выделяют **4 группы ВС**:

1) **ОКОД** - с одним потоком команд и одним потоком данных;
2) **ОКМД** - с одним потоком команд и множеством данных;
3) **МКОД** - с множеством команд и одним потоком данных;
4) **МКМД** - с множеством команд и множеством данных;

Соответствующая англоязычная система классификации:

1) *SISD* = Single Instruction Single Data
2) *MISD* = Multiple Instruction Single Data
3) *SIMD* = Single Instruction Multiple Data
4) *MIMD* = Multiple Instruction Multiple Data

Если обозначить:

ПД – поток данных;
ПК – поток команд;

П-1, П-2,……П-N –процессоры или процессорные элементы;
ЦУУ ВС – центральное устройство управления ВС.

Тогда, ***общие архитектуры*** такой классификации можно представить рисунками 1.1 а-г:

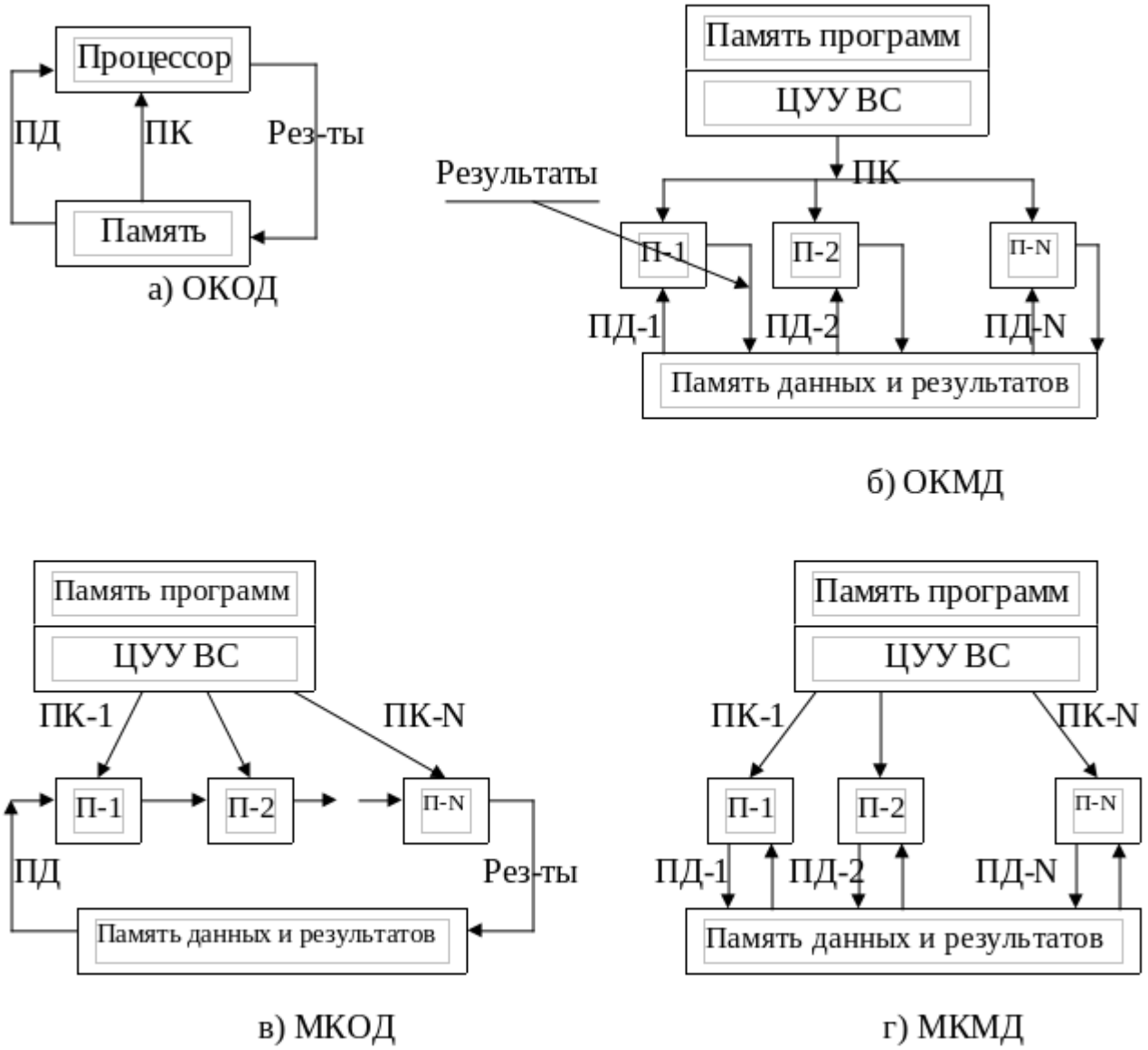

Рисунок 1.1 — Классификация архитектур вычислительных комплексов

К первой группе относятся *традиционные* или *классические ЭВМ* и построенные на их основе вычислительные системы.
На архитектуре типа ОКОД возможно *совместное решение нескольких задач*.
Такой режим работы системы называется *мультипрограммным режимом*:

программы и данные совместно решаемых задач хранятся в оперативной памяти, где всем программам *выделяются свои сегменты* (разделение оборудования);

время работы процессора разделено на небольшие периоды (*такты*), в

течение которых он выполняет команды для одной программы;

когда такт заканчивается, происходит прерывание, и передача управления операционной системе (*программе-супервизору*);

супервизор просматривает очередь задач и определяет, есть ли в очереди *задачи с более высоким или таким же приоритетом*, что и у прерванной программы.

если есть, то в следующий такт процессор выполняет команды другой программы,

если нет, то продолжает выполнение прерванной программы — *разделение времени процессора*.

ВС типа ОКМД иначе называют еще *системами с общим потоком команд*.

В разных процессорах выполняются *одни и те же команды* над *разными данными*.

Реализуется *синхронный параллельный вычислительный процесс*:

все данные для очередной команды, одновременно подаются на обработку процессору;

одновременно из процессора все данные после обработки передаются в память ЭВМ.

Применяются такие ВС для решения задач:

одномерное и двумерное прямое и обратное *преобразование Фурье*,

решение дифференциальных уравнений в частных производных,

выполнение операций над матрицами и векторами.

ВС типа МКОД реализуют *принцип конвейерной обработки одной команды* (функции):

в команде *выделены несколько операций* (подфункций);

операции *последовательно выполняются* на отдельных аппаратных блоках;

результат операции *записывается в память*.

В ВС типа МКМД реализуется *асинхронный параллельный принцип обработки данных*:

каждый процессор выполняет свою программу или участок (ветвь) одной большой программы над отдельными данными.

имеется в наличии *нескольких групп многоканальных связей* между аппаратными модулями; к ним относится *общая память*, каналы ввода-вывода (*КВВ*) и *процессоры*.

Многоканальные связи бывают:

постоянными или физическими *связями*;

логическими связями, которые устанавливаются по мере необходимости в ходе вычислительного процесса.

За счет многоканальных связей обеспечиваются *следующие условия работы*:

любой процессор может управлять передачей данных **к/от** любого *сегмента общей памяти*;

любой процессор может направлять команды к любому *КаналуВВ*;

любой КВВ может передавать данные **к/от** любого сегмента *общей памяти*;
любой КВВ может управлять передачей данных *между ОП и любым устройством* ввода-вывода.

1.2 Сравнение параллельной и конвейерной организации ВК

Параллелизм и **конвейеризация** имеют одинаковые цели – *повышение производительности ВК*.
Оба подхода предполагают достижение этой цели за счет “размножения” аппаратных средств (*избыточности аппаратуры ВК*).

Организация вычислительного процесса в обоих этих подходах сильно различается.
На приведенной ниже таблице 1.1 отражены наиболее существенные различия между этими подходами.

Таблица 1.1 – Сравнение параллельной и конвейерной организации ЭВМ

Наименование параметра	**Организация**	
	параллельная	***конвейерная***
Базовая структура	Независимое исполнение подзадач на отдельных блоках аппаратуры.	Разбиение функции на N подфункций.
Производительность	N результатов за каждые T секунд.	Один результат за каждые T/N секунд.
Основной период синхронизации	Время для вычисления одной функции.	Время для одной ступени (выполнение подфункции).
Типичная архитектура	ОКМД, МКМД	ОКОД, МКОД
Предпочтительная структура задачи	Матричные задачи с длинами векторов, кратными числу процессоров; процессы, поддающиеся разбиению на независимые части.	Предпочтительны одномерные векторы с произвольно большой длиной; ускорение выполнения традиционных наборов команд.
Типичная организация памяти	Многократно повторенные независимые модули памяти.	Одна многократно расслоенная память.

Особенности управления	Осуществляется пользователем	Во многом осуществляется аппаратурой.
Факторы, ограничивающие производительность	Стоимость, структура задач.	Элементная база, скорость доступа к памяти.
Обеспечение надежности	Легко достижима за счет “горячего” резерва.	Обходится дорого за счет немодульной организации.

О различиях базовых структур, производительности, предпочтительных задачах, особенностях программирования задач в параллельных ВК уже говорилось и дополнительных пояснений по этим пунктам не требуется.

Возможны вопросы по ограничивающим факторам:

- *какова стоимость* для параллельных ВК;
- *какова надежность* для параллельных ВК.

Значительная стоимость МКМД вызвана наличием *многоканальных связей между модулями*, которые обеспечивают гибкость вычислительного процесса.
Реализуются эти связи путем дополнения традиционной аппаратуры ВК *коммутаторами различной степени сложности*.
Эти дополнительные аппаратные средства и приводят к значительному удорожанию параллельных ВК, даже при наличии коммутаторов невысокой степени сложности.

Высокая степень обеспечения надежности, во многих параллельных **МКМД**, обусловлена тем, что можно, *за небольшую стоимость*, добавить дополнительные копии процессоров, которые при стандартных ситуациях в вычислительном процессе не будут задействованы, однако при выходе из строя какого-либо из основных процессоров, резервный процессор может быть подключен к системе.

Замечание

Такого простого решения *нельзя обеспечить в конвейерных ВК*, поскольку ступени конвейера различны между собой.
В общем случае, в конвейерных ВК, для каждой ступени разработчики ищут свой метод обеспечения надежности и он (метод) должен быть реализован на этапе проектирования такой системы.

1.3 SMP-архитектура

SMP (*symmetric multiprocessing*) – симметричная многопроцессорная архитектура. *Главной особенностью* систем SMP является *наличие общей физической памяти*, разделяемой всеми процессорами, что отображено на рисунке 1.2.

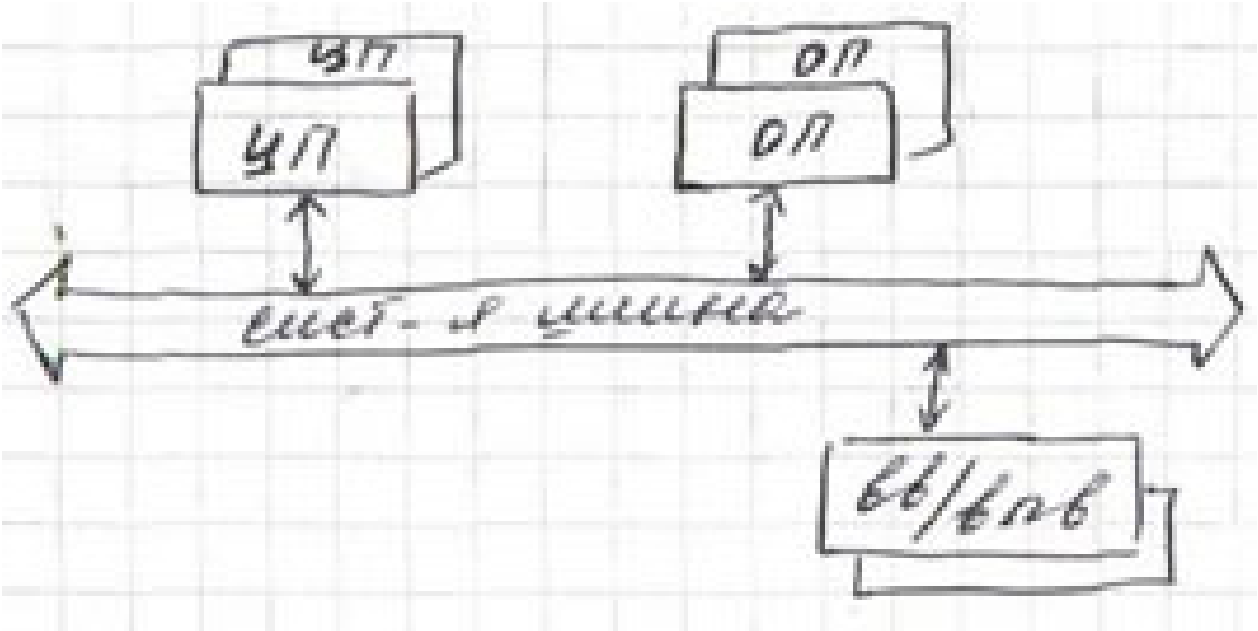

Рисунок 1.2 — Общая архитектура SMP

SMP - это ***один компьютер*** с несколькими *равноправными процессорами*.
Все остальное - в одном экземпляре:

1) одна память,
2) одна подсистема ввода/вывода,
3) одна операционная система.

Слово "равноправный", как и слово '*симметричная*', в названии архитектуры, означает, что *каждый процессор может делать все, что любой другой*.

Каждый процессор имеет:

1) доступ ко всей памяти;
2) может выполнять любую операцию ввода/вывода;
3) может прерывать другие процессоры и т.д.

Но это представление справедливо только на уровне программного обеспечения. *Умалчивается то*, что на самом деле, в SMP *имеется несколько устройств памяти*.

В традиционной SMP-архитектуре, *связи между кэшами ЦП* и *глобальной памятью* реализуются с помощью *общей шины памяти*, разделяемой между различными процессорами.

Как правило, эта шина становится *слабым местом конструкции системы* и стремится к насыщению при увеличении числа инсталлированных процессоров.
Это происходит потому, что *увеличивается трафик пересылок* между кэшами и памятью, а также между кэшами разных процессоров, которые конкурируют между собой за пропускную способность шины памяти.
При рабочей нагрузке, характеризующейся интенсивной обработкой транзакций, эта проблема является даже еще более острой.

В SMP, оперативная память (**ОП**) физически представляет *последовательное адресное пространство*, доступ к которому имеют одновременно все процессоры системы по единой коммуникационной среде:

- *либо шинной* архитектуры;
- *либо коммутатором* типа **crossbar**.

Основные достоинства технологии:

1) *Простота организации вычислительного процесса*, так как все процессоры обращаются к единой памяти по одному алгоритму.
2) *Эффективность организации программного кода задачи*, которая обеспечивается системным программным обеспечением, так как в процессе генерации кода нет необходимости учитывать разнообразие размещения данных в ОП.
3) *Проверенное, большим сроком эксплуатации, программно-аппаратное решение*, реализованное основными производителями вычислительных систем.

Недостатки технологии:

1) *Единый путь доступа к ОП*, который становится узким местом, при увеличении числа процессоров в системе. Попытка технологически решить эту проблему лишь отодвигает граничный трафик. Так архитектура с синхронной шиной доступа позволяла линейно увеличивать производительность системы в пределах до 8-ми процессоров. *Пакетная организация системной шины*, уменьшая количество взаимных блокировок, позволяет довести количество процессоров в системе до 16-ти. *Технология crossbar*, т.е. когда элементы вычислительной системы коммутируются напрямую друг с другом *по протоколу точка-точка*, позволила довести количество процессоров до 72-х. Однако, с увеличением количества коммутируемых элементов системы происходит резкий рост сложности ***crossbar*** и, как следствие, рост цены устройства.
2) *усложнение логической части ВС*, которая отвечает за работу с кэшем, в частности за когерентность (целостность), что также влияет на производительность и цену системы.

Примеры компьютеров с SMP архитектурой:

HP 9000 (до 32 процессоров),
Sun HPC 100000 (до 64 проц.),
Compaq AlphaServer (до 32 проц.)

1.4 MPP-архитектура

MPP - *massive parallel processing* – массивно-параллельная архитектура, показанная на рисунке 1.3.

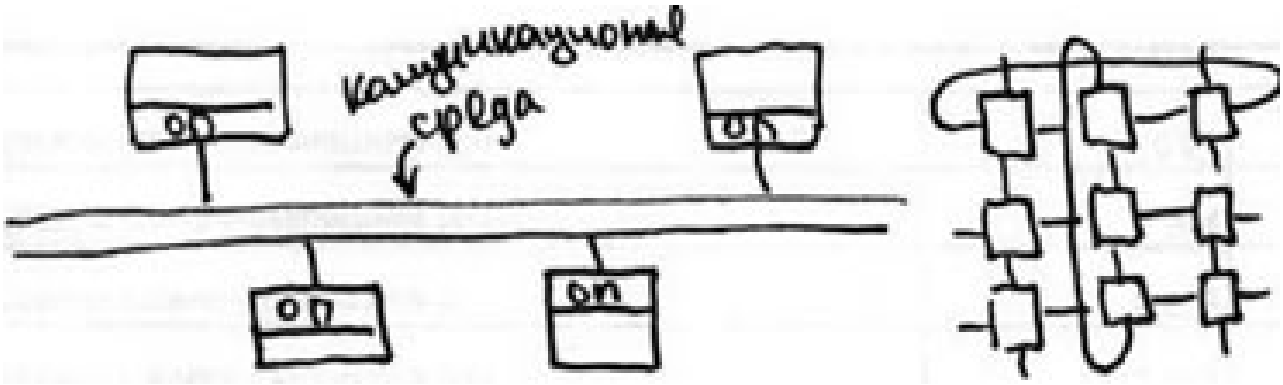

Рисунок 1.3 — Массивно-паралльельная архитектура MPP

В основе такого ВК лежит *транспьютер* – мощный универсальный процессор, особенность которого - *наличие 4 линков* (коммуникационные каналы связи).
Каждый линк состоит *из двух частей*, служащих для передачи информации в противоположных направлениях, и используется:

- *для соединения* транспьютеров между собой;
- *для подключения* внешних устройств.

Особенности такой системы:

- *Процессоры* обмениваются между собой данными.
- *После передачи байта данных*, пославший их транспьютер *ожидает получения подтверждающего сигнала*, указывающего на то, что принимающий транспьютер готов к дальнейшему приему информации.
- *Большая прикладная задача* разбивается на *процессы*, распределяемые на каждый отдельный процессор (*транспьютер*).

MPP система начинается со 128 процессоров.
Если число процессоров < 64, то это точно не MPP:

- хотя тоже оборудование;
- тот же компилятор.

Сообщения пересылаются через ряд процессоров.
Нет узкого «горлышка», как у SMP-архитектуры.

1.5 MPP-система Paragon

Система Paragon — это конкретная реализация MPP, созданная корпорацией *Intel* и показанная на рисунке 1.4.
Таких систем было выпущено несколько сотен, причем каждая из них была не похожа на другую по *количеству процессоров*, *размеру оперативной памяти* и другим характеристикам.

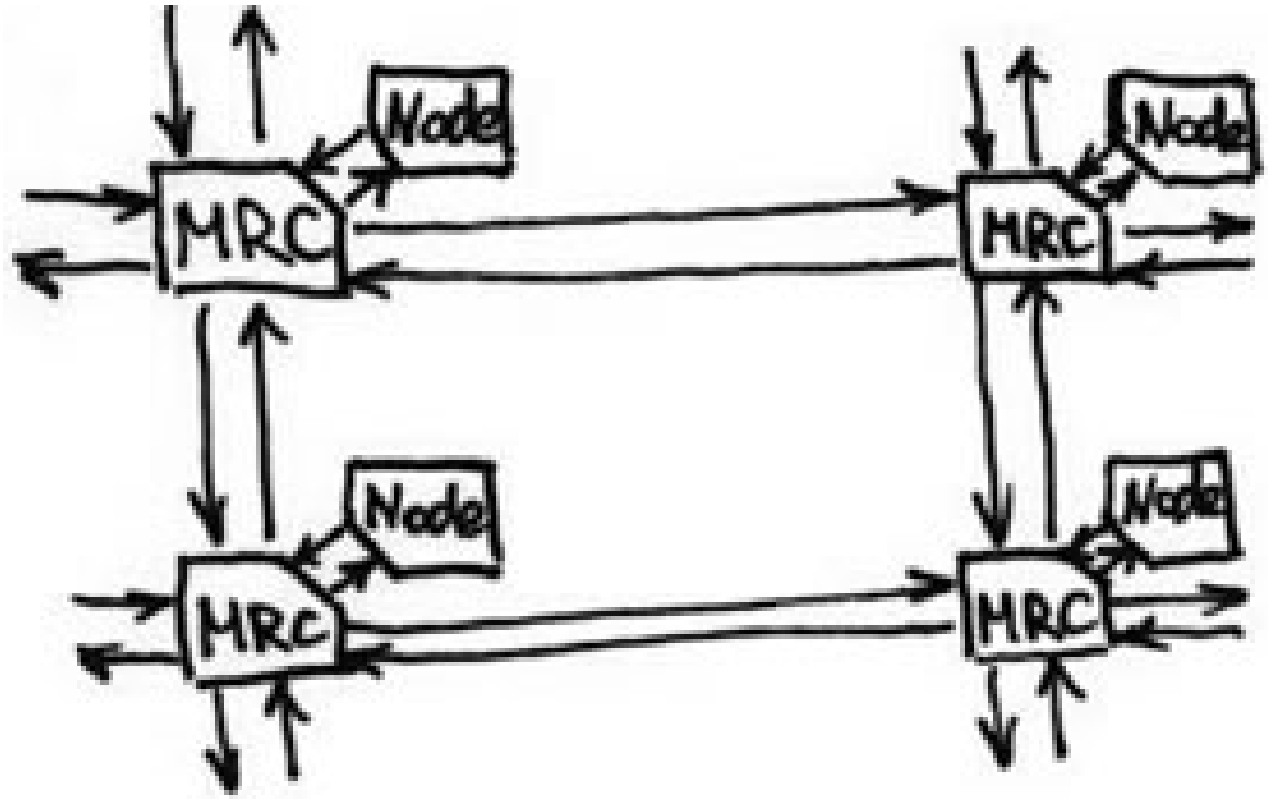

Рисунок 1.4 — Общая архитектура системы Paragon

Для реализации использовались:

- *маршрутизаторы* **MRC (*Mobile Router and Control*)**;
- *процессорные узлы* **Node**, на основе процессора ***i860***.

MRC - *маршрутизатор* – набор портов:
порты могут связываться между собой,
к каждому маршруту, может подключиться компьютер.

Node – *процессные узлы* трех типов:

1) *вычислительные*;
2) *сервисные* - возможности ОС UNIX, которые предназначены для разработки программ (узлы для взаимодействия программиста);
3) *узлы в/в* - могут подключаться либо к общим ресурсам (дисковым), либо через них реализуется интерфейс с другими сетями.

Число процессоров для Paragon достигало *5000-8000*.

Общая схема процессорного ядра представлена на рисунке 1.5.

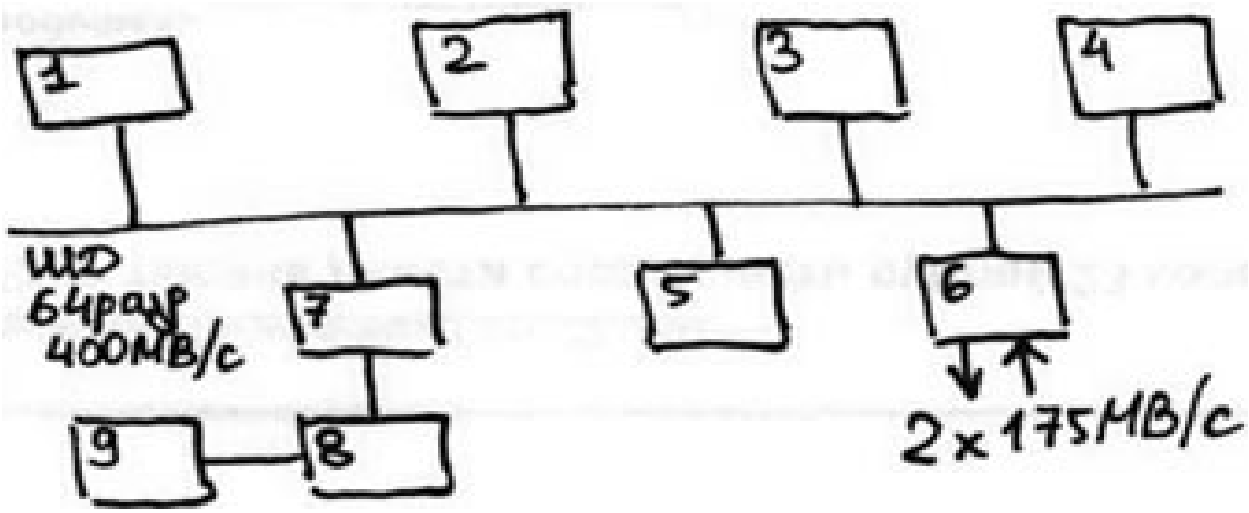

Рисунок 1.5 — Общая схема процессорного ядра системы Paragon

Схема процессорного ядра содержит:

1. *Исполнительный монитор*, позволяющий отлаживать, конролировать и записывать работу узла.
2. *Процессор приложений*.
3. *ОП* (32-64Mb);
4. *Машины передачи данных* (2 шт): Одна - на прием; другая - на передачу.
5. *Процессор сообщений* (i860).
6. *Контроллер сетевого интерфейса* (порты, которые выходят на MRC) .
7. *Порт расширения*, к которому через интерфейсные карты могли подключаться: Интерфейс в/в, либо ЛВС либо ЖД.

Примерами MPP систем можно назвать:

- IBM RS/6000 SP;
- NCR WorldMark 5100M (до 128 узлов, 4096 процессоров).

1.6 Кластерная архитектура

Кластерные системы представляют собой некоторое число недорогих рабочих станций или персональных компьютеров, объединенных в общую вычислительную сеть (*подобно массивно-параллельным системам*).
Причина возникновения – необходимость выполнения работ на нескольких компьютерах.

Термин "кластер" был введен в обиход *компанией DEC*.
Затем, этот подход удостоился *общепринятого названия*.
Сегодня, кластерная архитектура является козырной картой практически каждого поставщика компьютерных систем, ориентированных на применение *ОС UNIX, Novell Netware или Windows NT*.

Первоначальные общие кластерные схемы такой архитектуры представлены на рисунке 2.5.

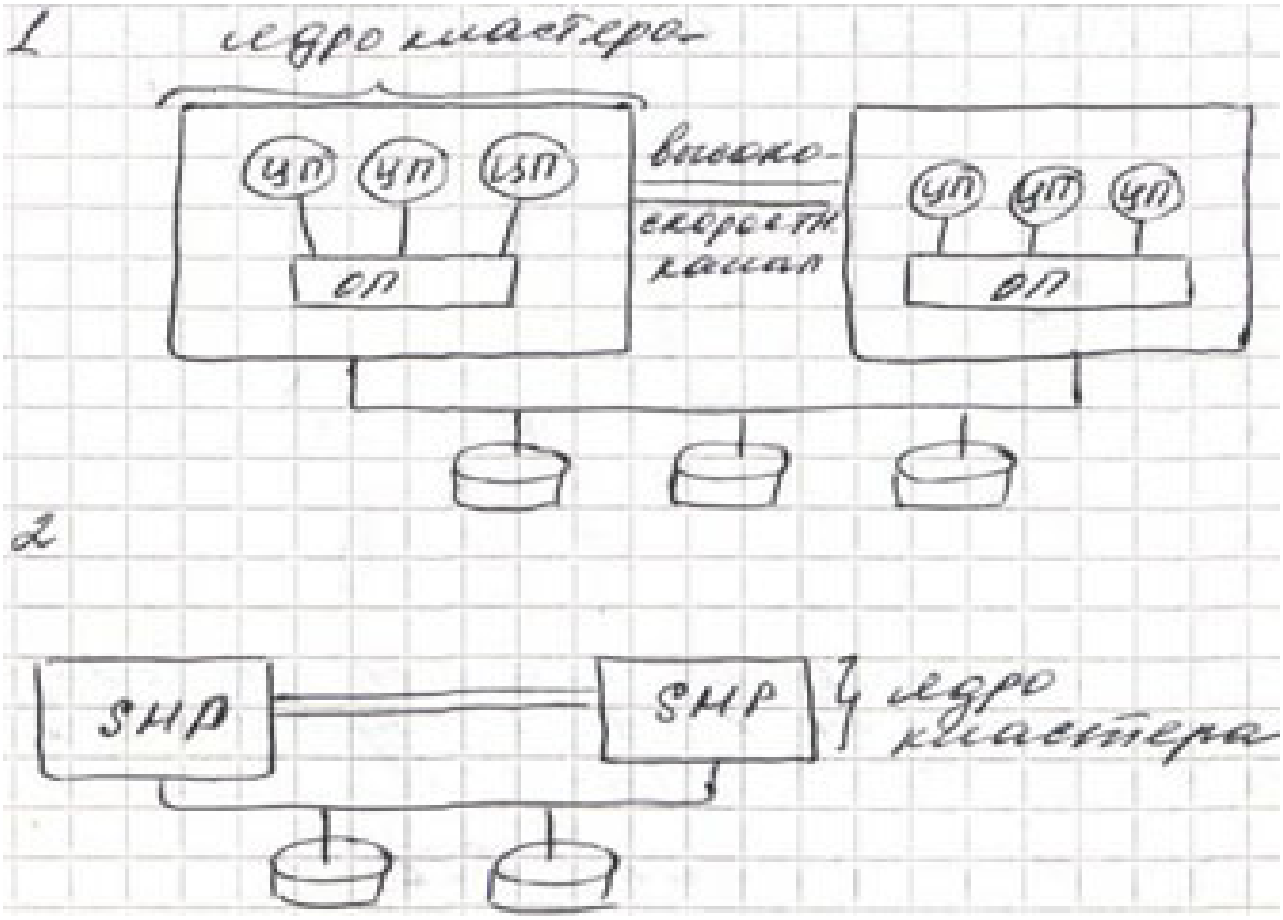

Рисунок 1.6 — Кластерные архитектуры

Кластер - это связанный *набор полноценных компьютеров*, используемый в качестве *единого ресурса*.
Под словосочетанием "*полноценный компьютер*" понимается завершенная компьютерная система, обладающая всем, что требуется для ее функционирования,

включая:

1) *процессоры*, память, подсистему ввода/вывода,
2) а также *операционную систему*, подсистемы, приложения и другое.

Обычно, для этого годятся готовые компьютеры, которые могут обладать архитектурой SMP.

Словосочетание "*единый ресурс*" означает наличие программного обеспечения, дающего возможность пользователям, администраторам и даже приложениям считать, что имеется только одна сущность - *кластер*.
У ведущих поставщиков систем баз данных имеются версии, работающие в параллельном режиме на нескольких машинах кластера.
В результате, приложения, использующие базу данных, не должны заботиться о том, где выполняется их работа.
СУБД отвечает за синхронизацию параллельно выполняемых действий и поддержание целостности базы данных.

Кластеры демонстрируют *высокий уровень доступности*, поскольку в них отсутствуют единая операционная система и совместно используемая память с обеспечением когерентности кэшей.
Кроме того, специальное программное обеспечение в каждом узле постоянно производит контроль работоспособности всех остальных узлов.
Этот контроль основан на периодической рассылке каждым узлом сигнала "*Я еще бодрствую*".
Если такой сигнал от некоторого узла не поступает, то *этот узел считается вышедшим из строя*; ему не дается возможность выполнять ввод/вывод, его диски и другие ресурсы переназначаются другим узлам (включая IP-адреса), а программы, выполнявшиеся в вышедшем из строя узле, перезапускаются в других узлах.

Возможность *практически неограниченного наращивания числа узлов* и *отсутствие единой операционной системы* делают кластерные архитектуры исключительно *хорошо масштабируемыми*.
Успешно используются массивно параллельные системы с сотнями и тысячами узлов.

Примером кластерного решения можно назвать системы: *Compaq AlphaServer* на базе своих серверов AlphaServer ES40.

Замечание

Развитие сетевых технологий расширило кластерные архитектуры, подразумевая компьютеры, объединенные локальной сетью.

2 Лабораторная работа №7

Рассматривая архитектуры различных вычислительных комплексов, нетрудно прийти к выводу, что общая тенденция создания таких архитектур направлена на распараллеливание вычислений, при большом желании эффективного использования аппаратных средств ЭВМ.
Поскольку современный уровень вычислительной техники является микропроцессорным, то существенную роль в создании вычислительных комплексов начинает играть специальное системное программное обеспечение, которое призвано облегчить и ускорить создание новых архитектур высокопроизводительных систем.

В лабораторных работах предыдущей темы было показано, что применение потоков (нитей, threads) позволяет, в ряде случаев, как упростить, так и повысить качество решения системных задач.
Следует ожидать, что расширение программного арсенала технологий, ориентированных на параллельные вычисления, должно привести к возможности постоения более совершенных систем.
Среди такого арсенала технологий следует отметить программные проекты ***OpenMP***, ***MPI*** и ***PVM***, которые мы и рассмотрим, прежде чем преходить к конкретной формулировке тем лабораторных работ.

2.1 Технологии параллельных вычислений

Сразу следует оговориться, что все рассматриваемые три проекта относятся к технологиям паралллельного программирования, поэтому могут оказаться наиболее эффективными и полезными при построении *вычислительных систем*.
С другой стороны, эти проекты простроены, или влючают в себя, технологии нитей (легковесных процессов), что:

- *дает основание* проводить параллели со стандартной технологией ***Ptheads***;
- *позволяет* более точно учитывать многопроцессорность компьютера, наличие у него ***SMP***-архитектуры ОС, а также сетевые возможности данных проектов.

2.1.1 Технология OpenMP

OpenMP (*Open Multi-Processing*) — открытый стандарт для распараллеливания программ на языках программирования ***C/C++*** и ***Fortran***.
Дает описание совокупности директив компилятору языка, которые позволяют программировать *многопоточные приложения* на *многопроцессорных системах* с архитектурой *общей памяти*.

Стандарт OpenMP был разработан ***в 1997 году***, как ***API***, ориентированный на написание портируемых многопоточных приложений для языка ***Fortran***.
В октябре 1998 года, появились спецификации ***OpenMP*** для языка ***C***.

Синтаксис *OpenMP* использует модель «*ветвление-слияние*» (***fork-join***), что хорошо подходит для распараллеливания циклов, в которых, как правило, выполняются однотипные операции.
Такая модель очень хорошо реализуется на аппаратной платформе ***SMP***-архитектур, а также позволяет воспользоваться преимуществами потоковой модели ***SIMD***.

Семантика *OpenMP* основана на понятии *нити* (***потока***, ***threads***), что позволяет сравнивать ее со стандарной моделью ***Ptreads*** стандарта ***POSIX***:

- *обе модели* используют понятие главной (***master***) нити, которая присутствует всегда, причем ***OpenMP*** нумерует нити целыми числами, начиная с нуля, что позволяет с минимальными затратами на кодирование обрабатывать массивы данных достаточно большого размера;
- *модель Pthreads* использует для реализации программы классическую парадигму функционального подхода, что проще воспринимается разработчиками ПО на языке С, но требует непосредственного его участия на всех этапах распараллеливания и синхронизации алгоритма решения задачи;
- *модель OpenMP* опирается на множество специальных директив, перекладывающих многие аспекты распараллеливания и синхронизации алгоритма задачи «на усмотрение» компилятора, что требует от программиста лишь указания последовательности параллельных и последовательных участков кода, а многие аспекты синхронизации выполняются по умолчанию;
- *хотя обе модели* позволяют использовать произвольное количество нитей, ограниченное лишь лимитами ОС, непосредственное распределение процессоров ЭВМ на конкретные нити процессов - скрыто от разработчика ПО;
- *хотя модель OpenMP* содержит большое число директив и, детализирующих их требования опций, эффективность применения этой модели для решения задач построения вычислительных комплексов вызывает серьезные сомнения.

Для учебных целей, прагматика использования модели ***OpenMP*** демонстрируется решением одной из задач мониторинга, подробно описанной в подразделе 2.2.
В качестве базовых основ описания модели ***OpenMP*** можно воспользоваться:

- методическим пособием М.В. Васильевой и других «Параллельное программирование на основе библиотек», электронный вариант которого расположен на сайте: http://edu.chpc.ru/parallel/main.html;
- учебными материалами сайта: http://parallel.ru/tech/tech_dev/openmp.html;
- справочным материалом по операторам и директовам OpenMP, размещенном в файле *~/Документы/OpenMP-4.5-1115-CPP-web.pdf* рабочей области пользователя ***upk***.

2.1.2 Технология MPI

MPI (Message Passing Interface) - интерфейс передачи сообщений или программный интерфейс (***API***) для передачи информации, который позволяет обмениваться

сообщениями между процессами, выполняющими одну задачу.

MPI было предложено ***Уильямом Гроуппом*** и ***Эвином Ласком***, в период 1993 - 1994 годов.
В 1994 году, вышла первая версия этого проекта.

12 июня 1995 года, опубликована спецификация ***MPI 1.1***, а ее первая реализация появилась ***в 2002 году***, в которой поддерживаются следующие функции:

- передача и получение сообщений между отдельными процессами;
- коллективные взаимодействия процессов;
- взаимодействия в группах процессов;
- реализация топологий процессов.

18 июля 1997 года, опубликована спецификация ***MPI 2.0***, дополнительно поддерживающая функции:

- динамическое порождение процессов и управление процессами;
- односторонние коммуникации (***Get/Put***);
- параллельный ввод и вывод;
- расширенные коллективные операции: процессы могут выполнять коллективные операции не только внутри одного коммуникатора, но и в рамках нескольких коммуникаторов.

Версия MPI 2.1 вышла в начале сентября 2008 года.
Версия MPI 2.2 вышла 4 сентября 2009 года.
Версия MPI 3.0 вышла 21 сентября 2012 года.

Главные отличия технологии ***MPI*** от технологий ***Ptreads*** и ***OpenMP***:

- *распараллеливание* реализации программного обеспечения производится на ***уровне процессов***, а не на уровне нитей;
- *коммуникации* между параллельными компонентами проводится на уровне ***передачи сообщений***, а не на уровне общей памяти.

Для учебных целей, прагматика использования этой модели демострируется на известной реализации ***Open MPI***.
Сама демонстрация проводится посредством решения одной из задач мониторинга, подробно описанной в следующей лабораторной работе.
В качестве базовых основ описания модели ***MPI*** можно воспользоваться:

- изучением главы 15 из книги: Jeffrey M. Squyres «Архитектура приложений с открытым исходным кодом», том 2, размещенной на сайте: http://rus-linux.net/MyLDP/BOOKS/Architecture-Open-Source-Applications/Vol-2/openmpi-1.html;
- учебными материалами сайта: http://parallel.ru/tech/tech_dev/mpi.html;
- электронным вариантом книги: Оленев Н.Н. «Основы параллельного программирования в системе MPI». - М.: ВЦ РАН. 2005, с.81, размещенном в файле *~/Документы/MPIbook1.pdf* рабочей области пользователя ***upk***.

2.1.3 Кластерное решение по технологии PVM

Parallel Virtual Machine (***PVM***) - «*виртуальная параллельная машина*», которая представляет собой общедоступный системный программный пакет, позволяющий объединять разнородный набор компьютеров в общий вычислительный ресурс.

Виртуальная параллельная машина (***PVM***) и предоставляет возможность управлять процессами с помощью технологии передачи сообщений.
Существуют реализации ***PVM*** для самых различных платформ, диапазон которых распространяется от лэптопов до суперкомпьютеров ***Gray***.
Поддерживается, посредством предоставления специальных библиотек программирования на языках ***Fortran***, ***C/C++***.
Является свободным ПО, распространяемым под двумя лицензиями: ***BSD License*** и ***GNU General Public License***.
По сравнению с технологией ***MPI***, имеет более расширенные возможности, в плане контроля вычислений: присутствует специализированная консоль управления параллельной системой, а также ее графический эквивалент ***XPVM***, позволяющий наглядно продемонстрировать работу всей системы.

PVM является плодом сотрудничества универсистетов ***Эмори*** и штата ***Теннсси***.
Работа началась в 1989 году и в этом же году была выпущена версия ***PVM 1.0***.
Последняя версия, ***PVM 3.4.6***, выпущена в феврале ***2009 года***.

Замечание

Проект ***PVM*** — достаточно громоздкий и его изучение не входит в программу нашего курса.
Для самостоятельного первоначального изучения, можно обратиться к источникам:

- документация сайта OpenNET под общим названием «PVM-параллельная виртуальная машина», - http://www.opennet.ru/docs/RUS/linux_parallel/node209.html;
- по адресу http://chaos.ssu.runnet.ru/dynamics/books/pvm/using_PVM.htm, - описание системы PVM под общим названием «Использование PVM. Введение в программирование»;
- по адресу http://www.portablecomponentsforall.com/edu/pvm-ru/, - описание, ссылки на документацию и другое, под общим названием «PVM. Parallel Virtual Machine».

2.1.4 Краткие выводы

Подводя итоги развития средств вычислительной техники и сопутствующих им средств программного обеспечения, нетрудно заметить существующую диалектику, присущую самому этому процессу:

- стремление *упорядочить* и *согласовать* последовательность выполняемых вычислений, что естественным образом приводить к понятию алгоритма и развитию технологий последовательных действий основанных на логике;
- стремление *ускорить* процесс вычислений, что естественным образом приводит к идеям и технологиям параллельных вычислений.

Другой аспект вычислительных технологий содержит диалектику связанную со стремлением:

- обеспечить *надежность вычислений*, что требует дублирования (распараллеливания) однотипных функциональных блоков, включая контроль их работы;
- обеспечить *эффективность использования* имеющихся вычислительных ресурсов, включая энергетические ресурсы, что актуально для мобильных систем.

Общая современная направленность технологических решений, обеспечивающих некоторое приемлемое обеспечение сразу всех указанных противоречивых требований основана на выделении *однотипных* и *максимально независимых* объектов данных, на основе которых и строится архитектура вычислительных машин и комплексов.

Это можно показать следующими примерами:

- эволюция *последовательных блочных устройств* от магнитных лент, перфолент и перфокарт к устройствам с *прямым доступом*, таким как магнитные диски, имеющим блоки (сектора) одинаковой длины;
- замена *сегментной организации* оперативной памяти ЭВМ на *страничную*;
- широкое распространение технологий RISC-архитектур процессоров, имеющих *одинаковый размер команд*, *конвейерную организацию* микропроцессорных вычислений;
- *SMP-архитектура* компьютеров с общей памятью и другие.

Серьезными препятствиями на этом пути являются:

- *последовательная природа* базовых вычислительных операций, например, операции суммирования, которые требуют переноса результата суммирования каждого бита слагаемых;
- *необходимость согласования кешей*, при построении SMP-архитектур ЭВМ;
- *неоднородная*, с точки зрения типизации объектов данных, память в сложных системах;
- *распределенная*, *динамичная* и *гетерогенная* природа большинства практически значимых приложений и задач.

Таким образом, мы приходим к выводу, что построение единой универсальной архитектуры ЭВМ или вычислительного комплекса является неприемлемым решением, поскольку в своей основе она должна отражать вычислительную базу будущих вычислительных систем.

В этой ситуации, наиболее важными как в учебном, так и в технологическом плане является изучение технологий ***OpenMP*** и ***Open MPI***, поскольку они отражают основную суть указанных выше диалектически разнонаправленных тенденций, на которых и должна строится целевая ахитектура любого вычислительного комплекса.

2.2 Технология OpenMP

Приступая к практическому освоению технологии ***OpenMP***, мы должны помнить и сравнивать ее с технологией ***Pthreads***, которая к тому же подтверждена стандартами ***POSIX***:

- обе технологии ориентированы на ***SMP***-архитектуру ЭВМ или комплекса;
- обе технологии проводят распараллеливание процесса посредством нитей («***легковесных***» ***процессов***, ***threads***);
- обе технологии только заказывают необходимое количество нитей, а непосредственное распределение их по процессорам ***SMP***-архитектуры осуществляется средствами ОС.

Несмотря на общее сходство, идеологическая и библиотечная реализации этих технологий имеют существенные различия, которые обязательно следует учитывать во время практического их применения.
Описание явных отличий этих технологий начнем с ***Ptreads***, а затем раскроем на конкретных примерах использования ***OpenMP***.

Технология *Pthreads*, практическое применение которой было продемонстрировано в лабораторной работе №6, имеет следующие особенности:

- использует функции явного *создания* и *уничтожения* нитей;
- каждая нить имеет свое *конкретное назначение* и *функциональную реализацию*;
- *для синхронизации* работы нитей изпользутся дополнительные средства ОС и, в частности, - мютексы.

Таким образом, технология ***Pthreads*** предоставляет разработчику мощный инструмент для построения системных программных решений.

Технология *OpenMP*, являющаяся предметом изучения в данной лабораторной работы, основана на парадигме «***ветвлений-слияний***», что поддерживается соответствующими конструкциями самого языка программирования.
Общий вид таких конструкций (для языка С) имеет следующий формат:

```
    #pragma omp <директива> [раздел [ [,] раздел]...]
```

или

```
    #pragma omp <directive> [clause [ [,] clause]...]
```

Раздел (***опция***, ***clause***) — является необязательным синтаксическим элементом языковой конструкции и предназначена для конкретизации применения операции <***директива***>.

Директива — это указание компилятору сформировать специальные команды, которые воздействуют на следущий после директивы блок операторов базового языка, обычно ограничиваемый фигурными скобками «***{...}***».

OpenMP поддерживает следующие основные директивы ***parallel***, ***for***, ***parallelfor***, ***section***, ***sections***, ***single***, ***master***, ***critical***, ***flush***, ***ordered*** и ***atomic***.

Каждая из указанных директив, конечно по своему, воздействует на нити процесса, конкретизируя общие идеи:

- нити процесса *имеют целочисленную нумерацию*, начинающуюся с ***номера 0***;
- нить с ***номером 0*** ассоциируется с *главной* (***master***) нитью процесса и существует всегда, в течении его жизненного цикла;
- после функционального блока директивы, создающей множество нитей, происходит *неявная процедура синхронизации*, соответствующая ожиданию завершения работы всех нитей, кроме нити с ***номером 0***.

Такие идеи очень хорошо ложатся на реализацию различных вычислительных алгоритмов:

- *освобождая* программиста от явного кодирования функций создания и удаления нитей;
- *обеспечивая* простейший тип синхронизации «***barrier***», после завершения параллельного блока, что поддерживается также и специальной директивой ***barrier***;
- *реализуя* удобную нумерацию нитей, обеспечивая индексацию алгоритмов работы с массивами данных.

Дополнительно, по сравнению с ***ptreads***:

- поддерживатся *механизмы защиты* общих переменных всего процесса, выделяя *приватные* (***private***) и общедоступные (***share***) данные;
- обеспечивается *вложенность директив*, также обеспеченная средствами синхронизации.

В целом, может показаться, что технология **OpenMP** мало подходит в качестве технологии, обеспечивающей реализацию системных решений вычислительных комплексов, следует учитывать ряд особенностей присущих сложным системам:

- сложные системы ***практически не реализуются*** в рамках одного процесса, что позволяет использовать разные технологии в отдельных ее компонентах;
- системное ПО вычислительных комплексов ***может содержать компоненты***, запускаемые в рамках отдельных процессов, которые реализуют параллельные алгоритмы обработки данных.

Все это оправдывает изучение и применение технологии ***OpenMP***.

2.2.1 Учебный тестовый пример технологии OpenMP

За неимением возможности изучать все директивы и варианты использования технологии ***OpenMP***, начнем нашу работу с тестового примера, который почти полностью формируется шаблоном проекта системы разработки ***EclipsePTP***.

По традиции, этот пример выводит на консоль терминала привествие, демонстрируя нам параллельную работу нитей, а заодно и готовность к использованию самой системы разработки.

Замечание

Отметим, что для применения технологии ***OpenMP*** нет острой необходимости использования специальных сред разработки. Все современные компиляторы способны реализовывать подобные решения.

Непосредственно, для компилятора ***gcc***, необходимо:

- включить в исходный текст программы заголовочный файл <***omp.h***>;
- указать компилятору опцию ***-fopenmp***;
- обеспечить линковщику доступ к библиотеке ***libomp.so***.

На листинге 2.1 приведен исходный текст тестового примера, реализующего технологию ***OpenMP***.

Текст примера снабжен необходимыми коментариями, поэтому не требует дополнительных пояснений.

Листинг 2.1 — Тестовый пример

```
/*
 ==============================================================================
 Name        : omp1.c
 Author      : Reznik V.G., 12.08.2017
 Version     :
 Copyright   : Your copyright notice
 Description : AVK in C, Ansi-style
 ==============================================================================
 */
#include <omp.h>

#include <stdio.h>
#include <stdlib.h>

/**
 * Hello OpenMP World печатает число нитей и
 * текущий id нити.
 */
int main (int argc, char *argv[]) {

  int numThreads, tid, end, nprocs;

  /**
   * Определяем количество процессоров.
   */
  nprocs = omp_get_num_procs();
  printf("Компьютер имеет %i процессоров\n", nprocs);
  /**
   * Характеристики нитей.
   */
  printf("Устанавливаю 10 нитей...\n");
  omp_set_num_threads(10);
  printf("omp_get_max_threads(): %d\n", omp_get_max_threads());

  printf("omp_get_thread_limit(): %d\n", omp_get_thread_limit());

  /**
   * Выполняем команду создания нитей;
   * каждая нить имеет собственную копию переменных.
   */
#pragma omp parallel private(numThreads, tid)
 {
```

```
  tid = omp_get_thread_num();
  printf("Привет от thread (нити) номер %d\n", tid);

  /**
   * Эта часть запускается только master-нитью (tid=0)
   */
  if (tid == 0)
    {
      numThreads = omp_get_num_threads();
      printf("Число нитей: %d\n", numThreads);
    }
}
end = omp_get_thread_num();
printf("После оператора #pragma: нить номер %d\n", end);

return 0;
}
```

Возможный вариант результатов работы тестовой программы приведен на рисунке 2.1.

```
Problems  Tasks  Console  Properties  Call Graph  Search
<terminated> (exit value: 0) omp1 [C/C++ Application] /home/vgr/workspaceC/omp1/D
Компьютер имеет 4 процессоров
Устанавливаю 10 нитей...
omp_get_max_threads(): 10
omp_get_thread_limit(): 2147483647
Привет от thread (нити) номер 0
Привет от thread (нити) номер 2
Привет от thread (нити) номер 1
Привет от thread (нити) номер 9
Привет от thread (нити) номер 8
Привет от thread (нити) номер 4
Привет от thread (нити) номер 7
Привет от thread (нити) номер 5
Привет от thread (нити) номер 6
Число нитей: 10
Привет от thread (нити) номер 3
После оператора #pragma: нить номер 0
```

Рисунок 2.1 — Результат работы тестовой программы листинга 2.1

Задание

Студенту следует:

- создать в среде системы разработки EclipseC проект ***omp1***;
- скопировать туда текст листинга 2.1 и выполнить действия, указанные в замечании, а затем — запустить проект на выполнение.

Обязательно:

- запустить проект без задания количества нитей;
- отразить в отчете полученные результаты.

2.2.2 Постановка учебной задачи

Применение технолоии ***OpenMP*** продемонстрируем на примере *задачи мониторинга* некоторого заданного набора системных ресурсов вычислительного комплекса.
С целью обощения и одновременного упрощения учебной задачи, контролируемые ресурсы промоделируем программным способом, подразумевая *некоторый абстрактный ресурс*, который задается вектором (структурой) своих параметров.
В качестве конкретного вектора, абстрактного ресурса, выберем набор следующих параметров:

- ***n*** — номер ресурса: целое число от ***1*** до ***N***;
- ***x*** — координата ***x***: целое число со значением в пределах экрана монитора;
- ***y*** — координата ***y***: целое число со значением в пределах экрана монитора.

Главное требование мониторинга ресурсов — контролируемые параметры *должны быть согласованы*: получены «одновременно».

Учебная задача данной лабораторной работы — написание программы мониторинга набора ресурсов вычислительного комплекса с применением технологии ***OpenMP***, которая бы отображала эти ресурсы на экране компьютера, при выполнении следующих дополнительных условий:

- *отслеживание* переключений виртуальных терминалов, с целью предотвращения вмешательства в их деятельность;
- отображение каждого отдельного ресурса производится непосредственно в фреймбуфер виртуального терминала;
- *отслеживание* нажатий на клавиши устройства клавиатуры, с целью обнаружения команды завершения работы программы, которая соответствует нажатию комбинации клавиш ***Ctrl-X***;
- *вывод* на экран монитора конкретного ресурса производится в виде изображения круга некоторого цвета с указанием в центре номера ресурса;
- *реализовать* задачу процесса в виде проекта ***avk_openmp***;
- *использовать* в процессе реализаци проекта *уровень прикладного программирования*, как это было определено в лабораторной работе №5;
- *провести* исследование работы программы данного проекта:
- *описать* в отчете полученные результаты.

Общая технология программной реализации поставленной задачи:

- *выделение* отдельных частей программного обеспечения, соответствующего каждой асинхронно работающей компоненте проекта: компоненты *виртуального терминала*, компоненты *фреймбуфера*, компоненты *получения данных вектора* отдельного контролируемого ресурса;
- *объединение* (композитинг) отдельных частей ПО проекта с помощью главной функции: функции-монитора (функция ***main()***).

Откроем в системе разработки ***EclipseC*** проект ***avk_openmp*** и приступим к поэтапной его реализации.

2.2.3 Формирование заголовочного файла задачи

Заголовочный файл нашего проекта содержит в себе глобальное описание нужных нам структур данных и функций, что вполне оправдано предыдущими примерами.
Учитывая, что даже простейшие задачи, наподобие нашей, используют множество не всегда тривильных функций, постараемся максимально использовать ПО, уже разработанное в других проектах.
Поскольку качественное рисование в фреймбуфере обеспечивается ПО библиотеки ***cairo***, то за основу будем брать ПО лабораторной работы №6, модифицируя его под требования нашей задачи.

Откроем в нашем проекте заголовочный файл ***avk_openmp.h*** и скопируем в него содержимое заголовочного файла ***avk_fb_compositor.h***.

Удаляем из заголовочного файла:

- переменные структуры ***avk_context_t***, касающиеся нитей ***pthreads*** и мютексов, а также все, что касается курсора и динамического изображения;
- описания функций, касающиеся курсора и динамического изображения.

Добавляем в заголовочный файл подключение файла <***omp.h***>.

Результирующий вариант заголовочного файла представлен на листинге 2.2.

Листинг 2.2 — Исходный текст заголовочного файла avk_openmp.h

```
/*
 * avk_openmp.h
 *
 *  Created on: 31 авг. 2017 г.
 *      Author: upk
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/vt.h>
#include <sys/ioctl.h>
#include <termios.h>
#include <string.h>

#include <fcntl.h>
#include <linux/input.h>
#include <signal.h>
#include <errno.h>

#include <stdint-gcc.h>
#include <linux/fb.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
```

```
/**
 * Добавляем заголовочный файл для работы с
 * библиотекой cairo.
 */
//#include <pthread.h>
#include <cairo/cairo.h>
#include <syslog.h>

/**
 * Заголовочный файл для работы с технологией OpenMP.
 */
#include <omp.h>

#define iprintf(...)    syslog(LOG_INFO, __VA_ARGS__)

#ifndef AVK_OPENMP_H_
#define AVK_OPENMP_H_

#define FBFILE    "/dev/fb0"

/**
 * Объявление структуры состояния терминала.
 */
typedef struct _avk_tty {
      struct termios oldtty;  // Старое состояние терминала.
      struct termios newtty;  // Новое  состояние терминала.
      int            changed; // Статус состояния терминала.

} avk_tty_t;

/**
 * Обявление общей контекстной структуры проекта.
 */
typedef struct _avk_contect {
      /**
       * Данные виртуального терминала.
       * Терминал отслеживает необходимость завершения работы
       * и возможность вывода в фреймбуфер.
       */
      avk_tty_t         *tty_status;// Структура состояния терминала.
      int               if_exit;    // Новое: 0 - продолжаем работу; 1 - выход.
      int               no_fb;      // Новое: 0 - можно, 1 - нельзя выводить в fb.
      /**
       * Данные фреймбуфера.
       * Добавляем указатели cairo
       * окна композитора и  фреймбуфера.
       * Композитор исполняется главной нитью!!!
       */
      int fdf;                      // Дескриптор фреймбуфера.
      int x, y, w, h;               // Размеры окна фреймбуфера.
      int stride;                   // Длина строки окна в байтах.
      uint32_t color32;             // Цвет заполнения всего экрана  = 0x00ff
      unsigned char * screen;       // Изменение: выровненный по байту указатель
                                    // на данные фреймбуфера.
      unsigned char * data;         // Новое: Указатель на  буфер  композитора.
      cairo_surface_t *surf;        // Новое: Указатель на  окно   фреймбуфера.
      cairo_t         *crf;         // Новое: Контекст  окна фреймбуфера.
      cairo_surface_t *sur;         // Новое: Указатель на  окно композитора.
      cairo_t         *cr;          // Новое: Контекст  окна  композитора.

} avk_context_t;
```

```c
/** ******************************************************
 * Обявления функций.
 ********************************************************/
/**
 * Функция восстановления состояния терминала,
 * которое сохранено в struct trmios oldtty.
 */
void
avk_restore_tty(avk_context_t *ctx);

/**
 * Установка нового состояния теринала.
 * Новое: аргументом является глобальна яструктура.
 */
avk_tty_t *
avk_setup_tty(avk_context_t *ctx);

/**
 * Получить номер текушей виртуальной консоли.
 * Если ошибка, то возвращает: 0xffff
 */
unsigned short
avk_get_current_vc(int fd);

/**
 * Чтение массива байт с терминала:
 *
 * Аргументы функции:
 * @param buff (o) - Внешний буфер для чтения данных;
 * @param length   - Заявленная длина буфера;
 * @return - возвращает число прочитанных байт
 * с ожиданием не более 20 миллисекунд.
 */
size_t
avk_read_tty(void *buff, size_t length);

/**
 * Закрытие устройства фреймбуфера.
 */
void
avk_fb_close(void *arg);

/**
 * Открытие устройства фреймбуфера.
 */
int
avk_fb_open(avk_context_t *ctx);

/**
 * Заполнение окна фреймбуфера цветом.
 */
int
avk_fb_paint(avk_context_t *ctx);


#endif /* AVK_OPENMP_H_ */
```

2.2.4 Формирование компоненты виртуального терминала

Открываем в проекте файл ***avk_tty.c*** и переносим в него содержимое аналогичного файла из проекта ***avk_compositor***.

В файле ***avk_tty.c*** проводим следующие действия:

- заменяем подключение заголовочного файла на ***avk_openmp.h***;
- удаляем функцию обработки нити `avk_tty_thread(void *arg);`
- в функциях `avk_restore_tty(avk_context_t *ctx)` и `avk_setup_tty(avk_context_t *ctx)` удаляем операторы связанные с созданием и удалением нитей ***pthreads***.

В результате, файл ***avk_tty.c***, содержащий функции управления виртуальным терминалом, примет вид показанный на листинге 2.3.

Листинг 2.3 — Исходный текст файла avk_tty.c

```
/*
 * avk_tty.c
 *
 *  Created on: 3 сент. 2017 г.
 *      Author: upk
 */

#include "avk_openmp.h"

/**
 * Изменено:
 * Функция восстановления состояния терминала,
 * которое сохранено в struct trmios oldtty.
 */
void
avk_restore_tty(avk_context_t *ctx){

      avk_tty_t *p =
                  ctx->tty_status;
      /**
       * Очищаем весь экран.
       * Курсор переводим в позицию (1,1).
       */
      printf("\033[2J\033[H\n");
      /**
       * Включаем аппаратный курсор
       */
     printf("\033[?25h\n");

    if(p->changed)
      /**
       * Возвращаем состояние терминала.
       */
       tcsetattr(STDIN_FILENO, TCSANOW, &p->oldtty);
    /**
     * Освобождаем память, выделенную структуре.
     */
    free(p);
}

/**
```

```
 * Изменено:
 * Установка нового состояния теринала.
 * Новое:
 * - аргументом является глобальна яструктура.
 * Результат:
 * - если инициализация не успешна, то выставляем if_exit = 1;
 * - если инициализация успешна, то выставляем if_exit = 0
 * и запускаем нить обслуживания.
 */
avk_tty_t *
avk_setup_tty(avk_context_t *ctx){
      /**
       * Выделяем память под структуру.
       */
      avk_tty_t *p = malloc(sizeof(avk_tty_t));
    if(p == NULL) return NULL;

    /**
       * Очищаем весь экран.
       * Курсор переводим в позицию (1,1).
       */
      printf("\033[2J\033[H\n");
      /**
       * Выключаем аппаратный курсор.
       */
     printf("\033[?25l\n");
    /**
     * Сохраняем состояние терминала.
     */
    tcgetattr(STDIN_FILENO, &p->oldtty);
    tcgetattr(STDIN_FILENO, &p->newtty);
    /**
     * Переводим терминал в новое состояние:
     * отключаем канонический режим и эхо.
     */
    p->newtty.c_lflag &= ~(ICANON | ECHO | ECHOE );
    if(tcsetattr(STDIN_FILENO, TCSANOW, &p->newtty) < 0){
        /**
         * Если - проблема!!!
         * Возвращаем старое состояние терминала.
         */
      printf("Не могу установить терминал...\n");
        tcsetattr(STDIN_FILENO, TCSANOW, &p->oldtty);
        p->changed   = 0;
        ctx->if_exit = 1;     // Всем на выход!!!
    }else{
      p->changed   = 1;
        ctx->if_exit = 0;
    }

    return p;
}

/**
 * Получить номер текушей виртуальной консоли.
 * Если ошибка, то возвращает: 0xffff
 */
unsigned short
avk_get_current_vc(int fd)
{

      struct vt_stat vs;

      if(ioctl(fd, VT_GETSTATE, &vs) < 0)
```

```
            return 0xffff;

      return(vs.v_active);
}

/**
 * Чтение массива байт с терминала:
 *
 * Аргументы функции:
 * @param buff (o) - Внешний буфер для чтения данных;
 * @param length   - Заявленная длина буфера;
 * @return - возвращает число прочитанных байт
 * с ожиданием не более 20 миллисекунд.
 */

size_t
avk_read_tty(void *buff, size_t length){
    ssize_t L = 0;
    fd_set rfds;
    struct timeval tv;
    int retval;
    FD_ZERO(&rfds);
    FD_SET(0, &rfds);
    /**
     * Ожидаем не более 20ms: 50 раз/сек
     */
    tv.tv_sec = 0; tv.tv_usec = 20*1000;
    /**
     * Проверяем наличие прочитанных данных.
     */
    retval = select(1, &rfds, NULL, NULL, &tv);
    if (!retval) return 0;
    /**
     * Читаем данные, если они доступны.
     */
    if(FD_ISSET(0, &rfds)){
        if((L = read(0, buff, length)) < 1) return 0;
    }
    return (size_t)L;
}
```

2.2.5 Формирование компоненты фреймбуфера

Открываем в проекте файл ***avk_fb.c*** и переносим в него содержимое аналогичного файла из проекта ***avk_compositor***.

Поскольку компонента фреймбуфера не формирует нитей ***Pthreads***, то изменения здесь — минимальны.

В файле ***avk_fb.c*** проводим следующие действия:

- заменяем подключение заголовочного файла на ***avk_openmp.h***;
- в функции `avk_fb_paint(avk_context_t *ctx)` заменяем текст «`AVK-композитор`» на текст «`AVK-OpenMP`».

В результате, файл ***avk_fb.c***, содержащий функции управления фреймбуфером и окном композитора, примет вид показанный на листинге 2.4.

Листинг 2.4 — Исходный текст файла avk_fb.c

```
/*
 * avk_fb.c
 *
 *  Created on: 3 сент. 2017 г.
 *      Author: upk
 */

#include "avk_openmp.h"

/**
 * Удаление cairo устройства для framebuffer
 */
void
asufb_device_destroy(void * arg)
{
      avk_context_t *ctx = (avk_context_t *)arg;

    if ((ctx->screen != NULL)&&(ctx->stride > 0)){
      /**
       * Завершаю работу.
       */
        munmap(ctx->screen, ctx->stride * ctx->h);
      close(ctx->fdf);
      ctx->fdf = -1;
    }
}

/**
 * Закрытие устройства фреймбуфера.
 */
void
avk_fb_close(void *arg){
      avk_context_t *ctx =
                  (avk_context_t *)arg;
      /**
       * Закрываем созданные объекты библиотеки cairo.
       */
      if(ctx->cr != NULL)
```

```
            cairo_destroy (ctx->cr);
      ctx->cr = NULL;

      if(ctx->sur != NULL)
            cairo_surface_destroy (ctx->sur);
      ctx->sur = NULL;
      if(ctx->data != NULL)
            free(ctx->data);
      ctx->data = NULL;

      if(ctx->crf != NULL)
            cairo_destroy (ctx->crf);
      ctx->crf = NULL;

      if(ctx->surf != NULL)
            cairo_surface_destroy (ctx->surf);
      ctx->surf = NULL;
}

/**
 * Открытие устройства фреймбуфера.
 */
int
avk_fb_open(avk_context_t *ctx){
      char * title =
                  "avk_fb_open():";

      /**
       * Открываем устройство фреймбуфера для
       * чтения и записи.
       */
      ctx->fdf = open(FBFILE, O_CLOEXEC | O_RDWR);
      if(ctx->fdf < 0){
            printf("%s\tНе могу открыть файл: %s\n",
                        title, FBFILE);
            return -1;
      }

      /**
       * Определяем параметры монитора.
       */
      struct fb_fix_screeninfo fix;
      if(ioctl (ctx->fdf, FBIOGET_FSCREENINFO, &fix) == -1){
            printf("%s\tНе могу получить информацию о буфере...\n",
                        title);
            close(ctx->fdf);
            return -1;
      }
      /**
       * Длина строки экрана в пикселях:
       * по 4 байта на пиксель.
       */
      ctx->stride = fix.line_length;
      /**
       * Читаю формат пикселя в битах.
       */
      struct fb_var_screeninfo var;
      var.bits_per_pixel = 0;
      ioctl (ctx->fdf, FBIOGET_VSCREENINFO, &var);
      /**
       * Если формат не 32 бита на пиксель, то
       * переустанавливаю это значение.
       */
      if(var.bits_per_pixel != 32){
```

```
        var.bits_per_pixel = 32;
        ioctl (ctx->fdf, FBIOPUT_VSCREENINFO, &var);
  }
  /**
   * Снова (для контроля) читаю структуру.
   */
  if(ioctl (ctx->fdf, FBIOGET_VSCREENINFO, &var) == -1){
        printf("%s\tНе могу получить информацию о буфере...\n",
                    title);
        close(ctx->fdf);
        return -1;
  }
  /**
   * Проверка формата пикселя.
   */
  if(var.bits_per_pixel != 32){
        printf("%s\tНа такой экран - не записываю!!!\n",
                    title);
        close(ctx->fdf);
        return -1;
  }
  /**
   * С фреймбуфером можно работать как с последовательным
   * символьным файлом, но лучше - как с матрицей целых чисел,
   * поэтому проводим mmap() на память ЭВМ.
   */
  ctx->w   = var.xres;                // Ширина фреймбуфера в пикселях.
  ctx->h   = var.yres;                // Высота фреймбуфера в пикселях.
  int nbuf =
              ctx->h*fix.line_length; // Длина фреймбуфера в байтах.
  ctx->screen = (unsigned char *)mmap (0, nbuf, PROT_READ |
              PROT_WRITE, MAP_SHARED,
              ctx->fdf, 0);
  if(ctx->screen <= 0){
        printf("%s\tНе могу сделать mmap() на фреймбуфер!!!",
                    title);
        close(ctx->fdf);
        return -1;
  }
/**
 * Создание cairo surface, для рисования в framebuffer
 */
ctx->surf = cairo_image_surface_create_for_data(ctx->screen,
        CAIRO_FORMAT_ARGB32, var.xres, var.yres, ctx->stride);
/**
 * Инициализация данными cairo surface: для рисования в framebuffer
 */
cairo_surface_set_user_data(ctx->surf, NULL, ctx,
                            &asufb_device_destroy);
ctx->crf = cairo_create(ctx->surf);

/**
 * Создание окна композитора.
 * Выделяем память для изображения окна композитора.
 */
  ctx->data = malloc(nbuf);
  if(ctx->data == NULL){
  printf("%s\t\tОшибка выделения памяти ...\n",
              title);
        return -1;
  }
/**
 * Создаем cairo_surface_t - поверхность композитора.
 */
```

```
    ctx->sur = cairo_image_surface_create_for_data(ctx->data,
            CAIRO_FORMAT_ARGB32, ctx->w, ctx->h, ctx->stride);
    /**
     * Создаем cairo_t - управление рисунком композитора.
     */
    ctx->cr = cairo_create(ctx->sur);

    /**
     * Неплохо бы и проверять созданное!!!
     */

    avk_fb_paint(ctx);

      return EXIT_SUCCESS;
}

/**
 * Заполнение окна фреймбуфера цветом.
 */
int
avk_fb_paint(avk_context_t *ctx){
      /**
       *  Заполняем фоновым цветом весь экран.
       *  Новое:
       *  Действия выполняются в буфере композитора, имитируя ситуацию,
       *  что вроде как композитор сам перенес в буфер родительское окно.
       */
      cairo_set_source_rgba(ctx->cr, 0, 0, 1.0, 1.0); // Цвет фона синий.
      cairo_move_to(ctx->cr, 0, 0);                   // В начало координат.
      cairo_rectangle(ctx->cr, 0, 0, ctx->w, ctx->h); // Задаем область.
      cairo_fill(ctx->cr);                            // Заполняем цветом.

    /**
     * Выводим текст
     */
    char text[] =
                  "AVK-OpenMP";
    cairo_select_font_face(ctx->cr, "Sans", CAIRO_FONT_SLANT_NORMAL,
            CAIRO_FONT_WEIGHT_BOLD);
    cairo_set_font_size(ctx->cr, ctx->h/20);

    cairo_text_extents_t ext; // Вычисляем длину текста.
    cairo_text_extents (ctx->cr, text, &ext);
    int wtext = ext.width;

    /**
     * Создаем тень.
     */
    int shade = 5;
    cairo_set_source_rgb (ctx->cr, 0.3, 0.3, 0.0);    // Темно-желтый.
    cairo_move_to(ctx->cr, (ctx->w - wtext)/2 + shade, ctx->h/2);
    cairo_show_text(ctx->cr, text);

    /**
     * Нормальный цвет.
     */
    cairo_set_source_rgb (ctx->cr, 0.6, 0.6, 0.0);    // Желтый.
    cairo_move_to(ctx->cr, (ctx->w - wtext)/2, ctx->h/2);
    cairo_show_text(ctx->cr, text);

      return 0;
}
```

2.2.6 Реализация проекта avk_openmp

Реализацию проекта проведем в три этапа:

- *на этапе 1*, создадим основу файла ***avk_openmp.c***, которая содержит уже имющееся ПО виртуального терминала и фреймбуфера;
- *на этапе 2*, реализуем функционал ПО, который моделирует контролируемые ресурсы;
- *на этапе 3*, завершим создание проекта, объеденив весь имеющийся функционал, в соответствии с требованиями поставленной задачи.

Этап 1. Реализация функционала виртуального терминала и фреймбуфера

За основу функционала, реализуемого функцией ***main()*** нашего проекта, возьмем исходный текст файла ***avk_compositor.c***, где:

- *заменим* файл подключаемого заголовка на ***avk_openmp.h***;
- *удалим* из исходного текста вызовы функций и определения переменных, касающиемя устройства мыши и динамического рисунка;
- *добавим* во вторую часть исходного теста функции ***main()***, реализуемого циклом, функционал, реагирующий на переключение виртуальных терминалов, восстановление содержимого фреймбуфера из содержимого буфера композитора, а также реакцию на завершение работы программы по нажатию комбинации клавиш ***Ctrl-X***.

Замечание

Указанные изменения студент может выполнить самостоятельно, учитывая опыт предыдущих лабораторных работ.

Далее, следует, посредством компилирования проекта в системе разработки, убедиться в отсутствии ошибок, а затем проверить реальную работу программы.

Замечание

Работа программы проверяется:

- переходом на виртуальный терминал ***tty3***;
- ***login*** в терминале от имени пользователя ***upk***;
- запуск приложения командой ***avk_openmp***.

Внимание!

Убедитесь, что имеется мягкий link из директории ***~/bin*** на запускаемую программу.
Если линка нет, то, перейдя в директорию ***~/bin***, создайте его командой:

```
ln -s ~/workspaceC/avk_openmp/Debug/avk_openmp  avk_openmp
```

В результате запуска проекта на этом этапе, экран монитора должен окрашиваться в синий цвет, а посередине экрана должна быть соответствующая надпись.
При нажатии комбинации клавиш ***Ctrl-X***, программа должна завераща́ть работу.

Этап 2. Реализация функционала моделируемых ресурсов

В соответствии с постановкой задачи, мы моделируем ***N*** — ресурсов, каждый из которых задается структурой типа:

```
typedef struct _resurs {// Структура отдельного ресурса.
      int   n;          // Номер ресурса.
      int   x;          // Координата x ресурса.
      int   y;          // Координата y ресурса.
} resurs_t;
```

Полный перечень исходных данных представлен на листинге 2.5.

Листинг 2.5 — Исходные данные модели для файла avk_opentp.c

```
/**
 * Общий контекст проекта.
 */
avk_context_t ctx;

/**
 * Модель ресурсов.
 */
typedef struct _resurs {      // Структура отдельного ресурса.
      int   n;          // Номер ресурса.
      int   x;          // Координата x ресурса.
      int   y;          // Координата y ресурса.
} resurs_t;

int N = 20;             // Число моделируемых ресурсов.

resurs_t N_VECTOR[20];  // Вектор новых  ресурсов.
resurs_t O_VECTOR[20];  // Вектор старых ресурсов.

time_t      nt = 3;                 // Время в сек между замерами ресурсов.
time_t      tt = 0;                 // Текущее время, в секундах.
time_t      tp = 0;                 // Время следующего замера, в секундах.
int n;                              // Рабочая переменная, номер ресурса.

/**
 * Функция, генерирующая отдельный ресурс.
 */
void
set_new_resurs(avk_context_t *pctx, resurs_t *pnew, int n){
      pnew->n = n+1;
      float f = rand();                   // Получаем случайное число.
      pnew->x = (f/RAND_MAX)*pctx->w;     // Устанавливаем координату x.
      f = rand();                         // Получаем случайное число.
      pnew->y = (f/RAND_MAX)*pctx->h;     // Устанавливаем координату y.
//printf("tt=%li tp=%li n=%i x=%i y=%i\n", tt, tp, pnew->n, pnew->x, pnew->y);
}
/**
 * Задаем параметры формы отображаемого ресурса.
 */
int ro = 20;      // Размер круга в пикселях.
double reds[3]   = {0.5, 0.7, 0.9}; // Цвета для ресурсов.
double greens[3] = {0.9, 0.7, 0.5};
int k;            // Рабочая переменная для выбора цвета окружности.
char snum[10];    // Рабочая переменная для текста номера ресурса.
```

Этап 3. Полная реализация проекта

Реализацию проекта завершаем модификацией головной функции ***main()***.
Для этого:

- до основного цикла, - генегируем начальные значения координат ресурсов;
- внутри основного цикла, измеряем текущее время и проверяем необходимость обновления координат ресурсов; когда время обновления — наступило, мы генерируем новые значения, а затем, в цикле, восстанавливаем изображение фреймбуфера из буфера композитора и рисуем окружности, в точках новых координат ресурсов.

Результирующий вариант файла ***avk_openmp.c*** приведен на листинге 2.6.

Листинг 2.6 — Результирующий вариант файла avk_openmp.c

```
/*
 ============================================================================
 Name        : avk_openmp.c
 Author      : Reznik V.G., 31.08.2017
 Version     :
 Copyright   : Your copyright notice
 Description : AVK in C, Ansi-style
 ============================================================================
 */

#include "avk_openmp.h"
#include <time.h>
#include <math.h>

/**
 * Общий контекст проекта.
 */
avk_context_t ctx;

/**
 * Модель ресурсов.
 */
typedef struct _resurs {// Структура отдельного ресурса.
      int   n;          // Номер ресурса.
      int   x;          // Координата x ресурса.
      int   y;          // Координата y ресурса.
} resurs_t;

int N = 20;             // Число моделируемых ресурсов.

resurs_t N_VECTOR[20];  // Вектор новых  ресурсов.
resurs_t O_VECTOR[20];  // Вектор старых ресурсов.

time_t      nt = 3;                 // Время в сек между замерами ресурсов.
time_t      tt = 0;                 // Текущее время, в секундах.
time_t      tp = 0;                 // Время следующего замера, в секундах.
int n;                              // Рабочая переменная, номер ресурса.

/**
 * Функция, генерирующая отдельный ресурс.
 */
void
set_new_resurs(avk_context_t *pctx, resurs_t *pnew, int n){
```

```
      pnew->n = n+1;
      float f = rand();                    // Получаем случайное число.
      pnew->x = (f/RAND_MAX)*pctx->w;      // Устанавливаем координату x.
      f = rand();                          // Получаем случайное число.
      pnew->y = (f/RAND_MAX)*pctx->h;      // Устанавливаем координату y.
//printf("tt=%li tp=%li n=%i x=%i y=%i\n", tt, tp, pnew->n, pnew->x, pnew->y);
}

/**
 * Задаем параметры формы отображаемого ресурса.
 */
int ro = 20;       // Размер круга в пикселях.
double reds[3]   = {0.5, 0.7, 0.9}; // Цвета для ресурсов.
double greens[3] = {0.9, 0.7, 0.5};
int k;             // Рабочая переменная для выбора цвета окружности.
char snum[10];     // Рабочая переменная для текста номера ресурса.

/**
 * Головная функция.
 */
int
main(void) {
      char *title =
                  "Проект avk_openmp:";
      printf("\nКомбинация клавишь Ctrl-X - завершение работы программы!\n");
      printf("Для продолжения - нажми Enter...\n");
      getchar();

      /** ****************************************
       * Первая часть ПО.
       * Инициализация данных и открытие устройств.
       ******************************************/
      /**
       * Инициализация общей контекстной структуры проекта.
       */
      /**
       * Данные виртуального терминала.
       */
      ctx.tty_status = avk_setup_tty(&ctx);// Структура состояния терминала.
      if(ctx.tty_status == NULL || ctx.tty_status->changed == 0){
            printf("%s\tНе могу инициализировать виртуальный терминал\n",
                        title);
            usleep(2000*1000);
            return -1;
      }
      unsigned short nvt = avk_get_current_vc(0);// Номер виртуального  терминала.

      /**
       * Данные фреймбуфера.
       */
      ctx.x = 0;
      ctx.y = 0;
      ctx.color32 = 0x00ff;
      /**
       * Запускаем устройство фреймбуфера.
       */
      if(avk_fb_open(&ctx) < 0){
            printf("%s\tНе могу открыть фреймбуфер\n",
                        title);
            usleep(2000*1000);
            avk_restore_tty(&ctx);
            return -1;
      }
```

```
      /** ************************************
       * Генерируем начальные значения ресурсов.
       ****************************************/
      omp_set_num_threads(N); // Задаю число нитей.
      resurs_t *pnew;         // Указатель на вектор ресурсов.
tt = time(NULL);
tp = tt;
#pragma omp parallel shared(ctx,N_VECTOR) private(n,pnew)
{
            n = omp_get_thread_num();           // Определяю номер нити.
            pnew = &N_VECTOR[n];
            set_new_resurs(&ctx, pnew, n);      // Генерирую данные ресурса.
            // Устанавливаю барьер синхронизации.
#pragma omp barrier
}

      /**
       * Рабочие переменные.
       */
      char buffer[20];                    // Буфер чтения данных с клавиатуры.
      int          dirty  = 1;            // Нужно полностью перерисовать экран.
      __useconds_t dt         = 20*1000;  // Период цикла для экспериментов.

      /** ******************************************
       * Вторая часть ПО.
       * Цикл композитинга, реализующего
       * программный мониторинг состояний устройств.
       ********************************************/
      while(1){
            /**
             * Проверяю, не нужно ли завершить работу.
             */
            if(avk_read_tty(buffer, 20) > 0){
                  if((int)buffer[0] == 24)
                        ctx.if_exit = 1;
            }
            if(ctx.if_exit)
                  break;      // Завершаю работу.

            usleep(dt);       // Засыпаем на заданный период (для экспериментов).
            /**
             * Проверяю, является ли терминал текущим.
             */
            if(avk_get_current_vc(0) != nvt)
                  ctx.no_fb = 1;
            else
                  ctx.no_fb = 0;

            if(ctx.no_fb)
                  dirty = 1;
            /**
             * Проверяю, нужно ли перерисовывать весь экран.
             */
            if(!ctx.no_fb && dirty){
                  cairo_set_source_surface (ctx.crf, ctx.sur, 0, 0);
                  cairo_paint(ctx.crf);

                  dirty  = 0; // Отмечаем выполнение.

            }

            /** ************************************
             * Генерируем начальные значения ресурсов.
             ****************************************/
```

```
        tt = time(NULL);// Определяем текущее время.
        if(tt >= tp){     // Генерируем и отображаем новый ресурс.
              omp_set_num_threads(N); // Задаю число нитей.

        #pragma omp parallel shared(ctx,N_VECTOR,O_VECTOR) private(n,pnew)
        {
                    n = omp_get_thread_num();// Определяю номер нити.
                    O_VECTOR[n] = N_VECTOR[n];// Сохраняю старое значение ресурса.
                    pnew = &N_VECTOR[n];
                    set_new_resurs(&ctx, pnew, n);// Генерирую данные ресурса.
                    // Устанавливаю барьер синхронизации.
        #pragma omp barrier
        }
              tp = tt + nt;// Новое значение порога времени обновления значений ресурсов.
              /** ********************************
               * Отображаю новое состояние ресурсов.
               *
               * Привязываю исходный рисунок композитора к окну фреймбуфера.
               */
              cairo_set_source_surface (ctx.crf, ctx.sur, 0, 0);
              /**
               * В цикле:
               * Восстанавливаем области, занятые окружностями ресурсов,
               * изображениями из окна композитора.
               */
              for(int i=0; i< N; i++){
                  cairo_rectangle(ctx.crf, O_VECTOR[i].x-ro, O_VECTOR[i].y-ro,
                                  2*ro, 2*ro);
                  cairo_fill(ctx.crf);
              }
              /**
               * В цикле:
               * Рисуем новое положение ресурсов.
               */
            cairo_select_font_face(ctx.crf, "Sans", CAIRO_FONT_SLANT_NORMAL,
                    CAIRO_FONT_WEIGHT_BOLD);
            cairo_set_font_size(ctx.crf, ro);
              for(int i=0; i< N; i++){
                    cairo_move_to(ctx.crf, N_VECTOR[i].x, N_VECTOR[i].y);
                    /**
                     * Устанавливаем цвет круга и рисуем его.
                     */
                    k = i%3;
        cairo_set_source_rgb(ctx.crf, reds[k], greens[k], 0); // Цвет круга.
                    cairo_arc(ctx.crf, N_VECTOR[i].x, N_VECTOR[i].y,
                                ro, 0, 2*M_PI);
                    cairo_fill(ctx.crf);
                    /**
                     * Рисуем номер ресурса.
                     */
                    sprintf(snum, "%i", N_VECTOR[i].n);
                    cairo_move_to(ctx.crf, N_VECTOR[i].x-ro/2,
                          N_VECTOR[i].y+ro/2);
        cairo_set_source_rgb (ctx.crf, 0.0, 0.0, 0.0);  // Цвет черный.
                  cairo_show_text(ctx.crf, snum);
              }
        }

    }
    /** ******************************
     * Третья часть ПО.
     * Закрытие устройств и завершение
     * работы программы.
     *******************************/
```

```
//    printf("%s\tЗавершаю работу main()\n",
//                title);
      usleep(2000*1000);
//    printf("Вызываю avk_fb_close()\n");
      avk_fb_close(&ctx);
//    printf("Вызываю avk_restore_tty()\n");
      avk_restore_tty(&ctx);


      return EXIT_SUCCESS;
}
```

Завершив отладку программы, студенту следует провести:

- исследование работы программы;
- описание в личном отчете результатов исследования.

Замечание

Запуск программы производится на отдельном терминале, например ***/dev/tty3***, после соответствующей процедуры ***login*** от имени пользователя ***upk***.

3 Лабораторная работа №8

В данной лабораторной работе рассматривается технология ***OpenMPI***.
Open MPI представляет собой объединение четырех исследовательских и академических реализаций ***MPI*** с открытым исходным кодом: *LAM/MPI*, *LA/MPI* (Лос-Аламосский вариант MPI) и *FT-MPI* (отказоустойчивый вариант MPI).
Вскоре, после создания группы Open MPI, к ней присоединилась команда проекта ***PACX-MPI***.
Первый вариант этой реализации был помещен в *Subversion*, ***22 ноября 2003 года***.

3.1 Архитектура OpenMPI

Архитектура ***OpenMPI***, состоящая из трех основных слоев абстракции, представлена на рисунке 3.1.

Рисунок 3.1 — Архитектура OpenMPI

Назначение уровней (слоев) абстракции, следующее:

- *Open, Portable Access Layer* (***OPAL***): Слой OPAL является нижним слоем абстракции проекта Open MPI, обеспечивающим выполнение отдельных процессов посредством утилит и связующего кода, например, связные списки общего назначения, обработка строк, управление отладкой и другие необходимые функции. В этом слое также реализуется *ядро переносимости* Open MPI между различными операционными системами, например, доступ к интерфейсам ***IP***, совместное использование памяти на одном и том же сервере, согласование процессора и памяти, высокоточные таймеры и другое;
- *Open MPI Run-Time Environment* (***ORTE***) (*«**ор-тей**»*): обеспечивает предоставление как *интерфейса API*, необходимый для передачи сообщений, так и

сопутствующую *систему времени выполнения* для запуска, отслеживания и уничтожения параллельно выполняемых заданий. Для ***Open MPI*** параллельно выполняемое задание состоит из одного или нескольких процессов, которые могут исполняться на нескольких экземплярах операционной системы и могут быть взаимосвязанны друг с другом так, чтобы они действовали как нечто единое. В простых средах со слабой поддержкой или без поддержки распределенных вычислений *слой ORTE* использует команды ***rsh*** или ***ssh*** для запуска отдельных процессов в параллельно выполняемых заданиях. В более продвинутых HPC-средах обычно есть планировщики и менеджеры ресурсов, применяемые для справедливого распределения вычислительных ресурсов между многими пользователями. В таких средах обычно предоставляются специализированные интерфейсы, предназначенные для запуска и регулирования процессов на вычислительных серверах. В слое *ORTE* поддерживается широкий спектр таких управляемых сред, например: ***Torque/PBS Pro***, ***SLURM***, ***Oracle Grid Engine*** и ***LSF***;

- *Open MPI* ***(OMPI)*** - самый высокий уровнь абстракции, который является единственным слоем видимым приложениям и в котором реализован интерфейс ***API*** для ***MPI***, содержащий семантику передачи сообщений определяемую *стандартом MPI*.

Замечание

Каждый слой собран в виде отдельной библиотеки:

- библиотека ORTE зависит от библиотеки OPAL;
- библиотеки OMPI зависит от библиотеки ORTE.

Хотя каждая абстракция представляет собой слой, расположенный поверх нижележащего слоя, как показано на рисунке 3.1, слои ORTE и OMPI могут обходить нижележащие слои абстракции и непосредственно взаимодействовать с операционной системой и/или аппаратным обеспечением.

3.1.1 Технология MPI

Технология MPI основана на ряде определений, составляющих ее понятийную основу.

MPI - *message passing interface* - библиотека функций, предназначенная для поддержки работы параллельных процессов в терминах передачи сообщений.

Номер процесса - целое неотрицательное число, являющееся уникальным атрибутом каждого процесса.

Атрибуты сообщения - номер процесса-отправителя, номер процесса-получателя и идентификатор сообщения. Для них заведена структура *MPI_Status*, содержащая три поля:

- *MPI_Source* - номер процесса отправителя;
- *MPI_Tag* - идентификатор сообщения;
- *MPI_Error* - код ошибки: могут быть и добавочные поля.

Идентификатор сообщения ***(msgtag)*** - атрибут сообщения, являющийся целым неотрицательным числом, лежащим в диапазоне от ***0*** до ***32767***.
Процессы объединяются в ***группы***, которые могут быть вложенными группами.
Внутри группы все процессы перенумерованы.
С каждой группой ассоциирован свой ***коммуникатор***, поэтому при осуществлении пересылки необходимо указать идентификатор группы, внутри которой производится эта пересылка.
Все процессы содержатся в группе с заранее предопределенным идентификатором ***MPI_COMM_WORLD***.

Технология MPI, по разным оценкам, содержит более *300 функций*.
Рассмотрим наиболе значимые функции, отражающие суть самой технологии.

Общие процедуры MPI

```
int MPI_Init( int* argc, char*** argv)
```

MPI_Init - инициализация параллельной части приложения.
Реальная инициализация для каждого приложения выполняется не более одного раза, а если MPI уже был инициализирован, то никакие действия не выполняются и происходит немедленный возврат из подпрограммы.
Все оставшиеся MPI-процедуры могут быть вызваны только после вызова *MPI_Init*.
Возвращает: в случае успешного выполнения - *MPI_SUCCESS*, иначе - код ошибки.

```
int MPI_Finalize( void )
```

MPI_Finalize - завершение параллельной части приложения.
Все последующие обращения к любым MPI-процедурам, в том числе к *MPI_Init*, запрещены.
К моменту вызова *MPI_Finalize* некоторым процессом все действия, требующие его участия в обмене сообщениями, должны быть завершены.

Сложный тип аргументов *MPI_Init* предусмотрен для того, чтобы передавать всем процессам аргументы ***main()***:

```
int main(int argc, char** argv)
{
      MPI_Init(&argc, &argv);
          ...
      MPI_Finalize();
}
```

```
int MPI_Comm_size( MPI_Comm comm, int* size)
```

Определение общего числа параллельных процессов в группе ***comm***.

- *comm* - идентификатор группы;
- *size* - размер группы.

```
int MPI_Comm_rank( MPI_Comm comm, int* rank)
```

Определение номера процесса в группе ***comm***.
Значение, возвращаемое по адресу ***&rank***, лежит в диапазоне от **0** до *size_of_group-1*.

- *comm* - идентификатор группы;
- *rank* - номер вызывающего процесса в группе ***comm***.

```
double MPI_Wtime(void)
```

Функция возвращает астрономическое время в секундах (вещественное число), прошедшее с некоторого момента в прошлом.
Гарантируется, что этот момент не будет изменен за время существования процесса.

Прием и передача сообщений между отдельными процессами (прием и передача сообщений с блокировкой)

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
    int dest, int msgtag, MPI_Comm comm)
```

- *buf* - адрес начала буфера посылки сообщения;
- *count* - число передаваемых элементов в сообщении;
- *datatype* - тип передаваемых элементов;
- *dest* - номер процесса-получателя;
- *msgtag* - идентификатор сообщения;
- *comm* - идентификатор группы.

Блокирующая посылка сообщения с идентификатором *msgtag*, состоящего из *count* элементов типа *datatype*, процессу с номером *dest*.
Все элементы сообщения расположены подряд в буфере *buf*.
Значение *count* может быть нулем.
Тип передаваемых элементов *datatype* должен указываться с помощью предопределенных констант типа.
Разрешается передавать сообщение самому себе.

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы.
Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу *dest*, остается за MPI.

Замечание

Следует специально отметить, что возврат из подпрограммы ***MPI_Send()*** не означает ни того, что сообщение уже передано процессу ***dest***, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший ***MPI_Send()***.

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
     int msgtag, MPI_Comm comm, MPI_Status *status)
```

- *buf* - адрес начала буфера приема сообщения;
- *count* - максимальное число элементов в принимаемом сообщении;
- *datatype* - тип элементов принимаемого сообщения;
- *source* - номер процесса-отправителя;
- *msgtag* - идентификатор принимаемого сообщения;
- *comm* - идентификатор группы;
- *status* - параметры принятого сообщения.

Прием сообщения с идентификатором *msgtag* от процесса *source* с блокировкой.
Число элементов в принимаемом сообщении не должно превосходить значения *count*.
Если число принятых элементов меньше значения *count*, то гарантируется, что в буфере *buf* изменятся только элементы, соответствующие элементам принятого сообщения.
Если нужно узнать точное число элементов в сообщении, то можно воспользоваться подпрограммой ***MPI_Probe()***.

Блокировка гарантирует, что после возврата из функции все элементы сообщения приняты и расположены в буфере *buf*.
В качестве номера процесса-отправителя можно указать предопределенную константу *MPI_ANY_SOURCE* - признак того, что подходит сообщение от любого процесса.
В качестве идентификатора принимаемого сообщения можно указать константу *MPI_ANY_TAG* - признак того, что подходит сообщение с любым идентификатором.
Если процесс посылает два сообщения другому процессу и оба эти сообщения соответствуют одному и тому же вызову ***MPI_Recv()***, то первым будет принято то сообщение, которое было отправлено раньше.

3.1.2 Учебный тестовый пример

Существенной особенностью технологии MPI является понятие *коммуникатора*.
Коммуникатор — распределенная среда, включающая в себя как локальные процессора компьютера, так и процессора удаленных компьютеров.
Коммуникатор имеет *имя* и предварительные *настройки* на множество компьютеров, которые видны прикладной программе только через *имя коммуникатора*.

Базовым именем коммуникатора является ***MPI_COMM_WORLD***, что освобождает программиста от явного программирования средств доставки сообщений и позволяет ему сосредоточиться на самом алгоритме решения задачи.

Таким образом, программист пишет программу, подразумевая, что:

- *выполняется* главная нить некоторого процесса;
- *различение* процессов производится по *целочисленному номеру*, что учитывается при написании программы.

На листинге 3.1 приведен исходный код тестового примера, предоставляемого системой разработки, который реализован в пределах проекта ***ompi1***.

Листинг 3.1 — Тестовый пример проекта ompi1

```
/*
 ============================================================================
 Name        : ompi1.c
 Author      : Reznik V.G.
 Version     :
 Copyright   : Your copyright notice
 Description : Hello MPI World in C
 ============================================================================
 */
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[]){
      int  my_rank; /* rank of process */
      int  p;       /* number of processes */
      int source;   /* rank of sender */
      int dest;     /* rank of receiver */
      int tag=0;    /* tag for messages */
      char message[100];    /* storage for message */
      MPI_Status status ;   /* return status for receive */

      /* start up MPI */

      MPI_Init(&argc, &argv);

      /* find out process rank */
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

      /* find out number of processes */
      MPI_Comm_size(MPI_COMM_WORLD, &p);

      if (my_rank !=0){
            /* create message */
            sprintf(message, "Hello MPI World from process %d!", my_rank);
            dest = 0;
            /* use strlen+1 so that '\0' get transmitted */
            MPI_Send(message, strlen(message)+1, MPI_CHAR,
               dest, tag, MPI_COMM_WORLD);
      }
      else{
            printf("Hello MPI World From process 0: Num processes: %d\n",p);
            for (source = 1; source < p; source++) {
                  MPI_Recv(message, 100, MPI_CHAR, source, tag,
                        MPI_COMM_WORLD, &status);
                  printf("%s\n",message);
```

```
            }
        }
        /* shut down MPI */
        MPI_Finalize();


        return 0;
}
```

Замысел алгоритма программы — достаточно прост:

- процессы, с номерами отличными от ***0***, посылают текстовое сообщение типа «***Hello MPI World...***» процессу с номером ***0***;
- процесс с номером ***0*** читает эти сообщения и распечатывает их на терминале.

На рисунке 3.2 представлены два варианта запуска результирующей программы:

- *вариант 1* — просто запуск;
- *вариант 2* — запуск с использованием утилиты ***mpirun***.

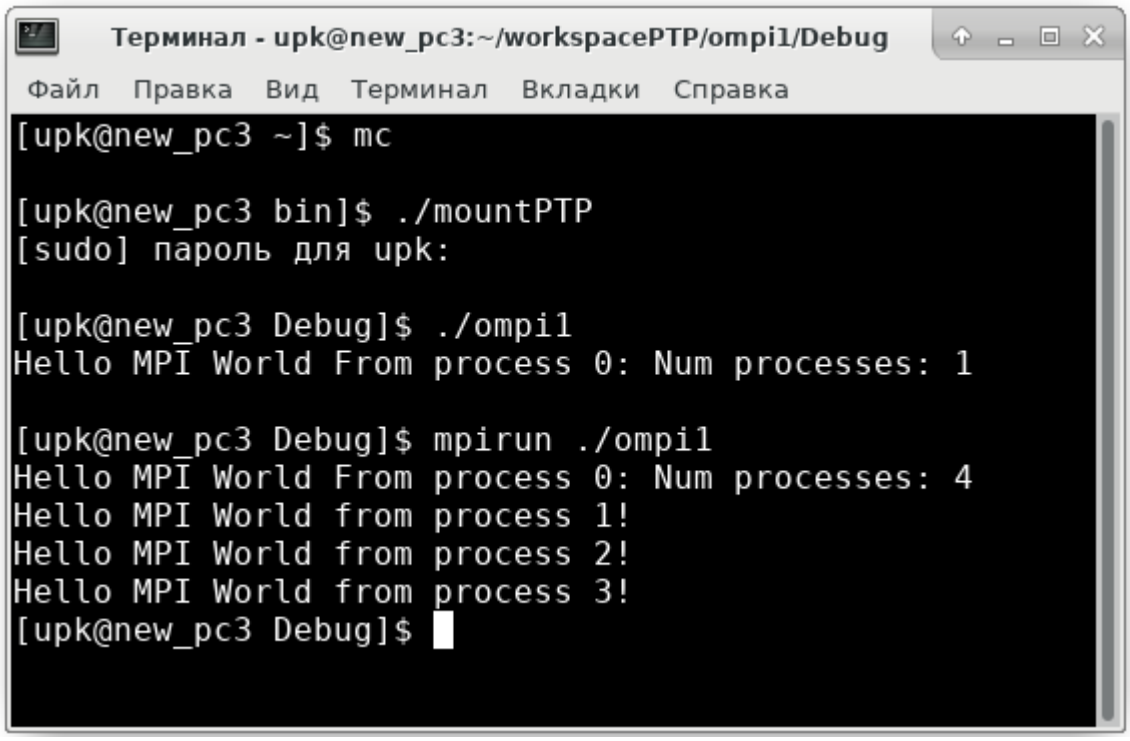

Рисунок 3.2 — Два варианта запуска программы ompi1

Из рисунка 3.2 хорошо видно, что сама программа реализует только один процесс, поэтому для ее запуска используется специальная утилита ***mpirun*** или ее эквиваленты: ***mpiexec*** и ***orterun***.

Основная их задача — параллельный запуск нужного количества процессов и распределение их по вычислительным ресурсам вычислительного комплекса.

Для этих целей используется общий синтаксис запуска:

```
mpirun -np <#>  <имя_программы> <аргументы>
```

где **<#>** - число параллельно запущенных процессов.

В общем случае, утилита ***mpirun*** имеет достаточно много аргументов, которые обеспечивают раличные варианты запуска приложений.
Для изучения всех опций запуска этой утилиты следует использовать руководство ***man***.

Замечание

При создании приложений по технологии *OpenMPI* необходим специальный препроцессор ***mpicc***, поэтому в данной лабораторной работе используется специальная среда разработки *EclipsePTP*.

Для запуска приведенного учебного примера следует:

- предварительно запустить сценарий ***~/bin/mountPTP***, который подмонтирует нужную среду разработки ***eclipse*** (см. содержимое сценария);
- запустить среду разработки ***EclipsePTP***, воспользовавщись соответствующим значком, расположенным на рабочем столе;
- создать проект ***ompi1***, выбрав в системе разработки соответствующий шаблон проекта.

Имеются также утилиты ***oshrun*** и ***shmemrun***, которые запускают параллельные процессы, используя разделяемую память компьютера: *SHMEM*.

В общем случае, выбор средств запуска программы ложится на плечи разработчика.

Задание:

- реализовать тестовый пример;
- исследовать запуск приложения по технологии OpenMPI;
- описать результаты в личном отчете.

3.2 Исползование OpenMPI в архитектуре ВК

В данном продразделе, сокращение ***ВК*** будет использоваться как абревиатура понятия «*вычислительный комплекс*».

Все реализации MPI обычно используются в средах «*высокопроизводительных вычислений*» (HPC).
HPC - *High-Performance Computing* — высокопроизводительные вычисления.

Интерфейс MPI, в сущности, предоставляет соединения типа ***IPC*** для программ моделирования, вычислительных алгоритмов и других приложений, требующих большого объема вычислений.
IPC - ***i****nter-****p****rocess* ***c****ommunication* — обмен данными между потоками (нитями) одного или разных процессов.
Он реализуется посредством механизмов, предоставляемых ядром ОС и реализующим новые возможности межпроцессного взаимодействия.
Взаимодействие между процессами может осуществляться как на одном компьютере, так и между несколькими компьютерами в сети или объединенными с помощью специальных адаптеров, например, ***InfiniBand***.
Основные механизмы, предоставляемые ОС и используемые ***IPC***:

- механизмы *обмена сообщениями*;
- механизмы *синхронизации*;
- механизмы *разделения памяти*;
- механизмы *удаленных вызовов* (***RPC*** -remote procedur call).

Infiniband (***IB***) — высокоскоростная коммутируемая компьютерная сеть, используемая в высокопроизводительных вычислениях, которая имеет:

- очень большую пропускную способность;
- низкую задержку, при передаче данных.

3.2.1 Архитектура плагинов OpenMPI

Ранее, на рисунке 3.1, уже была представлена многослойная архитектура ***OpenMPI***, которая реализована в виде отдельных бибилиотек.

Отдельные слои технологии *OMPI*, *ORTE* и *OLAP* имеют стандартизированный стабильный интерфейс, обеспечивающий надежную реализацию всей системы.
Для того, чтобы библиотеки системы было удобно использовать, их стали реализовывать в виде *компонент*, загружаемых во время исполнения программы.
Такие компоненты представляют собой ***динамически разделяемые объекты***, часто называемые «***DSO***» или «***плагины***».

Чтобы компоненты или «плагины» было удобно использовать, была проведена их *типизация*.
Каждый тип компонента представлен в виде *фреймворка*.

Фреймворк (***framework*** **—** каркас, структура) — программная платформа, определяющая структуру программной системы, облегчающая разработку и объединение разных компонентов большого программного проекта.

Каждый компонент технологии ***OpenMPI*** принадлежит ровно одному фреймворку, а фреймворк поддерживает ровно один вид компонента.
Набор слоев проекта Open MPI, его фреймворки и компоненты образуют модульную архитектуру компонентов - *Modular Component Architecture* (***MCA***).

На рисунке 3.3 приведена общая компоновка архитектуры проекта ***Open MPI***, на которой показаны несколько фреймворков ***Open MPI*** и некоторые из имеющихся компонентов. Остальные фреймворки и компоненты ***Open MPI*** подключены к проекту аналогичным образом.

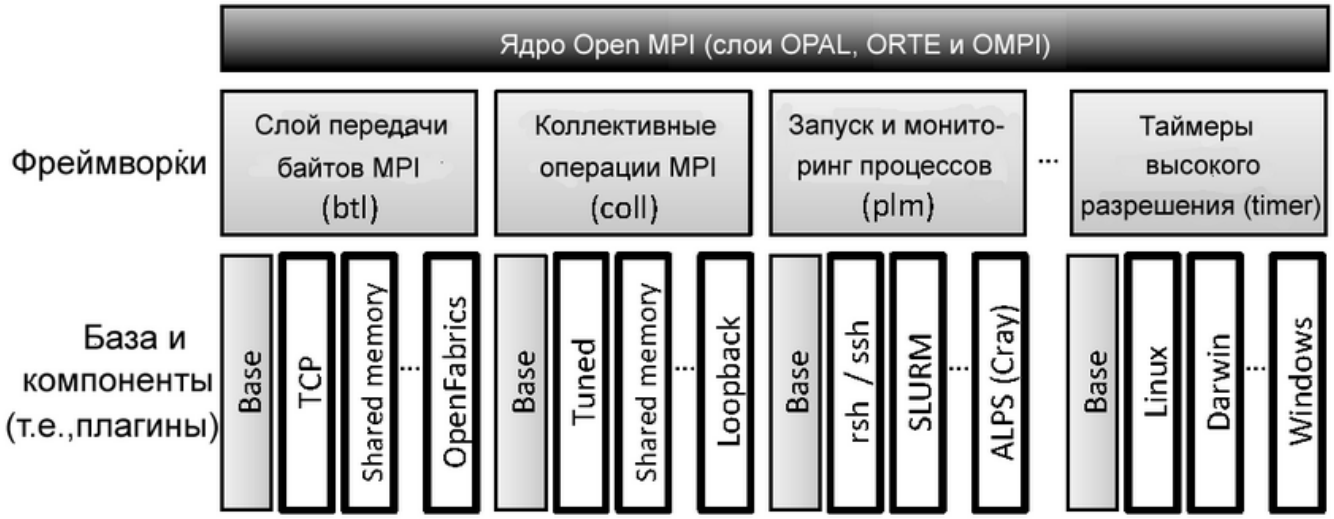

Рисунок 3.3 — Набор компонент технологии OpenMPI

Замечание

Каждый фреймворк, показанных на рисунке 3.3, содержит базовый код ***base*** и один или несколько компонентов.
Эта структура реплицируется (повторяется) в каждом из слоев.
На данном рисунке показаны примеры фреймворков всех трех слоев:

- слой *OMPI* - фреймворки ***btl*** и ***coll***;
- слой *ORTE* - фреймворк ***plm***;
- слой *OPAL* - фреймворк ***timer***.

Наличие в системе множества фреймворков предоставляет пользователям (администраторам) возможность по-разному соединять различные плагины различных типов и создавать программный стек, который наиболее эффективен в конструируемой архитектуре вычислительного комплекса.

3.2.2 Архитектура технических решений

Наиболее позитивной частью технологии ***Open MPI*** является возможность разделения *прикладной части* ПО разрабатываемой системы и ПО ее *системной части*.

Прикладная часть ПО представляет собой отдельную программу или набор различных программ, реализующих некоторый вычислительный алгоритм. Эти программы могут использовать инструментальные средства MPI или нет:

- программы, *использующие* инструментальные средства MPI, могут определить свой номер процесса (***rank***), а также передавать сообщения друг другу;
- программы, *не использующие* инструментальные средства MPI, не имеют возможности взаимодействовать друг с другом.

В любом случае, разработанные программы переносятся на те ЭВМ, на которых они будут в дальнейшем запускаться.

Системная часть ПО представляет собой инструментальные средства самого пакета ***Open MPI***. В нее входит набор библиотек и утилит.
Главная утилита системной части ПО — утилита ***mpirun***. Именно она создает отдельные *локальные и удаленные процессы*, в которых запускает прикладную часть ПО, обеспечивая его номерами процессов и средствами коммуникации.

Таким образом, технология MPI освобождает прикладного программиста от проблем, связанных с программированием низкоуровневого сетевого интерфейса и распределения нагрузки между системными средами ЭВМ, перекладывая эти заботы на архитекторов вычислительного комплекса.

Поскольку ЭВМ, входящие в вычислительный комплекс, могут иметь различные средства коммуникации, различную вычислительную нагрузку и даже различные ОС, все эти различия и дополнительные требования должны быть учтены при запуске утилиты ***mpirun***.
Для примера, приведем три из множества возможных вариантов использования утилиты ***mpirun***:

- ***запуск трех процессов на отдельной ЭВМ***

```
mpirun -np 3 программа
```

здесь три экземпляра одной программы запускаются на локальной ЭВМ и, для взаимодействия между ними, используется разделяемый «механизм» памяти;

- ***запуск трех процессов на трех разных ЭВМ***

```
mpirun -np 3 –host a,b,c программа
```

здесь три экземпляра одной программы запускаются на трех разных ЭВМ ***a***, ***b*** и ***c***, а для взаимодействия между ними, обычно, используется сеть ***TCP/IP***;

- ***запуск трех разных процессов на трех разных ЭВМ***

```
mpirun -np 1 –host a прог1 : -np 1 –host b прог2 : -np 1 –host C прог3
```

здесь три экземпляра разных программ (***прог1***, ***прог2*** и ***прог3***) запускаются на трех разных ЭВМ ***a***, ***b*** и ***c***, а для взаимодействия между ними, обычно, используется сеть ***TCP/IP***.

Утилита ***mpirun*** имеет множество ключей и вариантов запуска.
Наиболее часто, используются следующие:

- **--hostfile файл** — задает файл, в котором перечисляются имена хостов, требуемое количество используемых процессов, а также максимальное количество возможных процессоров;
- **--host список** — задает имя одного или список имен хостов, перечисленных через запятую, на которых должны запускаться процессы;
- **--np** (или **-np**) **число** — определяет старт требуемого количества процессов;
- **--mca** (или **-mca**) **список_MCA_parameters** — список параметров, уточняющих средства коммуникации запущенных процессов;
- **--wdir <directory>** - устанавливает рабочую директорию, в которой стартуют процессы. Если параметр не задан, то подразумевается текущая директория (или используется значение системной переменной ***$HOME***);
- **-x <env-variable-name>** - Установка и экспорт имени и значения системной переменной, которая будет использоваться при запуске параллельного приложения. Ключ (опция) **-x** может быть задана множество раз.

Замечание

Поскольку, имеется множество способов запуска утилиты ***mpirun***, а также множество настроек самого пакета ***Open MPI***, то следует воспользоваться перечнем ответов на вопросы, которые подробно, но в англоязычном варианте, размещены на сайте:
https://www.open-mpi.org/faq/

3.3 Учебный кластер ЭВМ

Из материала предыдущих разделов видно, что технология MPI, и в частности, технология Open MPI, - достаточно сложны для изучения.
С другой стороны, очевидно, что технология имеет достаточно большой потенциал, который можно использовать в различных прикладных проектах, требующих повышенного вычислительного ресурса.

Для целей учебного процесса, используется задание, которое имеет название «*Учебный кластер ЭВМ*».
Чтобы максимально упростить задание, кластер ЭВМ будет состоять из двух ЭВМ, объединенных сетью в пределах учебного класса кафедры АСУ.

Для выполнения лабораторной работы, студенты разделяются на проектные группы по два человека и занимают два отдельных компьютера, расположенного в одном учебном классе.

В процессе выполнения данной лабораторной работы, каждая учебная проектная группа последовательно решает следующие задачи:

- настройка сети на машинах кластера;
- настройка и тестирование работы пакета OpenSSH на отдельном компьютере;
- создание и распространение ключей SSH;
- тестирование ПО кластера.

Далее, по тексту данного методического пособия, излагается учебный материал по выполнению каждого пункта проекта.

Замечание
Поскольку компьютеры учебных классов кафедры АСУ используются для разных целей обучения, следует очень внимательно подойти к настройкам сети, чтобы в дальнейшем избежать проблем с выполнением данной работы.

3.3.1 Настройка сети

Этап 1. Проверка и настройка подключения к сети
Запустить виртуальный терминал и выполнить в нем команду:

```
sudo mc
```

которая обеспечит запуск файлового менеджера «***Midnight Commander***» от имени пользователя ***root***.

Затем, командой:

```
ip address
```

убедиться *в наличии* или *отсутствии* настроек компьютера на работу в сети.

Если ***IP-адрес*** отсутствует, то перейти в директорию ***/etc/netctl*** и убедиться в наличии файла ***ethernet-dhcp***.
Если файл ***ethernet-dhcp*** - отсутствует, то скопировать его шаблон из директории ***/etc/netctl/exemples***, после чего откорректировать в нем параметр *Interface*, который должен быть равен имени интерфеса сетевого устройства, полученного при выводе предыдущей команды.

Далее, последовательностью команд:

```
netctl stop    ethernet-dhcp
netctl disable ethernet-dhcp
netctl enable  ethernet-dhcp
netctl start   ethernet-dhcp
```

перезапустить сетевой интерфейс и убедиться в подключении компьютера к сети.

Этап 2. Установка уникального имени компьютера
Выполнить в терминале команду:

```
hostname
```

которая выведет имя компьютера.
Убедиться, что *это имя является уникальным* и больше не повторяется в именах компьютеров сети.

Замечание
Общее правило, обеспечивающее уникальность имени компьютера в сети, - использование имени, под которым студент проводит личный логин на кафедре АСУ.

Чтобы установить уникальное имя компьютера, следует:

- *подключить* к ЭВМ личный flashUSB студента;
- *отредактировать* файл ***/boot/grub/grub.cfg***, установив в первом пункте меню значение переменной ***UPK_HOST=имя-компьютера***;
- *завершить* (корректно) работу с ЭВМ;
- *включить* ЭВМ и *снова войти* в учебную систему.

Замечание
После указанных действий, проверить настройки компьютера по пунктам этапов 1 и 2.

Этап 3. Настройка файла /*etc/hosts*

Завершив первые два этапа, следует:

- *передать напарнику* по лабораторной работе свои данные об ***IP***-адресе и имени хоста (компьютера);
- *получить от напарника* его данные об ***IP***-адресе и имени хоста (сетевые данные удаленного хоста).

Отредактировать файл ***/etc/hosts***, добавив в него две строки:

```
ваш_IP_адрес имя_вашего_хоста
его_IP_адрес имя_его_хоста
```

После указанных действий, проверить взаимосвязь между компьютерами кластера командой:

```
    ping имя_его_хоста
```

В случае успешной проверки, настройка сети — закончена.

3.3.2 Настройка и тестирование OpenSSH

Технология ***Open MPI*** широко использует технологию ***OpenSSH*** для связи и запуска процессов на компьютерах кластера, которая обеспечивается запуском и настройкой серверного процесса ***sshd***.

ОС УПК АСУ содержит все необходимое программное обеспечение для работы с пакетом ***OpenSSH***, но первоначальная ее настройка не обеспечивает запуск ***sshd***, поэтому такую настройку следует осуществить.

Замечание

Неправильная настройка и запуск сервера ***sshd*** приводит к нарушению безопасности компьютера, поэтому, после запуска ЭВМ, от имени пользователя ***root*** следует выполнить две команды:

```
    systemctl disable sshd.service
    systemctl stop    sshd.service
```

которые отключат работу сервера.

Далее, работу с сервером ***sshd*** следует осуществлять в ограниченном объеме, как это используется в данной лабораторной работе.

Этап 4. Настройка серверной части технологии OpenSSH

Запустив терминал, как это показано ***на этапе 1***, и отключив работу сервера ***sshd***, как это показано в замечании, следует перейти в директорию ***/etc/ssh***, где следует удалить все файлы, кроме: *moduli*, *ssh_config* и *sshd_config*.

Отредактировать файл ***ssh_config***, установив значения параметров, как это показано на листинге 3.2, закомментировав все остальные парараметры.

Листинг 3.2 — Правильные параметры файла ssh_config

```
Host *
PasswordAuthentication no
ChallengeResponseAuthentication no
PubkeyAuthentication yes
IdentityFile ~/.ssh/id_rsa
Port 22
Protocol 2
```

Отредактировать файл ***sshd_config***, установив значения параметров, как это показано на листинге 3.3, закомментировав все остальные парараметры.

Листинг 3.3 — Правильные параметры файла sshd_config

```
AllowUsers asu upk
Port 22
HostKey /etc/ssh/ssh_host_rsa_key
PermitRootLogin yes
PubkeyAuthentication yes
AuthorizedKeyFile .ssh/authorized_keys
PasswordAuthentication no
PermitEmptyPassords yes
ChallengeResponseAuthentication no
Subsystem sftp /usr/lib/ssh/sftp-server
```

Запустить сервер ***sshd*** командой:

```
systemctl start sshd.service
```

Убедиться командой:

```
ps -ax | grep sshd
```

что сервер старовал.

Замечание

Перед стартом сервера ***sshd***, в директории ***/etc/ssh***, будут созданы все необходимые закрытые и открытые ключи сервера.
Предполагается, что удаленное подключение к ***sshd*** будет проводиться без пароля с использованием сертификатов ключей.

Этап 5. Локальная проверка ssh-соединений

Прежде чем проверять работу пакета ***OpenSSH*** в сети, это следует сделать на локальном компьютере.

Проверка будет осуществляться посредством подключения пользователя ***upk*** к пользователю ***asu***.

Для этого, пользователем upk следует открыть терминал, в котором выполнить команду:

```
ssh-keygen -t rsa
```

Замечание

При гененрации ключей, утилита ***ssh-keygen*** запросит ввод парольных фраз. На такие запросы необходимо просто нажимать клавишу ***Enter***.

В результате указанного действия появится сообщение, как это показано на рисунке 3.4, а в директории ***~/.ssh*** пользователя ***upk***, появятся два файла:

- **id_rsa** — *закрытый ключ* по сертификату *rsa*;
- **id_rsa.pub** — *открытый ключ* по сертификату *rsa*.

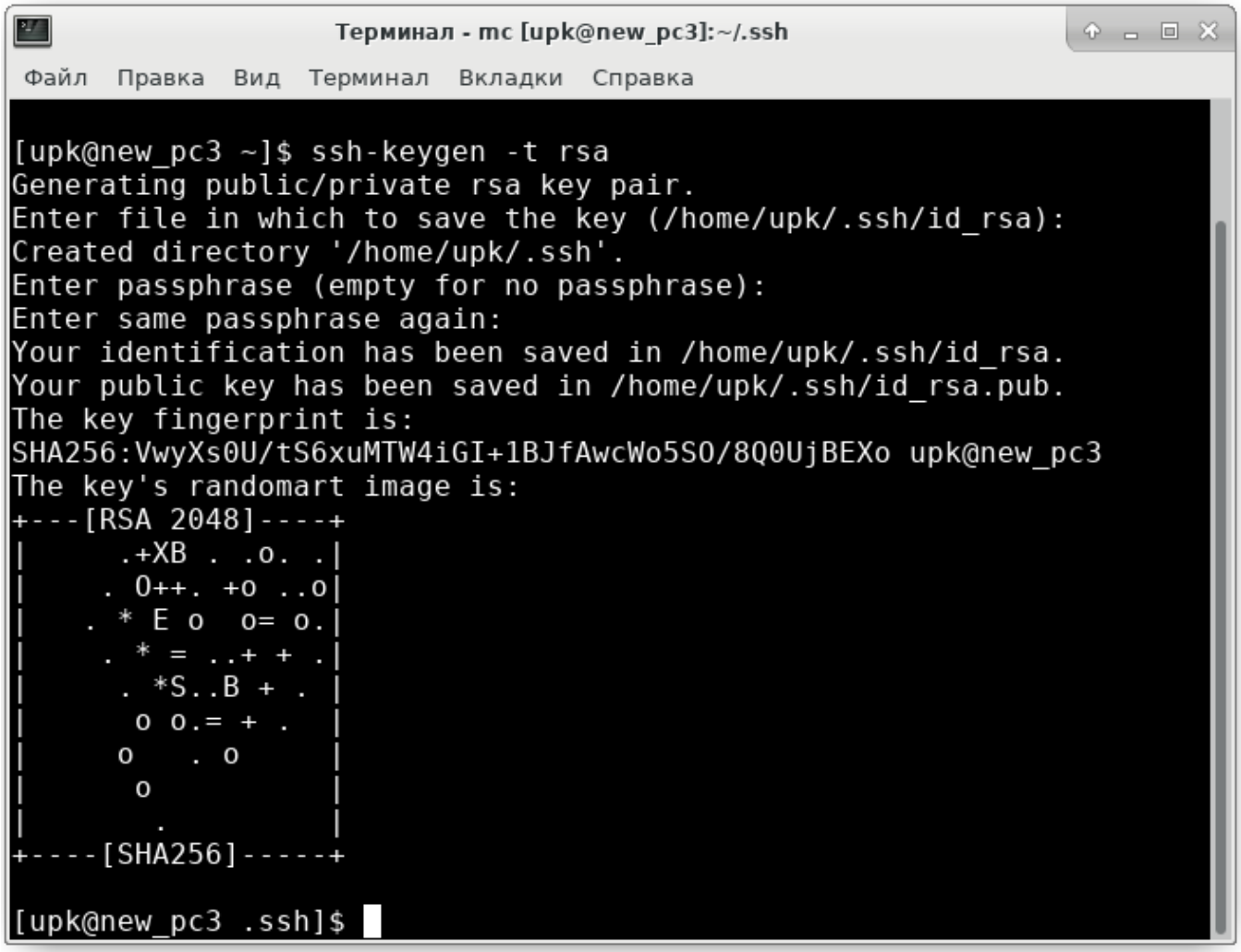

Рисунок 3.4 — Результат генерации ключей утилитой ssh-keygen

Открытый ключ, под именем ***aa.pub***, следует скопировать в директорию ***/tmp***.

Переключиться на виртуальный терминал ***/dev/tty3*** и произвести ***login*** пользователем ***asu***, после чего запустить ***mc***.
Пользователем ***asu*** скопировать файл ***/tmp/aa.pub*** в директорию ***~/.ssh*** и в этой директории выполнить команду:

```
cat ./aa.pub >> authorized_keys
```

после чего удалить файл ***aa.pub***.
В результате указанных действий, пользователь ***asu*** будет готов к подключению по ***ssh***.

Вернуться на ***/dev/tty1*** и удалить файл ***/tmp/aa.pub***.
Теперь можно провести проверку локального соединенияю

В виртуальном терминале, от имени пользователя ***upk***, выполнить команду:

```
ssh asu@имя_локального_компьютера
```

Первоначально, сервер спросит: *доверяем ли мы его сертификату?*
Необходимо ответить: ***yes***

В результате:

- пользователь ***upk*** должен зайти на компьютер пользователем ***asu***;
- в файл ***~/.ssh/known_hosts*** пользователя ***upk*** будет записан сертификат доверия указанному серверу, что в дальнейшем обеспечит полный беспарольный доступ к серверу для подключения пользователем ***asu***.

Чтобы убедиться в этом, следует выйти из логин пользователя ***asu*** и повторить соединение.

3.3.3 Создание и распространение ключей SSH

Этап 6. Распространение публичных ключей
Убедившись в осуществимости локального защищенного соединения по протоколу ***ssh***, следует:

- *передать* напарнику свой файл ***id_rsa.pub***;
- *получить* от напарника его файл ***id_rsa.pub***, который необходимо скопировать в директорию ***/home/upk/.ssh*** под любыть именем, например, ***bb.pub***;
- *выполнить* (от имени пользователя ***upk***) в директории команду:

```
cat ./bb.pub >> authorized_keys
```

- *удалить* файл ***bb.pub***.

Этап 7. Проверка защищенного удаленного соединения

После успешного проведения предыдущих действий, следует провести соединение с компьютером напарника, выполнив команду:

```
ssh upk@имя_удаленного_компьютера
```

Замечание

При первом соединении, обязательно согласие ***yes*** на запрос о доверии сертификату сервера.

3.3.4 Тестирование ПО кластера

Тестирование ПО кластера проводится только после успешного выполнения всех предыдущих этапов.

Обычно, тестирование ПО кластера осуществляется в два этапа:

- *упрощенное тестирование*, предполагающее запуск на удаленном компьютере общедоступной утилиты, не использующей технологии ***Open MPI***;
- *полное тестирование*, предполагающее запуск на удаленном компьютере программ, разработанных по технологии ***Open MPI*** и включающих, при необходимости, дополнительную настройку системной среды ОС ЭВМ.

Мы проведем только упрощенное тестирование.

Этап 8. Упрощенное тестирование ПО кластера

Запустим на локальной и удаленной ЭВМ утилиту ***hostname***, распечатывающую на терминале имя компьютера, на котором она запущена:

```
mpirun -np 1 –host имя_local hostname :  -np 1 –host имя_remote hostname
```

Желаю успеха!

Список использованных источников

1. Резник В.Г. Архитектура вычислительных комплексов. Самостоятельная и индивидуальная работа студента. Учебно-методическое пособие. – Томск, ТУСУР, 2017. – 13 с.

2. Резник В.Г. Учебный программный комплекс кафедры АСУ на базе ОС ArchLinux. Учебно-методическое пособие. – Томск, ТУСУР, 2017. – 38 с.

Учебное издание

Резник Виталий Григорьевич

АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ КОМПЛЕКСОВ

Учебно-методическое пособие предназначено для изучения темы №3 по дисциплине «Архитектура вычислительных комплексов» для студентов уровня основной образовательной программы магистратура направления подготовки: 09.04.01 «Информатика и вычислительная техника».

Учебно-методическое пособие

Усл. печ. л. . Тираж . Заказ .
Томский государственный университет
систем управления и радиоэлектроники
634050, г. Томск, пр. Ленина, 40