

Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Северо-Кавказский федеральный университет»
Невинномысский технологический институт (филиал)

Методические указания к выполнению
лабораторных работ по дисциплине
«Основы теории параллельных вычислений»
для студентов направления

15.04.04 – «Автоматизация технологических процессов и производств»

Невинномысск 2017

Настоящие методические указания предназначены для студентов направления 15.04.04 – «Автоматизация технологических процессов и производств». Они разработаны в соответствии с государственным образовательным стандартом и основной образовательной программой специальностей.

В методических указаниях рассмотрены правила создания многопоточных приложений, синхронизация доступа потоков к общему ресурсу, синхронизация процессов и протоколов в WIN 32. Приведены примеры программ на языках C++ Builder, VisualC++. Даны задания для самостоятельной работы и приведен список рекомендуемых источников литературы.

№ п/п	Содержание компетенции	Шифр
<i>Профессиональные компетенции</i>		<i>ПК-(№)</i>
1.	Способность формировать новые конкурентоспособные идеи и реализовывать их в проектах	ПК-27

Составитель

Ю.Н. Кочеров

Ответственный редактор

канд. техн. наук Д.В. Болдырев

СОДЕРЖАНИЕ

Лабораторная работа №1	4
Лабораторная работа №2	12
Лабораторная работа №3	14
Лабораторная работа №4	24
Лабораторная работа №5	26
Список источников.....	45

Лабораторная работа №1

Потоки в Windows

Для создания многопоточных приложений в среде программирования C++Builder используется абстрактный класс **TThread**, позволяющий создавать отдельные потоки процесса. Для этого необходимо создать потомка класса TThread, каждый новый потомок является новым потоком.

При организации многопоточных приложений следует придерживаться следующих правил:

- не следует организовывать слишком много потоков, рекомендуемый предел – 16 потоков на один процесс на однопроцессорном компьютере,
- когда несколько потоков используют общий ресурс, они должны синхронизироваться для избежания конфликтов.

Целью данной лабораторной работы является изучение свойств, методов и событий класса TThread для написания простых многопоточных приложений. Для повышения наглядности работы, потоки будут напрямую обращаться к свойствам и методам компонентов библиотеки VCL - меткам (Label), индикаторам процесса выполнения (ProgressBar) и кнопкам., что, вообще, не допустимо из-за потенциальной возможности возникновения проблем синхронизации доступа. Однако, учитывая учебный характер данных программ и минимальную вероятность одновременного использования ресурсов, мы сделаем допущение, что общих совместно используемых разными потоками ресурсов у нас нет и рассматривать вопросы синхронизации в данной работе не будем.

1. КЛАСС TTHREAD

СВОЙСТВА (PROPERTIES)	
FreeOnTerminate	
<i>ud:</i>	<code>B __property boolFreeOnTerminate = {read=FFreeOnTerminate, write=FFreeOnTerminate, nodefault};</code>

<i>Опис.:</i>	Определяет, уничтожится ли автоматически объект потока при завершении потока. Если <code>FreeOnTerminate = истина</code> , то объект уничтожается автоматически, в противном случае его необходимо уничтожать явно (<code>delete</code>).
<i>Прим.:</i>	<pre>void __fastcall TtestThread::Execute(){ int Result FreeOnTerminate=true; for(int i=0; i<10000; i++) Result+=sin(i); }</pre>
Handle	
<i>Вид:</i>	<code>__property int Handle = {read=FHandle, nodefault};</code>
<i>Опис.:</i>	Возвращает дескриптор потока. Необходим при использовании WinAPI функций.
<i>Прим.:</i>	<pre>P1->Resume(); int HT=P1->Handle; Form1->Label2->Caption = IntToStr(HT);</pre>
Priority	
<i>Вид:</i>	<pre>enum TThreadPriority {tpIdle, tpLowest, tpLower, tpNormal, tpHigher, tpHighest, tpTimeCritical}; __property TThreadPriority Priority = {read=GetPriority, write=SetPriority, nodefault};</pre>
<i>Опис.:</i>	Определяет относительный приоритет потока.
<i>Прим.:</i>	<pre>P1 = new IvanovP1(true); P1->Priority = tpNormal; P1->Resume();</pre>
Suspended	
<i>Вид:</i>	<code>__property bool Suspended = {read=FSuspended, write=SetSuspended, nodefault};</code>

<i>Опис.:</i>	Определение состояния, а также приостановка/продолжение выполнения потока. Если <code>Suspended = true</code> – поток приостанавливается. На <code>false</code> – продолжение выполнения.
<i>Прим.:</i>	<pre> first->Suspended=true; //Остановка выполнения потока ... first->Suspended=false; //Продолжение выполнения потока </pre>
Terminated	
<i>Вид:</i>	<code>__property bool Terminated = {read=FTerminated, nodefault};</code>
<i>Опис.:</i>	Определяет, был ли завершен поток. Метод <code>Execute</code> должен периодически проверять <code>Terminated</code> , и выходить, если <code>Terminated = true</code> . Метод <code>Terminate</code> устанавливает <code>Terminated</code> в <code>true</code> .
<i>Прим.:</i>	<pre> void __fastcall TMyThread::Execute() { while (!Terminated) { ... } } </pre>
ThreadID	
<i>Вид:</i>	<code>__property int ThreadID = {read=FThreadID, nodefault};</code>
<i>Опис.:</i>	Получение идентификатора потока.
<i>Прим.:</i>	<code>int id=first->ThreadID;</code>
МЕТОДЫ (METHODS)	
DoTerminate	
<i>Вид:</i>	<code>virtual void __fastcall DoTerminate(void);</code>
<i>Опис.:</i>	Генерирует событие <code>OnTerminate</code> , но не завершает поток.
Execute	
<i>Вид:</i>	<code>virtual void __fastcall Execute(void) = 0;</code>
<i>Опис.:</i>	Чистый виртуальный метод. Содержит исполняемый код потока.

<i>Прим.:</i>	<pre> void __fastcall IvanovP1::Execute() { float s = 0; for(int i=0;i<N;i++){ s+= sin(i) + cos(i); Form1->ProgressBar1->Position = i; if(Terminated) return; } } </pre>
Resume	
<i>Вид:</i>	<code>void __fastcall Resume(void);</code>
<i>Опис.:</i>	Выполнение или возобновление выполнения прерванного потока.
<i>Прим.:</i>	<code>first->Resume();</code>
Suspend	
<i>Вид:</i>	<code>void __fastcall Suspend(void);</code>
<i>Опис.:</i>	Приостановка выполнения потока.
<i>Прим.:</i>	<code>first->Suspend();</code>
Synchronize	
<i>Вид:</i>	<pre> typedef void __fastcall (__closure *TThreadMethod)(void); void __fastcall Synchronize(TThreadMethod&Method); </pre>
<i>Опис.:</i>	<p>Выполняет вызов метода в пределах основного потока VCL. Параметр Method имеет тип TThreadMethod (означающий процедурный метод, не использующий никаких параметров). Метод, передаваемый в качестве параметра Method, и является как раз тем методом, который затем выполняется из основного потока приложения. Synchronize используется для устранения многопоточных конфликтов. Если нет уверенности, что какой-то метод является потоко-безопасным, его следует запускать в пределах главного VCL потока посредством передачи в Synchronize.</p>

<i>Прим.:</i>	<pre> void __fastcall TMyThread::PushTheButton(void) { Button1->Click(); } void __fastcall TMyThread::Execute() { ... Synchronize(PushTheButton); ... } </pre>
Terminate	
<i>Вид:</i>	void __fastcall Terminate(void);
<i>Опис.:</i>	<p>Устанавливает свойство <code>Terminated</code> в истину, сигнализируя, что поток должен завершиться быстро, как только это возможно. В отличие от WinAPI функции <code>TerminateThread</code>, которая вынуждает поток завершиться, <code>Terminate</code> просто сигнализирует о необходимости завершения.</p>

<i>Прим.:</i>	<pre> void __fastcall IvanovP1::Execute() { float s = 0; for(int i=0;i<N;i++){ s+= sin(i) + cos(i); Form1->ProgressBar1->Position = i; if(Terminated)goto a1; } a1:; Fa = s; } //----- void __fastcall TForm1::Button5Click(TObject *Sender) { P1->Terminate(); } </pre>
TThread	
<i>Вид:</i>	__fastcall TThread(bool CreateSuspended);
<i>Опис.:</i>	<p>Конструктор. Создает образец объекта потока. Если CreateSuspended - false, метод Execute будет вызван немедленно, в противном случае поток будет создан в приостановленном состоянии и для его запуска необходимо вызвать метод Resume.</p>
<i>Прим.:</i>	<pre> TMyThread *Second = new TMyThread(true); // создать, нонезапускать Second->Priority = tpLower; // установка приоритета ниже нормального Second->Resume(); // запуск потока </pre>
WaitFor	
<i>Вид:</i>	int __fastcall WaitFor(void);

<i>Опис.:</i>	Ожидание завершения потока.
<i>Прим.:</i>	<pre> ... first->Resume(); first->WaitFor(); second->Resume(); ... </pre> <p>В данном примере два потока first и second будут выполняться последовательно. Поток second начнет выполняться только по завершении выполнения потока first.</p>
СОБЫТИЯ (EVENTS)	
OnTerminate	
<i>Вид:</i>	<code>__property TNotifyEventOnTerminate = {read=FOnTerminate, write=FOnTerminate};</code>
<i>Опис.:</i>	Событие происходит после выхода из метода Execute но прежде, чем поток будет уничтожен.
<i>Прим.:</i>	<code>p1->OnTerminate = ThreadDone;</code>

2. ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ № 1

1) Разработать программу для вычисления определенного интеграла от заданной функции на заданном отрезке методом прямоугольников. Программа должна разбивать отрезок на три равные части, запускать по выбору вычисления на каждом отрезке по 10000 значений либо параллельно, либо последовательно, обеспечивать приостановку и прерывание вычислений, установку приоритета каждого потока, по окончании вычислений сложить их результаты и получить ответ.

Вариант	Функция	Отрезок
1	$\sin(x) \cdot \cos(x^2)$	[0, 10]
2	$\ln(x) + 10 \cdot \cos(x)$	[0, 8]
3	$x^2 - 10 \cdot x + \sin(x)$	[3, 9]
4	$2^x / x^3$	[4, 10]
5	$15 \cdot \cos(2 \cdot x) / \ln(x)$	[2, 8]

6	$\ln(x) - 2 * \sin(3 * x)$	[3, 10]
7	$2^x - \lg(x)$	[1, 3]
8	$3 * x^3 / \sin(x)$	[0, 6]
9	$x^x - 10 * \sin(5 * x)$	[1, 4]

2) Разработать программу для сравнения эффективности двух заданных алгоритмов сортировки путем их одновременного запуска на случайном массиве из 50000 целых чисел. Обеспечить вывод отсортированной последовательности в файл. Программа должна отображать ход вычислений, допускать приостановку и прерывания вычислений.

Вариант	Алгоритм 1	Алгоритм 2
1	Быстрая	Обменами (пузырек)
2	Пирамидальная	Вставками
3	Слиянием	Быстрая
4	Выбором	Пирамидальная
5	Быстрая	Вставками
6	Слиянием	Обменами (пузырек)
7	Пирамидальная	Выбором
8	Обменами	Пирамидальная
9	Слиянием	Вставками

Лабораторная работа №2

Синхронизация доступа потоков к графическим компонентам. Использование потоков в задачах имитационного моделирования.

Целью данной лабораторной работы является изучение методов синхронизации доступа потоков к общему ресурсу (канве объекта PaintBox) с помощью компонентов библиотеки VCL.

СИНХРОНИЗАЦИЯ ДОСТУПА

1. Метод Synchronize класса TThread

Прямой доступ к свойствам или методам компонентов библиотеки VCL следует выполнять только из основного потока приложения. Т.е., любой код, получающий прямой доступ или обновляющий данные пользовательского интерфейса в приложении, должен выполняться в контексте основного потока.

В классе TThread определен метод Synchronize, который позволяет вызывать некоторые из методов этого класса прямо из основного потока приложения. Определение метода Synchronize имеет вид:

```
void __fastcall Synchronize(TThreadMethod&Method);
```

Параметр Method имеет тип TThreadMethod, означающий процедурный метод, не использующий никаких параметров. Метод, передаваемый в качестве параметра Method выполняется из основного потока приложения.

При первом создании вторичного потока в приложении, библиотека VCL создает и далее поддерживает скрытое окно потока в контексте своего основного потока. Единственная цель этого окна состоит в организации последовательности вызовов процедур, выполненных посредством метода Synchronize().

2. Методы, включенные в классы VCL

Некоторые объекты библиотеки VCL (TCanvas, TGraphicsObject, TCustomWinSocket, TInterfaceList, TThreadList) оснащены встроенным механизмом защиты внутренних данных – методами Lock и Unlock,

базирующимися на применении критических секций Win32 API. В данной лабораторной работе будем использовать методы Lock и Unlock класса TCanvas для синхронизации доступа к холсту объекта PaintBox.

ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ № 2

Разработать программу с использованием потоков для моделирования движения окружностей внутри прямоугольника. При столкновении окружностей со стенками и друг с другом происходит отражение по законам упругого соударения.

Программа должна содержать прямоугольник PaintBox с окружностями, должна позволять задавать начальное положение окружности, начальную скорость и направление движения, запускать и останавливать движение. Программа должна поддерживать выбор из трех методов синхронизации доступа к холсту: отсутствие синхронизации, синхронизации через метод Synchronize, синхронизации через метод Lock/Unlock.

Лабораторная работа №3

Средства синхронизации Windows 32

Для синхронизации процессов и потоков в Win32 предусмотрено четыре механизма:

- критический раздел;
- исключающий семафор (объект типа mutex);
- событие (eventobject);
- классический семафор.

Рассмотрим отдельно каждый из этих механизмов.

1. Критический раздел

Критический раздел - это часть кода, доступ к которому в данное время имеет только один поток. Другой поток может обратиться к критическому разделу, только когда первый выйдет из него.

Предположим, что потоки разделяют некоторые переменные или структуру данных. Часто эти сложные переменные или структуры данных должны быть согласованными между собой. Операционная система может прервать поток во время обновления этих переменных. В этом случае поток, который затем использует эти переменные, будет иметь дело с несогласованными данными. В результате может возникнуть ситуация, приводящая к краху программы. Средством исключения подобных ситуаций и является критический раздел. Для работы с критическими разделами используются следующие функции:

`VOID InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection)` - инициализация синхронизатора типа *критический раздел*.

`lpCriticalSection` - указатель на переменную типа `CRITICAL_SECTION`. Тип данных `CRITICAL_SECTION` является структурой, ее поля используются только Windows.

`VOID EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection)` - запрос на вход в критический раздел.

`VOID LeaveCriticalSection (LPCRITICAL_SECTION lpCriticalSection)` - выход из критического раздела.

`VOID DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection)` - удаление критического раздела (обычно при выходе из программы).

Итак, для создания критического раздела необходимо инициализировать структуру `CRITICAL_SECTION`. Создав объект `CRITICAL_SECTION`, мы можем работать с ним, т.е. можем обозначить код, доступ к которому для одновременно выполняющихся задач требуется синхронизировать. Если один поток вошел в критический раздел, то следующий поток, вызывая функцию `EnterCriticalSection` с тем же самым объектом типа `CRITICAL_SECTION`, будет задержан внутри функции. Возврат произойдет только после того, как первый поток покинет

критический раздел, вызвав функцию `LeaveCriticalSection`. В этот момент второй поток, задержанный в функции `EnterCriticalSection`, станет владельцем критического раздела, и его выполнение будет возобновлено.

Когда объект *критический раздел* больше не нужен программе, его можно удалить с помощью функции `DeleteCriticalSection`. Это приведет к освобождению всех ресурсов системы, задействованных для поддержки объекта *критический раздел*.

Заметим, что возможно определение нескольких объектов типа *критический раздел*. Если в программе имеется четыре потока, и два из них разделяют одни данные, а два других - другие, то потребуется два объекта типа *критический раздел*.

Рассмотрим такой пример. Мы хотим записывать и считывать значения из некоего глобального массива `mas`. Причем запись и считывание должны производиться двумя разными потоками. Вполне естественно, что лучше, если эти действия не будут выполняться одновременно. Поэтому введем ограничение на доступ к массиву.

Листинг 1. Ограничение доступа к массиву с использованием критических разделов

```
// Массив значений.
intmas[1000];
// Критическая секция, регулирующая доступ к массиву
CRITICAL_SECTION CritSec;
{
...
// Инициализируем критический раздел
InitializeCriticalSection(&CritSect);
// Запускаем потоки
Thread1 -> Resume;
Thread2 -> Resume;
... // Текст программы.
// Удаляем объект критического раздела
DeleteCriticalSection(&CritSec);
}

// Первый поток: запись в массив данных
{ // Запись значения в массив
// Запрос на вход в критический раздел
EnterCriticalSection(&CritSec);
// Выполнение кода в критическом разделе
for(int i = 0; i < 1000; i++)
{
    mas[i] = i;
}
```

```

// Выход из критического раздела:
// освобождаем критический раздел для доступа
// к нему других задач
LeaveCriticalSection(&CritSec);
// завершаем поток
}

// Второй поток: считывание данных из массива
{ // Считывание значений из массива
int j;
// Запрос на вход в критический раздел
EnterCriticalSection(&CritSec);
// Выполнение кода в критическом разделе
for(int i = 0; i < 1000; i++)
{
j = mas[i];
}
// Выход из критического раздела:
// освобождаем критический раздел для доступа
// к нему других задач
LeaveCriticalSection(&CritSec);
}

```

И хотя приведенный нами пример подобного ограничения (см. листинг 1) чрезвычайно упрощен, он хорошо иллюстрирует работу синхронизатора типа критический раздел: пока один поток "владеет" массивом, другой доступа к нему не имеет.

Вход в критическую секцию уже занявшим его потоком возможен любое количество раз (столько же раз необходимо выполнить операцию выхода).

2. Исключающий семафор (mutex)

Еще один вид синхронизаторов - исключающий семафор. Основное его отличие от критического раздела заключается в том, что последний можно использовать только в пределах одного процесса (одного запущенного приложения), а исключающими семафорами могут пользоваться разные процессы. Другими словами, критические разделы - это локальные объекты, которые доступны в рамках только одной программы, а исключающие семафоры могут быть глобальными объектами, позволяющими синхронизировать работу программ (т. е. разные запущенные приложения могут разделять одни и те же данные).

Рассмотрим основные функции для работы с объектами mutex.

1. Создание объекта mutex:

```
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpMutexAttributes,
BOOL bInitialOwner, LPCTSTR lpName );
```

Параметры:

lpMutexAttributes - указатель на структуру SECURITY_ATTRIBUTES (в Windows 95 данный параметр игнорируется);

bInitialOwner - указывает первоначальное состояние созданного объекта (TRUE - объект сразу становится занятым, FALSE - объект свободен);

lpName - указывает на строку, содержащую имя объекта. Имя необходимо для доступа к объекту других процессов, в этом случае объект становится глобальным и им могут оперировать разные программы. Если вам не нужен именованный объект, то укажите NULL. Функция возвращает указатель на объект mutex. В дальнейшем этот указатель используется для управления исключаящим семафором.

2. Закрытие (уничтожение) объекта mutex :

BOOL CloseHandle(HANDLE hObject)

3. Универсальная функция запроса доступа:

DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds) - универсальная функция, предназначенная для запроса доступа к синхронизирующему объекту (в данном случае к объекту mutex).

Параметры:

hHandle - указатель на синхронизирующий объект (в данном случае передается значение, возвращенное функцией CreateMutex);

dwMilliseconds - время (в миллисекундах), в течение которого происходит ожидание освобождения объекта mutex. Если передать значение INFINITE (бесконечность), то функция будет ждать бесконечно долго.

Данная функция может возвращать следующие значения:

WAIT_OBJECT_0 - объект освободился;

WAIT_TIMEOUT - время ожидания освобождения прошло, а объект не освободился;

WAIT_ABANDON - произошел отказ от объекта (т. е. процесс, владеющий данным объектом, завершился, не освободив объект). В этом случае система (а не "процесс-владелец") переводит объект в свободное состояние. Такое освобождение объекта не предполагает гарантий защищенности данных;

WAIT_FAILED - произошла ошибка.

4. Освобождение объекта mutex:

BOOL ReleaseMutex(HANDLE hMutex) - освобождает объект mutex, переводя его из занятого в свободное состояние.

Посмотрим, как выглядит наш пример с критическими разделами, если переписать его, используя исключаящие семафоры.

Листинг 2. Ограничение доступа к массиву с использованием исключаящих семафоров

```
// Массив значений.
```

```
int mas[1000];
```

```
// Объект, регулирующий доступ к разделяемому коду.
```

```
HANDLE CritMutex;
```

```

{
...
// Инициализируем семафор разделяемого кода.
CritMutex = CreateMutex(NULL,FALSE,NULL);
... // Текст программы.
// Закрываем объект доступа к разделяемому коду.
CloseHandle(CritMutex);
}

// Первый поток: запись в массив данных.
{ // Запись значений в массив.
// Запрос на вход в защищенный раздел.
DWORD dw = WaitForSingleObject(CritMutex,INFINITE);
if(dw == WAIT_OBJECT_0)
{ // Если объект освобожден корректно, то
// выполнение кода в защищенном разделе.
for(int i = 0;i<1000;i++)
{
mas[i] = i;
}
}

// Выход из защищенного раздела:
// освобождаем объект для доступа
// к защищенному разделу других задач.
ReleaseMutex(CritMutex);
}

// Второй поток: считывание данных из массива.
{ // Считывание значений из массива.
int j;
// Запрос на вход в защищенный раздел.
DWORD dw = WaitForSingleObject(CritMutex,INFINITE);
if(dw == WAIT_OBJECT_0)
{ // Если объект освобожден корректно, то
// выполнение кода в защищенном разделе.
for(int i = 0;i<1000;i++)
{
j = mas[i];
}
}

// Выход из защищенного раздела:
// освобождаем объект для доступа
// к защищенному разделу других задач.
ReleaseMutex(CritMutex);
}

```

```
}  
}
```

Исключающий семафор может быть занят неограниченное количество раз одним и тем же потоком.

3. События

Рассмотрим теперь механизм событий. Объект *событие* может быть либо свободным (signaled) или установленным (set), либо занятым (non-signaled) или сброшенным (reset).

Создать объект *событие* можно с помощью функции:

```
HANDLE CreateEvent(  
LPSECURITY_ATTRIBUTES lpEventAttributes, // SD  
BOOL bManualReset, // reset type  
BOOL bInitialState, // initial state  
LPCTSTR lpName // object name  
);
```

Первый параметр (указатель на структуру типа SECURITY_ATTRIBUTES) и последний параметр (имя объекта *событие*) имеют смысл только в том случае, когда объект *событие* разделяется между процессами. В случае с одним процессом эти параметры обычно имеют значение NULL.

Значение параметра fInitial устанавливается равным TRUE, чтобы объект *событие* был изначально свободным, или FALSE, чтобы он был занятым. Параметр fManual будет описан немного позже.

Для того, чтобы сделать свободным объект *событие*, нужно вызвать функцию:

```
SetEvent (hEvent);
```

Чтобы сделать объект *событие* занятым, нужно вызвать функцию:

```
ResetEvent (hEvent);
```

Чтобы сделать объект *событие* свободным, а затем сразу занятым, вызывается функция:

```
PulseEvent (hEvent);
```

Обычно программа вызывает функцию:

```
WaitForSingleObject (hEvent, dwTimeOut);
```

где второй параметр имеет значение INFINITE. Возврат из функции происходит немедленно, если объект *событие* свободен. В противном случае поток будет приостановлен, пока *событие* не станет свободным.

Если параметр fManual при вызове функции SetEvent имеет значение FALSE, то объект *событие* автоматически становится занятым, когда осуществляется возврат из функции WaitForSingleObject. Освобождается только один поток. Эта особенность позволяет избежать использования функции ResetEvent.

4. Механизм семафоров

В Win32 реализован также механизм классических семафоров. Для работы с классическими семафорами используются следующие функции.

Функция создания объекта типа семафор:

```
HANDLE CreateSemaphore( LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
LONG InitialCount, LONG IMaximumCount, LPCTSTR lpName );
```

Параметры:

lpSemaphoreAttributes - указатель на структуру SECURITY_ATTRIBUTES , который определяет, может ли возвращаемый дескриптор наследоваться порожденными процессами. Если NULL, то дескриптор не может наследоваться.

InitialCount - определяет начальное значение семафора, которое должно быть не меньше нуля и не больше IMaximumCount. Семафор установлен, если его значение больше нуля, и не установлен, если его значение равно нулю. Счетчик семафора увеличивается при вызове функции ReleaseSemaphore.

IMaximumCount - определяет максимальное значение семафора;

lpName - указывает на строку, определяющую имя семафора, если NULL, то семафор открывается без имени.

Функция возврата дескриптора существующего именованного семафора:

```
HANDLE OpenSemaphore(
    DWORD dwDesiredAccess, // access flag
    BOOL bInheritHandle,   // inherit flag
    LPCTSTR lpName         // pointer to semaphore-object name
);
```

Функция увеличения счетчика семафора на заданное число:

```
BOOL ReleaseSemaphore( HANDLE hSemaphore, LONG
IReleaseCount, LPLONG lpPreviousCount );
```

Параметры:

hSemaphore - дескриптор семафора;

IReleaseCount - определяет, на сколько увеличивать счетчик семафора;

lpPreviousCount - указатель на 32-битную переменную с предыдущим значением счетчика (NULL, если предыдущее значение не требуется).

Функция WaitForSingleObject определяет, свободен ли семафор, и если он свободен, то уменьшает значение счетчика семафора на 1, в противном случае поток будет приостановлен, пока семафор не освободится.

Посмотрим, как выглядит пример с критическими разделами, если переписать его, используя классические семафоры (см. листинг 3).

Листинг 3. Ограничение доступа к массиву с использованием классических семафоров

```
// Массив значений.
int mas[1000];
// Семафор, регулирующий доступ к критическому разделу.
HANDLE Sema;
{
...
// Создаем семафор;
```

```

Semaphore = CreateSemaphore(NULL, 1, 1, NULL);
// Запускаем потоки

// Удаляем семафор
CloseHandle(Semaph);
}

// Первый поток: запись в массив данных.
{ // Запись значения в массив.
// Запрос на вход в критический раздел.
WaitForSingleObject(Semaph, INFINITE);
// Выполнение кода в критическом разделе.
for(int i = 0; i < 1000; i++)
{
    mas[i] = i;
}
// Выход из критического раздела:
// освобождаем семафор для доступа
// других задач к критическому разделу
ReleaseSemaphore(Semaph, 1, NULL);
// завершаем поток
}
// Второй поток: считывание данных из массива.
{ // Считывание значений из массива.
int j;
// Запрос на вход в критический раздел.
WaitForSingleObject(Semaph, INFINITE);
// Выполнение кода в критическом разделе.
for(int i = 0; i < 1000; i++)
{
    j = mas[i];
}
// Выход из критического раздела:
// освобождаем семафор для доступа
// других задач к критическому разделу
ReleaseSemaphore(Semaph, 1, NULL);
// завершаем поток
}

```

5. Ожидание нескольких объектов синхронизации

Функция **WaitForMultipleObjects** возвращает управление, когда выполняется одно из следующих условий:

- Все указанные в ней объекты находятся в свободном состоянии.
- Истек интервал ожидания.

```

DWORD WaitForMultipleObjects(
DWORD nCount, // Количество дескрипторов в массиве

```

```
CONSTHANDLE*lpHandles, // Массив дескрипторов
BOOLbWaitAll, // Опция ожидания
DWORDdwMilliseconds // Интервал ожидания
);
```

bWaitAll определяет тип ожидания. Если он равен TRUE, то функция возвращает управление, когда состояние всех объектов, указанных в массиве дескрипторов установлено. Если FALSE, то возврат происходит, если хотя бы один объект освободится. Возвращаемое значение позволяет определить, какой именно объект вызвал возврат.

В случае успеха, если *bWaitAll*=TRUE, функция возвращает WAIT_OBJECT_0, если *bWaitAll*=FALSE, функция возвращает WAIT_OBJECT_0 + *nCount* - 1, где *nCount* - порядковый номер объекта, вызвавшего возврат.

ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ № 3

1. Задача об обедающих философх: на круглом столе находятся k тарелок с едой, между которыми лежит столько же вилок, $k=4, \dots, 6$. В комнате имеется k философов, чередующих философские размышления с принятием пищи. За каждым философом закреплена своя тарелка; для еды философу нужны две вилки, причем он может использовать только вилки, примыкающие к его тарелке. Требуется так синхронизировать философов, чтобы каждый из них мог получить за ограниченное время доступ к своей тарелке. Предполагается, что длительности еды и размышлений философа конечны, но заранее неопределенны (могут быть выбраны случайным образом из некоторого диапазона).

Решите задачу с использованием

а) Многопоточного приложения – синхронизация методом активного ожидания через разделяемые логические переменные.

б) Многопоточного приложения – синхронизация методом активного ожидания с использованием алгоритма билета.

в) Многопоточного приложения – синхронизация с помощью критических секций `TCriticalSection`.

г) Нескольких процессов (приложений) – с использованием именованных мьютексов.

Программы должны выводить на экран состояние работы и записывать протокол работы в файл.

2. Задача о производителях и потребителях: Найти все простые числа на интервале от 2 до N , которые являются палиндромами (читаются слева направо так же как справа налево). В этой задаче потоки-производители записывают в кольцевой буфер некоторого размера найденные числа-палиндромы, а потоки-потребители проверяют, являются ли эти числа простыми и записывают их в файл.

Решите задачу с использованием семафорной синхронизации.

Лабораторная работа №4

Технология OpenMP.

ВАРИАНТЫ ЗАДАНИЙ

Алгоритмы должны реализовываться с помощью распределенных вычислений. Создается СОМ-сервер, содержащий объект для выполнения элементарных действий алгоритма. Далее создается клиентская программа, управляющая распределенными вычислениями.

1. Напишите программу для вычисления интеграла на отрезке для заданной функции (аналогично лабораторной работе №1).

Рассмотрите следующие варианты:

а) Разбиение вручную отрезка на несколько частей и запуск вычислений в параллельных регионах.

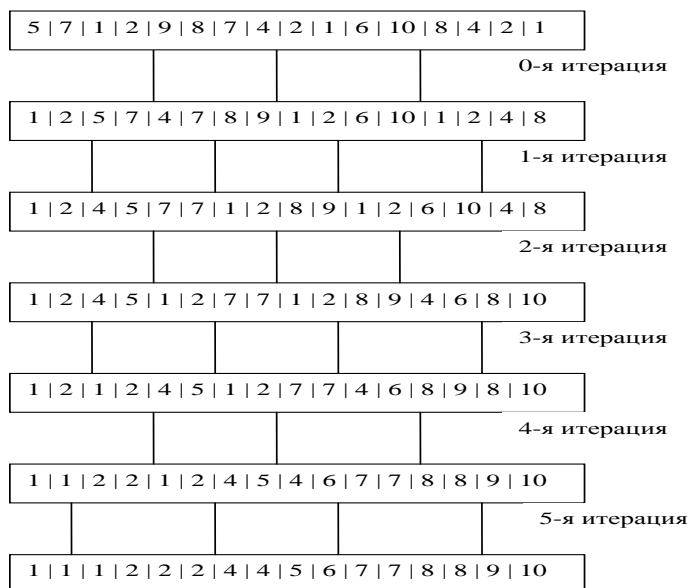
б) Параллельное вычисление с синхронизацией.

в) Параллельное вычисление с редукцией.

2. Напишите программу для вычисления произведения квадратных матриц. По аналогии напишите программу поиска минимальных путей в графе методом Флойда.

3. Параллельная сортировка

А) Сортировка транспозициями



Пример сортировки транспозициями 16-элементного массива за 6 итераций

При сортировке N -элементного массива M клиентами i -му из них на четных итерациях передается отрезок массива из N/M элементов с номерами $(N/M)(i-1)+1, \dots, (N/M)i$, $i=1, M$, на нечетных – с номерами $(N/M)(i-1)+1+N/(2M), \dots, (N/M)i+N/(2M)$, $i=1, M-1$. После сортировки каждый клиент возвращает обратно свою часть. При сортировке следует учитывать, что на всех итерациях, кроме 0-й (первой хронологически) получаемая каждым клиентом часть состоит из двух равных частей – старшей и младшей – уже отсортированных. На рисунке приведен пример сортировки.

Б) Сортировка Шелла

Необходимо реализовать параллельную сортировку Шелла. Идея алгоритма состоит в обмене элементов, расположенных не только рядом, но и далеко друг от друга, что значительно сокращает общее число операций перемещения элементов. Как пример рассматривается файл из $2n$ элементов. На j -м шаге файл делится на 2^j групп по 2^{n-j} элементов в каждой, при этом i -я группа содержит элементы $(i, i+1*2^{n-j}, i+2*2^{n-j}, \dots, i+(2^j-1)*2^{n-j})$, $i=1, \dots, 2^j$, $j=n-1, \dots, 1$ (номер шага убывает от $n-1$ до 1). На каждом шаге выполняется сортировка в каждой группе. При сортировке в каждой группе следует иметь в виду, что элементы с четными (нечетными) номерами образуют уже отсортированную последовательность, то есть для сортировки группы надо слить эти две отсортированные подпоследовательности.

В) Сортировка слиянием

Алгоритмы слияния основываются на процедуре слияния двух уже отсортированных отрезков массива в один. При параллельной сортировке весь процесс разбивается на итерации. На первой итерации массив разбивается на фрагменты, общее количество которых превышает количество клиентов в два раза. Затем каждая пара фрагментов сливается одним из процессов, после чего фрагмент записывается в сортируемый массив. Все следующие итерации идентичны первой за исключением уменьшения работающих процессов вдвое на каждой из итераций. Последняя пара фрагментов может быть слита на сервере.

Лабораторная работа №5

Технология MPI. Распределенные вычисления.

Целью настоящей работы является освоение работы с библиотекой MPI в среде разработки Visual C++ и исследование способов создания параллельных программ, работающих на нескольких компьютерах, соединенных локальной сетью и использующих обмен сообщениями для коммуникации между своими частями, расположенными на разных рабочих станциях сети.

Как правило, программирование для сетевых кластеров отличается от привычной модели программирования для многозадачных систем, построенных на базе одного или множества процессоров. Часто в таких системах необходимо обеспечить только синхронизацию процессов, и нет нужды задумываться об особых способах обмена информацией между ними, т.к. данные обычно располагаются в общей разделяемой памяти. В сети реализация такого механизма затруднительно из-за высоких накладных расходов, связанных с необходимостью предоставлять каждому процессу копию одной и той же разделяемой памяти для работы. Поэтому, обычно, при программировании для сетевых кластеров используется SPMD-технология (SingleProgram – MultipleData, одна программа – множественные данные). Идея SPMD в том, чтобы поделить большой массив информации между одинаковыми процессами, которые будут вести обработку своей части данных (рис. 1).

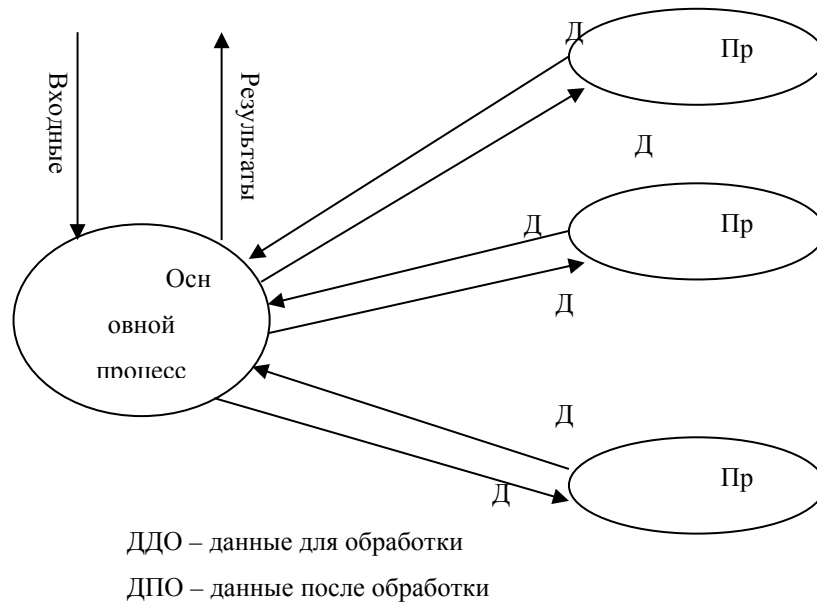


Рис. 1. Схема взаимодействия частей SPMD-программы.

В случае SPMD-подхода достаточно рассылать время от времени процессам блоки данных, которые требуют трудоемкой обработки, а затем собирать результаты их работы. Если время обработки блока данных одной машиной значительно больше, чем время пересылки этого блока по сети, то сетевая кластерная система становится очень эффективной.

Именно такой подход используется в MPI. Здесь всегда есть основной процесс, который производит распределение данных по другим машинам, а после окончания вычислений собирает результаты и показывает их пользователю. Обычно процесс-мастер после распределения данных также выполняет обработку их части, чтобы использовать ресурсы системы наиболее эффективно.

Как уже было сказано выше, MPI является стандартом обмена сообщениями. Он обеспечивает низкоуровневые коммуникации и синхронизацию процессов в сетевом кластере.

По сути дела, каждое сообщение представляет собой пакет типизированных данных, который один процесс может отправить другому процессу или группе процессов. Все сообщения обладают идентификатором

(не обязательно уникальным), который задается программистом и служит для идентификации типа сообщения на принимающей стороне.

MPI поставляется в виде библиотеки, подключаемой к среде программирования. Самыми распространенными являются библиотеки для языков Си и Fortran. Также частью MPI является резидент, который запускается на каждой машине кластера и, отвечая на запросы процессов, позволяет им взаимодействовать в сети, а также осуществляет начальный запуск всех процессов, участвующих в расчете и составляющих SPMD-программу, на указанных в файле конфигурации задачи машинах кластера.

Каждый MPI-процесс в пределах SPMD-программы уникально идентифицируется своим номером. Допускается объединять процессы в группы, которые могут быть вложенными. Внутри каждой группы все процессы перенумерованы, начиная с нуля. С каждой группой ассоциирован свой коммуникатор (уникальный идентификатор). Поэтому при осуществлении пересылки данных необходимо указать наряду с номером процесса и идентификатор группы, внутри которой производится эта пересылка. Все процессы изначально содержатся в группе с предопределенным идентификатором `MPI_COMM_WORLD`, который заводится для каждой запускаемой SPMD-программы (рис. 2). Разные SPMD-программы не знают о существовании друг друга и обмен данными между ними невозможен.

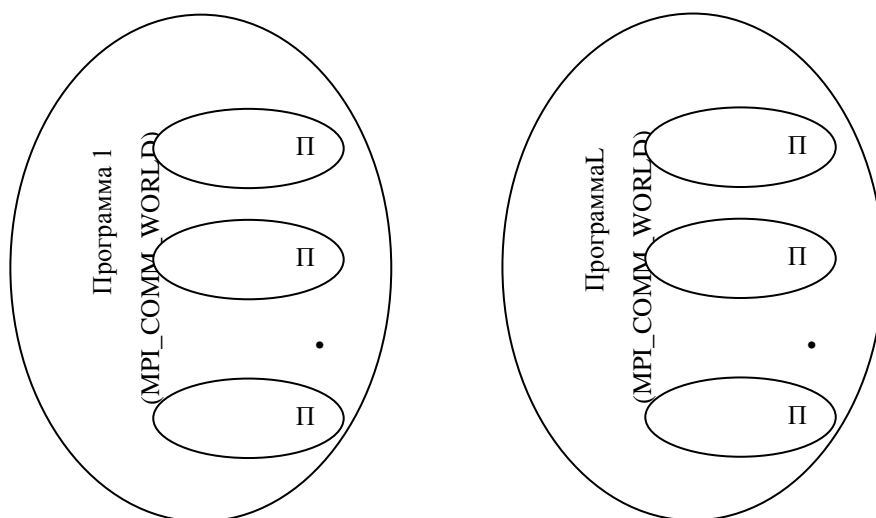


Рис 2. Нумерация процессов и коммутаторы в MPI

Рассмотрим основные функции библиотеки. Все они возвращают целое значение, которое либо равно константе `MPI_SUCCESS`, либо содержит код произошедшей ошибки.

Перед тем, как программа сможет работать с функциями библиотеки необходимо инициализировать MPI с помощью функции:

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

В качестве аргументов этой функции передаются параметры командной строки, поступающие в функцию `main()`.

Если процессу MPI необходимо узнать свой порядковый номер в группе, то используется функция:

```
intMPI_Comm_rank (MPI_Comm, int *rank)
```

Ее второй параметр будет выходным, содержащим номер процесса в указанной в первом параметре группе.

Чтобы узнать размер группы (количество процессов в ней) необходимо применить функцию:

```
intMPI_Comm_size (MPI_Comm, int *ranksize)
```

Для окончания работы с MPI необходимо использовать функцию:

```
intMPI_Finalize()
```

Перед тем, как начать рассмотрение функций передачи/приема сообщений, отметим, какие основные типы данных поддерживает MPI (таблица 1).

Таблица 1. Соответствие типов в MPI и языке Си

Тип MPI	Тип Си
<code>MPI_CHAR</code>	<code>char</code>
<code>MPI_BYTE</code>	<code>unsigned char</code>
<code>MPI_SHORT</code>	<code>short</code>
<code>MPI_INT</code>	<code>int</code>

MPI_LONG	long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_UNSIGNED_C HAR	unsigned char
MPI_UNSIGNED_SH ORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LO NG	unsigned long
MPI_LONG_DOUBL E	long double

Для передачи сообщений в MPI используется ряд функций, которые работают синхронно и асинхронно. В первом случае функция не возвращает управления, пока не завершит свою работу, во втором – сразу возвращает управление, инициировав соответствующую операцию, затем с помощью специальных вызовов есть возможность проконтролировать ход выполнения асинхронной отправки или приема данных.

Рассмотрим эти функции подробнее.

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

Функция выполняет синхронную (с блокировкой) отсылку сообщения с идентификатором msgtag (идентификатор выбирается самостоятельно программистом!), состоящего из count элементов типа datatype, процессу с номером dest и коммуникатором comm. Все элементы сообщения расположены подряд в буфере buf. Тип передаваемых элементов datatype должен указываться с помощью определенных констант типа (таблица 1). Разрешается передавать сообщение самому себе.

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. 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. В MPI имеется ряд специальных функций отправки сообщения, устраняющих подобную неопределенность, которые подробно описаны в литературе.

```
intMPI_Recv(void* buf, intcount,  
MPI_Datatype datatype, intsource,  
intmsgtag, MPI_Commcomm,  
MPI_Status *status)
```

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.

Блокировка гарантирует, что после возврата из подпрограммы 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.

Параметр status содержит служебную информацию о ходе приема, которая может быть использована в программе. status является структурой и содержит поля: MPI_SOURCE, MPI_TAG и MPI_ERROR (источник сообщения, идентификатор сообщения и возникшая ошибка, соответственно).

```
intMPI_Isend(void *buf, intcount,  
MPI_Datatypesdatatype, intdest,  
intmsgtag, MPI_Commcomm,  
MPI_Request *request)
```

Передача сообщения, аналогичная MPI_Send, однако, возврат из подпрограммы происходит сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере buf. Это означает, что нельзя повторно использовать данный буфер для других целей без получения дополнительной информации о завершении данной посылки. Окончание процесса передачи (т.е. тот момент, когда можно вновь использовать буфер buf без опасения испортить передаваемое сообщение) можно определить с помощью параметра request (идентификатор асинхронной операции определенного в MPI типа MPI_Request) и процедур MPI_Wait и MPI_Test, которые будут рассмотрены ниже.

Сообщение, отправленное любой из процедур MPI_Send и MPI_Isend, может быть принято как функцией MPI_Recv, так и MPI_Irecv.

```
intMPI_Irecv(void *buf, intcount,  
MPI_Datatypesdatatype, intsource,  
intmsgtag, MPI_Commcomm,  
MPI_Request *request)
```

Прием сообщения, аналогичный MPI_Recv, однако возврат из подпрограммы происходит сразу после инициализации процесса приема без ожидания получения сообщения в буфере buf. Окончание процесса приема можно определить с помощью параметра request и процедур MPI_Wait и MPI_Test.

```
intMPI_Wait(MPI_Request *request, MPI_Status *status)
```

С помощью этой функции производится ожидание завершения асинхронных процедур MPI_Isend или MPI_Irecv, ассоциированных с идентификатором request. Вслучаеприема, параметрысообщенияпоказываютсяв status.

```
intMPI_Test(MPI_Request *request, int *flag,  
MPI_Status *status)
```


Проверка завершенности асинхронных процедур `MPI_Isend` или `MPI_Irecv`, ассоциированных с идентификатором `request`. В параметре `flag` возвращает значение 1, если соответствующая операция завершена, и значение 0 в противном случае. Если завершена процедура приема, параметры сообщения оказываются в `status`. `MPI_Test` не блокирует программу до окончания соответствующих операций приема/передачи, чем и отличается от `MPI_Wait`.

`MPI` позволяет работать с некоторыми операциями коллективного взаимодействия. В операциях коллективного взаимодействия процессов участвуют все процессы коммутатора. Соответствующая процедура должна быть вызвана каждым процессом, быть может, со своим набором параметров. Возврат из процедуры коллективного взаимодействия может произойти в тот момент, когда участие процесса в данной операции уже закончено. Как и для блокирующих процедур, возврат означает то, что разрешен свободный доступ к буферу приема или отправки, но не означает ни того, что операция завершена другими процессами, ни даже того, что она ими начата (если это возможно по смыслу операции).

```
intMPI_Bcast(void *buf, int count,  
MPI_Datatype datatype, int source,  
MPI_Comm comm)
```

Рассылка сообщения от процесса `source` всем процессам, включая рассылающий процесс. При возврате из процедуры содержимое буфера `buf` процесса `source` будет скопировано в локальный буфер процесса. Значения параметров `count`, `datatype` и `source` должны быть одинаковыми у всех процессов.

```
intMPI_Gather(void *sbuf, int scount,  
MPI_Datatype stype, void *rbuf,  
int rcount, MPI_Datatype rtype,  
int dest, MPI_Comm comm)
```

Сборка данных со всех процессов в буфере `rbuf` процесса `dest`. Каждый процесс, включая `dest`, посылает содержимое своего буфера `sbuf` процессу

dest. Собирающий процесс сохраняет данные в буфере rbuf, располагая их в порядке возрастания номеров процессов. Параметр rbuf имеет значение только на собирающем процессе и на остальных игнорируется, значения параметров count, datatype и dest должны быть одинаковыми у всех процессов.

```
intMPI_Scatter(void *sbuf, intscout,  
MPI_Datatypestype, void *rbuf,  
intrcount, MPI_Datatypesstype,  
intsource, MPI_Commcomm)
```

Обратная к MPI_Gather функция. Процесс source рассылает порции данных из массива sbuf всем процессам группы, на которую указывает коммуникатор comm. Можно считать, что массив sbuf делится на n равных частей, состоящих из scout элементов типа stype, после чего i-я часть посылается i-му процессу. На процессе source существенным являются значения всех параметров, а на остальных процессах – только значения параметров rbuf, rcount, rtype, source и comm. Значения параметров source и comm должны быть одинаковыми у всех процессов.

Синхронизация MPI-процессов может быть выполнена с использованием блокирующих процедур приема/посылки сообщений или посредством функции, реализующей барьерную синхронизацию:

```
intMPI_Barrier(MPI_Commcomm)
```

MPI_Barrier блокирует работу процессов, вызвавших данную процедуру, до тех пор, пока все оставшиеся процессы группы comm также не выполнят эту процедуру.

MPI продолжает активно развиваться и совершенствовать механизмы работы в сетевом кластере. Так, начиная с версии 2.0 [3], в MPI появилась возможность запускать новые процессы из уже исполняющихся MPI-программ. Это необходимо для реализации модели обработки информации MPMD (MultipleProgram – MultipleData). Дляэтогоиспользуетсяпроцедура:

```
intMPI_Comm_spawn(char *command, char *argv[],  
intmaxprocs, MPI_Infoinfo,
```

```
introot, MPI_Commcomm,  
MPI_Comm *intercomm,  
intarray_of_errcodes[ ] )
```

Эта подпрограмма запускает `maxprocs` процессов, обозначенных командой `command` с аргументами, находящимися в массиве строк `argv`. В зависимости от реализации стандарта система может запустить меньшее количество процессов или выдать ошибки при невозможности запустить `maxprocs` процессов.

Нами рассмотрены самые основные функции MPI, составляющие менее половины всех имеющихся. Для более подробного изучения данной библиотеки можно использовать документацию по данному стандарту [1-3].

Для программирования с использованием MPI в среде Visual C++ в Windows мы будем использовать свободно распространяемую библиотеку WinMPI версии 1.3, которая в полной мере реализует стандарт MPI 1.1. Для данной версии справедливо все сказанное выше за исключением возможности динамического запуска процессов MPI.

Первым делом необходимо установить WinMPI в любую директорию (по умолчанию это `C:\WMPI1.3`) Для подключения компьютера к кластеру требуется запустить системную часть MPI одним из двух вариантов: в виде службы или консольного приложения.

Первый вариант пригоден для ОС типа WindowsNT, Windows 2000, WindowsXP и позволяет установить системную службу. Для этого необходимо войти в папку `SYSTEM\serviceNT`, находящуюся в каталоге установки MPI, и запустить командный файл `install_service.bat`, чтобы зарегистрировать службу в системе. Для запуска службы можно использовать или соответствующий апплет операционной системы или командный файл `start_service.bat`. Остановка сервиса осуществляется командным файлом `stop_service.bat`, а его удаление из системы – `remove_service.bat`. **Установка сервиса выполняется один раз**, а запуск по мере необходимости.

Второй вариант можно использовать на любой ОС семейства Windows и он предполагает запуск консольного приложения поддержки MPI. Для этого необходимо в папке SYSTEM\daemon запустить командный файл verbose daemon.bat.

Для корректной работы компиляции программ, работающих с библиотекой, в среде Visual C++ требуется наличие в каталоге проекта или в каталоге, указанном в переменной PATH, следующих файлов: Mpi.h, Mpi_errno.h, Mpi++.h, Mpi++P.h, cdvlibf.lib, cvwmpi.lib, binding.h. Кроме того, необходимо подключить директивой компилятора *#include* заголовочный файл mpi.h и указать в настройках линковщика библиотеку cvwmpi.lib (рис. 3).

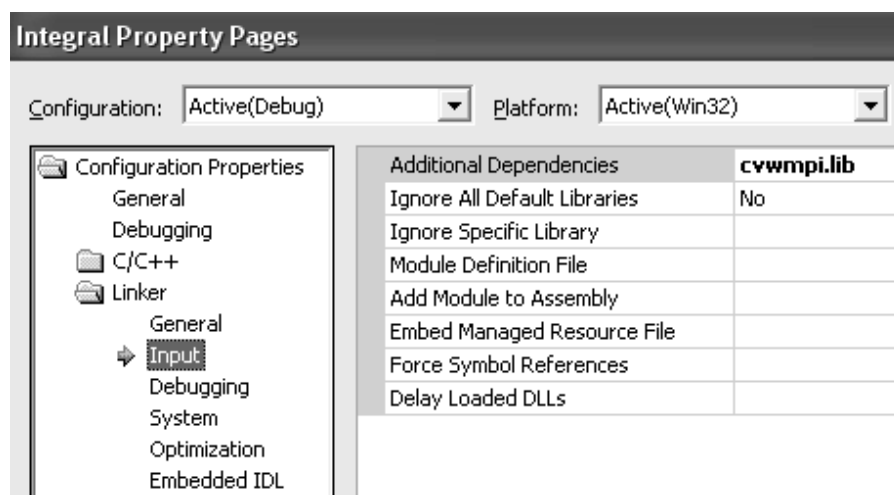


Рис. 3. Окно настроек линковщика проекта VisualC++ .NET

Перед запуском MPI-программы необходимо создать конфигурационный файл с именем, соответствующим имени созданного компилятором исполняемого файла, и расширением .rg, поместив его в той же директории, что и исполняемый файл. В конфигурационном файле предполагается использование двух основных директив.

```
local<количество>
```

В данной директиве указывается количество процессов MPI-программы, запускаемых на машине, откуда производится ее порождение (запуск процесса-мастера). Реально количество запускаемых процессов на 1 больше, т.к. процесс-мастер запускается всегда.

`<IP-адрес>|<DNS-имя><кол-во><путь к exe-файлу>`

Эта директива указывает, на какой рабочей станции сети, в каком количестве и какой файл, относящийся к MPI-программе, запускать.

Пример:

local 1

192.168.27.5 1 c:\integral.exe

Первая строчка указывает, что на основной машине необходимо запустить 2 (1+родительский процесс с номером 0) экземпляра процесса и на машине с адресом 192.168.27.5 запустить 1 экземпляр процесса, порожденного из файла integral.exe, хранящегося на ее диске C в корневом каталоге.

Иницируется запуск MPI-процессов запуском одного exe-файла на основной машине, все действия по порождению остальных процессов MPI выполняет самостоятельно.

Пример выполнения лабораторной работы в VisualC++

MPI-программа вычисления определенного интеграла

Приведенная ниже программа представляет собой консольное приложение Windows, вычисляющее определенный интеграл $\int_0^{100} x^2 dx$ методом Гаусса (прямоугольников) с шагом 10^{-7} .

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

const double a=0.0; //Нижнийпредел
const double b=100.0; //Верхнийпредел
const double h=0.0000001; //Шагинтегрирования

double fnc(double x) //Интегрируемаяфункция
```

```

{
return x*x;
}

int _tmain(int argc, _TCHAR* argv[])
{
int myrank, ranksize, i;
MPI_Status status;
    MPI_Init(&argc, &argv); //Инициализация MPI
//Определяем свой номер в группе:
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    //Определяем размер группы:
MPI_Comm_size(MPI_COMM_WORLD, &ranksize);
double cur_a, cur_b, d_ba, cur_h;
if(!myrank)
{ //Это процесс-мастер
    //Определяем размер диапазона для каждого
процесса:
    d_ba=(b-a)/ranksize;
//Оставляем первый диапазон для мастера:
cur_a=a+d_ba;
    cur_h=h;
//Рассылаем исходные данные подчиненным процессам:
for(i=1;i<ranksize;i++)
    {
cur_b=cur_a+d_ba-h;
MPI_Send(&cur_a, 1, MPI_DOUBLE, i, 98,
        MPI_COMM_WORLD);
MPI_Send(&cur_b, 1, MPI_DOUBLE, i, 99,
        MPI_COMM_WORLD);

```

```

MPI_Send(&cur_h, 1, MPI_DOUBLE, i, 100,
        MPI_COMM_WORLD);
cur_a+=d_ba;
    }
    cur_a=a;cur_b=a+d_ba-h;
}
else
{ //Это один из подчиненных процессов
//Получаем исходные данные:
MPI_Recv(&cur_a, 1, MPI_DOUBLE, 0, 98,
MPI_COMM_WORLD,&status);
MPI_Recv(&cur_b, 1, MPI_DOUBLE, 0, 99,
MPI_COMM_WORLD,&status);
MPI_Recv(&cur_h, 1, MPI_DOUBLE, 0, 100,
MPI_COMM_WORLD,&status);
}
//Расчет интеграла в своем диапазоне, выполняют
все
//процессы:
double s=0, s1;
printf("Process %d. A=%.4fB=%.4fh=%.10f\n",
myrank, cur_a, cur_b, cur_h);
for(cur_a+=cur_h;cur_a<=cur_b;cur_a+=cur_h)
    s+=cur_h*fnc(cur_a);
if(!myrank)
{ //Это процесс-мастер
//Собираем результаты расчетов:
for(i=1;i<ranksize;i++)
    {

```

```

        MPI_Recv(&s1,      1,      MPI_DOUBLE,      i,      101,
MPI_COMM_WORLD,&status);
        s+=s1;
    }
    //Печатьрезультата:
printf("Integral value: %.4f\n",s);
}
else
    //Это подчиненный процесс, отправляем результаты
//мастеру:
    MPI_Send(&s,      1,      MPI_DOUBLE,      0,      101,
        MPI_COMM_WORLD);
    MPI_Finalize();//Завершение работы с MPI
return 0;
}

```

Далее приводится листинг модифицированной программы расчета интеграла, использующей операции группового взаимодействия процессов.

```

#include "mpi.h"
#include "stdio.h"

const double a=0.0;//Нижнийпредел
const double b=100.0;//Верхнийпредел
const double h=0.0000001;//Шагинтегрирования

double fnc(double x)//Интегрируемаяфункция
{
return x*x;
}

int _tmain(intargc, _TCHAR* argv[])

```



```

{
intmyrank, ranksize,i;
MPI_Init(&argc, &argv); //Инициализация MPI
//Определяем свой номер в группе:
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    //Определяемразмергруппы:
MPI_Comm_size(MPI_COMM_WORLD, &ranksize);
double cur_a,cur_b,d_ba,cur_h;
double *sbuf=NULL;
if(!myrank)
{ //Это процесс-мастер
    //Определяем размер диапазона для каждого
процесса:
    d_ba=(b-a)/ranksize;
    sbuf=new double[ranksize*3];
    cur_a=a;
    cur_h=h;
    for(i=0;i<ranksize;i++)
    {
        cur_b=cur_a+d_ba-h;
        sbuf[i*3]=cur_a;
        sbuf[i*3+1]=cur_b;
        sbuf[i*3+2]=h;
        cur_a+=d_ba;
    }
}
doublerbuf[3];
    //Рассылка всем процессам, включая процесс-мастер
//начальных данных для расчета:

```

```

    MPI_Scatter(sbuf, 3, MPI_DOUBLE, rbuf, 3, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
    if(sbuf) delete []sbuf;
    cur_a=rbuf[0];cur_b=rbuf[1];cur_h=rbuf[2];
    //Расчет интеграла в своем диапазоне, выполняют все
//процессы:
    double s=0;
    printf("Process   %d.   A=%.4f   B=%.4f   h=%.10f\n",
myrank, cur_a, cur_b, cur_h);
    for(cur_a+=cur_h;cur_a<=cur_b;cur_a+=cur_h)
        s+=cur_h*fnc(cur_a);
    rbuf[0]=s;
    if(!myrank) sbuf=new double[ranksize];
    //Собираем значения интегралов от процессов:
    MPI_Gather(rbuf, 1, MPI_DOUBLE, sbuf, 1, MPI_DOUBLE, 0,
        MPI_COMM_WORLD);
    if(!myrank)
    {//Это процесс-мастер
        //Суммирование интегралов, полученных каждым
//процессом:
        for(i=0,s=0;i<ranksize;i++) s+=sbuf[i];
        //Печатьрезультата:
        printf("Integral value: %.4f\n",s);
        delete []sbuf;
    }
    MPI_Finalize();//Завершение работы с MPI
    return 0;
}

```

Таким образом, программы, использующие групповые операции получаются более короткими и более эффективными с точки зрения передачи данных по сети.

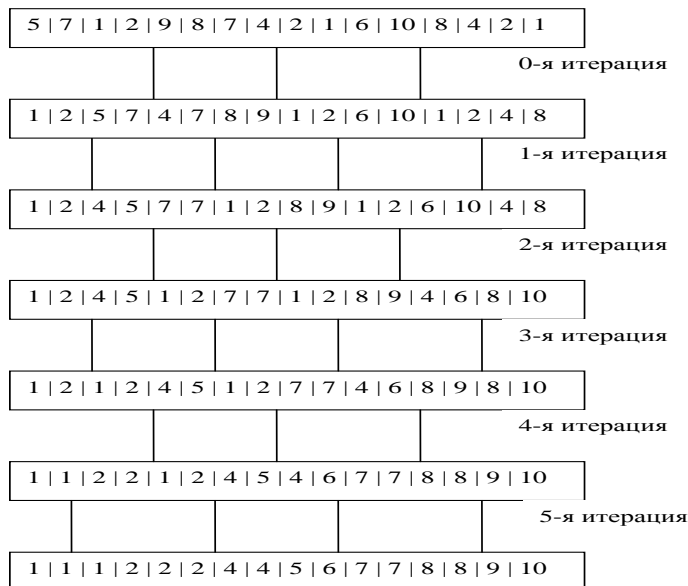
Задания к лабораторной работе

При выполнении работы по одному из выбранных заданий, варианты которых даны в следующем разделе, необходимо разработать параллельную программу, провести сравнение времени вычислений при разном количестве процессов. Отчет о работе, который должен содержать:

- постановку задачи;
- описание параллельного алгоритма;
- листинг программы;
- результаты работы, сравнительные характеристики работы программы для нескольких процессов.

1. Параллельная сортировка

А) Сортировка транспозициями



Пример сортировки транспозициями 16-элементного массива за 6 итераций

При сортировке N-элементного массива M клиентами i-му из них на четных итерациях передается отрезок массива из N/M элементов с номерами $(N/M)(i-1)+1, \dots, (N/M)i$, $i=1, M$, на нечетных – с номерами $(N/M)(i-1)+1+N/(2M), \dots, (N/M)i+ N/(2M)$, $i=1, M-1$. После сортировки каждый клиент

возвращает обратно свою часть. При сортировке следует учитывать, что на всех итерациях, кроме 0-й (первой хронологически) получаемая каждым клиентом часть состоит из двух равных частей – старшей и младшей – уже отсортированных. На рисунке приведен пример сортировки.

Б) Сортировка Шелла

Необходимо реализовать параллельную сортировку Шелла. Идея алгоритма состоит в обмене элементов, расположенных не только рядом, но и далеко друг от друга, что значительно сокращает общее число операций перемещения элементов. Как пример рассматривается файл из $2n$ элементов. На j -м шаге файл делится на $2j$ групп по $2n-j$ элементов в каждой, при этом i -я группа содержит элементы $(i, i+1*2n-j, i+2*2n-j, \dots, i+(2j-1)*2n-j)$, $i=1, \dots, 2j$, $j=n-1, \dots, 1$ (номер шага убывает от $n-1$ до 1). На каждом шаге выполняется сортировка в каждой группе. При сортировке в каждой группе следует иметь в виду, что элементы с четными (нечетными) номерами образуют уже отсортированную последовательность, то есть для сортировки группы надо слить эти две отсортированные подпоследовательности.

В) Сортировка слиянием

Алгоритмы слияния основываются на процедуре слияния двух уже отсортированных отрезков массива в один. При параллельной сортировке весь процесс разбивается на итерации. На первой итерации массив разбивается на фрагменты, общее количество которых превышает количество клиентов в два раза. Затем каждая пара фрагментов сливается одним из процессов, после чего фрагмент записывается в сортируемый массив. Все следующие итерации идентичны первой за исключением уменьшения работающих процессов вдвое на каждой из итераций. Последняя пара фрагментов может быть слита на сервере.

2. Параллельное шифрование

А) Шифрование методом замены (моноалфавитная подстановка)

В этом методе символы шифруемого текста заменяются символами, взятыми из одного (одноалфавитная или моноалфавитная подстановка) алфавита. Самой простой разновидностью является прямая замена, когда буквы шифруемого сообщения заменяются другими буквами того же самого или некоторого другого алфавита. Таблица замены может иметь следующий вид:

Символы шифруемого текста	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Заменяющие символы	S P X L R Z I M A Y E D W T B G V N J O C F H Q U K

Б) Шифрование методом замены (полиалфавитная подстановка)

При полиалфавитной подстановке для замены символов исходного текста используются несколько алфавитов, заданных данным ключом, причем смена алфавитов осуществляется последовательно и циклически, т.е. первый символ заменяется соответствующим символом первого алфавита, второй - символом второго алфавита и т.д. до тех пор, пока не будут использованы все выбранные алфавиты. После этого использование алфавитов повторяется.

В) Шифрование методом гаммирования.

Суть этого метода состоит в том, что символы шифруемого текста последовательно складываются с символами некоторой специальной последовательности, называемой гаммой. Иногда такой метод представляют как наложение гаммы на исходный текст, поэтому он получил название "гаммирование".

Процедуру наложения гаммы на исходный текст можно осуществить двумя способами. При первом способе символы исходного текста и гаммы заменяются цифровыми эквивалентами, которые затем складываются по модулю k , где k – число символов в алфавите, т.е.

$$t_{\text{ш}} = (t_o + t_r) \bmod k,$$

где $t_{\text{ш}}$ – символы зашифрованного текста;

t_o – символы исходного текста;

t_r – символы гаммы.

При втором методе символы исходного текста и гаммы представляются в виде двоичного кода, затем соответствующие разряды складываются по модулю два.

Список источников.

1. MPI для начинающих. – http://www.csa.ru:81/~il/mpi_tutor
2. MPI-2: Extensions to the Message-Passing Interface. - <http://parallel.ru/docs/Parallel/mipi2/mipi2-report.html>
3. Богачев К.Ю. Основы параллельного программирования. – М.:Бином, 2003. – 342 с.
4. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб: БХВ-Петербург, 2002. – 608 с.
5. Воеводин Вл. В. Технологии параллельного программирования. MessagePassingInterface (MPI). – <http://parallel.ru/vvv/mipi.html>
6. Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. – СПб.: БХВ-Петербург, 2002. – 400 с.

**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ
ФЕДЕРАЦИИ**
**Федеральное государственное автономное
образовательное учреждение высшего профессионального
образования**
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Кочеров Ю.Н.

**ВВЕДЕНИЕ В ТЕХНОЛОГИИ ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ
ВЫЧИСЛЕНИЙ**

Методические указания к самостоятельной работе

Направление подготовки 09.03.02 Информационные системы и
технологии

Профиль подготовки «Информационно-управляющие системы»

Форма обучения – очная/заочная.

Квалификация выпускника:

Бакалавр

Невинномысск 2015

СОДЕРЖАНИЕ

1. Цель и задачи освоения дисциплины.....	49
2. Место дисциплины в структуре ООП бакалавриата.....	49
3. Связь с предшествующими дисциплинами.....	49
4. Связь с последующими дисциплинами	50
5. Компетенции обучающегося, формируемые в результате освоения дисциплины.....	50
6. Структура и содержание дисциплины	52
7. Организационно-методические рекомендации по освоению дисциплины.....	52
8. Методические указания по выполнению самостоятельной работы студентов.....	53
9. Содержание самостоятельной работы	54
9.1. Примерная тематика заданий для самостоятельной работы студентов	55
10. План-график выполнения СРС	Ошибка! Закладка не определена.
11. Организация контроля знаний студентов.....	56
11.1 Формы контроля знаний студентов	56
11.2. Рекомендации по подготовке к зачету	57
12. Рекомендации по работе с литературой и источниками.....	61
13 Перечень рекомендуемой литературы.....	62

1. Цель и задачи освоения дисциплины

Цель изучения дисциплины «Введение в технологии высокопроизводительных вычислений» – формирование у студентов системы теоретических представлений о современных подходах к разработке и использованию информационных систем, ориентированных на обработку больших объемов информации с применением параллельных и распределенных архитектур.

Задачами дисциплины «Введение в технологии высокопроизводительных вычислений» являются:

- 1) изучение типологии многопроцессорных систем;
- 2) изучение классификации и возможностей каждого семейства многопроцессорных систем;
- 3) обучение основным принципам и технологиям параллельного программирования;
- 4) изучение механизмов программирования с использованием механизма потоков;
- 5) изучение основных принципов тестирования и отладки параллельных программ;
- 6) изучение аппаратных средств высокопроизводительных ИС;
- 7) изучение программных средств высокопроизводительных ИС.
- 8) формирование набора профессиональных компетенций (ПК-12) и (ПК-28).

2. Место дисциплины в структуре ОП магистратуры

Дисциплина «Основы теории параллельных вычислений» относится к вариативной части профессионального цикла (БЗ.В.ОД.4). Ее освоение происходит во 2 семестре.

3. Связь с предшествующими дисциплинами

Для успешного освоения дисциплины «Основы теории параллельных вычислений» необходимо освоить дисциплины «Объектно-ориентированное программирование» и «Технология программирования».

4. Связь с последующими дисциплинами

Успешное освоение дисциплины «Основы теории параллельных вычислений» необходимо для успешного прохождения итоговой государственной аттестации.

5. Компетенции обучающегося, формируемые в результате освоения дисциплины

Процесс изучения дисциплины направлен на формирование у студентов компетенций, представленных в таблице 1.

Таблица 1 – Содержание компетенций, формируемых в результате освоения дисциплины

Индекс	Формулировка:
ПК-27	способность формировать новые конкурентоспособные идеи и реализовывать их в проектах

В результате освоения дисциплины обучающийся должен:

Перечень компонентов	Технологии формирования компетенции	Средства и технологии оценки
Знать: <ul style="list-style-type: none">– информационные технологии высокопроизводительных информационных систем;– состав, назначение и функции основных компонентов аппаратного обеспечения высокопроизводительных ИС;– типологию многопроцессорных систем;– принципы построения SMP-систем, MPP-систем, кластеров;– средства разработки программного обеспечения для многопроцессорных систем;– принципы подхода GPGPU;	традиционные лекции, самостоятельная работа, лабораторные работы	собеседование, письменный отчет

<ul style="list-style-type: none"> – принципы разработки приложений с использованием библиотеки MPI; – теоретические основы функционирования вычислителей на основе GPU; – принципы разработки приложений с использованием технологии CUDA; – теоретические основы функционирования распределенных систем обработки информации; – математические средства реализации информационных технологий; – алгоритмические средства реализации информационных технологий. – современные библиотеки автоматизации процесса разработки объектов профессиональной деятельности; – принципы построения и особенности жизненного цикла ИС предметной направленности. 		
<p>Уметь:</p> <ul style="list-style-type: none"> – производить выбор инструментария для автоматизации процессов проектирования информационных технологий; – использовать системы на базе архитектуры передачи сообщений; – разрабатывать приложения на основе многопоточной технологии; – разрабатывать приложения для систем GPGPU; – применять программные средства в зависимости от архитектуры многопроцессорной ЭВМ; – проектировать архитектуру распределенного приложения и выбирать оптимальные методы тестирования и отладки; – анализировать и выбирать инструментальные средства, оптимальным образом подходящие для разработки объектов профессиональной деятельности. 	<p>традиционные лекции, самостоятельная работа, лабораторные работы</p>	<p>собеседование</p>
<p>Владеть:</p> <ul style="list-style-type: none"> – инструментарием разработки параллельных и распределенных приложений; – технологией построения и настройки кластерных систем и систем на базе GPU-вычислителей; – технологией разработки MPI-приложений; – технологией разработки CUDA-приложений; – технологией разработки приложений для распределенной обработки данных; 	<p>традиционные лекции, самостоятельная работа, лабораторные работы</p>	<p>собеседование</p>

<p>– методикой автоматизированного проектирования информационных технологий.</p> <p>– навыками проектирования, разработки, анализа эффективности, тестирования и отладки информационных технологий различной направленности.</p>		
--	--	--

6. Структура и содержание дисциплины

Общая трудоемкость дисциплины составляет:

4 зач.ед. (144 академических часа), из них:

1 зач.ед. (36 академических часа) – аудиторные занятия,

3 зач.ед. (108 академических часов) – самостоятельная работа.

7. Организационно-методические рекомендации по освоению дисциплины

Самостоятельная работа студентов является важнейшим условием формирования научного способа познания. Она проводится накануне каждого лабораторного занятия и включает изучение необходимого для выполнения лабораторной работы теоретического материала, а также подготовку отчета по выполненным на лабораторном занятии заданиям.

Подготовленный материал оформляется в виде отчета.

Самостоятельные занятия (СЗ) являются одной из активных форм обучения.

Самостоятельные занятия по дисциплине «Введение в технологии высокопроизводительных вычислений» имеют целью:

– закрепить и углубить знания, полученные студентами на лекциях и в процессе лабораторных занятий;

– привить практические навыки при разработке, эксплуатации и исследовании информационных процессов, систем и технологий.

Самостоятельные занятия проводятся в специализированных аудиториях, оборудованных СВТ и возможностью пользования Интернет-ресурсами, в библиотеке, а также дома.

Предлагаемые методические рекомендации содержат информацию для студентов, необходимую при подготовке и проведении лабораторных занятий по дисциплине «Введение в технологии высокопроизводительных вычислений».

8. Методические указания по выполнению самостоятельной работы студентов

Методические указания должны включать следующие разделы:

- цель работы;
- задание, которое должно быть выполнено студентом в результате проведения самостоятельной работы;
- варианты индивидуальных заданий;
- основные теоретические положения, необходимые для выполнения задания, они должны быть краткими и содержать ссылки на литературу, в которой эти положения изложены в объеме, достаточном для выполнения самостоятельной работы;
- этапы выполнения задания с указанием конкретных сроков выполнения каждого из этапов и всего задания в целом;
- требования к оформлению графической и текстовой части самостоятельной работы;
- пример выполнения одного из вариантов задания и оформления отчета;
- библиографический список использованных источников.

9. Содержание самостоятельной работы

Наименование тем дисциплины, цель, форма контроля, задания, требования к оформлению, перечень литературных источников представлен в таблице 2.

Таблица 2 – Наименование тем дисциплины, цель, форма контроля, задания и требования к оформлению самостоятельной работы

	Наименование разделов и тем дисциплины, их краткое содержание; вид самостоятельной работы	Форма контроля	Зачетные единицы (часы)
7 семестр			
	Подготовка к лабораторным занятиям	Собеседование	8
	Подготовка к лекционным занятиям	Собеседование	8
	Самостоятельное изучение материала тем: "Исследовать эффективность выполнения векторно-матричных операций в зависимости от вычислительного алгоритма: поэлементное, построчное и постолбцовое формирование результирующей матрицы с ее транспонированием"	Собеседование	6
	Самостоятельное изучение материала тем:	Собеседование	8

	" Исследовать эффективность решения задачи транзитивного замыкания в зависимости от затрат на управление ходом вычислений "		
	Самостоятельное изучение материала тем: " Исследовать эффективность решения задач сортировки от коэффициента распараллеливания вычислений " " Исследовать влияние коэффициента распараллеливания вычислений на их точность "	Собеседование	39
	Самостоятельное изучение материала тем: " Исследовать индифферентность и живучесть ассоциативных процессоров с микропрограммных уровнем доступа в зависимости от карты отказов "	Собеседование	39
Итого за 2 семестр			108(3)
ИТОГО			108(3)

9.1. Примерная тематика заданий для самостоятельной работы студентов

По дисциплине «Основы теории параллельных вычислений» предусмотрена самостоятельная работа студентов по изучению вопросов теоретического материала с контролем в форме собеседования. Вопросы для собеседования:

№ п/п	Темы для самостоятельного изучения	Рекомендуемые источники информации (№ источника)			
		Ос новная	Дополни- тельная	Методи- ческая	Интернет- ресурсы
	Раздел 1. Теоретические основы функционирования высокопроизводительных систем				
	Закон Амдала	2		1, 2	1, 2
	Техническая реализация многопроцессорных систем	1, 2	1,2,3	1, 2	1, 2
	Раздел 9. Реализация высокопроизводительных вычислений на многоядерных процессорах				
	Проблема многопоточного параллелизма	1, 2		1, 2	
	Синхронизация потоков	1,2	2	1, 2	1, 2
	Класс Thread	1, 2		1, 2	1, 2
	Передача данных потокам	1, 2	2	1, 2	
	Управление потоками	1, 2		1, 2	

10. Организация контроля знаний студентов

10.1 Формы контроля знаний студентов

Контроль и оценка знаний, умений и навыков студентов осуществляется на лабораторных занятиях, консультациях, при сдаче экзамена. В ходе контроля знаний преподаватель оценивает понимание студентом содержания дисциплины «Введение в технологии высокопроизводительных вычислений», его способность анализировать развитие информационных систем и технологий.

Контроль знаний студентов может осуществляться в следующих формах:

- текущий контроль знаний;
- итоговый контроль знаний.

Текущий контроль знаний студентов имеет целью: дать оценку работы каждого студента по усвоению им учебного материала, выявить недостатки в его подготовке и оказать практическую помощь в их устранении;

Основными формами текущего контроля знаний студентов являются:

- устный контрольный опрос;
- защита лабораторной работы;
- проверка конспектов лекций.

Устный контрольный опрос студентов проводится на лекциях (и лабораторных занятиях). По его результатам преподаватель оценивает качество подготовки студента к занятию.

На лабораторных занятиях знания и практические навыки студентов оцениваются по 4-балльной системе. Полученные оценки выставляются в журнале.

При проверке конспектов лекций дается анализ качества их ведения. Отмечаются допущенные ошибки, в рецензии преподавателя оценивается качество конспектирования учебного материала, даются рекомендации по улучшению качества конспектирования лекционного материала.

[10.2. Рекомендации по подготовке к зачету и экзамену](#)

Во 2 семестре формой промежуточной аттестации по дисциплине «Основы теории параллельных вычислений» является экзамен. Защита лабораторных работ и выполнение контрольной работы является основанием для допуска к экзамену.

Перед экзаменом студенту необходимо полностью выполнить все задания к лабораторным работам. При наличии задолженностей по текущей

аттестации по данной дисциплине студент к экзамену не допускается. Экзамен по дисциплине предусмотрен в устной форме по билетам.

Вопросы для проверки уровня обученности

Студент должен **знать**: информационные технологии высокопроизводительных информационных систем; состав, назначение и функции основных компонентов аппаратного обеспечения высокопроизводительных ИС; типологию многопроцессорных систем; принципы построения SMP-систем, MPP-систем, кластеров; средства разработки программного обеспечения для многопроцессорных систем; принципы подхода GPGPU; принципы разработки приложений с использованием библиотеки MPI; теоретические основы функционирования вычислителей на основе GPU; принципы разработки приложений с использованием технологии CUDA; теоретические основы функционирования распределенных систем обработки информации; математические средства реализации информационных технологий; алгоритмические средства реализации информационных технологий.

Базовый уровень. Вопросы для проверки знаний:

1. Назначение и основные концепции технологии GPGPU.
2. Особенности GPGPU подхода.
3. Технология CUDA.
4. Способы реализации приложений CUDA.
5. Языки разработки приложений CUDA.
6. Модели памяти CUDA.
7. Современные практические достижения GPGPU и CUDA.
8. Направление использования CUDA в научных лабораториях.
9. Направление использования CUDA в учебном процессе.
10. Направление использования CUDA в промышленности.
11. Архитектура вычислительной видеокарты Tesla.

12. Системы, основанные на парадигме Hadoop.
13. Перспективные технологии построения высокопроизводительных систем.
14. Области применения высокопроизводительных систем обработки информации.
15. Средства разработки многопоточных приложений для Windows.
16. Инструментарий .NET Framework для построения многопоточных приложений.
17. Инструментарий Java для построения многопоточных приложений.
18. Системы управления базами данных NoSQL.
19. Реляционные системы управления базами данных и их масштабирование.
20. Области применения объектно-ориентированных систем управления базами данных.
21. Реализации библиотеки MPI.
22. Коллективные операции MPI.
23. Использование MPI в рамках подхода GPGPU.
24. Использование OpenMP в рамках подхода GPGPU.
25. Библиотеки программного кода для вычислительных систем на базе GPU.
26. Рейтинг суперкомпьютеров: мировой и общероссийский.
27. Суперкомпьютерные технологии в современной науке.
28. Потоки Thread.
29. Класс Parallel.
30. Классы для обеспечения синхронизации.

Базовый уровень. Вопросы для проверки умений и владения практическими навыками

1. Использование оператора lock.
2. Использование класса Mutex.
3. Применение методов класса Parallel.
4. Возможности класса Task.
5. Реализация задач продолжения.
6. Применение пула потоков.
7. Использование фабрики задач.
8. Получение результата асинхронного делегата через ожидание.
9. Получение результата выполнения асинхронного метода через делегат обратного вызова.
10. Получение результата асинхронного делегата через дескриптор ожидания.

Повышенный уровень. Вопросы для проверки знаний

1. *Примеры использования NoSQL в реальных секторах.*
2. *Архитектура информационной системы на базе высокопроизводительной СУБД.*
3. *Возможности NoSQL-подхода при решении научных и учебных задач.*
4. *Язык R и его использование в рамках NoSQL.*
5. *NoSQL и реляционные СУБД: отличия и сходство.*
6. *NoSQL и объектные СУБД: отличия и сходство..*
7. *Задачи, ориентированные на высокопроизводительные ИС.*
8. *Гетерогенные системы на базе CPU и GPU.*
9. *Использование GPGPU в системах обработки данных.*
10. *Взаимодействие многопоточного приложения и интерфейса пользователя.*

Повышенный уровень. Вопросы для проверки умений и владения практическими навыками

1. *Реализация приложения с использованием класса Thread.*
2. *Разработка приложения с использованием асинхронных делегатов.*
3. *Применение асинхронных делегатов с параметрами.*
4. *Применение асинхронных делегатов с возвращаемым значением.*
5. *Реализация пула потоков.*
6. *Использование lock для потокобезопасного приложения.*
7. *Применение таймера для организации динамичных приложений.*
8. *Использование событий в интерфейсных классах.*
9. *Применение событий для организации взаимодействия объектов бизнес-логики.*
10. *Реализация приложения с заменой стандартного потока.*

11. Рекомендации по работе с литературой и источниками

Изучение литературы и источников необходимо начинать с прочтения соответствующих глав учебных изданий, учебных пособий или литературы, рекомендованной в качестве основной или дополнительной по дисциплине «Введение в технологии высокопроизводительных вычислений», которые прямо или косвенно относятся к изучаемой теме.

При изучении литературы и источников студенту рекомендуется вести краткий конспект. Однако не следует переписывать все содержание изучаемой темы, нужно выписывать лишь основные идеи и главные на ваш взгляд мысли. В отдельных случаях, когда встречаются важные определения, понятия, необходимый фактический материал и примеры, статистическая информация, имеющие отношение к изучаемой теме, студенту следует выписать их в виде цитат с полным указанием библиографических источников.

Конспектирование рекомендуемой литературы и источников необходимо вести с распределением собранных материалов по отдельным главам и параграфам согласно учебно-тематическому плану. Необходимо

выписывать все выходные данные по используемой литературе и источникам.

Основой технологии интенсификации обучения на платформе цифровых образовательных технологий являются учебно-иллюстрационные материалы (опорный конспект) по дисциплине «Введение в технологии высокопроизводительных вычислений».

Работа с учебно-иллюстрационными материалами имеет следующие этапы.

1. Изучение теоретических основ учебного материала в аудитории: изложение преподавателем изучаемого материала студентам с объяснением по опорному конспекту;

2. Самостоятельная работа: индивидуальная работа студентов по опорному конспекту; фронтальное закрепление по блокам опорного конспекта.

3. Первое повторение – воспроизведение содержания заданной темы опорного конспекта по памяти.

4. Устное проговаривание материала опорного конспекта – необходимый этап внешне речевой деятельности при усвоении учебного материала.

5. Второе повторение – взаимопрос и взаимопомощь студентов друг другу.

Применение учебно-иллюстрационных материалов позволяет обобщить сложный по содержанию материал, активизировать мыслительную деятельность студентов.

Необходимо помнить, что главное для студента в самостоятельной работе с рекомендуемой литературой и источниками - это формирование своего индивидуального стиля, который может стать основой в будущей профессиональной деятельности.

12 Перечень рекомендуемой литературы

12.1. Рекомендуемая литература

12.1.1. Основная литература:

1. Клементьев, И.П. Введение в облачные вычисления [Электронный ресурс] / И.П. Клементьев, В.А. Устинов. - М. : Интернет-Университет Информационных Технологий, 2011. - 427 с. - URL: <http://biblioclub.ru/index.php?page=book&id=23404>.

12.1.2. Дополнительная литература:

1. Васильков, Ю.В. Компьютерные технологии вычислений в математическом моделировании [Электронный ресурс] / Ю.В. Васильков, Н.Н. Василькова. - М. : Финансы и статистика, 2002. - 256 с. - URL: <http://biblioclub.ru/index.php?page=book&id=237054>.

12.1.3. Методическая литература:

1. Методические указания к выполнению лабораторных работ по дисциплине «Введение в технологию высокопроизводительных вычислений» для студентов специальности 09.03.02 "Информационные системы и технологии" / сост. Николаев Е.И. – Ставрополь : СКФУ, 2014. – __ с.

2. Учебное пособие «Введение в технологию высокопроизводительных вычислений» для студентов специальности 09.03.02 «Информационные системы и технологии» / сост. Николаев Е.И. – Ставрополь : СКФУ, 2014. – __ с.

12.1.4. Интернет-ресурсы:

1. <http://www.intuit.ru> – Сайт национального открытого университета «Интуит»

2. <http://www.citforum.ru> – Сайт центра информационных технологий

3. <http://www.parallels.com/ru> – Сайт компании Parallels. Содержит описания новых коммерческих программных продуктов в области облачных и параллельных технологий.

12.2.5. Программное обеспечение:

- Операционная система Windows,
- Microsoft Visual Studio (версия не ниже 2012).

12.2. Материально-техническое обеспечение дисциплины

Компьютерный класс. Высокопроизводительная ЭВМ для разработки и тестирования в среде . MS Visual Studio, создания отчетов по выполненным лабораторным работам в пакете Microsoft Office.

**МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ
К САМОСТОЯТЕЛЬНОЙ РАБОТЕ СТУДЕНТОВ
по дисциплине
«Основы теории параллельных вычислений»**

Направление подготовки 15.04.04 «Автоматизация технологических
процессов и производств»

Профиль подготовки «Безопасность информационных систем»

Форма обучения – очная/заочная.

Квалификация (степень) выпускника:

Бакалавр

Составители: Ю.Н. Кочеров