МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ 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

« Технология параллельных вычислений»

для студентов направления подготовки

15.03.04 – «Автоматизация технологических процессов и производств»

Невинномысск 2021

Настоящие методические указания предназначены для студентов направления 15.03.04 – «Автоматизация технологических процессов и производств». Они разработаны в соответствии с федеральным государственным образовательным стандартом и основной образовательной программой.

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 выполнения лабораторных работ и приведен список рекомендуемых источников литературы.

Составитель канд. техн. наук Ю.Н. Кочеров

Ответственный редактор канд. техн. наук Д.В. Болдырев

СОДЕРЖАНИЕ

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 изза потенциальной возможности возникновения проблемам 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	
Вид:	__property boolFreeOnTerminate = {read=FFreeOnTerminate, write=FFreeOnTerminate, nodefault};

Опис.:	Определяет, уничтожится ли автоматически объект потока при завершении потока. Если FreeOnTerminate = истина, то объект уничтожается автоматически, в противном случае его необходимо уничтожать явно (delete).

Прим.:	`void __fastcallTtestThread::Execute(){`
	`int Result`
	`FreeOnTerminate=true;`
	`for(inti=0; i<10000; i++) Result+=sin(i);`
	`}`

Handle	
Вид:	__property int Handle = {read=FHandle, nodefault};
Опис.:	Возвращает дескриптор потока. Необходим при использовании WinAPI функций.

Прим.:	`P1->Resume();`
	`int HT=P1->Handle;`
	`Form1->Label2->Caption = IntToStr(HT);`

Priority	
Вид:	enumTThreadPriority {tpIdle, tpLowest, tpLower, tpNormal, tpHigher, tpHighest, tpTimeCritical}; __property TThreadPriority Priority = {read=GetPriority, write=SetPriority, nodefault};
Опис.:	Определяет относительный приоритет потока.

Прим.:	`P1 = new IvanovP1(true);`
	`P1->Priority = tpNormal;`
	`P1->Resume();`

Suspended	
Вид:	__property bool Suspended = {read=FSuspended, write=SetSuspended, nodefault};

Опис.:	Определение состояния, а также приостановка/продолжение выполнения потока. Если Suspended = true – поток приостанавливается. На false – продолжение выполнения.

Прим.:	`first->Suspended=true; //Остановка выполнения потока`
	`...`
	`first->Suspended=false; //Продолжение выполнения потока`

Terminated	
Вид:	__property bool Terminated = {read=FTerminated, nodefault};
Опис.:	Определяет, был ли завершен поток. Метод Execute должен периодически проверять Terminated, и выходить, если Terminated = true. Метод Terminate устанавливает Terminated в true.

Прим.:	`void __fastcallTMyThread::Execute()`
	`{`
	`  while (!Terminated){`
	`  …`
	`  }`
	`}`

ThreadID	
Вид:	__property intThreadID = {read=FThreadID, nodefault};
Опис.:	Получение идентификатора потока.
Прим.:	`int id=first->ThreadID;`
МЕТОДЫ (METHODS)	
DoTerminate	
Вид:	virtual void __fastcallDoTerminate(void);
Опис.:	Генерирует событие OnTerminate, но не завершает поток.
Execute	
Вид:	virtual void __fastcall Execute(void) = 0;
Опис.:	Чистый виртуальный метод. Содержит исполняемый код потока.

Прим.:	`void __fastcall IvanovP1::Execute()`
	`{`
	`float s = 0;`
	`for(inti=0;i<N;i++){`
	`s+= sin(i) + cos(i);`
	`Form1->ProgressBar1->Position = i;`
	`if(Terminated) return;`
	`}`
	`}`

Resume	
Вид:	void __fastcall Resume(void);
Опис.:	Выполнение или возобновление выполнения прерванного потока.
Прим.:	`first->Resume();`
Suspend	
Вид:	void __fastcall Suspend(void);
Опис.:	Приостановка выполнения потока.
Прим.:	`first->Suspend();`
Synchronize	
Вид:	typedef void __fastcall (__closure *TThreadMethod)(void); void __fastcall Synchronize(TThreadMethod&Method);
Опис.:	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.

Прим.:	`void __fastcallTMyThread::PushTheButton(void)`
	`{`
	`  Button1->Click();`
	`}`
	`void __fastcallTMyThread::Execute()`
	`{`
	`  ...`
	`Synchronize(PushTheButton);`
	`  ...`
	`}`

Terminate	
Вид:	void __fastcall Terminate(void);
Опис.:	Устанавливает свойство Terminated в истину, сигнализируя, что поток должен завершиться быстро, как только это возможно. В отличие от WinAPI функции TerminateThread, которая вынуждает поток завершиться, Terminate просто сигнализирует о необходимости завершения.

Прим.:	void __fastcall IvanovP1::Execute()
	{
	float s = 0;
	
	for(inti=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();
	}

TThread	
Вид:	__fastcallTThread(boolCreateSuspended);
Опис.:	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.

Прим.:	TMyThread *Second = new TMyThread(**true**); // создать, нонезапускать
	Second->Priority = tpLower; // установкаприоритетаниженормального
	Second->Resume(); // запускпотока

WaitFor	
Вид:	int __fastcallWaitFor(void);

Опис.:	Ожидание завершения потока.

Прим.:	`...`
	`first->Resume();`
	`first->WaitFor();`
	`second->Resume();`
	`...`
	В данном примере два потока first и second будут выполняться последовательно. Поток second начнет выполняться только по завершении выполнения потока first.

СОБЫТИЯ (EVENTS)	
OnTerminate	
Вид:	__property TNotifyEventOnTerminate = {read=FOnTerminate, write=FOnTerminate};
Опис.:	Событие происходит после выхода из метода Execute но прежде, чем поток будет уничтожен.
Прим.:	`P1->OnTerminate = ThreadDone;`

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	$\sin(x)*\cos(x^2)$	[0, 10]
2	$\ln(x)+10*\cos(x)$	[0, 8]
3	$x^2-10*x+\sin(x)$	[3, 9]
4	$2^x / x^3$	[4, 10]
5	$15*\cos(2*x)/\ln(x)$	[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 классаTThread

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 __fastcallSynchronize(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 PaintBoxc 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) - запроснавходвкритическийраздел.

VOID LeaveCriticalSection (LPCRITICAL_SECTION lpCriticalSection) - выходизкритическогораздела.

VOID DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection) - удалениекритическогораздела (обычнопривыходеизпрограммы).

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);
 // Запускаемпотоки
Thread1 -> Resume;
Thread2 ->Resume;
... // Текстпрограммы.
// 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(inti = 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(inti = 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. Созданиеобъектамutex:

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

Параметры:

lpMutexAttributes - указательнаструктурySECURITY_ATTRIBUTES (вWindows 95 данныйпараметригнорируется);

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 исключающим семафором.

2. Закрытие (уничтожение) объектамutex :

BOOLCloseHandle(HANDLEhObject)

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 исключающие семафоры.

Листинг 2. Ограничение доступа к массиву с использованием исключающих семафоров

```
// Массив значений.
intmas[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(inti = 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(inti = 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 lInitialCount, LONG lMaximumCount, 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.

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

lMaximumCount - 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 lReleaseCount,LPLONGlpPreviousCount);

Параметры:

hSemaphore - дескриптор семафора;

lReleaseCount - 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.
intmas[1000];
// Семафор, регулирующий доступ к критическому разделу.
HANDLESemaph;
{
...
// Создаемсемафор;
```

```
Semaph  =CreateSemaphore(NULL, 1, 1, NULL);
// Запускаемпотоки

// Удаляемсемафор
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(inti = 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(inti = 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(
DWORDnCount,              // 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 философах: на круглом столе находятся ktарелок с едой, между которыми лежит столько же вилок, k=4,…,6. В комнате имеется kфилософов, 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 COM-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.

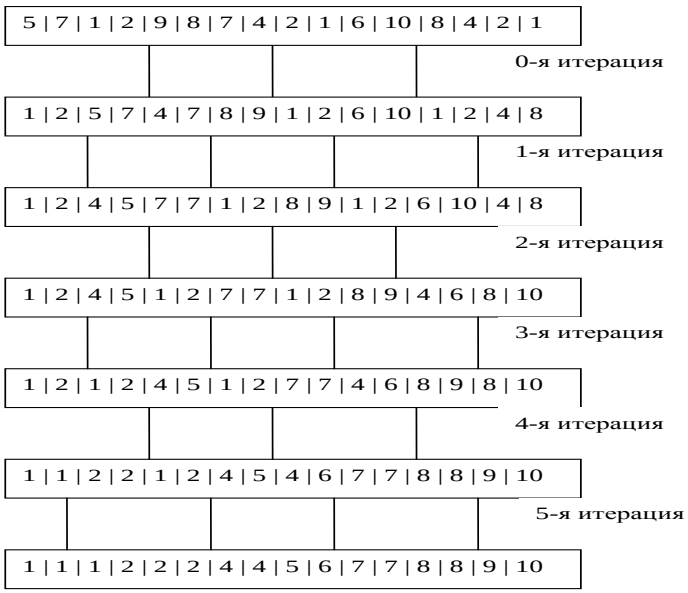
б) Параллельное вычисление с синхронизацией.

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,…, (N/M)i, i=1,M, на нечетных – с номерами (N/M)(i-1)+1+N/(2M),…, (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 , .., i+(2j -1)*2n-j), i=1,.., 2j , j=n-1,..,1 (номер шага убывает от n-1 до 1). На каждом шаге выполняется сортировка в каждой группе. При сортировке в каждой группе следует иметь ввиду, 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

ТехнологияMPI. 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).

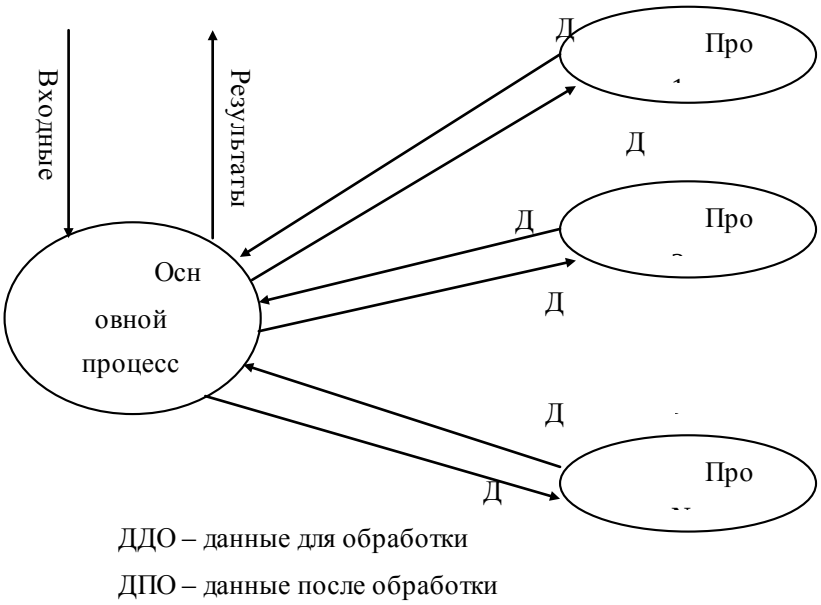


Рис. 1. Схема взаимодействия частей SPMD-программы.

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 знаю о существовании друг друга и обмен данными между ними невозможен.

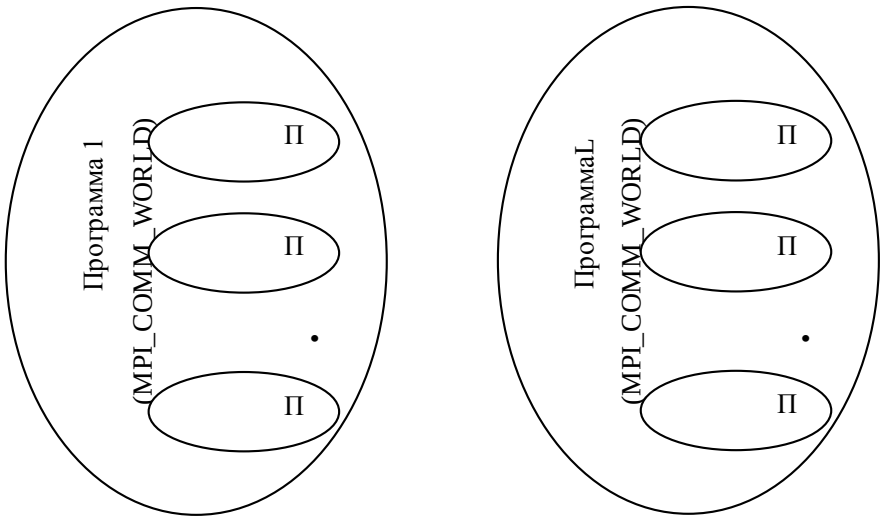


Рис 2. Нумерация процессов и коммуникаторы в MPI

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
MPI_CHAR	char
MPI_BYTE	unsigned char
MPI_SHORT	short
MPI_INT	int

MPI_LONG	long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_LONG_DOUBLE	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 посылки или приема данных.

Рассмотримэтифункцииподробнее.

```
intMPI_Send(void* buf, intcount,
MPI_Datatypedatatype, intdest,
intmsgtag, MPI_Commcomm)
```

Функция выполняет синхронную (с блокировкой) посылку 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 предопределенных 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 посылки сообщения, устраняющих подобную неопределенность, которые подробно описаны в литературе.

```
intMPI_Recv(void* buf, intcount,
MPI_Datatypedatatype, 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_Datatypedatatype, 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_Datatypedatatype, 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, параметрысообщенияоказываютсяв 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 коммуникатора. 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 посылки, 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_Datatypedatatype, int source,
MPI_Commcomm)
```

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, intscount,
MPI_Datatypestype, void *rbuf,
intrcount, MPI_Datatypertype,
intdest, MPI_Commcomm)
```

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, intscount,
MPI_Datatypestype, void *rbuf,
intrcount, MPI_Datatypertype,
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 scount элементов типа stype, после чего i-я часть посылается i-му процессу. На процессе source существенным являются значений 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).

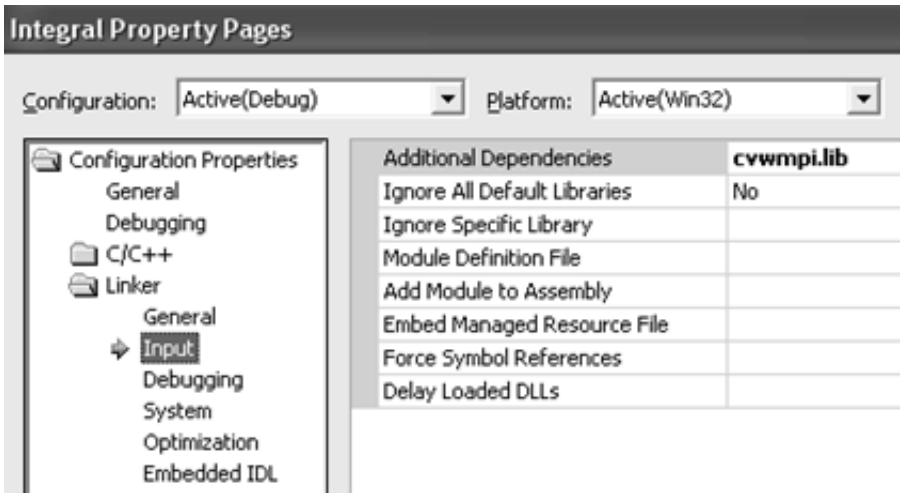


Рис. 3. Окно настроек линковщика проекта VisualC++ .NET

Перед запуском MPI-программы необходимо создать конфигурационный файл с именем, соответствующим имени созданного компилятором исполняемого файла, и расширением .pg, 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.

Инициируется 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(intargc, _TCHAR* argv[])
{
intmyrank, ranksize,i;
MPI_Status status;
  MPI_Init(&argc, &argv);//ИнициализацияMPI
//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;
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
    //Получаемисходныеданные:
    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:
    doubles=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.

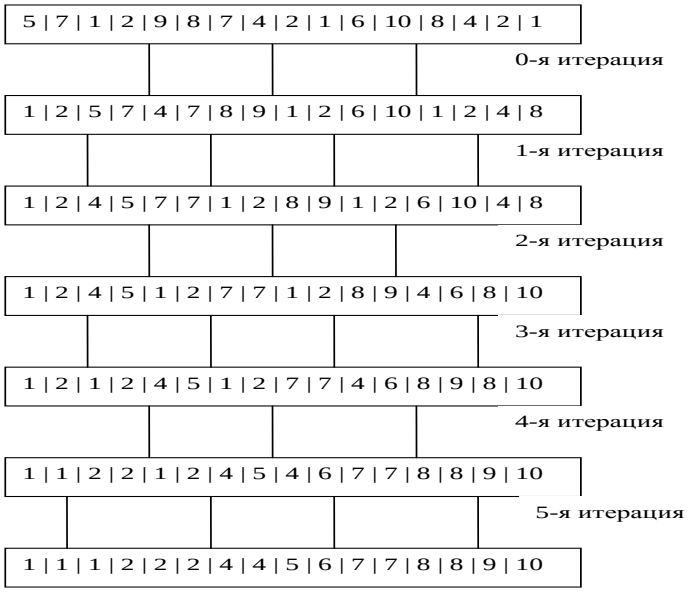
Задания к лабораторной работе

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,…, (N/M)i, i=1,M, на нечетных – с номерами (N/M)(i-1)+1+N/(2M),…, (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 , .., i+(2j -1)*2n-j), i=1,.., 2j , j=n-1,..,1 (номер шага убывает от n-1 до 1). На каждом шаге выполняется сортировка в каждой группе. При сортировке в каждой группе следует иметь ввиду, 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.

$t_ш = (t_o + t_г) \bmod k$,

где $t_ш$ – символы зашифрованного текста;

t_o – символы исходного текста;

$t_г$ – 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/mpi2/mpi2-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-Петербуг, 2002. – 608 с.
5. Воеводин Вл. В. Технологии параллельного программирования. MessagePassingInterface (MPI). – http://parallel.ru/vvv/mpi.html
6. Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. – СПб.: БХВ-Петербург, 2002. – 400 с.