БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ РАДИОФИЗИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ
Кафедра информатики и компьютерных систем

Н. В. Серикова

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ
ТЕХНОЛОГИЯ MPI

Методические указания
к лабораторному практикуму по курсу
«Параллельные вычисления и программирование»
для студентов факультета радиофизики
и компьютерных технологий

МИНСК
2016

УДК 004.42.032.24(075.8)(076.5)+004.272.2(075.8)(076.5)
ББК 32.973.2-018.2я 73-1
С 33

Утверждено
на заседании кафедры информатики и компьютерных систем
факультета радиофизики и компьютерных технологий БГУ
7 декабря 2015 г., протокол № 5

Р е ц е н з е н т
кандидат технических наук, доцент *В. С. Садов*

Серикова, Н. В.
С33 Параллельные вычисления. Технология MPI: метод. указания / Н. В. Серикова. – Минск: БГУ, 2016. – 55 с.

Приводятся необходимые сведения и описание лабораторных работ для проведения практикума по спецкурсу «Параллельные вычисления и программирование». Методические указания Методические указания состоят из четырех частей. В первой части рассмотрены вопросы, связанные с организацией параллельных вычислений на вычислительном кластере. Во второй части описаны функции библиотеки MPI. В третьей части приведено большое число примеров параллельных программ. В четвертой части описаны 3 лабораторные работы и приведены 24 варианта самостоятельных заданий по каждой из них.

Предназначено для студентов факультета радиофизики и компьютерных технологий БГУ.

УДК 004.42.032.24(075.8)(076.5)+004.272.2(075.8)(076.5)
ББК 32.973.2-018.2я 73-1

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Отдельный класс параллельных архитектур представляют кластерные системы**. Кластер** – это совокупность вычислительных узлов, объединенных сетью. Параллельное приложение для кластерной системы представляет собой несколько процессов, которые общаются друг с другом по сети. Таким образом, если пользователь сумеет эффективно распределить свою задачу между несколькими процессорами на узлах кластера, то он может получить выигрыш в скорости работы, пропорциональный числу процессоров

Методические указания содержат необходимые сведения и описание лабораторных работ для проведения практикума «Параллельные вычисления. Технология MPI». Цель практикума – изучить организацию вычислений на кластерах, исследовать ускорение и эффективность реализованных параллельных алгоритмов.

Методические указания состоят из четырех частей.

Первая часть посвящена организации параллельных вычислений на кластерах под управлением ОС *Windows* с использованием библиотеки ***MPI*** (*The **M**essage **P**assing **I**nterface*) и пакета ***MPICH***. В пособии содержатся сведения по организации кластера, описаны настройки компиляторов ***Visual Studio*** **2005/2008/2010/2012** для создания параллельных программ, способы запуска параллельных программ с использованием пакета *MPICH* (версии 1 и 2), приведены сведения об отладке параллельных программ.

Во второй части приведены основные функции *MPI*, в третьей - примеры параллельных программ.

В четвертой части содержатся лабораторные работы для проведения практикума, приведены по 24 варианта самостоятельных заданий по каждой из них.

1. ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ НА ВЫЧИСЛИТЕЛЬНОМ КЛАСТЕРЕ

Под **вычислительным кластером** будем понимать совокупность процессоров, объединенных в рамках некоторой сети для решения одной задачи. Для функционирования кластера необходимо соответствующее программное обеспечение. ***MPI*** (*The **M**essage **P**assing **I**nterface*) является библиотекой программирования для кластеров. Реализацией *MPI* для кластеров является пакет ***MPICH***. *MPICH* — самая известная реализация *MPI*, созданная в Арагонской национальной лаборатории (США). Существуют версии этой библиотеки для популярных ОС. В качестве среды разработки используется *Visual Studio* 2005/2008/2010/2012.

1.1. НАСТРОЙКА КОМПИЛЯТОРА ДЛЯ РАБОТЫ С MPICH

1. Настройка компилятора *Visual Studio* 2005/2008

Чтобы скомпилировать программу необходимо изменить настройки проекта**:**

1. Добавить путь к заголовочным файлам объявлений *MPI* (**.h* файлам, которые используются в проекте). Для этого выбрать пункт меню ***Project->Project Properties***. В пункте ***Configuration Properties->C++->General->Additional Include Directories*** ввести путь к заголовочным файлам *MPI* **<Директория установки *MPICH*>*include*** (рис. 1).

2. Добавить путь к библиотечным файлам (**.lib* файлам, которые используются в проекте). Для этого выбрать пункт меню ***Project->Project Properties***. В пункте ***Configuration Properties->Linker->General->Additional Library Directories*** ввести путь до библиотечных файлов *MPI* ***<Директория установки MPICH>\lib*** (рис. 2).

3. Добавить библиотечный файл с реализацией функций *MPI*. Для этоговыбрать пункт меню ***Project->Project Properties***. В пункте ***Configuration Properties->Linker->Input->Additional Dependencies*** ввести название библиотечного файла ***mpi.lib*** (рис. 3).

1.1.2. Настройка компилятора *Visual Studio* 2010/2012

Чтобы скомпилировать программу необходимо изменить настройки проекта**:**

1. Добавить путь к заголовочным файлам объявлений *MPI* (*.*h* файлам, которые используются в проекте). Для этого нужно выбрать ***Project → Properties***, в дереве слева выбрать ***Configuration Properties → VC++ Directories***. Справа в поле ***Include Directories*** добавить путь к *h*-файлам (рис. 4).

2. Добавить путь к библиотечным файлам (*.*lib* файлам, которые используются в проекте). Для этого нужно выбрать ***Project → Properties***, в дереве слева выбрать ***Configuration Properties → VC++ Directories***. Справа в поле ***Library Directories*** добавить путь к *lib*-файлам (рис. 5).

3. Добавить библиотечный файл с реализацией функций *MPI*. Для этого открыть окно настроек проекта ***Project → Properties***, в дереве слева выбрать ***Configuration Properties → Linker → Input***. Добавить `mpi.lib` в поле ***Additional Dependencies*** справа (рис. 6).

Необходимо отметить, что все настройки, описанные выше, можно не делать, если создать проект с шаблоном ***MPI Project***. Тогда достаточно при создании проекта с данным шаблоном снять галочку с "***Precompiled Header***".

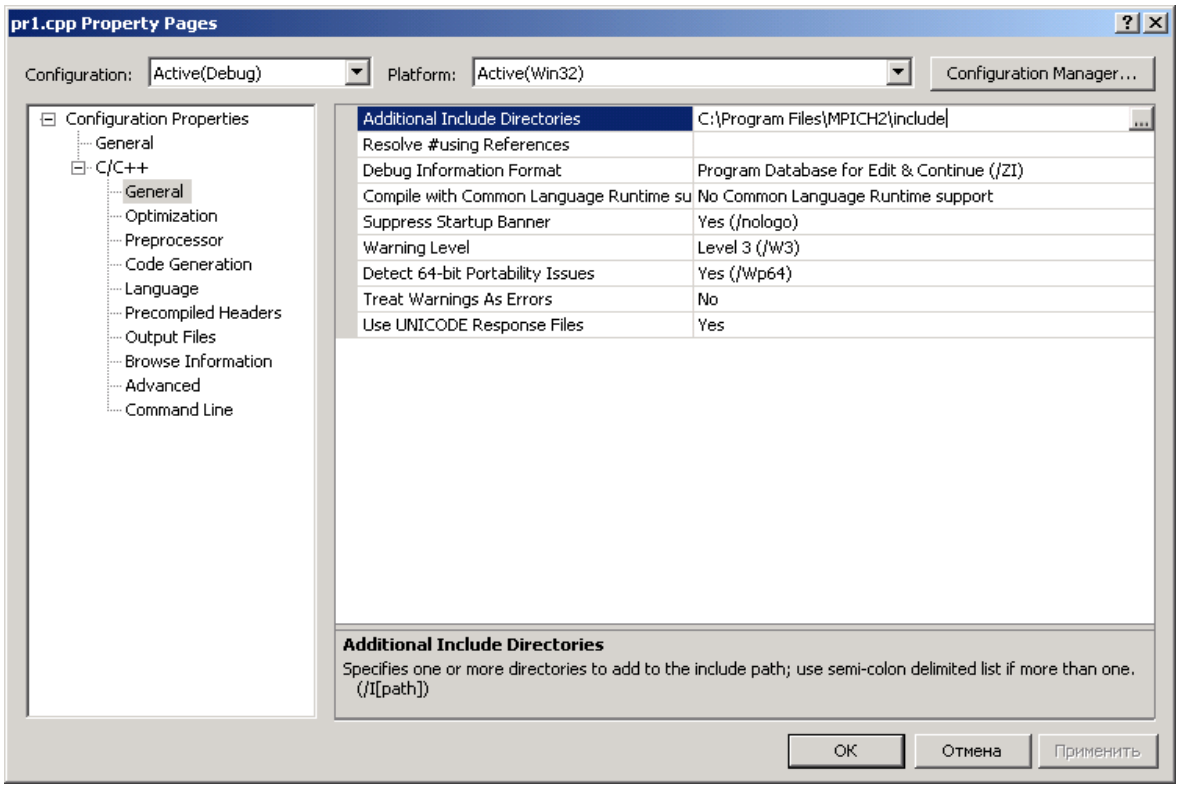


Рис. 1. Настройка пути к заголовочным файлам *MPICH*

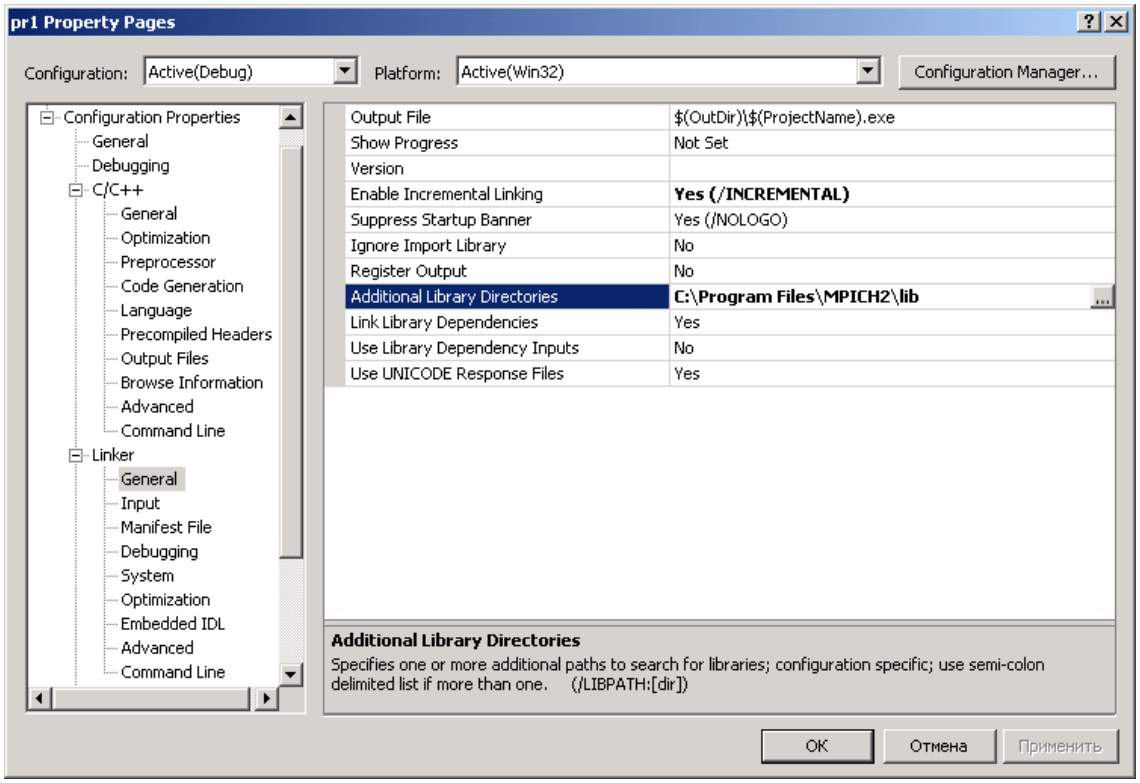


Рис. 2. Настройка пути к библиотечным файлам *MPICH*

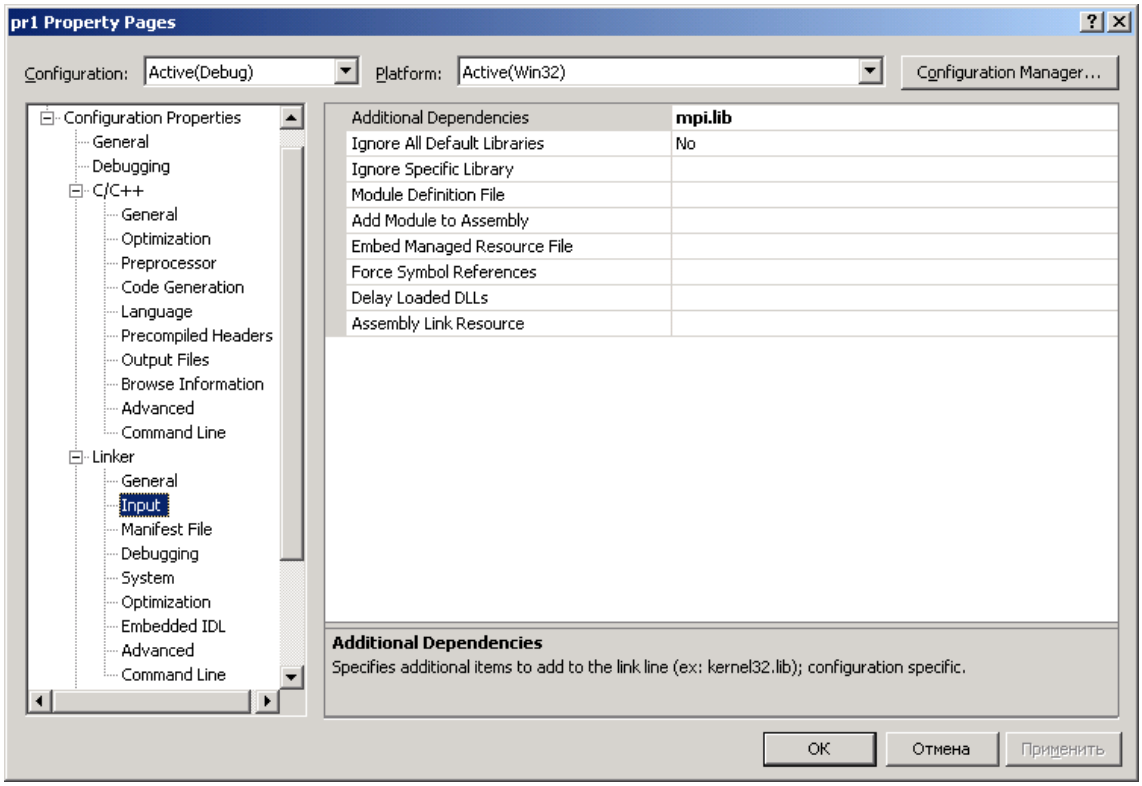


Рис 3. Добавление в проект библиотеки *mpi.lib*

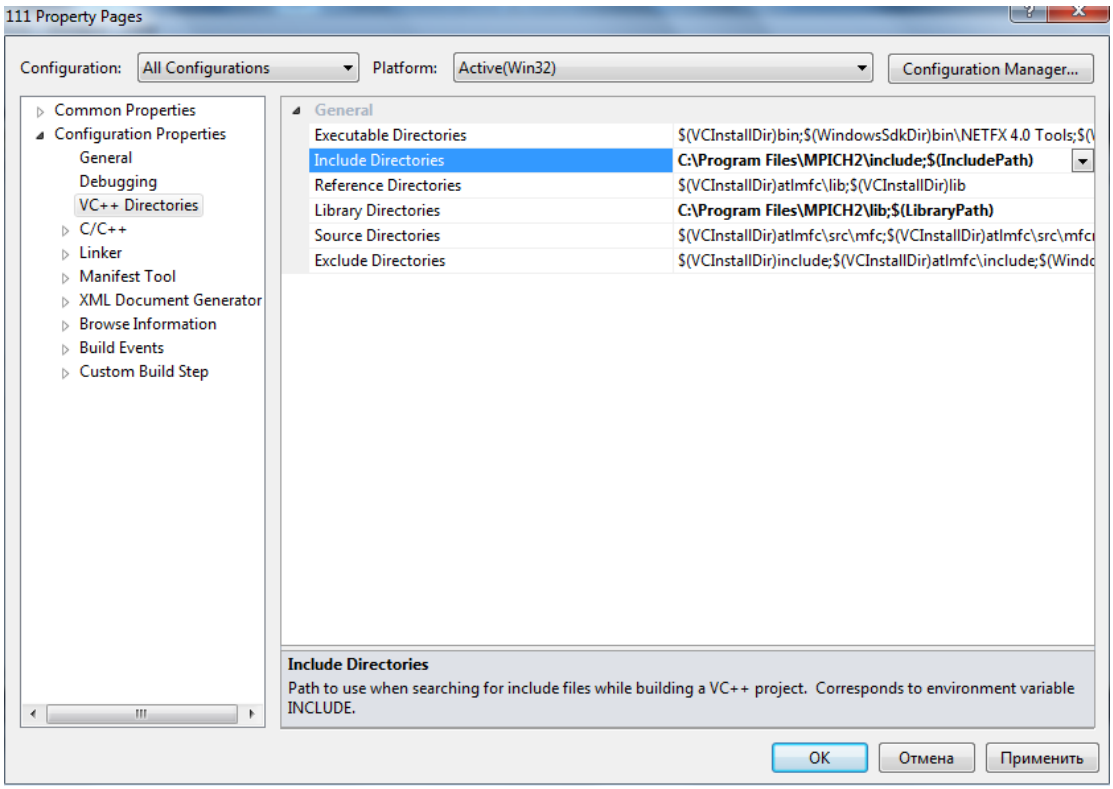


Рис 4. Настройка пути к заголовочным файлам *MPICH*

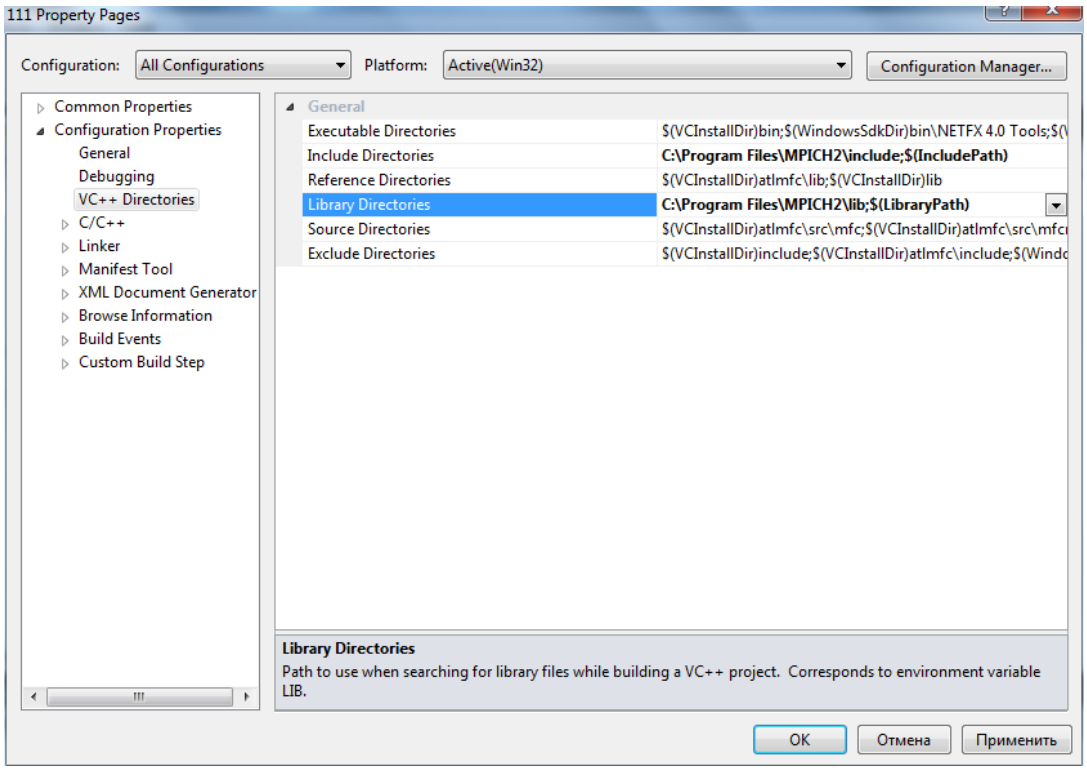


Рис. 5. Настройка пути к библиотечным файлам *MPICH*

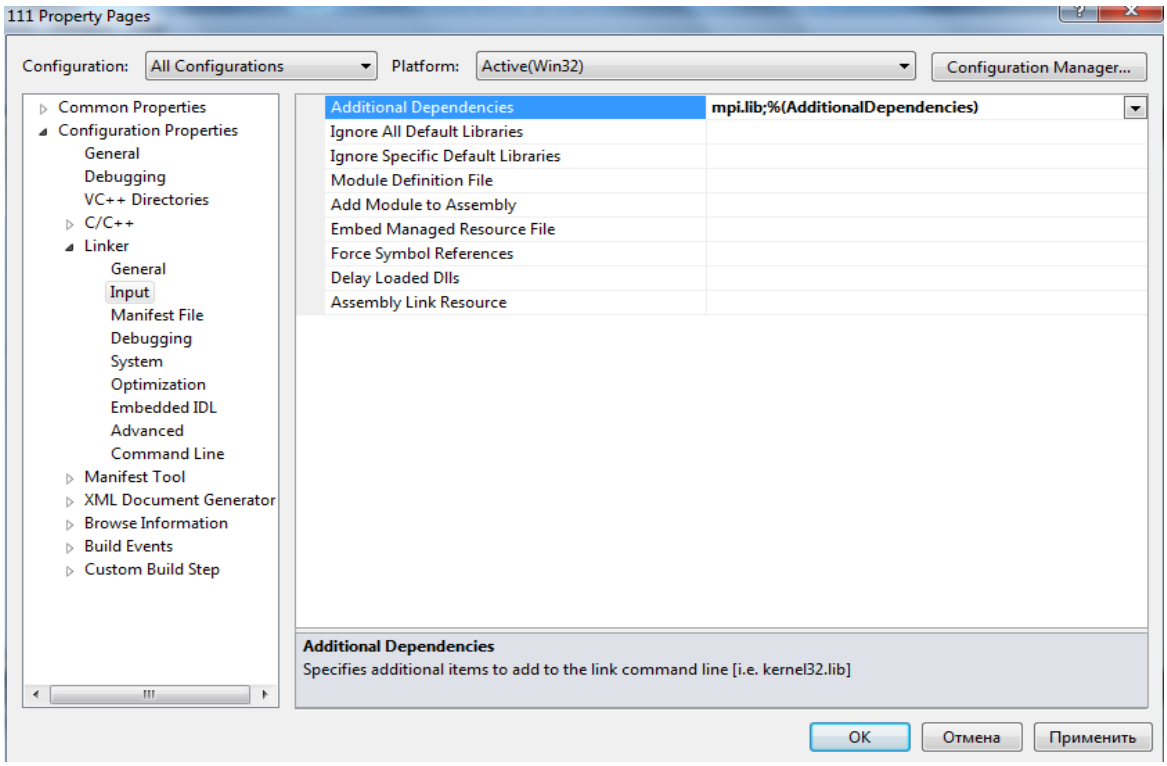


Рис. 6. Добавление в проект библиотеки *mpi.lib*

1.2. ЗАПУСК MPI-ПРОГРАММ

1.2.1. MPICH1

Необходимо установить пакет MPICH1 для запуска параллельных программ [3].

Для запуска *MPI*-программ из командной строки служит утилита ***mpirun***, которая находится в папке *%mpich%\bin*. Сведения о форматах запуска можно получить, запустив программу *mpirun* без параметров либо с параметром *-help*.

Запуск в локальном режиме программ на одном компьютере осуществляется с помощью команды

mpirun -localonly num program

где *num* – количество процессов, *program* – имя бинарного файла параллельной программы. Программа запускается на локальной машине, на которой произведён вызов команды. Режим полезен для отладки программы. Есть возможность запуска программы в локальном режиме с опцией *-tcp*, при этом программа выполняется локально, но коммуникация между процессами происходит через *TCP/IP*. Программу, откомпилированную с *MPICH*, можно запустить, как обычную программу без *mpirun*, что равносильно запуску с одним процессом.

Если параллельная программа использует параметры командной строки, параметры указываются после имени программы. Если программа читает системные переменные, эти переменные могут устанавливаться для программы с помощью опции *-env*.

mpirun -localonly num -env "var1=val1|var2=val2|..." program args

num – число процессов;
program – имя бинарного файла;
*var** – имена системных переменных описываются в виде строки,
*val** – значения соответствующих переменных;
args – аргументы программы, если они есть.

Запуск в многопроцессорном режиме осуществляется:

1. **mpirun -np num program**
2. **mpirun file.cfg**

В первом случае вызов программы *mpirun* аналогичен вызову в локальном режиме. Опция *-np* указывает на многопроцессорный режим, *num* – количество процессов, *program* – имя бинарного файла параллельной программы. Компьютеры для выполнения программы и количество процессов на каждом из них определяется MPICH.

Если необходимо указать конкретное расположение процессов по процессорам, необходим вызов программы *mpirun* с файлом конфигурации. Имя файла конфигурации может быть любым, формат файла следующий:

```
exe mpiprogram.exe
[env var1=val1var2=val2var3=val3...]
[dir drive:\my\working\directory]
[map drive:\\host\share]
[args arg1 arg2 ...]
hosts
hostname1 $procs1 [path\mpiprogram.exe]
hostname2 $procs2 [path\mpiprogram.exe]
hostname3 $procs3 [path\mpiprogram.exe]
#...
```

В квадратных скобках указаны необязательные параметры. Комментарии начинаются с символа # и заканчиваются концом строки. После ключевого слова *exe* указывается имя запускаемого файла. За ключевым словом *hosts* перечисляются сетевые имена узлов (в данном примере *hostname**). После каждого имени узла ставится количество процессов ($*procs**), запускаемых на этом узле, для каждого узла может быть разным.

Необязательные параметры используются в следующих случаях:

- *env* – задает переменные среды для запускаемой программы,
- *args* – задает аргументы запускаемой программе,
- *map* и *dir* позволяют подключить к файловой системе сетевой диск для программы, установить рабочую директорию.

Для каждой программы пишется свой файл конфигурации, нет ограничений на размещение этого файла в файловой системе, так как имя файла конфигурации указывается при вызове *mpirun*.

1.2.2. MPICH2

Установить последнюю версию MPICH2 можно с этой страницы : *https://www.mpich.org/downloads/versions/(***mpich2-1.4.1p1-win-ia32.msi** *)*

В случае использования MPI в качестве интерфейса передачи сообщений параллельные задачи необходимо запускать с использованием специальной утилиты ***mpiexec.exe***, осуществляющей одновременный запуск нескольких экземпляров параллельной программы на выбранных узлах кластера. Утилита находится в папке %*mpich2*%*bin*. Сведения о форматах запуска можно получить, запустив программу без параметров либо с параметром *–help* или *–help2*.

Запуск в локальном режиме программ на одном компьютере осуществляется с помощью команды

mpiexec –n num –localonly program

где *num* – количество процессов, *program* – имя бинарного файла параллельной программы. Программа запускается на локальной машине, на которой произведён вызов команды. Режим полезен для отладки программы.

Запуск в многопроцессорном режиме осуществляется:

mpiexec –n num program

num – количество процессов, *program* – имя бинарного файла параллельной программы. Компьютеры для выполнения программы и количество процессов на каждом из них определяется MPICH.

Если необходимо указать конкретное расположение процессов по процессорам, необходим вызов программы с файлом конфигурации. Имя файла конфигурации может быть любым, формат файла описан в 1.2.1.

mpiexec -configfile filename

Для запуска MPI-программ в комплект *MPICH2* входит программа с графическим интерфейсом ***wmpiexec***, которая представляет собой

оболочку вокруг соответствующей утилиты командной строки *mpiexec*. Окно программы *wmpiexec* представлено на рис. 7.

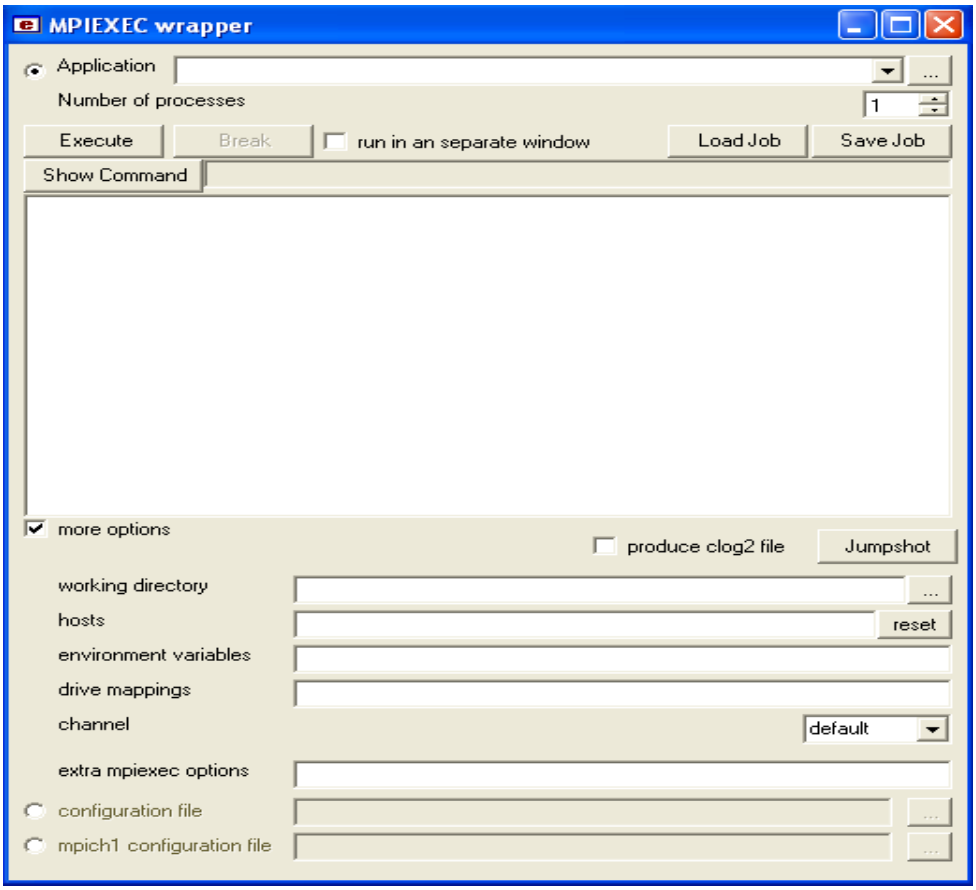


Рис. 7. Окно программы wmpiexec

Элементы управления окна:

- Поле ввода «***Application***»: вводится путь к MPI-программе. Путь передаётся в неизменном виде на все компьютеры сети, поэтому программа должна располагаться в общей сетевой папке.
- «***Number of processes***»: число запускаемых процессов. По умолчанию процессы распределяются поровну между компьютерами сети, однако это поведение можно изменить при помощи конфигурационного файла.
- Кнопка «***Execute***» запускает программу;
- Кнопка «***Break***» принудительно завершает все запущенные экземпляры.
- Флажок «***run in a separate window***» перенаправляет вывод всех экземпляров MPI-программы в отдельное консольное окно.

- Кнопка «***Show Command***» показывает в поле справа командную строку, которая используется для запуска MPI-программы. Командная строка собирается из всех настроек, введённых в остальных полях окна.
- В текстовое поле попадает ввод-вывод всех экземпляров MPI-программы, если не установлен флажок «*run in a separate window*».
- Флажок «***more options***» показывает дополнительные параметры.
- «***working directory***»: можно ввести рабочий каталог программы. Этот путь должен быть верен на всех вычислительных узлах. Если путь не указан, то в качестве рабочего каталога будет использоваться место нахождения MPI-программы.
- «***hosts***»: можно указать через пробел список вычислительных узлов, используемых для запуска MPI-программы. Если это поле пустое, то используется список, хранящийся в настройках менеджера процессов текущего узла.
- «***environment variables***»: в этом поле можно указать значения дополнительных переменных окружения, устанавливаемых на всех узлах на время запуска MPI-программы. Синтаксис следующий: имя1=значение1, имя2=значение2.
- «***drive mappings***»: можно указать сетевой диск, подключаемый на
- каждом вычислительном узле на время работы MPI-программы. Синтаксис: Z:\\winsrv\wdir.
- «***channel***»: позволяет выбрать способ передачи данных между экземплярами MPI-программы
- «***extra mpiexec options***»: в это поле можно ввести дополнительные ключи для командной строки *mpiexec*.

Чтобы MPICH не распределял запускаемые процессы между имеющимися узлами, нужно отключить работу с сетью; для этого существует ключ командной строки ***-localonly***. При его добавлении менеджер процессов не используется. Это очень полезно, если по каким-либо причинам *MPICH* не удаётся настроить. Нужно ввести этот ключ в поле «***extra mpiexec options***» (рис. 8). Результат работы MPI-программы на двух компьютерах кластера показан на рис.9.

1.3. ОТЛАДКА ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

1.3.1. Отладка параллельной программы

Отладка параллельных MPI-программ имеет ряд особенностей, обусловленных природой программирования для кластерных систем. В параллельной программе над решением задачи работают одновременно несколько процессов, каждый из которых, в свою очередь, может иметь несколько потоков команд.

Это обстоятельство существенно усложняет отладку, так как помимо ошибок, типичных для последовательного программирования, появляются ошибки, совершаемые только при разработке параллельных программ. Задача отладки параллельных программ в большинстве случаев сводится к отладке каждого процесса в отдельности плюс изучение поведения процессов при их взаимодействии. При этом отдельный процесс отлаживается стандартными средствами, а исследование взаимодействий требует других средств.

Отладчик параллельных MPI-программ в *Microsoft Visual Studio* появился впервые в версии *Visual Studio* 2005. Однако компиляцию и отладку MPI-программ можно производить и в более ранних версиях среды разработки. Обычными приемами в случае отсутствия истинной поддержки *MPI* со стороны отладчика являются следующие:

Использование текстовых сообщений, выводимых в файл или на экран, со значениями интересуемых переменных и/или информацией о том, какой именно участок программы выполняется. Такой подход часто называют “***printf* отладкой**” (“***printf debugging***”), так как чаще всего для вывода сообщений на консольный экран используется функция “***printf***”. Данный подход хорош тем, что он не требует от программиста специальных навыков работы с каким-либо отладчиком, и при последовательном применении позволяет найти ошибку. Однако для того, чтобы получить значение очередной переменной, приходится каждый раз писать новый код для вывода сообщений, перекомпилировать программу и производить новый запуск. Это занимает много времени. Кроме того, добавление в текст программы нового кода может приводить к временному исчезновению проявлений некоторых ошибок,

Использование последовательного отладчика. Так как MPI-программа состоит из нескольких взаимодействующих процессов, то для отладки можно запустить несколько копий последовательного отладчика, присоединив каждую из них к определенному процессу MPI-задания.

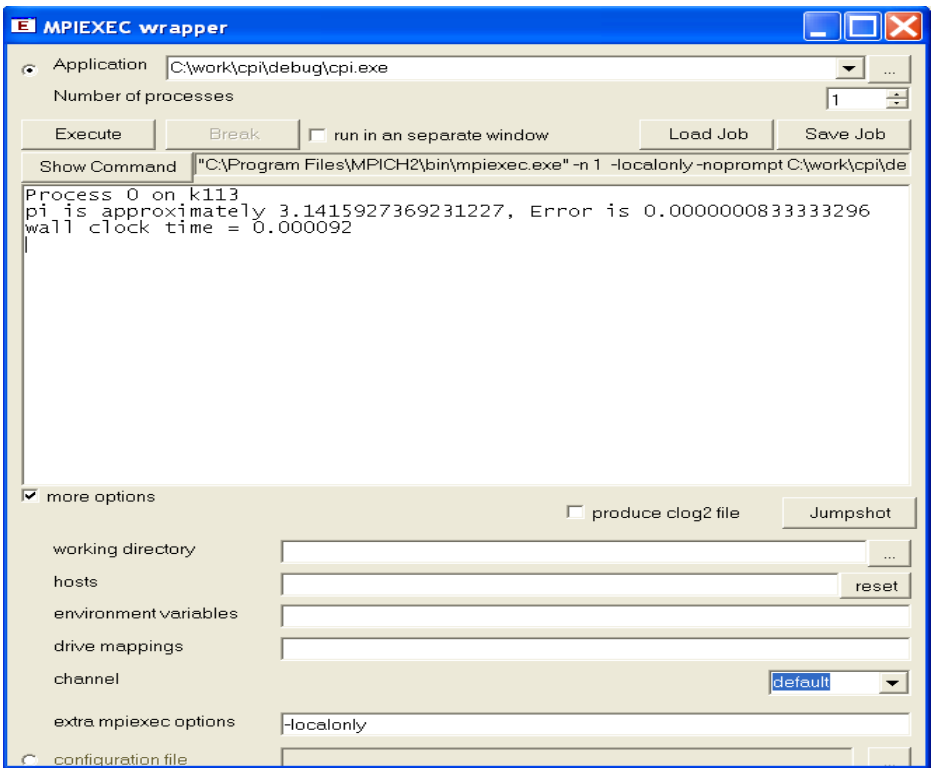


Рис. 8. Запуск MPI-программы в локальном режиме

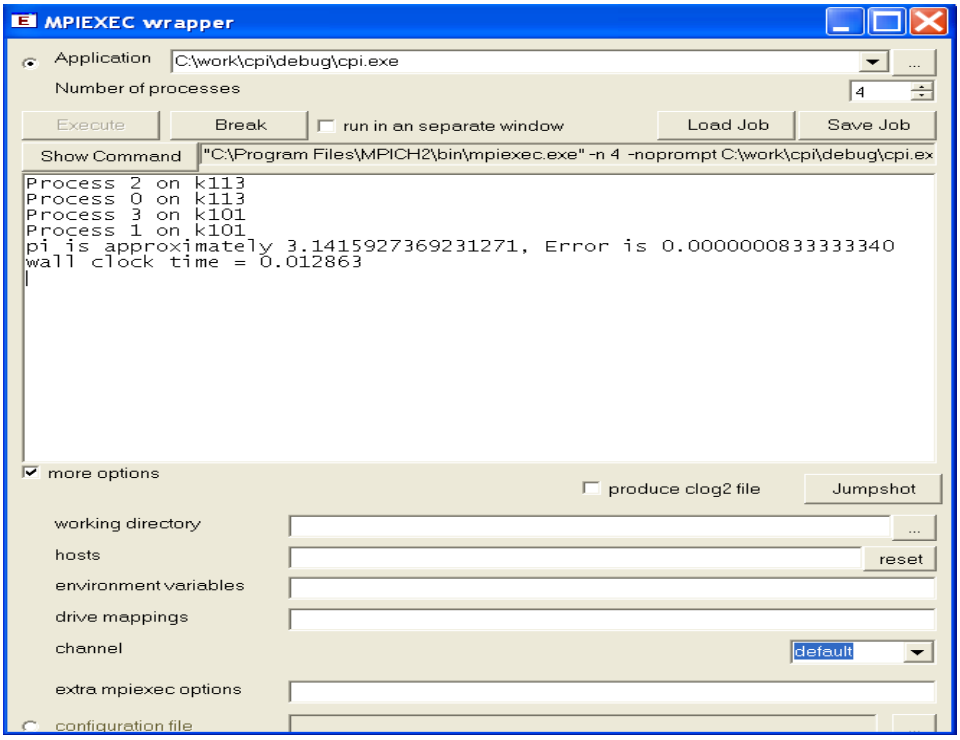


Рис. 9. Результат запуска MPI-программы на двух компьютерах кластера

Данный подход позволяет в некоторых случаях более эффективно производить отладку, чем при использовании текстовых сообщений. Главным недостатком подхода является необходимость ручного выполнения многих однотипных действий. В случае использования последовательного отладчика для приостановки 32 процессов MPI-задания придется переключиться между 32 процессами отладчика и вручную дать команду приостановки. Чтобы параллельную программу запустить под последовательными отладчиками, необходимо запускать каждый процесс не с помощью программы *mpirun*, а «вручную». Для запуска MPI-процесса нужно установить необходимые системные переменные, затем запустить процесс.

Для *MS Visual C++* вызов будет следующим:

```
set MPICH_IPROC=$myid
set MPICH_NPROC=$numprocs
set MPICH_ROOT=$host:$port
msdev.exe $project
```

Здесь:

- $*myid* – номер запускаемого процесса,
- $*numprocs* –общее количество процессов,
- $*host* – имя машины,
- $*port* – номер порта, через который процессы соединяются,
- $*project* – имя отлаживаемого проекта.

Каждая такая последовательность команд вызывает один процесс. Так как программа запускается на $*numprocs* процессах, то такая последовательность должна быть повторена в $*numprocs* консолях, при этом $*myid* должен составлять все значения от 0 до $*numprocs* – 1. $*numprocs* должны быть одинаковыми по всех вызовах. Итак, нужно создать $*numprocs* bat-файлов, в каждый из которых поместить последовательность вышеописанных команд, но $*myid* в каждом файле будет разным, от 0 до $*numprocs* – 1. Описанный метод позволяет отлаживать параллельные программы в исходных кодах.

Например, в случае отладки программы cpi.dsw на двух процессах следует создать два bat-файла со следующим содержанием.

`rem start1.bat`
`set MPICH_IPROC=0`
`set MPICH_NPROC=2`
`set MPICH_ROOT=comp67-1:12345`
`msdev cpi.dsw`
	`rem start2.bat`
	`set MPICH_IPROC=1`
	`set MPICH_NPROC=2`
	`set MPICH_ROOT=comp67-1:12345`
	`msdev cpi.dsw`

Результатом запуска обоих файлов будут два окна *VC++*, отлаживаемый в каждом окне процесс будет обладать свойствами *MPICH*-процесса.

1.3.2.Отладка параллельной программы в Visual Studio

Параллельный отладчик *Microsoft Visual Studio* лишен указанных недостатков и позволяет существенно экономить время на отладку. Это достигается за счет того, что среда рассматривает все процессы одной MPI-задачи как единую параллельно выполняемую программу, максимально приближая отладку к отладке последовательных программ. К числу особенностей параллельной отладки относится окно "***Processes***", позволяющее переключаться между параллельными процессами, а также дополнительные настройки среды.

Интегрированная среда разработки *Microsoft Visual Studio* имеет в своем составе отладчик, предоставляющий широкие возможности для поиска и устранения ошибок, в том числе и для отладки MPI-приложений.

К числу других часто используемых инструментов отладки *Microsoft Visual Studio* относятся:

- Окно **"*Call Stack*"** – окно показывает текущий стек вызова функций и позволяет переключать контекст на каждую из функций стека (при переключении контекста программист получает доступ к значениям локальных переменных функции),
- Окно **"*Autos*"** – окно показывает значения переменных, используемых на текущей и на предыдущих строках кода. Кроме того, это окно может показывать значения, возвращаемые вызываемыми функциями. Список отображаемых значений определяется средой автоматически,
- Окно **"*Watch*"** – окно позволяет отслеживать значения тех переменных, которых нет в окне **"*Autos*"**. Список отслеживаемых переменных определяется пользователем,
- Окно **"*Threads*"** – окно позволяет переключаться между различными потоками команд процесса,
- Окно **"*Processes*"** – окно позволяет переключаться между различными отлаживаемыми процессами (например, в случае отладки MPI- программы).

Для того чтобы получить возможность отладить параллельные MPI-программы, необходимо соответствующим образом настроить ***Microsoft Visual Studio***:

1. Открыть проект (параллельного вычисления числа *Pi* ***parallelpi***).

2. Выбрать пункт меню ***Project->paralellpi Properties…***. В открывшемся окне настроек проекта выбрать пункт ***Configuration Properties->Debugging*** (рис.10) **.**

3. Ввести следующие настройки:

- В поле ***Debugger to launch*** выбрать ***MPI Cluster Debugger*** .
- В поле ***MPIRun Command*** ввести ***mpiexec.exe*** – имя программы, используемой для запуска параллельной MPI-программы (“*c:\Program Files\Mpich2\Bin\mpiexec.exe*”).
- В поле ***MPIRun Argument*** ввести аргументы программы ***mpiexec.exe***. Например, ввести ***“-np 2 -localonly***” для указания числа процессов, которые будут открыты.
- В поле ***Application Command*** ввести путь до исполняемого файла программы.
- В поле ***Application Argument*** ввести аргументы командной строки запускаемой программы.
- В поле ***MPIShim Location*** введите путь до ***mpishim.exe*** – специальной программы, поставляемой вместе с *Microsoft Visual Studio,* используемой для отладки удаленных программ:

c:\Program Files\Microsoft Visual Studio 8\Common7\IDE\Remote Debugger\x86\mpishim.exe.

Нажать “**OK**” для сохранения внесенных изменений.

4. Выбрать пункт меню ***Tools->Options***. В открывшемся окне настроек проекта выбрать пункт ***Debugging->General***. Поставить флаг около пункта “***Break all processes when one process breaks***” для остановки всех процессов параллельной программы в случае, если один из процессов будет остановлен (рис. 11). Или снять флаг для остановки одного процесса при условии, что остальные процессы продолжат работу (рекомендуется).

Запустив программу из меню *Microsoft Visual Studio* (команда “***Debug->Start Debugging***”), будут запущены сразу несколько процессов (их число соответствует настройкам). Каждый из запущенных процессов имеет свое консольное окно. Можно приостанавливать любой из процессов средствами *Microsoft Visual Studio* и проводить отладку.

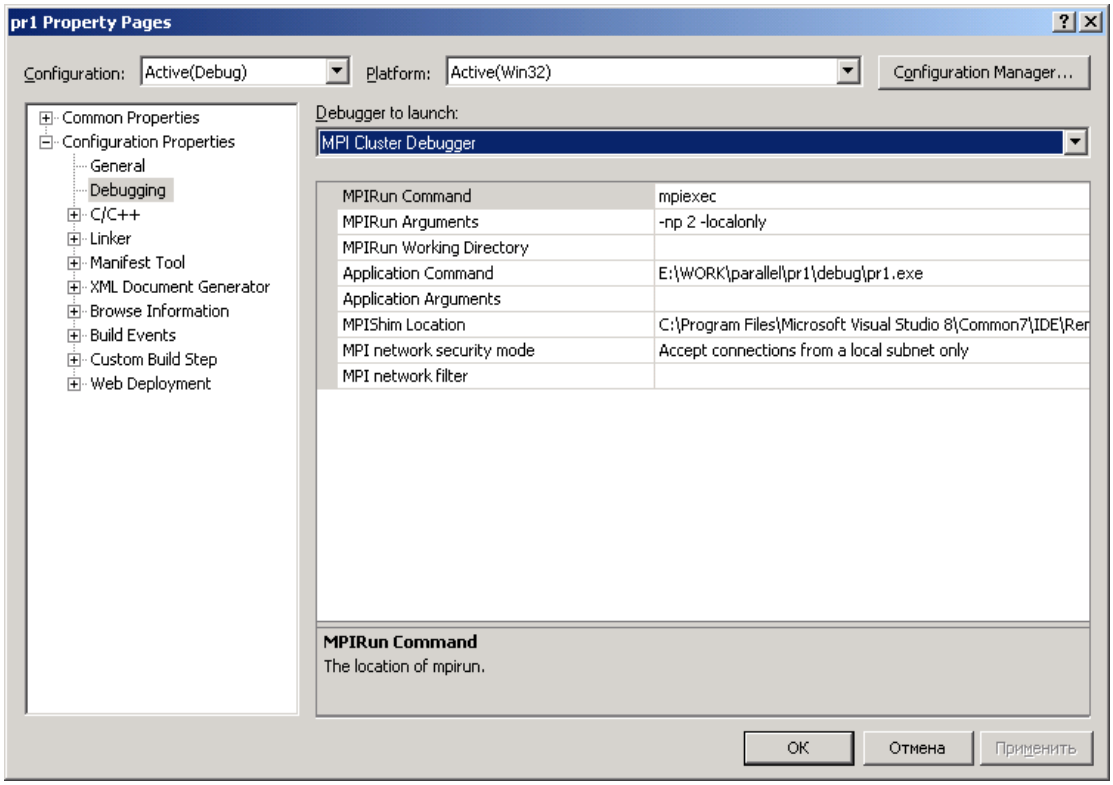


Рис. 10. Настройки среды для работы с отладчиком

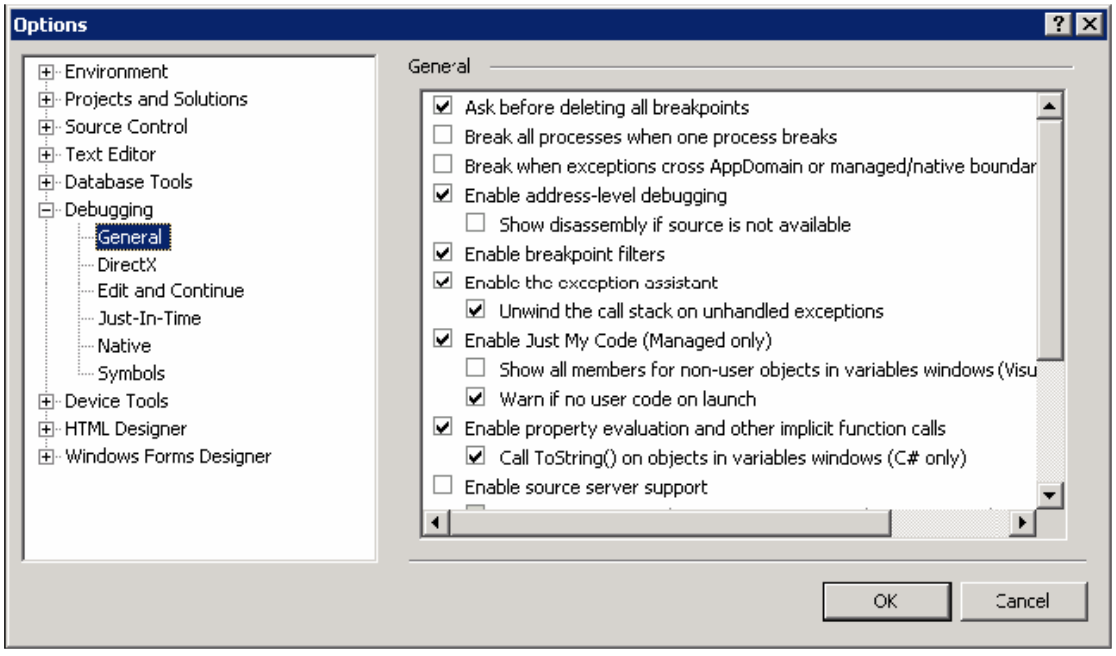


Рис. 11. Настройки для остановки всех процессов

Инструкция по выполнению отладки

1. Открыть проект параллельного вычисления числа *Pi* (***parallelpi***), выполнить настройки, необходимые для проведения отладки MPI-программ, указав число запускаемых процессов (например, равное 2).
2. Открыть файл "***parallelpi.cpp***" (дважды щелкните на файле в окне "***Solution Explorer***").
3. Установить точку останова (установить мигающий указатель вводимого текста на ту строку, на которой нужно установить точку останова и нажать клавишу "***F*9**").
4. Выбрать отладочную конфигурацию ("***Debug***") в выпадающем списке конфигураций проекта на панели инструментов. Запустить программу в режиме отладки: выполнить команду меню "***Debug->Start Debugging***" или нажать на кнопке с зеленым треугольником в панели инструментов *Microsoft Visual Studio*. Программа запустится, откроется 3 консольных окна: окно процесса "***mpiexec.exe***" и 2 консольных окна параллельной программы. По достижении точки остановки выполнение процессов приостановится, так как оба процесса достигнут указанной точки.
5. Открыть окно "***Call Stack***": выполнить команду меню "***Debug->Windows->Call Stack***". Окно показывает текущий стек вызова функций и позволяет переключать контекст на каждую из функций в стеке. При отладке это помогает понять, какая именно текущая функция была вызвана, а так же просмотреть значения локальных переменных функций в стеке.
6. Щелкнуть 2 раза на функции "***main***" в окне "***Call Stack***". Открыть окно "***Autos***" (выполнить команду меню "***Debug->Windows->Autos***"). Окно "***Autos***" показывает значения переменных, используемых на текущей и на предыдущих строках кода. Нельзя изменить состав переменных, так как он определяется средой *Microsoft Visual Studio* автоматически.
7. Открыть окно "***Watch***" (выполнить команду меню "***Debug->Windows-> Watch->Watch 1***"). В этом окне можно просматривать значения переменных, которых нет в окне "***Autos***".
8. Открыть окно "**Processes**" (выполнить команду меню "***Debug->Windows -> Processes***"). Появится окно со списком процессов параллельной MPI-программы. Ввести в окне "***Watch***" переменную "***MyID***" – идентификатор текущего процесса. Затем изменить текущий процесс, дважды щелкнув по другому процессу в окне "***Processes***". Значение переменной "***MyID***" должно отличаться от предыдущего, так как у каждого процесса в одной MPI- задаче идентификаторы различны.

При использовании окна “***Processes***” необходимо учитывать следующие моменты: можно сделать активным только приостановленный процесс (для приостановки процесса можно поставить точку остановки или выделить работающий процесс в окне “***Processes***” и выполнить команду “***Break Process***”– кнопка в виде двух вертикальных линий в окне “***Processes***”), в окне “***Processes***” есть колонка “***ID***” – идентификатор процесса в операционной системе *Windows*. Важно помнить, что указанный идентификатор не имеет отношения к рангу (идентификатору) процесса в MPI-задаче

9. Перейти на процесс с нулевым идентификатором “***MyID***”. Ввести в окне “***Watch***” переменные “***processor_name***” и “***all_proc_names***”. “***all_proc_names***” – массив названий узлов, на которых запущены параллельные процессы. Массив используется только на процессе с индексом 0. Для отображения значений, например, двух элементов массива введите “***all_proc_names,2***”.

10. Для пошагового выполнения программы служит клавиша “***F*10**”. После каждого шага можно видеть, как изменяются значения внутренних переменных. Для входа “внутрь” функции служит клавиша “***F*11**”.

2. БИБЛИОТЕКА MPI

Библиотека ***MPI*** предназначена для многопроцессорных систем с обменом сообщениями. *MPI* – «интерфейс передачи сообщений» – это стандартизованный механизм для построения параллельных программ в модели обмена сообщениями [1,2].

Существуют стандартные «привязки» *MPI* к языкам *C/C++*, *Fortran*. Существуют реализации почти для всех суперкомпьютерных платформ, а также для сетей рабочих станций *UNIX* и *Windows*. *MPI* – попытка собрать самые лучшие возможности многих систем с передачами сообщений, улучшить и стандартизировать их. *MPI* – библиотека, а не язык. Она определяет имена, вызовы процедур и результаты их работы. Программы компилируются обычными компиляторами и связаны с MPI-библиотекой. *MPI* – спецификация, а не специфическая реализация. Все поставщики параллельных компьютерных систем предлагают MPI-реализацию для своих машин. Правильная MPI-программа должна быть способна выполняться на всех MPI-реализациях без изменений.

Процесс это исполнение программы на одном процессоре, безотносительно к тому, содержит эта программа внутри параллельные ветви или просто последовательный программный код. Приложения могут

быть разбиты на некоторое количество процессов, а процессы могут быть выполнены на кластере любой конфигурации.

Группа представляет собой совокупность процессов, каждый из которых имеет внутри группы уникальное имя, используемое для взаимодействия с другими процессами посредством коммуникатора группы.

Коммуникатор реализует обмены данными между процессами и их синхронизацию. Каждый коммуникатор имеет собственное коммуникационное пространство. Сообщения, использующие разные коммуникаторы, не оказывают влияния друг на друга и не взаимодействуют. Созданные группы процессов используют отдельный коммуникатор и имеют внутри группы номера от 0 до $n - 1$, где n – количество процессов в группе. Имеется начальная группа, которой принадлежат все процессы в реализации MPI.

Программирование в стандарте MPI заключается в том, что параллельная программа содержит код всех ветвей. MPI-загрузчиком запускается указываемое количество экземпляров программы. Каждый экземпляр определяет свой порядковый номер в группе, и в зависимости от номера и размера группы выполняет ту или иную ветвь алгоритма. Каждая ветвь имеет пространство данных, полностью изолированное от других ветвей. Обмениваются данными ветви в виде сообщений MPI. Все ветви запускаются загрузчиком одновременно. Количество ветвей фиксировано – в ходе работы порождение новых ветвей невозможно.

2.1. ОСНОВНЫЕ ФУНКЦИИ MPI

1. Инициализация библиотеки.

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

Обращение к ***MPI_Init*** должно быть первым обращение в MPI-программе. После инициации посредством ***MPI_Init*** определен коммуникатор ***MPI_COMM_WORLD***, задающий стандартную коммуникационную среду с именами процессов 0, ... , $n - 1$, где n – число процессоров, определенное при инициализации.

2. Определение числа процессов.

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

Значение *size* – это размер группы процессов в коммуникаторе *comm*. Если коммуникатор – *MPI_COMM_WORLD*, то вызов функции

MPI_Comm_size возвращает число процессов, которые пользователь запустил в этой программе.

3. Определение номера процесса.

int MPI_Comm_rank (MPI_Comm comm, int *rank);

Процессы в любой группе нумеруются последовательными целыми числами, начиная с 0, номер процесса называется рангом. Вызывая ***MPI_Comm_rank***, каждый процесс выясняет свой ранг в группе, связанной с коммуникатором *comm*. Таким образом, хотя каждый процесс в этой программе получает тот же самый номер *size*, каждый будет иметь различный номер в *rank.*

4. Передача сообщений.

int MPI_Send (void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm);

buf	начальный адрес передающего буфера
count	число элементов данных в передающем буфере
datatype	тип передаваемых данных
dest	номер принимающего процесса
tag	тэг сообщения
comm	коммуникатор

5. Прием сообщений.

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

buf	начальный адрес принимающего буфера
count	число элементов данных в принимающем буфере
datatype	тип принимаемых данных
source	номер передающего процесса
tag	тэг сообщения
comm	коммуникатор
status	статус объекта

Передаем или получаем количество последовательных элементов, тип которых указан в поле *datatype*, начиная с элемента по адресу *buf*. Длина сообщения *count* задается числом элементов, а не числом байт. Длина получаемого сообщения должна быть равна или меньше длины буфера получения, в противном случае будет иметь место ошибка переполнения. Если сообщение меньше размера буфера получения, то в

нем модифицируются только ячейки, соответствующие длине сообщения.

В дополнение к описанию данных сообщение несет информацию, которая используется, чтобы различать и выбирать сообщения. Эта информация состоит из фиксированного количества полей, которые в совокупности называются атрибутами сообщения. Эти поля таковы: *sourse, destination, tag, communicator, status* (номер процесса-отправителя, номер процесса-получателя, тэг, коммуникатор, статус).

Аргумент *comm* описывает коммуникатор, который используется в операции обмена. Целочисленный аргумент *tag* используется, чтобы различать типы сообщений. Диапазон значений тэга находится в пределах 0, …, *UB*, где верхнее значение *UB* зависит от реализации.

Процесс-получатель может задавать значение ***MPI_ANY_SOURCE*** для отправителя и/или значение ***MPI_ANY_TAG*** для тэга, определяя, что любой отправитель и/или тэг разрешен. Аргумент отправителя, если он отличен от *MPI_ANY_SOURCE*, задается как номер внутри группы процессов, связанной с тем же самым коммуникатором. Следовательно, диапазон значений для аргумента отправителя есть (0, ..., n – 1) U {*MPI_ANY_SOURCE*}, где n есть количество процессов в этой группе.

Операция приема допускает получение сообщения от произвольного отправителя, но в операции посылки должен быть указан уникальный получатель. Допускается ситуация, когда имена источника и получателя совпадают, то есть процесс может посылать сообщение самому себе (однако это небезопасно, поскольку может привести к дедлоку (deadlock)). Если источник или тэг принимаемого сообщения неизвестны, или требуется возвратить коды ошибок для каждого запроса, то эта информация возвращается с помощью аргумента *status* операции *MPI_Recv*. В языке Си *status* есть структура, следовательно, *status.MPI_SOURCE, status.MPI_TAG* и *status.MPI_ERROR* содержат источник, тэг и код ошибки принятого сообщения.

6. Нормальное закрытие библиотеки:

```
MPI_Finalize();
```

7. Определение времени выполнения.

```
double MPI_Wtime();
```

Функция MPI_Wtime возвращает количество секунд, представляя полное время по отношению к некоторому моменту времени. Эта функция универсальна, поскольку возвращает секунды, а не количество тактовых импульсов. Функцию можно использовать для определения времени выполнения приложения.

2.2. КОЛЛЕКТИВНЫЕ ФУНКЦИИ MPI

Функции выполняются всеми процессами группы, посредством вызовов коммуникационных процедур с соответствующими аргументами во всех процессах. Ключевым аргументом служит коммуникатор, определяющий участвующие процессы. Вызов коллективной функции является корректным, только если произведен из всех процессов с этим коммуникатором в качестве аргумента. В этом и заключается коллективность: либо функция вызывается всем коллективом процессов, либо никем.

1. Барьерная синхронизация процессов.

int MPI_Barrier (MPI_Comm comm);

Вызов этой процедуры в процессе не возвращает кода возврата до тех пор, пока все процессы коммуникатора *comm* не произведут вызовов ***MPI_Barrier***.

2. Широковещательная передача сообщения.

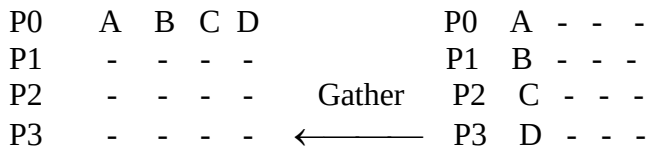
int MPI_Bcast (void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

buffer	начальный адрес буфера
count	число элементов в буфере
datatype	тип элементов
root	номер передающего процесса
comm	коммуникатор

Функция широковещательной передачи ***MPI_Bcast*** посылает сообщение из корневого процесса всем процессам коммуникатора, включая себя. Она вызывается всеми процессами с одинаковыми аргументами для *comm*, *root*. В момент возврата управления содержимое корневого буфера обмена будет скопировано во все процессы.

3. Сбор данных из всех процессов в один процесс коммуникатора.

int MPI_Gather (void* send_buf, int send_count, MPI_Datatype send_type,
void* recv_buf, int recv_count, MPI_Datatype recv_type,
int root, MPI_comm comm)



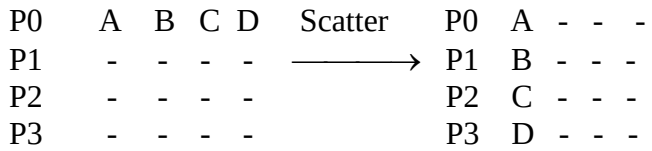
Каждый процесс коммуникатора *comm*, включая корневой, посылает содержимое передающего буфера корневому процессу. Все процессы должны послать и только корневой должен принять в требуемом порядке.

int MPI_Gatherv(void* send_buf, int send_count, MPI_Datatype send_type,
void* recv_buf, int *recv_count, int *displs, MPI_Datatype recv_type,
int root, MPI_comm comm);

По сравнению с ***MPI_Gather*** при использовании функции ***MPI_Gatherv*** разрешается принимать от каждого процесса переменное число элементов данных.

4. Раздача данных из одного всем процессам коммуникатора.

int MPI_Scatter (void* send_buf, int send_count, MPI_Datatype send_type,
void* recv_buf, int recv_count, , MPI_Datatype recv_type,
int root, MPI_Comm comm)



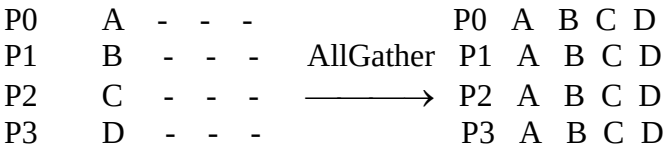
Каждый процесс коммуникатора *comm*, включая корневой, принимает содержимое буфера корневого процесса. Все процессы должны принимать и только корневой должен посылать в требуемом порядке.

int MPI_Scatterv (void* send_buf, int *send_count, int *displs,
MPI_Datatype send_type,
void* recv_buf, int recv_count, MPI_Datatype recv_type,
int root, MPI_Comm comm);

Операция ***MPI_Scatter*** обратна операции ***MPI_Gather***.

5. Сбор данных из всех процессов с получением результата сборки всеми процессами коммуникатора.

int MPI_Allgather (void* send_buf, int send_count, MPI_Datatype send_type,
void* recv_buf, int recv_count, MPI_Datatype recv_type,
MPI_comm comm)



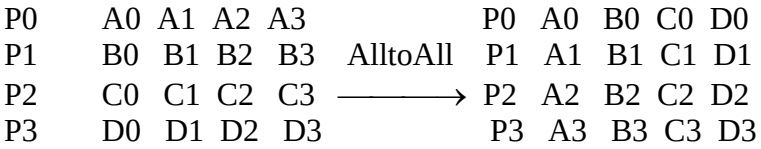
Функцию ***MPI_Allgather*** можно представить как функцию ***MPI_Gather***, где результат принимают все процессы, а не только корневой.

int MPI_Allgatherv (void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int *recvcounts, int *displs,
MPI_Datatype recvtype, MPI_Comm comm);

MPI_Allgatherv можно представить как ***MPI_Gatherv***, но при ее использовании результат получают все процессы, а не только корневой.

6. Раздача/сборка данных из всех процессов во все процессы

int MPI_Alltoall (void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype,
MPI_Comm comm)



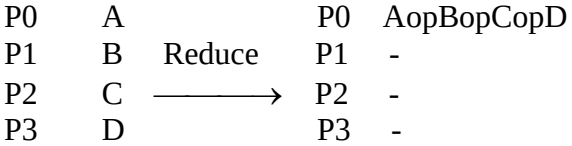
MPI_Alltoall – расширение функции ***MPI_Allgather*** для случая, когда каждый процесс посылает различные данные каждому получателю. ***j***-й блок, посланный процессом ***i,*** принимается процессом ***j*** и помещается в ***i***-й блок буфера *recvbuf***.**

int MPI_Alltoallv (void* sendbuf, int *sendcounts, int *sdispls,
MPI_Datatype sendtype,
void* recvbuf,int *recvcounts,int *rdispls,
MPI_Datatype recvtype, MPI_Comm comm);

MPI_Alltoallv обладает большей гибкостью, чем функция *MPI_Alltoall*, поскольку размещение данных на передающей стороне определяется аргументом *sdispls*, а на стороне приема – независимым аргументом *rdispls*. *j*-й блок, посланный процессом i, принимается процессом *j* и помещается в *i*-й блок *recvbuf*.

7. Определены глобальные операции редукции такие как: сложение, максимум, минимум или другие операции, определенные пользователем.

```
int MPI_Reduce (void* operand, void* result, int count,
                MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
```



Результат глобальной операции *op* передается в один (***MPI_Reduce***) или все процессы коммуникатора (***MPI_Allreduce***).

```
int MPI_Allreduce(void* sbuf, void* rbuf, int count,
                  MPI_Datatype datatype, MPI_Op op,  MPI_Comm comm);
```

2.3. ЭФФЕКТИВНОСТЬ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

При создании параллельных алгоритмов и соответствующих им параллельных программ к традиционным проблемам установления корректности последовательных алгоритмов добавляется ряд новых задач, связанных со спецификой асинхронно протекающих процессов. Среди этих новых задач особо важную роль играет выявление однозначности и беступиковости параллельных программ. **Однозначность** предполагает независимость результатов вычислений от скоростей протекания различных процессов и порядка их выполнения, а **беступиковость** – отсутствие бесконечных ожиданий и блокировок одних процессов другими.

Одной из главных характеристик параллельной программы является **ускорение**, которое определяется выражением:

$$R = T_1 / T_n,$$

где T_1 – время решения задачи на однопроцессорной системе, а T_n – время решения той же задачи на *n*-процессорной системе.

Ускорение зависит от потенциального параллелизма задачи и параметров аппаратуры.

Эффективностью параллельной программы называется величина

$$E=R/n,$$

где n – количество процессоров системы, R – ускорение программы на многопроцессорной системе. Если при выполнении программы достигается максимальное ускорение $(R = n)$, то $E = 1$.

Критерии оценки качества параллельных программ:

- ускорение программы от числа процессоров,
- эффективность параллельной программы,
- величина затрат на синхронизацию,
- величина затрат на обмен сообщениями,
- влияние размера задачи на ускорение,
- максимальное число занятых процессоров и т. д.

При выполнении параллельной программы основным препятствием к достижению максимального ускорения являются потери на межпроцессорный обмен сообщений. Эти потери могут не только снизить ускорение вычислений, но и замедлить вычисления по сравнению с однопроцессорным вариантом. Поэтому основным принципом при создании параллельных алгоритмов должен быть **крупноблочный иерархический подход**, при котором параллельный алгоритм строится из возможно более крупных и редко взаимодействующих блоков.

3. ПРИМЕРЫ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

1. Программа выводит на экран от каждого процесса сообщение: «*Hello from process i of n*», где *i* – номер процесса в *MPI_COMM_WORLD*, а *n* – размер *MPI_COMM_WORLD*. Программа выполняется для заданного количества процессов *n*.

```
{   int rank, n;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &n );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    cout<<"Hello from process"<< rank <<" of "<<n <<endl;
    MPI_Finalize();
    return 0;
}
```

2. Процесс с номером 0 посылает сообщение: «*Hello*» процессу с номером 1, процесс с номером 1 принимает данное сообщение. Каждый процесс выводит на экран свой номер в *MPI_COMM_WORLD*. Программа может выполняться только для количества процессов $n = 2$. Программа неверна. Функция *MPI_Recv* в процессе с номером 0 не может быть выполнена.

```
{  char message[20];
   int myrank;
   MPI_Status status;
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
   if (myrank == 0)
   { strcpy(message,"Hello");
     MPI_Send(message,strlen(message),MPI_CHAR,1,0,
              MPI_COMM_WORLD);
   }
   MPI_Recv(message, 20, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
            &status);
   cout<<myrank<<endl;
   MPI_Finalize();
}
```

3. Правильный вариант предыдущей программы**.** Процесс с номером 0 посылает сообщение: «*Hello*» процессу с номером 1, процесс с номером 1 принимает данное сообщение. Процесс 1-й выводит на экран свой номер в *MPI_COMM_WORLD*. Программа выполняется только для количества процессов $n = 2$.

```
{ char message[20];
  int myrank;
  MPI_Status status;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
  if (myrank == 0)
   { strcpy(message,"Hello");
     MPI_Send(message,strlen(message),MPI_CHAR,1,0,
              MPI_COMM_WORLD);
    }
   else
    { MPI_Recv(message, 20, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
               &status);
      cout<<myrank<<endl;
    }
    MPI_Finalize();
}
```

4. Еще один правильный вариант предыдущей программы**.** Процесс с номером 0 посылает сообщение: «Hello» процессам с номерами 0,1, процессы с номерами 0,1 принимают данное сообщение. Каждый процесс выводит на экран свой номер в *MPI_COMM_WORLD*. Программа выполняется только для количества процессов $n = 2$.

```
{  char mess[20];    int myrank;
   MPI_Status status;
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
   if (myrank == 0)
    { strcpy(message,"Hello");
      MPI_Send(mess,strlen(message),MPI_CHAR,1,0,
               MPI_COMM_WORLD);
      MPI_Send(mess,strlen(message),MPI_CHAR,0,0,
               MPI_COMM_WORLD);
    }
 MPI_Recv(mess, 20, MPI_CHAR, 0, 0,MPI_COMM_WORLD, &status);
 cout<<myrank<<endl;
 MPI_Finalize();
}
```

5**.** Процесс с номером 0 посылает сообщение: «*Hello*» всем процессам коммуникатора *MPI_COMM_WORLD*, исключая процесс с номером 0. Все процессы принимают данное сообщение. Каждый процесс выводит на экран свой номер в *MPI_COMM_WORLD*. Программа выполняется для заданного количества процессов n.

```
{   char message[20];  int myrank, n,i;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    MPI_Comm_size( MPI_COMM_WORLD, &n );
    if (myrank == 0)
    {     strcpy(message,"Hello");
          for (i=1;i<n;i++)
             MPI_Send(message,strlen(message),MPI_CHAR,i,0,
                      MPI_COMM_WORLD);
    }
    else
    { MPI_Recv(message, 20, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
               &status);
      cout<<" message recive process"<<myrank<<endl;
    }
    MPI_Finalize();    }
```

6. Процесс с номером 0 посылает сообщение: «*Hello*» всем процессам коммуникатора *MPI_COMM_WORLD*, включая процесс с номером 0. Все процессы принимают данное сообщение. Каждый процесс выводит на экран свой номер в *MPI_COMM_WORLD*. Программа выполняется для заданного количества процессов *n*.

```
{ char message[20];   int myrank, n,i;
  MPI_Status status;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
  MPI_Comm_size( MPI_COMM_WORLD, &n );
  if (myrank == 0)
  { strcpy(message,"Hello");
    for (i=0;i<size;i++)
       MPI_Send(message,strlen(message),MPI_CHAR,i,0,
                MPI_COMM_WORLD);
  }
  MPI_Recv(message, 20, MPI_CHAR, 0,
                    0,MPI_COMM_WORLD,&status);
  cout<<" message recive process"<<myrank<<endl;
  MPI_Finalize();
}
```

7. Передача данного по конвейеру. Процесс с номером 0 посылает некоторое введенное значение процессу с номером 1. Каждый из оставшихся процессов с номером *i* получает значение от процесса с номером *i* - 1, посылает процессу с номером *i* + 1. Программа выполняется для заданного количества процессов *n*.

```
{ int rank, value, n;
  MPI_Status status;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &n );
  if (rank == 0)
    { cin >> value;
      MPI_Send(&value,1,MPI_INT,rank+1,0,MPI_COMM_WORLD );
    }
    else
    { MPI_Recv(&value,1,MPI_INT,rank-1,0, MPI_COMM_WORLD,
               &status);
      MPI_Send(&value,1,MPI_INT,rank+1,0,MPI_COMM_WORLD );
    }
  cout<< "Process "<< rank<<" got "<< value <<endl;
  MPI_Finalize( );   }
```

Программа неверна. Функция *MPI_Send* в процессе с номером *n* - 1 не может быть выполнена.

8. Правильный вариант предыдущей программы. Передача данного по конвейеру. Процесс с номером 0 посылает введенное значение процессу с номером 1. Каждый из оставшихся процессов с номером *i* получает значение от процесса с номером *i* - 1, посылает процессу с номером *i* + 1. Программа выполняется для заданного количества процессов *n*.

```
{ int rank, value, n;
  MPI_Status status;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &n );
  if (rank == 0)
  { cin >> value;
    MPI_Send(&value,1,MPI_INT,rank+1,0,MPI_COMM_WORLD );
  }
  else
  {   MPI_Recv(&value,1,MPI_INT,rank-1,0,
                  MPI_COMM_WORLD,&status);
     if (rank < n - 1)
      MPI_Send(&value,1,MPI_INT,rank+1,0,MPI_COMM_WORLD );
 }
   cout<< "Process "<< rank<<" got "<< value <<endl;
   MPI_Finalize( );
}
```

9. Передача некоторого значения из корневого процесса всем процессам приложения. Программа неверна. Функция *MPI_Bcast* вызвана только в процессе с номером 0.

```
{    int rank, value;
     MPI_Init( &argc, &argv );
     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
     value = rank;
     if (rank == 0)
     {value = 10;
      MPI_Bcast( &value, 1, MPI_INT, 0, MPI_COMM_WORLD );
     }
     cout<<"Process "<<rank<<"got "<<value<<endl;
     MPI_Finalize();
}
```

10. Правильный вариант предыдущей программы. Передача некоторого значения из корневого процесса всем процессам приложения.

```
{    int rank, value;
     MPI_Init( &argc, &argv );
     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
     value = rank;
     if (rank == 0)    value = 10;
     MPI_Bcast( &value, 1, MPI_INT, 0, MPI_COMM_WORLD );
     cout<<"Process "<<rank<<<<"got "<<value<<endl;
     MPI_Finalize( );
     return 0;
}
```

11. Еще один правильный вариант предыдущей программы.

```
{    int rank, value;
     MPI_Init( &argc, &argv );
     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
     value = rank;
     if (rank == 0)
      {    value = 10;
           MPI_Bcast( &value, 1, MPI_INT, 0, MPI_COMM_WORLD );
      }
      else
           MPI_Bcast( &value, 1, MPI_INT, 0, MPI_COMM_WORLD );
     cout<<"Process "<<rank<<<<"got "<<value<<endl;
     MPI_Finalize( );
     return 0;
}
```

12. Сбор данных. Каждый процесс, включая корневой, посылает содержимое своего буфера в корневой процесс. Корневой процесс получает сообщения, располагая их в порядке возрастания номеров процессов.

```
{    int summ=0, myid, i, root=0, *rbuff=0;
     int gsize, msize, sendarray[10];
     MPI_Comm comm = MPI_COMM_WORLD;
     MPI_Init(&argc,&argv);
     MPI_Comm_size(comm, &gsize);
     MPI_Comm_rank(comm, &myid);
          msize = gsize*10;
     if (myid == root)
     rbuff = new int[msize];
```

```
          for (i = 0; i < 10; ++i)
          sendarray[i] = myid;

MPI_Gather(sendarray,10,MPI_INT,rbuff,10,MPI_INT,root,comm);
          if (myid == root)
     {       for (i = 0; i < msize; ++i)
                       summ += rbuff[i];
                cout<<summ<<endl;
         }
    MPI_Finalize();
}
```

13. Раздача данных. Корневой процесс раздает данные всем процессам коммуникатора *MPI_COMM_WORLD* в порядке возрастания номеров процессов. Каждый процесс, включая корневой, принимают содержимое буфера.

```
{    int rank, k=77, i, *buf=0, size;
     MPI_Init( &argc, &argv );
     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
     MPI_Comm_size( MPI_COMM_WORLD, &size );
     if (rank==0)
     { buf= new int [size];
       for (i=0;i<size;++i)
            buf[i]=size-i;
      }
   MPI_Scatter(buf, 1, MPI_INT, &k,1,MPI_INT,0,
               MPI_COMM_WORLD );
  cout<<" Process "<<rank<<" k = "<< k<<endl;
  MPI_Finalize( );
}
```

14. Сложение значений *j* каждого процесса коммуникатора *MPI_COMM_WORLD* в переменную *sum_j* корневого процесса**.**

```
{ int i,j,sum_j;
       MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &i);
  j = i % 2;
  MPI_Reduce(&j, &sum_j, 1, MPI_INT, MPI_SUM, 0,
             MPI_COMM_WORLD);
  if (i == 0)  cout<<jj<<endl;
  MPI_Finalize();
}
```

15. Программа вычисления числа π. Число π можно определить:

$$\int_0^1 \frac{dx}{1+x^2} = arctg(1)\text{-}arctg(0) = \pi/4.$$

Вычисление интеграла заменяют вычислением суммы для заданного количества интервалов разбиения *n*:

 , где: $x_i = \frac{1}{n}(i - 0.5)$.

Параллельная программа вычисления числа π

```
{ int n, myid, numprocs, i;
  double mypi, pi, h, sum, x, t1, t2,PI=3.141592653589793;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (1)
      {  if (myid == 0)
        { cout<<"Enter the number of intervals: (0 quits) ";
           cin>>n;
           t1 = MPI_Wtime();
         }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0)  break;
        else
        {  h  = 1.0/ (double) n;    sum = 0.0;
          for (i = myid +1; i <= n; i+= numprocs)
             {    x = h * ( (double)i - 0.5);
                   sum += (4.0 / (1.0 + x*x));
             }
           mypi = h * sum;
           MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM,
                       0,MPI_COMM_WORLD);
           if (myid == 0)
             {  t2 = MPI_Wtime();
                 cout<<"pi is approximately "<<pi<<endl;
                 cout<<"Error is "<<fabs(pi - PI)<<endl;
                 cout<<"time "<< t2-t1<<endl;
             }
        }
      }
      MPI_Finalize();
  }
```

Распределим вычисления между *numprocs* процессами следующим образом: каждый процесс вычисляет соответствующую частичную сумму для *n*/*numprocs* интервалов разбиения.

Обращение к *MPI_Init* должно быть первым обращением в MPI-программе, оно устанавливает «среду» MPI. В каждом выполнении программы может выполняться только один вызов *MPI_Init*. Коммуникатор *MPI_COMM_WORLD* описывает состав процессов и связи между ними. Вызов *MPI_Comm_size* возвращает в *numprocs* число процессов, которые пользователь задал при запуске программы. Процессы в любой группе нумеруются последовательными целыми числами, начиная с 0. Вызывая *MPI_Comm_rank*, каждый процесс выясняет свой номер (*myid*) в группе, связанной с коммуникатором *MPI_COMM_WORLD*.

Затем корневой процесс (который имеет *myid* = 0) получает от пользователя значение числа интервалов разбиения *n*:

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Первые три параметра обозначают адрес, количество и тип данных. Четвертый указывает номер источника данных (корневой процесс), пятый параметр – название коммуникатора. Таким образом, после обращения к *MPI_Bcast* все процессы имеют значение *n* и собственные идентификаторы, что является достаточным для каждого процесса, чтобы вычислить *mypi* – свой вклад в вычисление π. Для этого каждый процесс вычисляет область каждого прямоугольника, начинающегося с *myid* + l.

Затем все значения *mypi*, вычисленные индивидуальными процессами, суммируются с помощью вызова:

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM,0,
                                    MPI_COMM_WORLD);
```

Первые два параметра описывают адреса источника и результата, соответственно. Третий и четвертый параметр описывают число данных и их тип, пятый параметр – тип арифметико-логической операции, шестой – номер процесса для размещения результата. Функция *MPI_Wtime()* используется для измерения времени исполнения участка программы, расположенного между двумя включениями в программу этой функции. Затем управление передается на начало цикла. Этим предоставляется возможность задать новое *n* и повысить точность вычислений. Когда пользователь вводит 0 в ответ на запрос о новом значении *n*, цикл завершается, и все процессы выполняют вызов *MPI_Finalize*, после которого операции MPI

выполняться не будут.

16. Рассмотрим задачу получения значений элементов массива B из исходного массива A размерности N. Например, каждый элемент $B[i]=k*A[i]$, где k – константа. Вычисления значений результирующего массива – независимые действия, поэтому параллельный алгоритм заключается в следующем.

1. Разбиение массива A на $m = N / size$ частей, где $size$ – количество процессов.
2. Рассылка частей массива A по процессам.
3. Каждый процесс вычисляет свою часть результирующего массива B $B[i]=m*A[i]$.
4. Получение частей массива B от всех процессов.

```
{ int myrank, size,i,m;
  double *A=0,*B=0;    // входной и выходной массив
  int     N;           // размер массивовов
  double *a=0,*b=0;    // части массивов для каждого процесса

   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   if (myrank == 0)
     {cout<<"Vvedite N";   cin>>N; }
   MPI_Bcast(&N,1,MPI_INT,0,MPI_COMM_WORLD);
   if (myrank == 0)         // выделение памяди
     { A = new double[N];   // для входного
       B = new double[N];   // и выходного массивов
     }
       // определения количества элементов в каждом процессе
    m = N/size;
    a = new double[m];   b = new double[m];
    if (myrank == 0)
        f1(A,N);    // заполнение исходной матрицы A

   MPI_Scatter(A, m,  MPI_DOUBLE, a, m, MPI_DOUBLE, 0,
               MPI_COMM_WORLD );
                   // рассылка частей матрицы по процессам

   f(a,b,m,myrank+1);    // основная функция обработки

   MPI_Gather(b, m, MPI_DOUBLE, B, m,  MPI_DOUBLE, 0,
              MPI_COMM_WORLD );
                     // сбор результатов в корневой процесс
   if (myrank == 0)
       f2(B,N);         // вывод результата
    MPI_Finalize();
}
```

17. В предыдущем примере предполагалось, что размерности

массивов кратны количеству процессов. В общем случае, это не так. Каждый процесс должен определить количество элементов массива *A*, которые будут вычисляться в нем (разные процессы, возможно, будут иметь разные значения *m*).

```
{ int myrank, size,i,N;
  double *A=0,*B=0;          // входной и выходной массив
  double *a=0,*b=0;    // части массивов для каждого процесса
  int *sendcounts=0, //массив количества элементов в процес-
се
      *displs=0;       // массив смещений
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   if (myrank == 0)  {cout<<"Vvedite N";   cin>>N; }
   MPI_Bcast(&N,1,MPI_INT,0,MPI_COMM_WORLD);
   if (myrank == 0)
      // выделение памяди для входного и выходного массивов
     { A = new double[N];  B = new double[N]; }
          // определения количества элементов в каждом про-
цессе
    m = N/size+ (N%(size)>(myrank)?1:0);
    sendcounts = new int[size];
                 // заполнение массива sendcounts
    MPI_Gather(&m,1,MPI_INT,sendcounts,1,MPI_INT,0,
               MPI_COMM_WORLD);
    displs =  new int[size];  // заполнение массива displs
    displs[0] = 0;
    for (i = 1;i<size; ++i)
       displs[i] = displs[i-1] + sendcounts[i-1];
    a = new double[m];   b = new double[m];
    if (myrank == 0)
          f1(A,N);
        // заполнение исходной матрицы A
   MPI_Scatterv(A, sendcounts, displs,  MPI_DOUBLE, a, m,
                MPI_DOUBLE, 0,  MPI_COMM_WORLD );
                 // рассылка частей матрицы по процессам
   f(a,b,m,myrank+1);     //  основная функция обработки

   MPI_Gatherv(b, m, MPI_DOUBLE, B, sendcounts, displs,
               MPI_DOUBLE, 0, MPI_COMM_WORLD );
                // посылка результата в корневой процесс
    if (myrank == 0)  f2(B,N);       // вывод результата
    MPI_Finalize();
    }
```

4. ЗАДАНИЯ ДЛЯ ВЫПОЛНЕНИЯ

4.1. ФУНКЦИИ ОБМЕНОВ

Цель работы: изучение аппаратных и программных средств для организации параллельных вычислений на кластере, создание и запуск параллельных программ, осуществляющих обмены данными.

Порядок выполнения задания

1. Изучить структуру учебного вычислительного кластера.

2. Ознакомиться с процессом создания и запуском параллельных программ на кластере с помощью пакета MPICH.

3. Изучить основные функции технологии MPI .

4. Для заданного n и заданной топологии процессов (см. задания для самостоятельной работы) с помощью функций обменов осуществить обмен данными.

5. Выполнить написанную параллельную программу на вычислительном кластере для различных начальных данных.

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Топология процессов – полносвязная система из n (n – любое заданное значение) процессов (рис. 1). С помощью функций обменов осуществить:

- передачу данных от процесса № i процессу № j, где $0 \le i, j < n$;
- передачу различных данных (каждому процессу свое данное) от процесса № i всем остальным процессам.

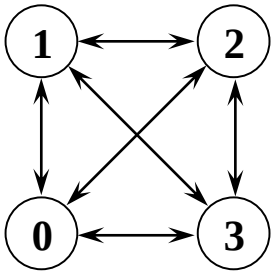


Рис. 1. Топология «полностью связанная система» при $n = 4$

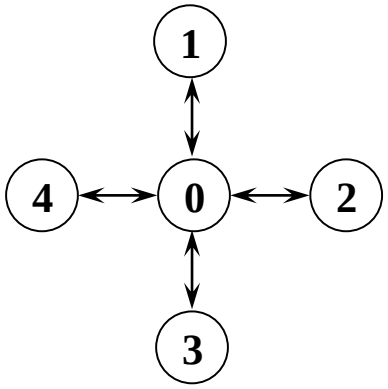


Рис. 2. Топология «звезда» при $n = 5$

2. Топология процессов – полносвязная система из n (n – любое заданное значение) процессов (рис. 1). С помощью функций обменов осуществить:

- передачу данных от процесса № i процессу № j, где $0 \le i, j < n$;

- передачу различных данных (каждому процессу свое данное) от каждого процесса каждому.

3. Топология процессов – «звезда» из n (n – любое заданное значение) процессов (рис. 2). С помощью функций обменов осуществить:
- передачу данных от процесса № i процессу № j, где $0 \le i, j < n$;
- передачу данных от процесса № i остальным процессам $0 \le i < n$.

4. Топология процессов – «звезда» из n (n – любое заданное значение) процессов (рис. 2). С помощью функций обменов осуществить:
- передачу данных от процесса № i процессу № j, где $0 \le i, j < n$;
- передачу различных данных (каждому процессу свое данное) от процесса № i остальным процессам $0 \le i < n$.

5. Топология процессов – «звезда» из n (n – любое заданное значение) процессов (рис. 2). С помощью функций обменов осуществить:
- передачу данных от процесса № i процессу № j, где $0 \le i, j < n$;
- передачу различных данных (каждому процессу свое данное) от каждого процесса каждому.

6. Топология процессов – «линейный массив» из n (n – любое заданное значение) процессов (рис. 3). С помощью функций обменов осуществить:
- передачу данных от процесса № i процессу № j, где $0 \le i, j < n$, $i < j$;
- передачу различных данных (каждому процессу свое данное) от процесса № i всем остальным процессам с большим номером.

7.Топология процессов – «линейный массив» из n (n – любое заданное значение) процессов (рис. 3). С помощью функций обменов осуществить:
- передачу данных от процесса № i процессу № j, где $0 \le i, j < n$, $i < j$;
- передачу различных данных (каждому процессу свое данное) от каждого процесса каждому с большим номером.

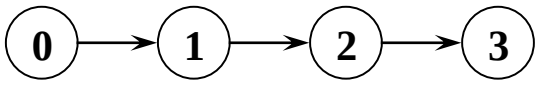


Рис. 3. Топология «линейный массив » при $n = 4$

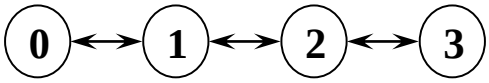


Рис. 4. Топология «линейный массив» при $n = 4$

8. Топология процессов – «линейный массив» из n (n – любое заданное значение) процессов (рис. 4). С помощью функций обменов осуществить:

- передачу данных от процесса № i процессу № j, где $0 \le i, j < n$;
- передачу данных от процесса № i всем остальным процессам, где $0 \le i < n$.

9.Топология процессов – «линейный массив» из n (n – любое заданное значение) процессов (рис. 4). С помощью функций обменов осуществить:

- передачу данных от процесса № i процессу № j, где $0 \le i, j < n$;
- передачу различных данных (каждому процессу свое данное) от процесса № i всем остальным процессам, где $0 \le i < n$.

10. Топология процессов – «линейный массив» из n (n – любое заданное значение) процессов (рис. 4). С помощью функций обменов осуществить:

- передачу данных от процесса № i процессу № j, где $0 \le i, j < n$;
- передачу различных данных (каждому процессу свое данное) от каждого процесса каждому.

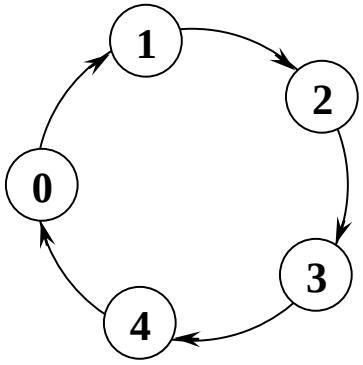


Рис. 5. Топология «кольцо» при $n = 5$

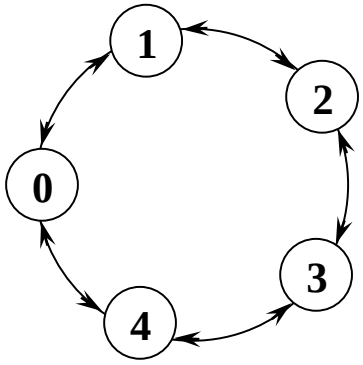


Рис. 6. Топология «кольцо» при $n = 5$

11. Топология процессов – «кольцевой линейный массив» из n (n – любое заданное значение) процессов (рис. 5). С помощью функций обменов осуществить:

- передачу данных от процесса № i процессу № j, где $0 \le i, j < n$;
- передачу данных от процесса № i всем остальным процессам, где $0 \le i < n$.

12.Топология процессов – «кольцевой линейный массив» из n (n – любое заданное значение) процессов (рис. 5). С помощью функций обменов осуществить:

- передачу данных от процесса № i процессу № j, где $0 \le i, j < n$;
- передачу различных данных (каждому процессу свое данное) от процесса № i всем остальным процессам, где $0 \le i < n$.

13.Топология процессов – «кольцевой линейный массив» из n (n – любое заданное значение) процессов (рис. 5). С помощью функций обменов осуществить:

- передачу данных от процесса № i процессу № j, где $0 \le i, j < n$;
- передачу различных данных (каждому процессу свое данное) от каждого процесса каждому.

14. Топология процессов – «кольцевой линейный массив» из n (n – любое заданное значение) процессов (рис. 6). С помощью функций обменов осуществить:

- передачу данных от процесса № i процессу № j, где $0 \le i, j < n$;
- передачу данных от процесса № i всем остальным процессам, где $0 \le i < n$.

15.Топология процессов – «кольцевой линейный массив» из n (n – любое заданное значение) процессов (рис. 6). С помощью функций обменов осуществить:

- передачу данных от процесса № i процессу № j, где $0 \le i, j < n$;
- передачу различных данных (каждому процессу свое данное) от процесса № i всем остальным процессам, где $0 \le i < n$.

16. Топология процессов – «кольцевой линейный массив» из n (n – любое заданное значение) процессов (рис. 6). С помощью функций обменов осуществить:

- передачу данных от процесса № i процессу № j, где $0 \le i, j < n$;
- передачу различных данных (каждому процессу свое данное) от каждого процесса каждому.

17.Топология процессов – «двумерная решетка» из n^2 процессов (рис. 7). С помощью функций обменов осуществить:

- передачу данных от процесса № 0 процессу № j, где $0 \le j < n^2$;

18.Топология процессов – «двумерная решетка» из n^2 процессов (рис. 7). С помощью функций обменов осуществить:

- передачу данных от процесса № 0 всем остальным процессам;

19. Топология процессов – «двумерная решетка» из n^2 процессов (рис. 7). С помощью функций обменов осуществить:

- передачу данных от процесса № i процессу № j, где $0 \le i, j < n^2$;

20. Топология процессов – «двумерная решетка» из n^2 процессов, (рис. 7). С помощью функций обменов осуществить:

- передачу данных от процесса № i всем остальным процессам.

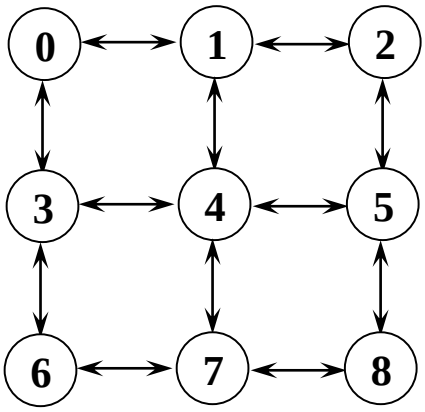


Рис. 7. Топология «двумерная решетка» при $n = 3$

21.Топология процессов – «двоичное полное дерево» из $\sum_{i=1}^{n} 2^{i-1}$ процессов (рис. 8). С помощью функций обменов осуществить:

- передачу данных от процесса № 0 процессу № j, где $0 \le j < n$;

22. Топология процессов – «двоичное полное дерево» из $\sum_{i=1}^{n} 2^{i-1}$ процессов (рис. 8). С помощью функций обменов осуществить:

- передачу данных от процесса № 0 всем остальным процессам.

23. Топология процессов – «гиперкуб» из 2^n процессов (рис. 9). С помощью функций обменов осуществить:

- передачу данных от процесса № 0 процессу № j, где $0 \le j < n$;

24. Топология процессов – «гиперкуб» из 2^n процессов (рис. 9). С помощью функций обменов осуществить:

- передачу данных от процесса № 0 всем остальным процессам.

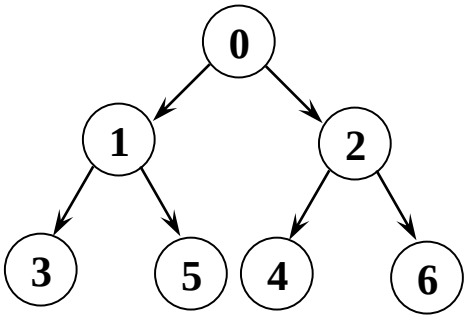


Рис. 8. Топология «полное двоичное дерево» при $n = 3$

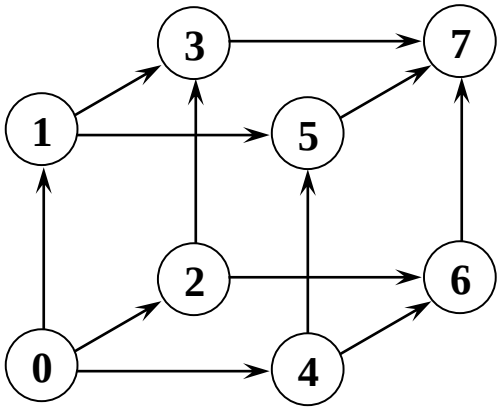


Рис. 9. Топология «гиперкуб» при $n = 3$

4.2. КОЛЛЕКТИВНЫЕ ФУНКЦИИ

Цель работы: изучение коллективных функций MPI, запуск параллельных программ с коллективными функциями.

Порядок выполнения задания

1. Изучить основные коллективные функции MPI и особенности их использования.
2. Разработать параллельную программу для выполнения задания (см. задания для самостоятельной работы) двумя способами:
   - с использованием соответствующих коллективных функций;
   - без коллективных функций – через обмен сообщениями (функции *Send*, *Recv*).
3. Выполнить написанную параллельную программу на вычислительном кластере для различных начальных данных ($100 \le m, n \le 10000$).
4. Сравнить результаты выполнения программы в обоих случаях.

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. ***MPI_Gather***. В каждом из p процессов создать матрицу A $n \times m$, заполненную значениями $a_{ij} = rank$, где *rank* – номер процесса. Каждый процесс посылает k-ую строку своей матрицы в корневой процесс. В корневом процессе с номером q получить вектор B из $m \cdot p$ элементов, составленный из элементов строк с номером k матриц A каждого процесса (сначала элементы 0-го процесса, затем 1-го и т.д.).

2. ***MPI_Gather***. В каждом из p процессов создать матрицу A $n \times m$, заполненную значениями $a_{ij} = rank + i$, где $rank$ – номер процесса. Каждый процесс посылает k-ый столбец своей матрицы в корневой процесс. В корневом процессе с номером q получить вектор B из $n \cdot p$ элементов, составленный из элементов столбцов с номером k матриц A каждого процесса (сначала элементы 0-го процесса, затем 1-го и т.д.).

3. ***MPI_Scatter***. В каждом из p процессов создать матрицу A $n \times m$, заполненную значениями $a_{ij} = rank$, где $rank$ – номер процесса. В корневом процессе с номером q создать вектор B из $m \cdot p$ элементов, заполненный значениями $b_i = i$. Корневой процесс посылает в каждый из p процессов по m соответствующих элементов (сначала m элементов в 0-й процесс, затем следующие m элементов в 1-й и т.д.). Каждый процесс должен разместить полученные элементы в k-ую строку своей матрицы A.

4. ***MPI_Scatter***. В каждом из p процессов создать матрицу A $n \times m$, заполненную значениями $a_{ij} = rank$, где $rank$ – номер процесса. В корневом процессе с номером q создать вектор B из $n \cdot p$ элементов, заполненный значениями $b_i = i$. Корневой процесс посылает в каждый из p процессов по n элементов (сначала n элементов в 0-й процесс, затем следующие n элементов в 1-й и т.д.). Каждый процесс должен разместить полученные элементы в k-ый столбец своей матрицы A.

5. ***MPI_Gather***. В каждом из p процессов создать матрицу A $n \times m$, заполненную значениями $a_{ij} = rank$, где $rank$ – номер процесса. Каждый процесс посылает k-ую строку своей матрицы в корневой процесс. В корневом процессе с номером q получить матрицу B размерности $p \times m$, составленную из k –ых строк матриц A каждого процесса (i-ая строка матрицы B – k-ая строка матрицы A i-го процесса).

6. ***MPI_Gather***. В каждом из p процессов создать матрицу A $n \times m$, заполненную значениями $a_{ij} = rank$, где $rank$ – номер процесса. Каждый процесс посылает k-ый столбец своей матрицы в корневой процесс. В корневом процессе с номером q получить матрицу B размерности $n \times p$, составленную из k-*ых* столбцов матриц A каждого процесса (i-ый столбец матрицы B – k-ый столбец матрицы A i-го процесса).

7. ***MPI_Scatter***. В каждом из p процессов создать матрицу A $n \times m$, заполненную значениями $a_{ij} = rank$, где $rank$ – номер процесса. В корневом процессе с номером q задать матрицу B размерности $p \times m$, заполненную значениями $b_{ij} = i$. Корневой процесс посылает в каждый процесс соответствующую строку матрицы B из m элементов(0-ая строка

матрицы B – в 0-й процесс, 1-ая – в 1-й и т.д.). Каждый процесс должен разместить полученные элементы в k-ую строку своей матрицы A.

8. ***MPI_Scatter***. В каждом из p процессов создать матрицу A $n \times m$, заполненную значениями $a_{ij} = rank$, где $rank$ – номер процесса. В корневом процессе с номером q задать матрицу B размерности $n \times p$, заполненную значениями $b_{ij} = i$. Корневой процесс посылает в каждый процесс соответствующий столбец матрицы B из n элементов (0-ой столбец матрицы B – в 0-й процесс, 1-ый – в 1-й и т.д.). Каждый процесс должен разместить полученные элементы в k-ый столбец своей матрицы A.

9. ***MPI_Allgather***. В каждом из p процессов создать вектор A из p элементов, заполненный значениями $a_i = rank$ и число $B = rank$, где $rank$ – номер процесса. Каждый процесс посылает в каждый процесс свое значение B. Каждый процесс должен разместить полученные от каждого процесса элементы в свой вектор A (сначала значение B 0-го процесса, затем 1-го и т.д.).

10. ***MPI_Allgather***. В каждом из p процессов создать вектор A из $n \cdot p$ элементов, заполненный значениями $a_i = rank$ и вектор B из n элементов, заполненный значениями $b_i = rank$, где $rank$ – номер процесса. Каждый процесс посылает в каждый процесс свой вектор B из n элементов. Каждый процесс должен разместить полученные от каждого процесса элементы в свой вектор A (сначала элементы вектора B 0-го процесса, затем 1-го и т.д.).

11. ***MPI_Alltoall***. В каждом из p процессов создать вектор B размерности p, заполненный значениями $b_i = rank$, где $rank$ – номер процесса. Получить в каждом процессе вектор A размерности p, заполненный по следующему правилу. Каждый процесс посылает элементы своего вектора B в соответствующее место вектора A соответствующего процесса, т.е. j-ый элемент вектора B i-го процесса – в i-ый элемент вектора A j-го процесса.

12. ***MPI_Alltoall***. В каждом из p процессов создать вектор B размерности $p \cdot n$, заполненный значениями $b_i = rank$, где $rank$ – номер процесса. Получить в каждом процессе вектор A размерности $p \cdot n$, заполненный по следующему правилу. Каждый процесс посылает каждый из p блоков по n элементов вектора B в соответствующее место вектора A соответствующего процесса, т.е. элементы j-го блока вектора B i-го процесса – в i-ый блок вектора A j-го процесса.

13. ***MPI_Gatherv***. В каждом из p процессов создать матрицу A $n \times m$, заполненную значениями $a_{ij} = rank + i \cdot m + j$, где $rank$ – номер процесса. Каждый процесс посылает k-ую строку своей матрицы в корневой процесс. В корневом процессе с номером q получить вектор B из $m \cdot p + p \cdot (p - 1) / 2$ элементов, составленный из строк с номером k матриц A каждого процесса. Элементы в векторе разместить со смещениями между ними, равными номерам процессов в коммуникаторе (сначала элементы от 0-го процесса, затем со смещением 1 элементы от 1-го процесса, со смещением 2 элементы от 2-го процесса и т. д.).

14. ***MPI_Gatherv***. В каждом из p процессов создать матрицу A $n \times m$, заполненную значениями $a_{ij} = rank + i \cdot m + j$, где $rank$ – номер процесса. Каждый процесс посылает k-ую строку своей матрицы в корневой процесс. В корневом процессе с номером q получить матрицу B размерности $p \times m$, составленную из $m - rank$ элементов строк с номером k матриц A каждого процесса (i-й строка матрицы B – это $m - rank$ элементов k-ой строки матрицы A i-го процесса), остальные элементы строки положить равными 0.

15. ***MPI_Gatherv***. В каждом из p процессов создать матрицу A $n \times m$, заполненную значениями $a_{ij} = rank + i \cdot m + j$, где $rank$ – номер процесса. Каждый процесс посылает k-ый столбец своей матрицы в корневой процесс. В корневом процессе с номером q получить вектор B из $n \cdot p + p \cdot (p - 1) / 2$ элементов, составленный из столбцов с номером k матриц A каждого процесса. Элементы в векторе разместить со смещениями между ними, равными номерам процессов в коммуникаторе (сначала элементы от 0-го процесса, затем со смещением 1 элементы от 1-го процесса, со смещением 2 элементы от 2-го процесса и т. д.).

16. ***MPI_Gatherv***. В каждом из p процессов создать матрицу A $n \times m$, заполненную значениями $a_{ij} = rank + i \cdot m + j$, где $rank$ – номер процесса. Каждый процесс посылает k-ый столбец своей матрицы в корневой процесс. В корневом процессе с номером q получить матрицу B размерности $n \times p$, составленную из $n - rank$ элементов столбцов с номером k матриц A каждого процесса (i-й столбец матрицы B – это $n - rank$ элементов k-ого столбца матрицы A i-го процесса), остальные элементы столбца положить равными 0.

17. ***MPI_Scatterv***. В каждом из p процессов создать матрицу A $n \times m$, заполненную значениями $a_{ij} = rank$, где $rank$ – номер процесса. В корневом процессе с номером q задать вектор B из $m \cdot p + p \cdot (p - 1) / 2$ элементов, заполненный значениями $b_i = i$. Корневой процесс посылает

в каждый процесс по m элементов. Множества посылаемых значений расположены в векторе B со смещениями равными номерам процессов в коммуникаторе (сначала n элементов, затем через 1 следующие n элементов, затем через 2 и т.д.). Каждый процесс должен разместить полученные элементы в k-ую строку своей матрицы A.

18. ***MPI_Scatterv***. В каждом из p процессов создать матрицу A $n \times m$, заполненную значениями $a_{ij} = rank$, где $rank$ – номер процесса. В корневом процессе с номером q задать матрицу B размерности $p \times m$, заполненную значениями $b_{ij} = i$. Корневой процесс посылает в каждый процесс соответствующую строку из $m - rank$ элементов (0-й процесс – m элементов 0-ой строки, 1-й процесс – $m - 1$ элемент 1 строки и т.д.). Каждый процесс должен разместить полученные элементы в k-ую строку своей матрицы A, а остальные элементы этой строки обнулить.

19. ***MPI_Scatterv***. В каждом из p процессов создать матрицу A $n \times m$, заполненную значениями $a_{ij} = rank$, где $rank$ – номер процесса. В корневом процессе с номером q задать вектор B из $n \cdot p + p \cdot (p - 1) / 2$ элементов, заполненный значениями $b_i = i$. Корневой процесс посылает в каждый процесс по n элементов. Множества посылаемых значений расположены в векторе B со смещениями равными номерам процессов в коммуникаторе (сначала n элементов, затем через 1 следующие n элементов, затем через 2 и т.д.). Каждый процесс должен разместить полученные элементы в k-й столбец своей матрицы A.

20. ***MPI_Allgatherv***. В каждом из p процессов создать вектор A $n \cdot p$, заполненный значениями $a_i = i$ и вектор B из $n - rank$ элементов, заполненный значениями $b_i = rank$, где $rank$ – номер процесса. Каждый процесс посылает в каждый процесс свой вектор B из $n - rank$ элементов. Каждый процесс должен разместить полученные элементы в вектор A со смещениями равными номерам процессов (сначала n элементов от 0-го процесса, затем $n - 1$ элемент от 1-го процесса со смещением 1 и т.д.).

21. ***MPI_Allgatherv***. В каждом из p процессов создать матрицу A $p \times n$, заполненную значениями $a_{ij} = i$ и вектор B из $n - rank$ элементов, заполненный значениями $b_i = rank$, где $rank$ – номер процесса. Каждый процесс посылает в каждый процесс свой вектор B из $n - rank$ элементов. Каждый процесс должен разместить полученные элементы в соответствующие строки своей матрицы A (в 0-й строку – от 0-го процесса, в 1-ую – от 1-го и т.д.).

22. ***MPI_Allgatherv***. В каждом из p процессов создать матрицу A $p \times n$, заполненную значениями $a_{ij} = i$ и вектор B из $n - rank$ элементов, заполненный значениями $b_i = rank$, где $rank$ – номер процесса. Каждый процесс посылает в каждый процесс свой вектор B из $n - rank$ элементов. Каждый процесс должен разместить полученные элементы в соответствующие строки своей матрицы A (в 0-ую строку – от 0-го процесса, в 1-ую – от 1-го и т.д.), остальные элементы строк обнулить.

23. ***MPI_Alltoallv***. В каждом из p процессов создать вектор B $p \cdot n$, заполненный значениями $b_i = rank + i$, где $rank$ – номер процесса. Получить в каждом процессе вектор A размерности $p \cdot n$, заполненный по следующему правилу. Каждый процесс посылает каждый из p блоков по $n - rank$ элементов вектора B в соответствующее место вектора A соответствующего процесса, т.е. $n - i$ элементов j-го блока вектора B i-го процесса – в i-ый блок вектора A j-го процесса, остальные элементы каждого блока положить равными нулю.

24. ***MPI_Alltoallv***. В каждом из p процессов создать матрицу B $p \times n$, заполненную значениями $b_{ij} = rank + i$, где $rank$ – номер процесса. Получить в каждом процессе матрицу A размерности $p \times n$, заполненную по следующему правилу. Каждый процесс посылает $n - rank$ элементов из каждой строки матрицы B в соответствующую строку матрицы A соответствующего процесса, т.е. $n - i$ элементов j-ой строки матрицы B i-го процесса – в i-ую строку матрицы A j-го процесса, а остальные элементы строк заполнить нулями.

4.3. РЕШЕНИЕ МАТРИЧНЫХ ЗАДАЧ

Цель работы: Разработка параллельных алгоритмов для решения матричных задач. Программная реализация алгоритмов на вычислительном кластере. Исследование зависимости ускорения от числа процессоров, размеров задачи.

Порядок выполнения задания

1. Реализовать параллельный алгоритм (см. задания для самостоятельной работы) на вычислительном кластере из 2, 4, 6 компьютеров.
2. Вычислить время выполнения программы для массивов разной размерности ($100 \le l, m, n \le 10000$).
3. Определить ускорение и эффективность выполнения задачи на кластере для массивов разных размерностей.

4. Найти минимальную размерность массивов, для которых реализованный алгоритм позволяет получить ускорение > 1.

5. Построить графики зависимости ускорения и эффективности от размерностей массивов и количества процессоров в кластере.

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Создать векторы A и B размерности n в корневом процессе с номером q. Вычислить скалярное произведение векторов $k=\sum_{i=1}^{n} x_i y_i$, используя следующий алгоритм. Каждый из p процессов получает $\lfloor n / p \rfloor$ элементов каждого вектора, вычисляет свою часть скалярного произведения и передает его корневому процессу.

2. Создать векторы A, B и C размерности n в корневом процессе с номером q. Определить, являются ли вектора ортогональными. Каждый из p процессов получает $\lfloor n / p \rfloor$ элементов каждого вектора, вычисляет свою часть скалярного произведения и передает его корневому процессу.

3. Создать матрицу A размерности $l \times m$ в корневом процессе с номером q. Вычислить величину $C = \sum_{i=0}^{l} \prod_{j=0}^{m} A_{ij}$ используя следующий алгоритм. Каждый из p процессов получает $\lfloor l / p \rfloor$ строк матрицы A, вычисляет соответствующую сумму, передает ее в корневой процесс.

4. Создать матрицу A размерности $l \times m$ в корневом процессе с номером q. Вычислить величину $C = \prod_{i=0}^{l} \sum_{j=0}^{m} A_{ij}$ используя следующий алгоритм. Каждый из p процессов получает $\lfloor l / p \rfloor$ строк матрицы A, вычисляет соответствующее произведение, передает его в корневой процесс.

5. Создать матрицу A размерности $l \times m$ в корневом процессе с номером q. Вычислить величину $C = \sum_{j=0}^{m} \prod_{i=0}^{l} A_{ij}$ используя следующий алгоритм. Каждый из p процессов получает $\lfloor l / p \rfloor$ столбцов матрицы A, вычисляет соответствующую сумму, передает ее в корневой процесс.

6. Создать матрицу A размерности $l \times m$ в корневом процессе с номером q. Вычислить величину $C = \prod_{j=0}^{m} \sum_{i=0}^{l} A_{ij}$ используя следующий алгоритм. Каждый из p процессов получает $|l / p|$ столбцов матрицы A, вычисляет соответствующее произведение, передает его в корневой процесс.

7. Создать матрицу A размерности $l \times m$ в корневом процессе с номером q. Для заданной матрицы A найти значение $\min_j \sum_i |a_{ij}|$, используя следующий алгоритм. Каждый из p процессов получает $|l / p|$ строк матрицы A, вычисляет соответствующий min, передает его в корневой процесс.

8. Создать матрицу A размерности $l \times m$ в корневом процессе с номером q. Найти норму заданной матрицы A, определенную как $\max_i \sum_j |a_{ij}|$, используя следующий алгоритм. Каждый из p процессов получает $|l / p|$ строк матрицы A, вычисляет соответствующий max, передает его в корневой процесс.

9. Создать матрицу A размерности $l \times m$, вектор B размерности m в корневом процессе с номером q. Вычислить вектор $C = A * B$ размерности l, используя следующий алгоритм. Каждый из p процессов получает $|l / p|$ строк матрицы A, вектор B, вычисляет соответствующие элементы результирующего вектора C, передает их в корневой процесс.

10. Создать матрицу A размерности $l \times m$, вектор B размерности l в корневом процессе с номером q. Вычислить вектор $C = B * A$ размерности m, используя следующий алгоритм. Каждый из p процессов получает $|m / p|$ столбцов матрицы A, вектор B, вычисляет соответствующие элементы результирующего вектора C, передает их в корневой процесс.

11. Создать матрицы A размерности $l \times m$, $B - m \times n$ в корневом процессе с номером q. Вычислить матрицу $C = A * B$ размерности $l \times n$, используя следующий алгоритм. Каждый из p процессов получает $|l / p|$ строк матрицы A, всю матрицу B, вычисляет соответствующие строки элементов результирующей матрицы C, передает их в корневой процесс.

12. Создать матрицы A размерности $l \times m$, B – $m \times n$ в корневом процессе с номером q. Вычислить матрицу $C = A * B$ размерности $l \times n$, используя следующий алгоритм. Каждый из p процессов получает $\lfloor n / p \rfloor$ столбцов матрицы B, всю матрицу A, вычисляет соответствующие столбцы элементов результирующей матрицы C, передает их в корневой процесс.

13. Создать матрицы A размерности $l \times m$, B – $m \times n$ в корневом процессе с номером q. Вычислить матрицу $C = A * B$ размерности $l \times n$, используя следующий планирующий алгоритм. Каждый из p процессов получает одну строку матрицы A, столбец матрицы B, вычисляет соответствующий элемент результирующей матрицы C, передает его в корневой процесс, получает следующее задание и т.д. до вычисления всех элементов результирующей матрицы.

14. Создать матрицы A размерности $l \times m$, B – $m \times n$ в корневом процессе с номером q. Вычислить матрицу $C = A * B$ размерности $l \times n$, которая должна быть распределена по процессам (по $\lfloor l / p \rfloor$ строк в каждом процессе), используя следующий планирующий алгоритм. Каждый из p процессов получает всю матрицу B, затем одну строку матрицы A, вычисляет соответствующую строку элементов результирующей матрицы C, передает ее в соответствующий процесс; получает следующую строку матрицы A и т.д. до вычисления всех элементов результирующей матрицы.

15. Создать матрицу A размерности $n \times n$ и векторы x, y размерности n и в корневом процессе с номером q. Получить вектор $c = A \cdot (x + y)$.

16. Создать матрицу A размерности $n \times n$ и векторы x, y размерности n и в корневом процессе с номером q. Вычислить скалярное произведение векторов $k = \sum_{i=1}^{n} x_i y_i$. Получить матрицу $B = A \cdot k$.

17. Создать матрицу A размерности $n \times n$ в корневом процессе с номером q. Получить матрицу $B = A \cdot A^*$.

18. Создать матрицы A,B,C размерности $n \times n$ в корневом процессе с номером q. Получить матрицу $D = A \cdot (B + C)$

19. Создать матрицу A размерности $n \times n$ в корневом процессе с номером q. Определить, является ли заданная матрица ортонормированной, т. е. такой, в которой скалярное произведение каждой пары

различных строк равно 0, а скалярное произведение каждой строки на себя равно 1.

20. Создать матрицу A размерности $n \times n$ и вектор b размерности n в корневом процессе с номером q. Получить вектор $c = A^2 \cdot b$.

21. Создать матрицу A размерности $n \times n$ в корневом процессе с номером q. Возвести матрицу в степень m, где m – натуральное заданное число.

22. Создать матрицы A,B размерности $n \times n$ в корневом процессе с номером q. Получить матрицу $C = A \cdot B + B \cdot A$.

23. Создать матрицу A размерности $n \times n$ в корневом процессе с номером q. Получить матрицу $B = E + A + A^2 + \ldots + A^m$, где m – заданное натуральное число.

24. Создать матрицу A размерности $n \times n$ в корневом процессе с номером q. Получить матрицу $B = A + A^2 + A^4 + A^8$.

ЛИТЕРАТУРА

1. Информационные материалы по MPI (http://www.mpi-forum.org)
2. *Шпаковский Г.И.* Программирование для многопроцессорных систем в стандарте MPI: Пособие / Г.И. Шпаковский , Н.В. Серикова. - Мн.: БГУ, 2002. 323 с.
3. *Шпаковский Г.И.* Организация работы на вычислительном кластере: учеб. пособие/ Г.И. Шпаковский, А.Е. Верхотуров, Н.В. Серикова. - Мн.: БГУ, 2004. 182 с.

Учебное издание

Серикова Наталья Владимировна

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ
Технология MPI

Методические указания к лабораторному практикуму
по курсу «Параллельные вычисления и программирование»
для студентов
факультета радиофизики и компьютерных технологий

В авторской редакции

Ответственный за выпуск *Н. В. Серикова*

Подписано в печать 01.02.16. Формат 60×84/16. Бумага офсетная.
Усл. печ. л. 3,25. Уч.-изд. л. 2,29. Тираж 50 экз. Заказ

Белорусский государственный университет.
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/270 от 03.04.2014
Пр. Независимости, 4, 220030, Минск.

Отпечатано с оригинал-макета заказчика
на копировально-множительной технике
факультета радиофизики и компьютерных технологий
Белорусского государственного университета.
Ул. Курчатова, д. 5, 220064, Минск.