МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С.П. КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)» (СГАУ)

ПАРАЛЛЕЛЬНАЯ РЕАЛИЗАЦИЯ ВЫЧИСЛИТЕЛЬНЫХ АЛГОРИТМОВ НА КЛАСТЕРЕ СГАУ «СЕРГЕЙ КОРОЛЁВ»

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский государственный аэрокосмический университет имени академика С.П. Королева (национальный исследовательский университет)» в качестве методических указаний

САМАРА
Издательство СГАУ
2015

УДК 004(075) + 004.382.2(075)
ББК: 32.973

Составители: ***Т.М. Кузьмишина, Е.В. Гошин***
Рецензенты: д-р техн. наук, проф. С.А. В о с т о к и н

Параллельная реализация вычислительных алгоритмов на кластере СГАУ «Сергей Королёв»: метод. указания / сост.: *Т.М. Кузьмишина, Е.В. Гошин.* – Самара: СГАУ, 2015. – 32 с.

Целью работы является практическое знакомство с технологией работы на суперкомпьютере. Работа выполняется на кластере «Сергей Королёв» СГАУ. В основу работы положено MPI-приложение для решения задач линейной алгебры. В качестве языка программирования используется язык С++.

Предназначены для аудиторной работы студентов младших курсов СГАУ, изучающих технологию параллельного программирования, а также самостоятельной работы аспирантов и преподавателей.

Подготовлены на кафедре суперкомпьютеров и общей информатики.

УДК 004(075) + 004.382.2(075)
ББК: 32.973

© СГАУ, 2015

ПАРАЛЛЕЛЬНАЯ РЕАЛИЗАЦИЯ ВЫЧИСЛИТЕЛЬНЫХ АЛГОРИТМОВ НА КЛАСТЕРЕ СГАУ «СЕРГЕЙ КОРОЛЁВ»

Лабораторная работа выполнена на кафедре Суперкомпьютеры и общая информатика Самарского аэрокосмического университета и предназначена для студентов первого курса в рамках бакалавриата с целью знакомства с многопроцессорными системами и решением простых задач с использованием языка программирования C++ и коммуникативной многопроцессорной среды MPI.

Часть 1. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Архитектура параллельных вычислительных систем

Знание особенностей вычислительной системы очень важно на самом первом этапе разработки и реализации параллельных алгоритмов. Это позволит правильно выбрать среду программирования и реализовать эффективное решение задачи. В литературе [1-3] и других источниках с той или иной степенью детализации приводится описание архитектур многопроцессорных вычислительных систем. Остановимся на кратком обзоре наиболее популярных архитектур по способу организации подсистем оперативной памяти.

При разработке параллельного алгоритма наиболее важным является деление вычислительных систем на системы с общей (разделяемой) и распределённой памятью.

Системы с разделяемой памятью (мультипроцессоры) имеют общую виртуальную память и все процессоры имеют одинаковый доступ к данным и командам, хранящимся в этой памяти (*uniform memory access* или UMA). Достаточно один раз создать соответствующую структуру данных и разместить её в оперативной памяти.

Системы с распределённой памятью (мультикомпьютеры) имеют отдельную локальную оперативную память для каждого процессора, при этом у других процессоров доступ к этой памяти отсутствует. При работе на компьютере с распределённой памятью необходимо создавать копии исходных данных на каждом процессоре.

Реальный вариант этой задачи слишком сложен для начального восприятия многопроцессорных вычислительных систем. Рассмотрим упрощённую модель.

Для простоты введём понятие «вычислительный узел». В общем случае под узлом будем понимать некую одноядерную модель процессора (CPU). Рассмотрим задачу реализации параллельных вычислительных процессов на нескольких узлах.

Современные многоядерные процессоры насчитывают до 4-х ядер (*kernels*), способных обрабатывать до 8-и потоков (*threads*) одновременно.

Будем рассматривать ситуацию, когда одна программа выполняется на нескольких узлах с разделённой (распределенной) памятью.

Назовём выполнение программы на отдельном узле процессом.

В данной лабораторной работе рассматривается реализация параллельной задачи на многопроцессорном компьютере с распределённой памятью.

Вычислительные узлы этого класса (массивно-параллельных) компьютеров объединяются друг с другом посредством коммуникационной среды. Каждый узел имеет один или несколько процессоров и свою собственную локальную память. Распределённость памяти означает, что каждый процессор имеет непосредственный доступ только к локальной памяти своего узла. Доступ к памяти других узлов осуществляется посредством коммуникационной среды.

Преимущества этой архитектуры – низкое значение отношения цена/производительность и возможность практически неограниченно наращивать число процессоров. Различия компьютеров данного класса сводятся к различиям в организации коммуникационной среды. На рис. 1.1. показана общая схема связей основных элементов системы в архитектуре многопроцессорных систем с распределённой памятью.

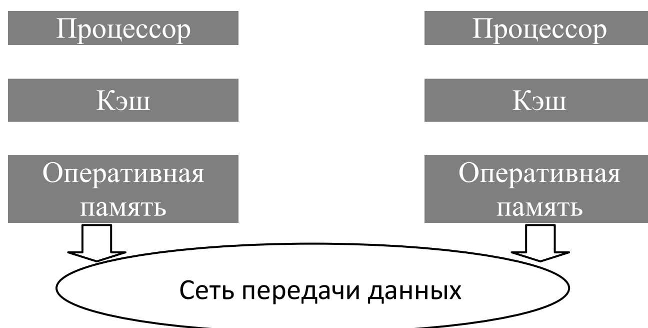

Рис. 1.1. Общая схема связей основных элементов системы с распределённой памятью

Различия в архитектуре предопределяют и различия в парадигмах программирования. Наиболее распространённым способом создания параллельных приложений для систем с распределённой памятью является организация процессов, взаимодействующих при помощи передачи сообщений. На этой парадигме основаны библиотеки MPI (*Message Passing Interface*), PVM (*Parallel Virtual Machine*) и многие другие средства программирования.

Параллельные процессы, взаимодействующие с помощью передачи сообщений

Наиболее распространённой технологией, применяемой для программирования многопроцессорных систем с распределённой памятью, является использование параллельных процессов, взаимодействующих с помощью передачи сообщений. Такая модель представляется также и наиболее естественной для подобных систем, так как передача сообщений по сети – практически единственный возможный способ взаимодействия процессов, выполняющихся на различных процессорах, не обладающих общей памятью.

Основным средством разработки программ в рассматриваемой парадигме является MPI (*Message-Passing Interface*). В настоящее время MPI входит в стандартный комплект программного обеспечения практически любого многопроцессорного вычислительного комплекса.

В состав среды программирования MPI входит библиотека с интерфейсом для одного или нескольких языков программирования (обычно Фортран, Си, Си++), а также средства для запуска и сборки параллельного приложения (программы пользователя)

Стандарты на языки программирования высокого уровня позволяют достаточно быстро освоить эффективное использование компьютера и поддерживают переносимость создаваемых программ. В настоящее время разработаны стандарты на параллельные компьютерные системы и на параллельное программирование. Одним из наиболее распространённых стандартов параллельного программирования является стандарт MPI.

Интерфейс – это совокупность средств и правил, обеспечивающих логическое или физическое взаимодействие устройств и/или программ вычислительной системы.

Стандарт MPI

Аббревиатура MPI расшифровывается как *Message Passing Interface*, что можно перевести как «интерфейс передачи сообщений». Этот стандарт был развит группой *Message Passing Interface Forum* (MPIF). Цель его разработки состояла в создании удобных средств параллельного программирования со свойствами эффективности и гибкости при практическом применении.

К числу требований, которые были предъявлены к результирующей версии стандарта MPI, относятся:

1) реализация для языков C и Fortran 77 при использовании различных платформ;

2) удобство использования, позволяющее программисту минимальным образом представлять себе архитектуру параллельной системы;

3) высокая степень переносимости и масштабируемости разрабатываемой пользователем программы.

Принято различать два подхода параллельного программирования:

- идеология SIMD (*Single Instruction Multiple Data*): одна инструкция – много данных;
- идеология MIMD (*Multiple Instructions Multiple Data*): много инструкций – много данных.

Идеология SIMD означает, что общая инструкция параллельно обрабатывается всеми модулями параллельной системы.

Заметим, что идеология SIMD не подразумевает тождественности работ всех модулей, что было бы довольно бессмысленно – программа может иметь разветвления в зависимости от номера модуля, так что, конечно, каждый модуль производит обработку по предписанному именно ему алгоритму.

Идеология MIMD предполагает обработку различных инструкций разными модулями.

Стандарт MPI скорее относится к первой идеологии, чем ко второй.

Основными понятиями MPI являются:

- процесс;
- группа процессов;
- коммуникатор.

Коммуникатор

Коммуникатор идентифицирует группу процессов, которые с точки зрения данного коммуникатора рассматриваются как параллельно исполняемые последовательные программы; последние, однако, могут на самом деле представлять группы процессов, вложенных в исходную группу.

MPI допускает многократное ветвление программы и создание на базе одного процесса очередной группы процессов, идентифицируемых новым коммуникатором. Таким образом, процесс ветвления может представлять собой дерево; для завершения программы не требуется возвращение к исходному процессу.

Коммуникатор реализует синхронизацию и обмены между идентифицируемыми им процессами; для прикладной программы он выступает как коммуникационная среда. Каждый коммуникатор имеет собственное коммуникационное пространство, а сообщения, задействующие разные коммуникаторы, не оказывают влияния друг на друга и не взаимодействуют. Таким образом, каждая группа процессов использует свой собственный коммуникатор, процессы внутри группы нумеруются от 0 до k-1, где k – число процессов в группе (параметр k может принимать различные натуральные значения, задаваемые пользователем, но не превосходящие значения, определяемого реализацией).

MPI управляет системной памятью для буферизации сообщений и хранения внутренних представлений объектов (групп, коммуникаторов, типов данных и т.п.).

В структуре языков Фортран и Си стандарт MPI реализуется как библиотека процедур с вызовами определённого вида. Требуется, чтобы программа, написанная с использованием стандарта MPI, могла быть выполнена на любой параллельной системе без специальной настройки.

Для применения MPI на параллельной системе к программе должна быть подключен заголовочный файл mpi.h (include “mpi.h”), содержащий определения функций, типов и констант MPI.

Перед применением процедур стандарта MPI следует вызвать процедуру MPI_INIT; она подключает коммуникатор MPI_COMM_WORLD, задающий стандартную коммуникационную среду с номерами процессов 0, 1, … , n-1, где n – число процессов, определяемых при инициализации системы.

Функция MPI_INIT принимает на вход адреса переменных, соответствующих аргументам командной строки.

int MPI_INIT(int*pargc, char***pargv)
pargc – указатель на счётчик аргументов командной строки;
pargv – указатель на массив аргументов командной строки.

При завершении использования MPI в программе должна быть вызвана процедура MPI_FINALIZE; вызов этой процедуры обязателен для правильной работы программы. Функция MPI_Finalize не имеет параметров.

Основной концепцией выполнения MPI-программы является модель SPMD (*Single Program Multiple Data* – «одна программа, разные данные»). Суть парадигмы состоит в том, что все задействованные процессоры вычислительной системы выполняют одну и ту же MPI-программу, которая может обрабатывать различные данные и выполняется по различным путям на разных процессорах. Модель SPMD является удобной с точки зрения разработки параллельных программ, и хотя следование этой модели не является обязательным требованием MPI, большинство MPI-приложений на сегодняшний день ей соответствуют.

Функции для работы с коммуникатором

MPI_COMM_size – определяет число процессов, ассоциированных с коммуникатором.

Синтаксис:
#include “mpi.h”
int MPI_Comm_size (MPI_Comm comm, int size)*
Входной параметр:
comm – коммуникатор.
Выходной параметр:
size – число процессов в коммуникаторе comm (указатель на область памяти, в которую будет сохранено общее число процессов в коммуникаторе comm.);

MPI_Comm_rank – определяет ранг (номер) вызывающего процесса в коммуникаторе.
Синтаксис:
#include “mpi.h”
int MPI_Comm_size (MPI_Comm comm, int size)*

comm – коммуникатор;

size – ранг (номер) вызывающего процесса в коммуникаторе comm (указатель на область памяти, в которую будет сохранен номер процесса, вызывающего функцию в коммуникаторе comm).

Первый параметр в вызове этих функций называется *коммуникатором* и служит для обозначения совокупности процессов, по отношению к которой вычисляется число процессов и номер процесса, выполнившего вызов функции

Следует представлять себе ситуацию таким образом, что рассматриваемая программа копируется во все модули параллельной системы.

В соответствии с концепцией, принятой в MPI, каждый процессор обрабатывает **одну и ту же программу,** но поскольку его деятельность может быть поставлена в зависимость от его номера, возможно делать поведение различных процессов различным и наблюдать за их работой. Процессы в MPI, вообще говоря, равноправны (нет заранее выделенного процесса): программист вправе рассматривать любой из них как основной.

Пересылка данных между двумя процессами

Для организации параллельных вычислений необходим обмен данными между процессами. Обмены всегда производятся в пределах одного коммуникатора, который указывается в качестве параметра в вызовах функций обмена. Номера процессов, принимающих участие в обменах, вычисляются по отношению к указанному коммуникатору.

Для взаимодействия в MPI двух процессов используются *точечные обмены* (*point-to-point communications*). В этом случае у взаимодействия есть две точки: процесс-отправитель и процесс-получатель сообщения.

Точечные, так же как и обмены других типов, всегда производятся в пределах одного коммуникатора, который указывается в качестве параметра в вызовах функций обмена. Номера процессов, принимающих участие в обменах, вычисляются по отношению к указанному коммуникатору.

Рассмотрим функции точечного обмена, реализующие процесс обмена с использованием буфера.

Отправка сообщения осуществляется функцией MPI_Send, которая реализует передачу данных из буфера процесса-отправителя процессу-получателю.

Приём сообщения осуществляется функцией MPI_Recv, которая реализует передачу данных от процесса-отправителя в буфер процесса-получателя.

Базовые функции точечных обменов

MPI_Send – отправка сообщения.

Синтаксис:
#include “mpi.h”
*int MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

Входные параметры:

buf – указатель на начало буфера передаваемых данных;

count – число элементов передаваемых данных;

datatype – MPI-тип передаваемых данных;

dest – ранг (номер) процесса, которому передаются данные;

tag – тэг сообщения (целое число, передаваемое вместе с сообщением и проверяемое при его приёме);

comm – коммуникатор, в пределах которого передаются данные.

Примечание. Эта функция может (но не обязана) блокировать процесс-отправитель до момента вызова парной операции приёма.

MPI_Recv – приём сообщения.
Синтаксис:

#include “mpi.h”

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

Выходные параметры:

buf – указатель на начало буфера, в который будут сохранены полученные данные;

status – указатель на структуру MPI_Status, в которую будет записан статус завершения операции. Содержит информацию о том, каким образом завершилась операция.

Входные параметры:

count – максимальное число элементов данных, которое предполагается принять (обычно это число выбирается таким образом, чтобы сохранение данных не привело к переполнению буфера);

datatype – MPI-тип данных каждого пересылаемого элемента данных в сообщении;

source – ранг (номер) процесса, от которого ожидается приём данных (сообщения) Сообщения, пришедшие с другим номером, не принимаются;

tag – ожидаемый тэг сообщения. Сообщения, имеющие тэг, отличный от указанного, не принимаются;

comm – коммуникатор, в пределах которого передаются данные.

Примечание. Аргумент count определяет максимальную длину сообщения; текущее количество полученных элементов можно определить при помощи функции MPI_Get_count.

MPI_Get_count – возвращение числа полученных элементов указанного типа.

Синтаксис:

#include “mpi.h”

*int MPI_Get_count (MPI_Status *status, MPI_Datatype datatype, int *count)*

Входные параметры:

status – статус операции приёма сообщения;

datatype – MPI-тип данных принимаемых элементов.

Выходной параметр:

count – число полученных элементов (указатель на начало буфера, в который будут сохранены полученные данные.

MPI-типы данных

MPI-тип является объектом данных типа MPI_DATATYPE и должен соответствовать типу данных, хранимых в буфере – области памяти buf. В таблице приводится предопределённые MPI-типы и соответствующие им типы языка Си.

Тип MPI	Тип Си
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	Long double

Часть 2. ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ НА КЛАСТЕРЕ «СЕРГЕЙ КОРОЛЁВ» СГАУ

Постановка задачи

Рассмотрим MPI-приложение для программы умножения матрицы на вектор, написанной на языке программирования C++.

Решение задачи

Известно, что при умножении матрицы $A = (a_{ij})$, порядка m x n ($i = 0, \dots , m - 1; j = 0,\dots, n - 1$ на вектор $X = (x_0, \dots x_{n-1})^T$ элементы вектора $Y = (y_0, \dots y_{m-1})^T$ – вычисляются по формуле

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j \,. \qquad (1)$$

Рассмотрим параллельный алгоритм решения этой задачи на многопроцессорном компьютере (многопроцессорная вычислительная система).

Введём некоторые допущения:

пусть $n = 4$, $m = 6$, тогда $X = (x_0, x_1, x_2, x_3)^T$;

$$A = \begin{pmatrix} a_{01} & a_{02} & a_{03} & a_{04} \\ a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \\ a_{51} & a_{52} & a_{53} & a_{54} \end{pmatrix}.$$

MPI-интерфейс предполагает распределённое использование памяти. Это означает, что каждый процесс будет выполнять одну и ту же программу в объёме только своих инструкций в собственном адресном пространстве. При этом все процессы работают параллельно в один и то же период времени. Набор инструкций для каждого процесса определяется в программе согласно алгоритму распараллеливания задачи.

Другими словами, для нашей задачи будет выделено определённое количество процессов, каждый из которых примет исходные данные (вектор X и матрицу A) в своё адресное пространство и выполнит операцию умножения вектора X на вполне определённое количество строк матрицы A. Каждый процесс вычисляет собственные номера строк для обработки (номер начальной строки и номер конечной стро-

ки) и организует собственный цикл. Все циклы выполняются одновременно (параллельно). Результат вычисления записывается отдельно в адресное пространство каждого процесса.

В результате выполнения этих операций в каждом процессе будет получена своя доля искомого вектора.

На следующем этапе решения задачи реализуется процедура «сборки». В интерфейсе MPI все процессы соединены единым коммуникатором. MPI-команды передачи сообщений последовательно (по определённому алгоритму программы) передают частичные результаты процессов в один главный процесс. В результате в адресном пространстве главного процесса будет собран искомый вектор Y. После чего главный процесс выведет результат работы программы в файл.

Пусть для решения задачи операционная система многопроцессорного компьютера выделяет 3 процесса с номерами 0, 1, 2. Главным из них будет процесс с номером 0.

Введём обозначения:

start – первая строка для обработки;

final – последняя строка для обработки;

my_rank – номер процесса;

rows – общее число строк в матрице;

rank_size – общее число процессов.

Пусть для каждого процесса

первая строка для обработки вычисляется по формуле:

start=(my_rank*rows)/ran_size; (2)

последняя строка для обработки вычисляется по формуле:

final=((my_rank+1)*rows)/rank_size. (3)

Вычислим значения *start* и *final* для каждого процесса нашего примера.

	0 процесс	1 процесс	2 процесс
start	(0*6)/3=**0**	(1*6)/3=**2**	(2*6)/3=**4**
final	((0+1)*6)/3=**2**	((1+1)*6)/3=**4**	((1+2)*6)/3=**6**

После этого в каждом процессе организуется цикл для вычисления соответствующих элементов y_i вектора Y согласно формуле (1). Параметр цикла изменяется от значения *start* до значения *final*-1 соответственно для каждого процесса. Таким образом, в адресном про-

странстве каждого процесса будут сохранены отдельные элементы вектора *Y*.

	y_i
0 процесс	y_0-y_1
1 процесс	y_2-y_3
2 процесс	y_4-y_5

Элементы вектора согласно алгоритму будут передаваться от одного процесса в другой, пока не займут каждый своё место, соединив таким образом вектор целиком. Пересылка и приём реализуются через общий коммуникатор командами MPI.

Текст программы

```
#include "mip.h"
#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char*argv[])
{
	int my_rank, rank_size; // Номер процесса, число процессов

	MPI_Init(&argc, &argv);		// Важно! Инициализация MPI
	MPI_Status status;	//Статус передачи (вспомогательная пе-
ременная)

	MPI_Comm_rank(MPI_COMM_WORLD,		&my_rank);
	//Получение номера процесса
	MPI_Comm_size(MPI_COMM_WORLD,		&rank_size);
	//Получение числа процессов

	ifstream f_in_a (arg[1]);	//Поток для чтения матрицы
	ifstream f_in_x (arg[2]);	//Поток для чтения вектора
	ofstream f_out_y (argv[3]);	//Поток для записи результата

	int rows, columns;		// Размеры матрицы
	f_in_a >> rows;
	f_in_a >> columns;
	int rows_on_process = rows / rank_size;		//Число	строк,
обрабатываемых процессом

	double a[rows][columns];
	double x[columns];
	double y[rows];

	for (int i=0; i<rows; i++)
	  {for (int j=0; j<columns; j++)
```

```
        {  f_in_a >> a[i][j];                      //Чтение матрицы
         }
      }

    for (int j=0; j<columns; j++)
      {f_in_x >> x[j];                    //чтение вектора
      }

    int start = (my_rank*rows)/rank_size;   //Первая строка для обра-
ботки
    int final = (my_rank+1)*rows)/rank_size;        //Последняя  стро-
ка для обработки

    for (int i=start; i<final; i++)
      {y[i]=0.0;
      for (int j=0; j<columns; j++)
     {y[i] += a[i][j]*x[j];                     //Вычисление результата
     }
      }

    if(my_rank%2 == 1)
   {MPI_Send    (&y[start],     rows_on_process,     MPI_DOUBLE,
my_rank-1, 0, MPI_COMM-WORLD);
   }
    if(my_rank%2 == 0)
       {MPI_Recv(&y[start+rows_on_process],        rows_on_process,
MPI_DOUBLE, my_rank+1, 0, MPI_COMM_WORLD, &status);
   }
  if(my_rank%4 == 2)
   {MPI_Send    (&y[start],    2*rows_on_process,    MPI_DOUBLE,
my_rank-2, 0, MPI_COMM-WORLD);
   }
  if(my_rank%4 == 0)
       {MPI_Recv(&y[start+2*rows_on_process],  2*rows_on_process,
MPI_DOUBLE, my_rank+2, 0, MPI_COMM_WORLD, &status);
   }
    if(my_rank%8 == 4)
   {MPI_Send    (&y[start],    4*rows_on_process,    MPI_DOUBLE,
my_rank-4, 0, MPI_COMM-WORLD);
```

```
        }
    if(my_rank%8 == 0)
            {MPI_Recv(&y[start+4*rows_on_process], 4*rows_on_process,
MPI_DOUBLE, my_rank+2, 0, MPI_COMM_WORLD, &status);
        }
        if(my_rank ==0)
        {for (int i=0; i<rows; i++)
           {f_out_y << y[i] << endl;                    //Вывод в файл
               }
        }
        MPI_Finalize ();            //Финализация MPI. Важно!
        return 0;
```

Комментарии к инструкциям программы

Инструкция	Комментарии
Main. cpp	Название программного модуля на языке С++.
#include “mpi.h”	Препроцессор вставляет сюда содержимое заголовочного файла ***mpi.h***
#include <iostream>	Компилятор С++ *включает* сюда все объявления стандартных потоковых средств ввода-вывода (I/O facilities) из файла ***iostream.*** Заголовочный файл <iostream> содержит описания классов для ввода/вывода и предопределённые объекты: cin (класс istream); cout (класс ostream); cerr (класс ostream); clog (класс ostream). Становятся доступными связанные с ними средства ввода/вывода.
#include ,<fstream>	Вставляет содержимое заголовочного файла fstream для работы с файловыми потоками.
Using namespace std;	Делает доступными все имена стандартной области имён.
Int main(int argc, char *argv[])	Каждая программа на С++ должна иметь функцию с именем ***main.*** Это точка входа в программу, с которой начинается её выполнение. Значение типа ***int*** предназначено для системы, запустившей программу. Аргументы функции main: **int argc** – количество параметров (целое), передаваемых функции, включая имя самой программы (=4); **char *argv[]** – указатель на массив указателй типа char. Каждый элемент массива содержит указатель на отдельный параметр командной строки **mpirun ./test.mpi a.txt x.txt y.txt**

Инструкция	Комментарии
	первый элемент массива (argv[0]) ссылается на полное имя запускаемого на выполнение файла **mpicxx –o test.mpi main.cpp;** *второй элемент* массива (argv[1]) указывает на первый параметр **a.txt;** *третий элемент* массива (argv[2]) указывает на второй параметр **x.txt;** *четвёртый элемент* массива (argv[3]) указывает на третий параметр **y.txt.ЫЫ**
Int my_rank, rank_size	Описание типа **Int** (целый) данных для переменных **my_rank** и **rank_size**. *my_rank* – номер процесса; *rank_size* – число процессов. Под величины этого типа на каждом узле отводится участок оперативной памяти, соответствующий разрядности процессора (например, по 4 байта для 32-разрядного процессора).
MPI_Init (&arc, &argv)	Важно! Инициализация MPI. Функция MPI_Init принимает на вход адреса переменных, соответствующих аргументам командной строки. **Int MPI_Init(int*pargc, char***pargv)** **pargc** – указатель на счётчик аргументов командной строки; **pargv** – указатель на массив аргументов командной строки.
Status MPI_Status	Статус завершения операции приёма. Указатель на структуру MPI_Status, которая заполняется после завершения операций приёма и содержит информацию о том, каким образом завершилась операция.
MPI_Comm_rank (MPI_Comm_World, &my_rank)	Получение номера процесса. Каждый процессор получает свой номер в коммуникаторе и размещает его в сво-

Инструкция	Комментарии
	ей переменной my_rank.
MPI_Comm_size (MPI_Comm_World, &rank_size)	Получение числа процессов. Каждый процессор получает общее число процессов коммуникатора.
Ifstream f_in_a (argv[1])	Открывается файловый поток **f_in_a** на вход (для приёма матрицы из первого аргумента (команда С++)).
Ifstream f_in_x (argv[2])	Открывается файловый поток **f_in_x** на вход (для приёма вектора из второго аргумента (команда С++)).
ofstream f_out_y (argv[3])	Открывается файловый поток **f_out_y** на выход (для записи результата в третий аргумент (команда С++)).
Int rows, columns	Описание типа данных Int для переменных **rows, columns.** На каждом узле выделяется память для размещения этих переменных.
f_in_a >> rows	Из входного файлового потока **f_in_a** читается в переменную **rows** число строк матрицы. Выполняется на каждом узле.
f_in_a >> columns	Из входного файлового потока **f_in_a** читается в переменную **columns** число столбцов матрицы. Выполняется на каждом узле.
int rows_on_process=rows/rank_size	Вычисляется количество строк матрицы **rows_on_process** для каждого (одного) процесса: число строк входной матрицы делится на общее количество процессов.
double a [rows][columns] double x [rows] double y [columns]	Описание переменных типа **double** (вещественные числа с двойной точностью) для элементов: **a [rows][lomuns]** двумерного массива; **x [rows]** одномерного массива; **y [columns]** одномерного массива. Выполняется на каждом узле.
for (int i = 0; I < rows; i++) {for (int j = 0; j < columns; j++) {f_in_a >> a [i] [j];} }	Вложенный цикл для чтения по строкам и столбцам элементов исходной матрицы из входного файлового потока **f_in_a** в двумерный массив **a**. Выполняется на каждом узле.

Инструкция	Комментарии
for (int j = 0; j < columns; j++) {f_in_x >> x[j];}	Цикл для чтения элементов исходного вектора из входного файлового потока **f_in_x** в одномерный массив **x**. Выполняется на каждом узле.
int start =(my_rank*rows)/rank_size;	Вычисление для каждого узла **начальной** строки матрицы для обработки на узле с номером **my_rank.** Выполняется на каждом узле.
int final=((my_rank+1)*rows)/ rank_size;	Вычисление для каждого узла **конечной** строки матрицы для обработки на узле с номером **my_rank**. Выполняется на каждом узле.
For (int I = start; i<final; i++) {y[i]=0.0; For (int j=0; j<columns; j++) {y[i]+=a[i][j]*x[j]; }}	Вычисление результата: элементов вектора y с номерами от start до final. Выполняется на каждом узле с определённым набором строк матрицы: от номера start до номера final.
If (my_rank%2 ==1) {MPI_Send(&y[start], rows_on_process, MPI_DOUBLE, my_rank-1, 0, MPI_COMM_WORLD);}	
If (my_rank%2 ==0) {MPI_Recv(&y[start+rows_on_process], rows_on_process, MPI_DOUBLE, my_rank+1, 0, MPI_COMM_WORLD, &status); }	
If (my_rank%4 ==2) {MPI_Send(&y[start], 2*rows_on_process, MPI_DOUBLE, my_rank-2, 0, MPI_COMM_WORLD); }	
If (my_rank%4 ==0) {MPI_Recv(&y[start+2*rows_on_process], 2*rows_on_process, MPI_DOUBLE, my_rank+2, 0, MPI_COMM_WORLD, &status) ; }	
If (my_rank%8 ==4) {MPI_Send(&y[start], 4*rows_on_process, MPI_DOUBLE, my_rank-2, 0,	

Инструкция	Комментарии
MPI_COMM_WORLD) ; }	
If (my_rank%8 ==0) {MPI_Recv(&y[start+4*rows_on_process], 4*rows_on_process, MPI_DOUBLE, my_rank+2, 0, MPI_COMM_WORLD, &status) ; }	
If (my_rank==0) { for (int=0; i<rows; i++){f_out_y<<y[i]<<endl}};	**Вывод в файл результата**
	Операция «вставить в поток» осуществляет вывод своего правого аргумента в левый аргумент.
f_out_y<<y[i]	Вывод в поток f_out_y *i*-го элемента одномерного массива *y*.
f_out_y<<y[i]<<endl	Цепочка операций помещения в поток f_out_y. Вывод выполняется слева направо. Манипулятор ***endl*** при выводе включает в поток символ новой строки и выгружает в буфер.
for (int=0; i<rows; i++){f_out_y<<y[i]<<endl}	Цикл for с параметром i (от 0 до rows-1 с шагом 1) для перебора всех элементов (=rows) одномерного массива *y*.
If (my_rank==0) { for (int=0; i<rows; i++){f_out_y<<y[i]<<endl}};	Для процессора с рангом (номером 0) выполнить действие: вывести в поток f_out_y (фактически в связанный с ним файл y.txt) элементы y[i] в столбец (после каждого элемента выводится символ перехода на новую строку).
MPI_Finalize()	Функция вызывается для завершения работы с библиотекой. Не имеет параметров.
return 0	Оператор возврата из функции main в вызвавшую её систему. 0, указанный после return неявно преобразуется к типу возвращаемого функцией значения и передаётся в точку вызова функции. В данном случае системе передаётся сообщение об успешном завершении работы функции main.

Состав передаваемых параметров

первый аргумент: число строк, число столбцов, элементы матрицы;
второй аргумент: (вектор);
третий аргумент: (вектор).

Компиляция, сборка и запуск программы в среде MPI

Одна и та же программа **test.mpi** одновременно начинает выполняться на всех процессорах. Каждый процессор имеет свою собственную (видимую) область памяти. Обмен данными между процессорами осуществляется через интерфейс передачи сообщений (MPI).

Работа на кластере «Сергей Королёв»

Общая характеристика кластера

Кластер состоит из одного управляющего и 136 вычислительных серверов. Управляющий сервер имеет локальное имя **mgt1**, вычислительные узлы именуются:

n1-n42, n57-n70, n113-n168, n225-n243,n246-n250.

На каждом узле кластера установлен жёсткий диск для хранения временных данных, смонтированный в директорию **/tmp.** При завершении задачи на узле эта директория очищается.

На узлах **n1-n34** доступно **56** Гб; **n35-n42, n57-n70, n113-n168** доступно **120** Гб; **n239-n242, n246-n250** доступно 244 Гб.

Организация удалённого доступа к кластеру

Удалённый доступ к кластеру возможен только с ip-адресов корпоративной сети СГАУ.

Сетевое имя суперкомпьютера «Сергей Королёв»: **sk.ssau.ru.**

Для организации доступа к суперкомпьютеру с локального компьютера используются следующие программы:

- эмулятор терминала с поддержкой протокола SSH версии 2, например **PuTTY** – для входа на кластер;
- программа работы с файлами по протоколу SFTP, например **WinSCP** – для передачи файлов на кластер.

Актуальные инструкции по настройке соединения в PuTTY и выбору необходимых параметров в WinSCP находится на сайте СГАУ hpc.ssau.ru.

Исходные файлы

До выхода на кластер пользователь должен подготовить исходные MPI-файлы (mpicc, mpicxx, mpif77, mpif90 и т.д.) и командный файл задания для системы пакетной обработки заданий **(*.pbs)**.

Исходные MPI-файлы создаются на языке С++ с использованием MPI-функций.

Команды на кластере СКЦ СГАУ можно выполнять только через систему пакетной обработки заданий, как в пакетном, так и в интерактивном режиме.

С помощью программы WinSCP подготовленные исходные файлы копируются на кластер в директорию пользователя.

Создание пакетного задания

Основные команды работы с системой пакетной обработки заданий (СПО):

qsub – постановка заданий в очередь;
qstat – просмотр статуса выполнения задания;
qdel – еудаление задания из очереди.

Основные опции команды ***qsub,*** часто используемые при формировании пакетного задания:

Опция	**Назначение**
-N *job*	Имя задания
-A *account*	Аккаунт
-l walltime =*время*	Время выполнения задания в формате чч:мм:сс (например 1 час: walltime=01:00:00)
-l nodes=*ноды*: ppn=*процессы*	Количество вычислительных узлов *nodes* с ppn процессами на каждом узле. Примеры: nodes=1:ppn=2 – запрос двух ядер на одном ноде.
-l software=*имя_ресурса [+n]*	Запрос использования *n* лицензий определённого программного обеспечения
-j oe	Перенаправлять поток стандартного вывода ошибок в стандартный вывод
-m ae	Посылать электронные сообщения об окончании и ошибках выполнения задания
-M user@mail.ru	Адрес электронной почты, на который отправляются сообщения СПО.

Основные опции команды ***qstat***

qstat –q – список очередей и их параметры;
qstat –a – список задач с расширенной информацией;
qstat -f <номер задачи> – полная информация о задаче;
qstat –n – информация на каких узлах запущена задача.;

Основные опции команды ***qdel***

qdel <номер задачи> – удаление задачи из очереди;
qdell all – удаление всех своих задач из очереди.

Для получения списка узлов с параметрами и состоянием рекомендуется использовать команду ***pestat*.**

Приведём пример простого командного файла задания **run.pbs** для запуска откомпилированной программы ***test.mpi***.

```
# ! /bin/bash          (настройка синтаксиса командных строк)
# PBS   -N   testjob
# PBS   -l  walltime=00:10:00
# PBS   -l  nodes=1:ppn:8
# PBS   -j  oe
cd $PBS_O_WORKDIR                (смена директория для qsub)
mpirun   ./test.mpi  a.txt  x.txt  y.txt (запуск программы test.mpi с параметрами a.txt, x.txt, y.txt).
```

Компиляция и запуск MPI программ

Процесс запуска MPI-программы включает в себя следующие шаги:

1. выбор реализацию MPI;
2. компиляцию MPI программы;
3. запуск MPI программы.

Выбор реализации MPI

Работа выполняется в командном режиме программы PuTTY.

Для выбора MPI среды выполните следующие команды:

[user@mgt1 ~] $ *module avail*

-----------– /etc/modulefiles --------------------------

impi/3 impi/4 openmpi

[user@mgt1 ~] $ *module load impi/4*

[user@mgt1 ~] $

В результате выполнения этих команд модуль impi/4 загружает переменные среды для работы с Intel MPI библиотекой версии 4. Загруженные переменные сохраняются в течении текущей сессии.

После выбора реализации MPI можно использовать программы для компиляции и запуска параллельных приложений (mpicc, mpicxx, mpif77, mpif90 и т.д.).

Компиляция MPI программ

Для компиляции программ, написанных на языке С++, используется команда

mpicc – o <имя файла с объектным кодом> <имя файла с исходным кодом>

Приведём пример простой команды (файл для объектного кода ***test.mpi***, файл с исходным кодом ***main.cpp***).

[user@mgt1 ~] $ *mpicxx – o test.mpi main.cpp*

Запуск программы

Для постановки задания в очередь и передачи всех переменных среды на вычислительные ноды (узлы), необходимо использовать команду *qsub* с ключом ***–V.***

Например, для командного файла задания **run.pbs**:

[user@mgt1 ~] $ *qsub –V run.pbs*

Для получения информации о статусе запущенной задачи (списка очередей и их параметры; списка задач с расширенной информацией; полной информации о задаче; информации на каких узлах запущена задача) можно использовать команду ***qstat***.

Литература

1. *Геркель, В.П.* Лекции по параллельным вычислениям: учеб. пособие / *В.П. Геркель, В.А. Фурсов.* – Самара: Изд-во СГАУ, 2009. – 164 с.
2. Архитектуры и топологии многопроцессорных вычислительных систем. Курс лекций: учеб. пособие / *А.В. Богданов, В.В. Корхов* [и др.] / – М.: ИНТУИТ.РУ «Интернет-университет Информационных Технологий», 2004. – 176 с.
3. Лупин, С.А. Технология параллельного программирования / *С.А. Лупин, М.А. Посыпкин.* – М.: ИД «ФОРУМ»: ИНФРА-М, 2008. – 2008. – 208 с.

Учебное издание

\

Составители:
Татьяна Михайловна Кузьмишина,
Егор Вячеславович Гошин

ПАРАЛЛЕЛЬНАЯ РЕАЛИЗАЦИЯ ВЫЧИСЛИТЕЛЬНЫХ АЛГОРИТМОВ НА КЛАСТЕРЕ СГАУ «СЕРГЕЙ КОРОЛЁВ»

Методические указания

Редактор И.И. Спиридонова
Доверстка И.И. Спиридонова

Подписано в печать 30.06.2015. Формат 60 х 84 1/16.
Бумага офсетная. Печать офсетная. Печ. л. 2,0.
Тираж 200 экз. Заказ . Арт. – 46/2015.

федеральное государственное автономное образовательное учреждение высшего образования «Самарский государственный аэрокосмический университет имени академика С.П. Королева (национальный исследовательский университет) (СГАУ)

443086 Самара, Московское шоссе, 34.
Изд-во СГАУ 443086 Самара, Московское шоссе, 34.