Высокопроизводительные параллельные вычисления

Лекция №4
Тема: Параллельное программирование на основе MPI

Козбакова Айнур Холдасовна– PhD, профессор, кафедра «Информационные системы» Факультет «Инжиниринг и информационных технологии» 01.09.2025 г.

В вычислительных системах с *распределенной памятью* процессоры работают независимо друг от друга. Для организации параллельных вычислений в таких условиях необходимо иметь возможность *распределять вычислительную нагрузку и организовать информационное взаимодействие* (*передачу данных*) между процессорами.

Решение всех перечисленных вопросов и обеспечивает интерфейс передачи данных (*message passing interface – MPI*).

В общем плане, для распределения вычислений между процессорами необходимо проанализировать алгоритм решения задачи, выделить информационно независимые фрагменты вычислений, провести их программную реализацию и затем разместить полученные части программы на разных процессорах. В рамках *MPI* принят более простой подход – для решения поставленной задачи разрабатывается одна программа и эта единственная программа запускается одновременно на выполнение на всех имеющихся процессорах. При этом для того чтобы избежать идентичности вычислений на разных процессорах, можно, во–первых, подставлять разные данные для программы на разных процессорах, а во–вторых, использовать имеющиеся в *MPI* средства для идентификации процессора, на котором выполняется программа (тем самым предоставляется возможность организовать различия в вычислениях в зависимости от используемого программой процессора).

Подобный способ организации параллельных вычислений получил наименование модели *"одна программа множество процессов"* (*single program multiple processes or* SPMP).

Для организации информационного взаимодействия между процессорами в самом минимальном варианте достаточно операций приема и передачи данных (при этом, конечно, должна существовать техническая возможность коммуникации между процессорами – *каналы* или *линии связи*). В *MPI* существует целое множество операций передачи данных. Они обеспечивают разные способы пересылки данных, реализуют практически все коммуникационные операции. Именно данные возможности являются наиболее сильной стороной *MPI* (об этом, в частности, свидетельствует и само название *MPI*).

Следует отметить, что попытки создания программных средств передачи данных между процессорами начали предприниматься практически сразу с появлением локальных компьютерных сетей. Однако подобные средства часто были неполными и, самое главное, являлись несовместимыми. Таким образом, одна из самых серьезных проблем в программировании – переносимость программ при переводе программного обеспечения на другие компьютерные системы – проявлялась при разработке параллельных программ в максимальной степени.

MPI – это стандарт, которому должны удовлетворять средства организации передачи сообщений и *MPI* – это программные средства, которые обеспечивают возможность передачи сообщений и при этом соответствуют всем требованиям стандарта *MPI*. Так, по стандарту, эти программные средства должны быть организованы в виде библиотек программных функций (*библиотеки MPI*) и должны быть доступны для наиболее широко используемых алгоритмических языков C и Fortran. Подобную "двойственность" *MPI* следует учитывать при использовании терминологии. Как правило, аббревиатура *MPI* применяется при упоминании стандарта, а сочетание "библиотека *MPI*" указывает на ту или иную программную реализацию стандарта. Однако достаточно часто для краткости обозначение *MPI* используется и для библиотек *MPI*, и, тем самым, для правильной интерпретации термина следует учитывать контекст.

Приведем ряд важных положительных моментов *MPI*:

– *MPI* позволяет в значительной степени снизить остроту проблемы переносимости *параллельных программ* между разными компьютерными системами – *параллельная программа*, разработанная на алгоритмическом языке C или Fortran с использованием библиотеки *MPI*, как правило, будет работать на разных вычислительных платформах;

– *MPI* содействует повышению эффективности параллельных вычислений, поскольку в настоящее время практически для каждого типа вычислительных систем существуют реализации библиотек *MPI*, в максимальной степени учитывающие возможности компьютерного оборудования;

– *MPI* уменьшает, в определенном плане, сложность разработки *параллельных программ*, т. к., с одной стороны, большая часть основных операций передачи данных предусматривается стандартом *MPI*, а с другой стороны, уже имеется большое количество библиотек параллельных методов, созданных с использованием *MPI*.

3.1 MPI– основные понятия и определения

Рассмотрим ряд понятий и определений, являющихся основополагающими для стандарта *MPI*.

Понятие параллельной программы

Под ***параллельной программой*** в рамках *MPI* понимается множество одновременно выполняемых *процессов*. *Процессы* могут выполняться на разных процессорах, но на одном процессоре могут располагаться и несколько *процессов* (в этом случае их исполнение осуществляется в режиме разделения времени). В предельном случае для выполнения *параллельной программы* может использоваться один процессор – как правило, такой способ применяется для начальной проверки правильности *параллельной программы*.

Каждый *процесс параллельной программы* порождается на основе копии одного и того же программного кода (*модель SPMP*). Данный программный код, представленный в виде исполняемой программы, должен быть доступен в момент запуска *параллельной программы* на всех используемых процессорах. Исходный программный код для исполняемой программы разрабатывается на алгоритмических языках C или Fortran с применением той или иной реализации библиотеки *MPI*.

Количество *процессов* и число используемых процессоров определяется в момент запуска *параллельной программы* средствами среды исполнения *MPI*–программ и в ходе вычислений не может меняться без применения специальных, но редко задействуемых средств динамического порождения *процессов* и управления ими, появившихся в стандарте *MPI* версии 2.0. Все *процессы* программы последовательно перенумерованы от 0 до p–1, где p есть общее количество *процессов*. Номер *процесса* именуется рангом *процесса*.

Операции передачи данных.

Основу *MPI* составляют операции передачи сообщений. Среди предусмотренных в составе *MPI* функций различаются *парные* (*point–to–point*) операции между двумя *процессами* и *коллективные* (*collective*) коммуникационные действия для одновременного взаимодействия нескольких *процессов*.

Для выполнения парных операций могут использоваться разные *режимы передачи*, среди которых синхронный, блокирующий и др. В стандарт *MPI* включено большинство основных коллективных операций передачи данных.

Понятие коммуникаторов.

Процессы параллельной программы объединяются в *группы*. Другим важным понятием *MPI*, описывающим набор *процессов*, является понятие коммуникатора. Под

коммуникатором в *MPI* понимается специально создаваемый служебный объект, который объединяет в своем составе группу *процессов* и ряд дополнительных параметров (контекст), используемых при выполнении операций передачи данных.

Парные операции передачи данных выполняются только для *процессов*, принадлежащих одному и тому же коммуникатору. Коллективные операции применяются одновременно для всех *процессов* одного коммуникатора. Как результат, указание используемого коммуникатора является обязательным для операций передачи данных в *MPI*.

В ходе вычислений могут создаваться новые и удаляться существующие группы *процессов* и коммуникаторы. Один и тот же *процесс* может принадлежать разным группам и коммуникаторам. Все имеющиеся в *параллельной программе процессы* входят в состав конструируемого по умолчанию коммуникатора с идентификатором MPI_COMM_WORLD.

В версии 2.0 стандарта появилась возможность создавать глобальные коммуникаторы (*intercommunicator*), объединяющие в одну структуру пару групп при необходимости выполнения коллективных операций между *процессами* из разных групп.

Типы данных.

При выполнении операций передачи сообщений для указания передаваемых или получаемых данных в функциях *MPI* необходимо указывать *тип пересылаемых данных*. *MPI* содержит большой набор *базовых типов данных*, во многом совпадающих с типами данных в алгоритмических языках C и Fortran. Кроме того, в *MPI* имеются возможности создания новых *производных типов* данных для более точного и краткого описания содержимого пересылаемых сообщений.

Виртуальные топологии

Как уже отмечалось ранее, парные операции передачи данных могут быть выполнены между любыми *процессами* одного и того же коммуникатора, а в коллективной операции принимают участие все *процессы* коммуникатора. Логическая топология линий связи между *процессами* имеет структуру полного графа (независимо от наличия реальных физических каналов связи между процессорами). Вместе с этим для изложения и последующего анализа ряда параллельных алгоритмов целесообразно логическое представление имеющейся коммуникационной сети в виде тех или иных топологий. В *MPI* имеется возможность представления множества *процессов* в виде решетки произвольной размерности. При этом граничные *процессы* решеток могут быть объявлены соседними, и, тем самым, на основе решеток могут быть определены структуры типа тор. Кроме того, в *MPI* имеются средства и для формирования логических (виртуальных) топологий любого требуемого типа.

Приступая к изучению *MPI*, можно отметить, что, с одной стороны, *MPI* достаточно сложен – в стандарте *MPI* предусматривается наличие более чем 120 функций. С другой стороны, структура *MPI* является тщательно продуманной – разработка *параллельных программ* может быть начата уже после рассмотрения всего лишь 6 функций *MPI*. Все дополнительные возможности *MPI* могут осваиваться по мере роста сложности разрабатываемых алгоритмов и программ.

Основы MPI

Приведем минимально необходимый набор функций *MPI*, достаточный для разработки сравнительно простых *параллельных программ*.

Инициализация и завершение MPI–программ

Первой вызываемой функцией *MPI* должна быть функция:

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

где

– argc — указатель на количество параметров командной строки,

– argv — параметры командной строки,

– применяемая для инициализации среды выполнения *MPI*–программы. Параметрами функции являются количество аргументов в командной строке и адрес указателя на массив символов текста самой командной строки.

Последней вызываемой функцией MPI обязательно должна являться функция:

int MPI_Finalize(void).

Как результат, можно отметить, что структура *параллельной программы*, разработанная с использованием *MPI*, должна иметь следующий вид:

```
#include "mpi.h"
int main(int argc, char *argv[]) {
 <программный код без использования функций MPI>
 MPI_Init(&agrc, &argv);
 <программный код с использованием функций MPI>
 MPI_Finalize();
 <программный код без использования функций MPI>
 return 0;
}
```

Следует отметить:

– файл mpi.h содержит определения именованных констант, прототипов функций и типов данных библиотеки *MPI*;

– функции MPI_Init и MPI_Finalize являются обязательными и должны быть выполнены (и только один раз) каждым *процессом параллельной программы*;

– перед вызовом MPI_Init может быть использована функция MPI_Initialized для определения того, был ли ранее выполнен вызов MPI_Init, а после вызова MPI_Finalize – MPI_Finalized аналогичного предназначения.

Рассмотренные примеры функций дают представление синтаксиса именования функций в *MPI*. Имени функции предшествует префикс *MPI*, далее следует одно или несколько слов названия, первое слово в имени функции начинается с заглавного символа, слова разделяются знаком подчеркивания. Названия функций *MPI*, как правило, поясняют назначение выполняемых функцией действий.

Определение количества и ранга процессов.

Определение *количества процессов* в выполняемой *параллельной программе* осуществляется при помощи функции:

int MPI_Comm_size(MPI_Comm comm, int *size),

где

– comm — коммуникатор, размер которого определяется,

– size — определяемое количество процессов в коммуникаторе.

Для определения ранга *процесса* используется функция:

int MPI_Comm_rank(MPI_Comm comm, int *rank),

где

– comm — коммуникатор, в котором определяется ранг процесса,

– rank — ранг процесса в коммуникаторе.

Как правило, вызов функций MPI_Comm_size и MPI_Comm_rank выполняется сразу после MPI_Init для получения общего количества *процессов* и ранга текущего *процесса*:

```
#include "mpi.h"
int ProcNum, ProcRank;
 <программный код без использования функций MPI>
 MPI_Init(&agrc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
 MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
```

```
  <программный код с использованием функций MPI>
 MPI_Finalize();
 <программный код без использования функций MPI>
 return 0;
}
```

Следует отметить:

– коммуникатор MPI_COMM_WORLD создается по умолчанию и представляет все процессы выполняемой параллельной программы;

– ранг, получаемый при помощи функции MPI_Comm_rank, является рангом процесса, выполнившего вызов этой функции, т. е. переменная ProcRank примет различные значения у разных процессов.

Передача сообщений.

Для *передачи сообщения процесс*– отправитель должен выполнить функцию:

```
int MPI_Send(void *buf, int count, MPI_Datatype type, int dest,
 int tag, MPI_Comm comm),
```

где

– buf — адрес буфера памяти, в котором располагаются данные отправляемого сообщения;

– count — количество элементов данных в сообщении;

– type — тип элементов данных пересылаемого сообщения;

– dest — ранг процесса, которому отправляется сообщение;

– tag — значение–тег, используемое для идентификации сообщения;

– comm — коммуникатор, в рамках которого выполняется передача данных.

Для указания типа пересылаемых данных в *MPI* имеется ряд базовых типов, полный список которых приведен в табл. 3.1.

Таблица 3.1. Базовые (предопределенные) типы данных *MPI* для алгоритмического языка С

Тип данных *MPI*	**Тип данных С**
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int

MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Следует отметить:

– отправляемое сообщение определяется через указание блока памяти (буфера), в котором это сообщение располагается. Используемая для указания буфера триада (buf, count, type) входит в состав параметров практически всех функций передачи данных;

– процессы, между которыми выполняется передача данных, в обязательном порядке должны принадлежать коммуникатору, указываемому в функции MPI_Send;

– параметр tag используется только при необходимости различения передаваемых сообщений, в противном случае в качестве значения параметра может быть использовано произвольное положительное целое число

Сразу же после завершения функции MPI_Send *процесс*–отправитель может начать повторно использовать буфер памяти, в котором располагалось отправляемое сообщение. Также следует понимать, что в момент завершения функции MPI_Send состояние самого пересылаемого сообщения может быть совершенно различным: сообщение может располагаться в *процессе*–отправителе, может находиться в состоянии передачи, может храниться в *процессе*–получателе или же может быть принято *процессом*–получателем при помощи функции MPI_Recv. Тем самым, завершение функции MPI_Send означает лишь, что операция передачи начала выполняться и пересылка сообщения рано или поздно будет выполнена.

Прием сообщений.

Для приема сообщения *процесс*–получатель должен выполнить функцию:

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

где

– buf, count, type — буфер памяти для приема сообщения, назначение каждого отдельного параметра соответствует описанию в MPI_Send;

– source — ранг процесса, от которого должен быть выполнен прием сообщения;

– tag — тег сообщения, которое должно быть принято для процесса;

– comm — коммуникатор, в рамках которого выполняется передача данных;

– status – указатель на структуру данных с информацией о результате выполнения операции приема данных.

Следует отметить:

– буфер памяти должен быть достаточным для приема сообщения. При нехватке памяти часть сообщения будет потеряна и в коде завершения функции будет зафиксирована ошибка переполнения; с другой стороны, принимаемое сообщение может быть и короче, чем размер приемного буфера, в таком случае изменятся только участки буфера, затронутые принятым сообщением;

– типы элементов передаваемого и принимаемого сообщения должны совпадать;

– при необходимости приема сообщения от любого процесса– отправителя для параметра source может быть указано значение MPI_ANY_SOURCE (в отличие от функции передачи MPI_Send, которая отсылает сообщение строго определенному адресату).