Система программирования MPI

Система MPI (Message Passing Interface) это стандартная библиотека подпрограмм (Fortran) или функций (C) которые может вызвать программа использующая технологию передачи сообщений. MPI осуществляет координацию между копиями программы, выполняющимися на разных вычислительных и является достаточно гибкой для использованной даже в системах с разделяемой (общей) памятью. Стандартизация библиотеки MPI позволяет разрабатывать программные коды работающие на различных платформах. Может использоваться также на гетерогенных вычислительных системах.

Модели параллельного программирования

1. модель передачи сообщений
2. модель инструкций при параллелизации данных

Эти походы отличаются организацией адресного пространства. Кратко рассмотрим обе модели.

Модель передачи сообщений

В данной модели процессы обмениваются сообщениями между собой. Эта модель наиболее распространена на системах с распределенной памятью. Вычислительный кластер как раз такая система. И хотя MPI программа может выполняться на системе с общей (разделяемой) памятью, такая программа непосредственно не воспользуется возможностями общей памяти на узле.

Инструкции основанные на параллелизации данных

В этой модели, программа на языке программирования дополняется инструкциями, которые являются коментариями в коде и используются только если заданы определенные ключи компилятора. Эти инструкции компилятору реализуют распределение данных по процессорам (или вычислительным ядрам) Одной из таких систем являетс OpenMP.

Новый подход

Новым многообещающим подходом является смешанное использование обеих систем (MPI и OpenMP) в одной программной модели. Такой подход продуктивен при использовани таких кластеров, в которых каждый вычислительный узел является многоядерным SMP компьютером.

Разработка параллелльной программы

Параллельные программы состоят из множественных копий обменивающихся между собой библиотечными вызовами. Эти вызовы можно грубо разбить на 4 класса:

1. Вызовы (подпрограммы) для инициализации, управления и завершения сообщений. Эти вызовы используются для начала передачи сообщения, идентификации процесса, создания групп процессов и определения
2. Обмен сообщениями между парами процессов.
3. Коллективные взаимодействия в группах процессов.
4. Создание произвольных типов данных для передачи их в сообщениях.

Основная цель разработки параллельной программы это достижение более высокой скорости вычислений по сравнению с последовательной версией программы. Для этого необходимовыполнить декомпозицию задачи, равномерно распределить нагрузку между узлами, минимизировать время простоя узлов.

Распараллеливание на основе декомпозиции массивов

При данном подходе данные делятся на фрагменты распределяемые по разным просессорам. Каждый процессор работает только со своей порцией данных. Процессы должны иногда обмениваться сообщениями для обмена или сбора данных.

Декомпозиция данных позволяет имет единый поток инструкций. Подход иногда называется SPMD (Single Program — Multiple Data одна программа — множественные данные).
Программа в этом подходе одна и та же на каждом процессоре.

Такая стратегия обычно используется когда в алгоритмах использующих конечно-разностные схемы, когда каждый процессор работает над своим относительно большим фрагментом данных, а объем сообщений, которыми необходимо обмениваться на каждой итерации относительно невелик.

Пример распараллеливания данных

Типичный пример это решение дифференциального уравнения в частных производных. Рассмотрим решение уравнения Пуассона на единичном квадрате с периодическими граничными условиями:

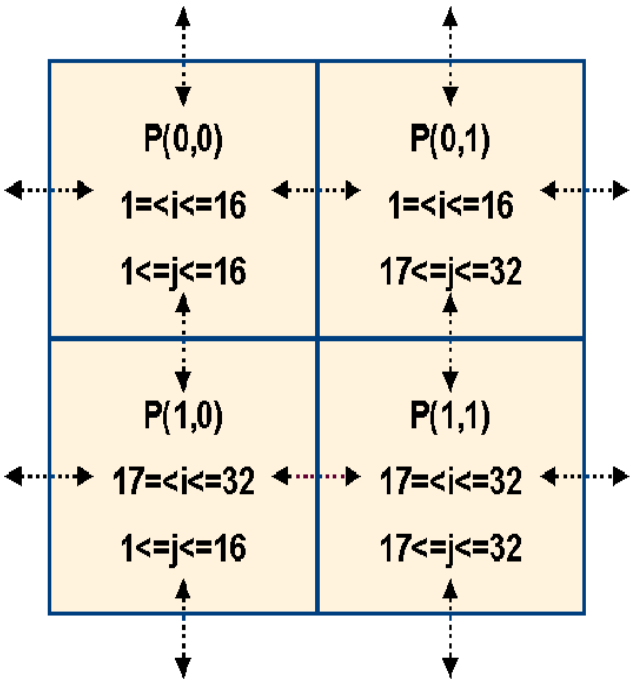
$$\nabla^2 u(x,y) = f(x,y)$$
$$u(x,0) = u(x,1) = u(0,y) = u(1,y) = 0$$

Эту задачу можно решить на двумерной сетке (массиве) значения массива *u(i,j)* нулями а затем выполняя итерационную процедуру по формуле:

```
u(i,j) = f(i,j) + 0.25*(u(i,j-1) + u(i,j+1) + u(i-1,j) + u(i+1,j))
```

на всей сетке до тих пор пока не выполнится какой-нибудь критерий сходимости.

Пусть индексы *i* и *j* изменяются от 1 до 32. И пусть для решения мы имеем 4 процессора организованных в матрицу 2x2, как показано на картинке.



Декомпозиция массива для дискретного решения уравнения Пуассона.

Тогда данные удовлетворяющие соотношению:

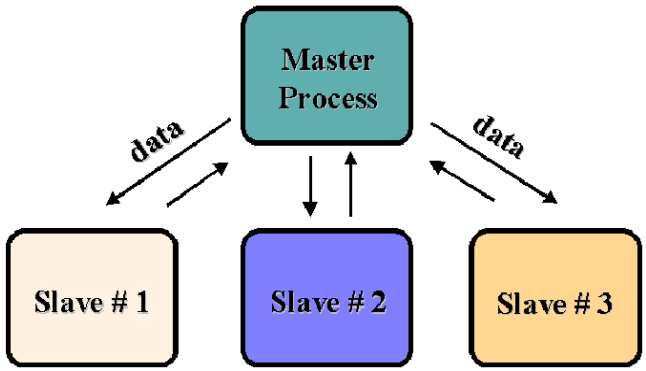
$$16*k+1<=i<=16*k+16,$$
$$16*m+1<=j<=16*m+16$$

будут обрабатываться процессором *P(k, m)* , где *k* и *m* равны 0 или 1. Каждый процессор выполняет итерации только на своих данных, останавливаясь только для обмена данными на границах после каждой итерации.

Функциональная декомпозиция

Для некоторых больших и сложных задач декомпоцизия данных может оказаться не самым эффективным методом параллелизации. Может оказаться, что использование декомпозиции данных может приводит к тому, что производительность будет низкая из-за того, что будет ограничиваться самым медленным процессом, в то время когда остальные процессоры будут простаивать. Функциональная декомпозиция или параллелизм задач может оказаться выигрышной стратегией с точки зрения вычислительной эффективности. Здача разбивается на множество задач меньшей сложности и они раздаются последовательно освободившимся процессорам. Тот процессор, который раньше заканчивает свою работу на частичной задачей сразу получает новую.

Функциональная декомпозиция представлена как клиент-серверная парадигма. Задачи размещаются на группах подчиненных процессоров, в то время как управляющий процессор тае же может быть занят какой-то частичной задачей. Простой пример: запуск копий одной и той же программы на группе процессоров, но с различными входными данными. По мере того как процессор заканчивает с задачей, ему поступают новые входные данные. Параллелизм задач также может быть использован и более сложным образом.



Распределение нагрузки

Необходимо распределять равномерно нагрузку между всеми процессами таким образом, чтобы минимизировать простой процессоров уже закончивших выполнение своей задачи и ожидающих отстающих. Задача является нетривиальной, поскольку время выполнения может зависеть от данных. В случае большой вариации времени выполнения в зависимости от данных, декомпозиция данных может быть не эффективной.

Время выполнения

Полное время выполнения для параллельной программы является основным параметром подлежащим анализу, оптимизации, сравнению с целью улучшения программы. Следующие три компоненты составляют время выполнения параллельной программы:

1. вычислительное время
2. время простоя
3. время обмена сообщениями

Идеальной была бы ситуация, когда все свое время процессоры заняты вычислениями на заданном наборе данных. Тогда бы программа завершала бы свою работу на N процессорах за 1/N часть времени затрачиваемого на одном процессоре. К сожалению, не все время процессора в параллельной системе расходуется на вычисления.

Время простоя расходуется когда процесс ожидает прихода данных от другого процесса. В течение этого времени ожидания процессор не выполняет полезной работы. Простой пример это осуществление операции ввода/вывода параллельной программой, которые как правило выполняются одним процессором, в то время как остальные простаивают.

Время обмна сообщениями расходуется на посылку и прием сообщений от других процессоров. Это время можно разбить на время подготовки сообщения и на собственно его пересылку. Поскольку последовательная программа вообще не занимается пересылкой сообщений, то для достижения преимущества параллельной программой перед последовательной необходимо минимизировать затраты на обмен сообщениями.

Время простоя

Необходимо минимизировать время простоя процессоров. Есть несколько способов продвинуться в этом направлении. Один из способов это загрузить процессор в то время, пока он ожидает события по передаче/приему сообщения. Аккуратное сочетание неблокирующих сообщений и вычислений не связанных с передаваемыми данными — вот путь оптимизации программы в этом направлении. На практике это, однако, бывает очень

трудно реализовать.

Подпрограммы MPI

Стандартные подпрограммы MPI включают в себя следующие типы операций:

- *Сообщения точка-точка*
- *Коллективные сообщения*
- Создание групп процессов
- Создание топологии процессов
- Контроль окружения

Обмен сообщениями между отдельными процессами

Элементарная операция MPI по пересылке сообщения представляет собой прямую передачу сообщения между двумя процессорами, когда один *посылает* данные, а другой *принимает* эти самые данные. В общем случае сообщение содержит в себе некоторые блоки данных, часть из которых представляет собой идентификатор сообщения, содержащий вспомогательные сведения (идентификаторы процессоров участвующих в передаче и приеме сообщения, и т. п.) и собственно данные, которые будут испоьзованы процессором в вычислениях.

MPI использует три типа информации, чтобы характеризовать тело сообщения, т. е. данные передавае от процессора процессору:

1. **Буфер** — указатель на адрес в памяти с которго отправляемые или получаемые данные могут быть найдены. В Фортране это собственно имя массива, которое указывает на первый его элемент.
2. **Тип данных** — тип данных передаваемых или получаемых. Это могут быть как базовые типы языка программирования (Fortran, C), так и производные типы определенные пользователем из базовых.
3. **Число** элементов передаваемых/принимаемых.

MPI стандартизует обозначения стандартных типов так, что пользователю нет нужды беспокоиться о различии представления типов на различных машинах в гетерогенной вычислительной среде.

Режимы связи и критерии завершения

MPI обеспечивает определенную гибкость при посылке сообщений. Например, *синхронная* посылка сообщения завершается когда получено подтверждение о его приеме от принимающей стороны, а *асинхронная* посылка сообщения завершается когда сообщение скопировано в локальный буфер. При этом посылающая сторона ничего не получает в слуае доставки сообщения принимающей стороне. Во всех случаях пользователь может не беспокоиться о потере данных в перезаписываемых областях памяти.

Имеется четыре режима посылки сообщений:

- Стандартная
- Синхронная
- Асинхронная
- По готовности

Для принимающей стороны есть только один режим приема сообщения. Прием завершается когда принимаемые данные реально получены и готовы для дальнейшего использования программой.

Блокирующие и неблокирующие сообщения

При блокирующих посылке или приеме выход из подпрограммы не происходит пока операция по передаче или приему не завершится фактически. При этом если произошел выход из процедуры, то это означает, что переменная в которую принимаются данные уже перезаписана на стороне принимающего процессора.

Неблокирующая посылка осуществляет выход из процедуры немедленно без всякой информации о том удевлетворен ли критерий завершения передачи. Такой подход имеет то преимущество, что процессор освобождается для дальнейшей работы, в то время как передача сообщения выполняется в фоновом режиме. Впоследствии можно проверить была ли завершена передача. Например, неблокирующая синхронная посылка возвращается из процедуры немедленно, хотя передача данных не будет завершена пока не будет получено подтверждение. А до тех пор нельзя быть уверенным, что передача данных произошла наверняка.

Коллективные взаимодействия

Коммуникатор это объект MPI определяющий группу процессов, которым разрешено взаимодействие через передачу/прием сообщений. Каждое сообщение MPI должно содержать коммуникатор определенный по имени и входящий в список параметров процедуры MPI. По умолчанию, все процессы являются членами группы с коммуникатором MPI_COMM_WORLD.

Процедуры *коллективного обмена сообщениями*, также называемые коллективными операциями передают данные среди всех процессоров в группе. Эти процедуры позволяют использовать разное количество прцессоров на стороне пердачи сообщения. Например, осуществлять передачу от одного нескольким процессорам или наоборот, от нескольких одному. Все коллективные операции блокирующие.

Имеется три основных типа событий MPI при коллективных операциях:

1. Синхронизация — каждый процесс ждет пока все процессы включенные в группу достигнут определенной точки синхронизации в программе.
2. Перемещение данных — данные передаются всем процессам в группе.
3. Коллективные вычисления — один процесс в группе собирает данные от других процессов в группе и выполняет с ними операции (складывеет, умножает, ...).

Основными преимуществами коллективных операций по сравнению с обменом сообщений точка-точка являются:

- Вероятность допустить ошибку в исходном коде ниже. Одна строка вызова коллективной процедуры обычно заменяется несколькими вызовами процедур типа точка-точка.
- Исходный код легче читаем, что означает более простую отладку.
- Оптимизированные формы коллективных операций часто быстрее выполняются, чем эквивалентные конструкции выполненные в терминах операций типа точка-точка.

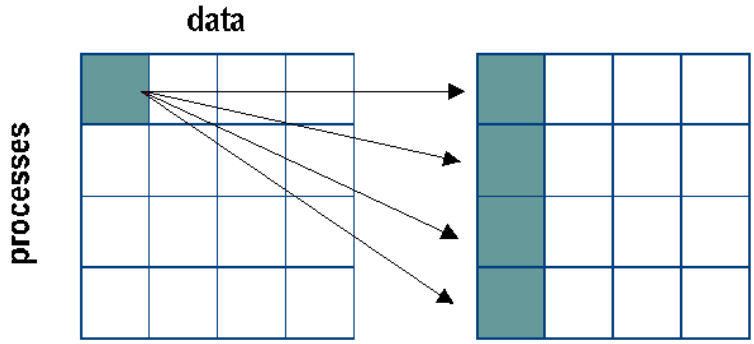
Краткоо рассмотрим следующие коллективные операции:

- Широковещательные (broadcast) операции

- Разбрасывание данных нескольким процессорам и сбор данных (scatter and gather operations)
- Операции редукции(reduction)

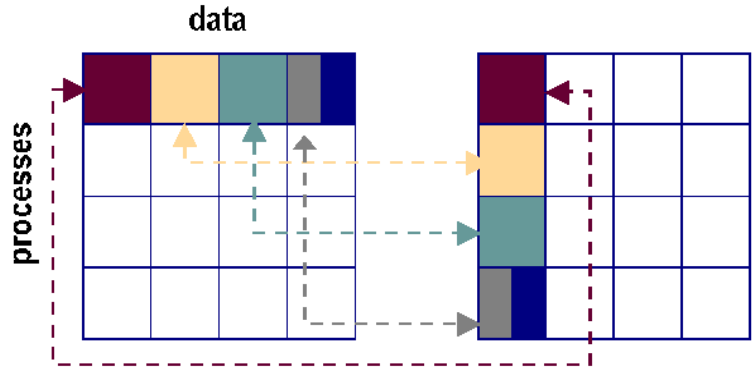
Широковещательные операции

Это самый простой вариант коллективных операций. При широковещании единственный процесс рассылает копии одних и тех же данных всем другим процессам в группе. Такая операция отображена на иллюстрации ниже. Каждый ряд представляет различные процессоры, а каждый закрашенный прямоугольник отображает фрагмент данных.



Операции распределения и сбора данных

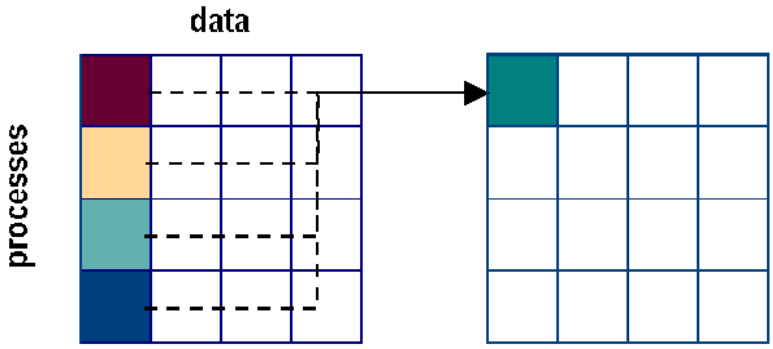
Операция распределения данных показана на рисунке внизу. Данные распределяются равными частями на группе процессоров.



В операции распределения, все данные, скажем некоторый массив, первоначально находятся на одном процессоре, после операции распределения различные фрагменты данных (массива) распределяются по различным процессорам. Нижний квадрат с различными цветами отражает то обстоятельство, что данных может не хватить, чтобы быть поровну распределенными между всеми поцессами. Операция сбора дaанных обратна операции распределения, она собирает фрагменты данных с разных процессоров в один блок (массив) в правильном порядке на одном процессоре.

Операции редукции

В таких коллективных операциях единственный процесс собирает данные с других процессоров в группе и выполняет над ними операцию таким образом, что выходным параметрм является одно число. Это может быть суммирование, перемножение, поиск максимума или минимума, побитовые операции, логические операции.



Группы процессов

Группа процессов это нумерованное множество процессов, в котором каждому *процессу* поставлено в соответствии число, называемое *рангом* процесса или *идентификатором* процесса. Идентификаторы всегда начинаются от 0 и заканчиваются N-1, где N — это число процессов в группе.

Хотя число процессоров при выполнении программы MPI фиксировано, группы процессов и комуникаторы им соответствующие могут создаваться динамически в процессе выполнения программы. Более того, процесс может быть членом нескольких групп или коммуникаторов и иметь уникальный идентификатор в каждой из групп.

Топология процессов

Топология в MPI это механизм сопоставления различных схем идентификации процессоров в группе. То есть топология отображает нумерацию процессов на геометрическую структуру. MPI поддерживает два основных типа топологии — Декартову (сетка, grid) и топологию графов. Все топологии MPI виртуальны и не имеют простого отношения к физическому устройству многопроцессорной вычислительной системы.

Виртульная топлогия в MPI призвана обеспечить *эффективность* взаимодействия (обмена сообщениями) и удобство разработки программного кода. Например, сеточная топология удобна для приложений использующих взаимодействие между ближайшими соседями на прямоугольной сетке. Какую топологию использовать определяет разработчик ПО.

Контроль окружения

В библиотеке MPI имеется некоторое количество процедур для контроля и получения информации об окружении. Эти процедуры используются для инициализации и завершения испоьзования среды MPI, завершения процессов принадлежащих заданному коммуникатору, определению числа процессов принадлежащих тому или иному коммуникатору, определению ранга вызывающего процесса в заданной группе (коммуникаторе).

Компиляция и выполнение программ MPI

Стандарты MPI не предписывают как именно должны запускаться программы MPI, поэтому реализация может варьироваться от компьютера к компьютеру. При компиляции программы главное связать программу с библиотекой MPI.

Чтобы запустить программу MPI чаще всего используют команду вроде этой:

```
$  mpirun -np 4 execfile
```

Здесь программа с именем `execfile` запускается на 4-х процессорах.

Простая программа

```
      PROGRAM mpitest
      INCLUDE 'mpif.h'
      INTEGER ierr, rank, size

      CALL MPI_INIT(ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      PRINT *, 'I am ', rank, ' of ', size
      CALL MPI_FINALIZE(ierr)
      END
```

Откомпилировать эту программу можно командой

```
mpif77  -o  mpitest  mpitest.f
```

или командой

```
mpif90  -o  mpitest  mpitest.f90
```

Запуск осуществляется командой (для примера на четырех процессорах)

```
mpirun  -np  4  mpitest
```

Вывод программы помещается в файл `output` в каталоге `./mpitest.1`

Вышеописанные команды верны для кластера um32.imm.uran.ru.