Введение в MPI

Параллельно поздоровавшись с миром, **Михаил Остапкевич** и **Евгений Балдин** призадумались о Жизни с большой буквы.

Наш эксперт

Михаил Остапкевич
Романтик, очарованный компьютерами и создаваемыми в них идеальными мирами; верит, что сложнейшие новые технологии могут и должны служить во благо человечеству.

Наш эксперт

Евгений Балдин
Физик, который действительно знает, что такое нехватка вычислительных ресурсов.

Времена больших векторных суперкомпьютеров прошли, ну или пока не настали. Что мы имеем взамен? Множество независимых «писишек», даже если они и установлены в стойки и управляются квалифицированными администраторами! Запустить на них пачку задач легко, но как подотчетные процессы будут общаться? Да с помощью MPI!

MPI или Message Passing Interface или, по-простому, интерфейс передачи сообщений – это стандартный программный интерфейс для передачи информации между процессами, выполняющими одну задачу. Стандарт поддерживается консорциумом MPI Forum, членами которого являются практически все крупные производители вычислительной техники и программного обеспечения. Первый стандарт MPI 1.0 принят в 1994 году. Формально разработка началась в 1991 году, когда небольшая группа ученых-информатиков собралась в уединенном австрийском горном пансионате и начала активно обмениваться мнениями. В сентябре 2012 года вышла спецификация MPI 3.0.

Технология MPI ориентирована в первую очередь на кластеры и на массивно-параллельные системы (MPP), но применяется также в системах SMP и NUMA. Это означает, что MPI-программы можно оптимизировать как для самых мощных решений из TOP500, так и для домашнего компьютера. Почти на каждом современном десктопе сейчас стоит более одного вычислительного ядра. Да что десктопы – телефоны тоже не отстают от этой моды. И эти ядра просто необходимо использовать!

Исполнение программы производится несколькими параллельно исполняемыми процессами, которые представляют собой разные экземпляры одной и той же программы. Каждый процесс имеет свой набор данных. Режим работы, когда одна и та же программа запущена на разных узлах или процессорах и обрабатывает разные наборы данных, называют SPMD (Single Program Multiple Data). Данные других процессов процессу не видны, поэтому для обмена данными между процессами требуется MPI, а именно посылка сообщений между ними. Механизмы MPI гарантируют доставку сообщений и соответствие порядка, в котором они посылались, порядку их приема. Существуют и другие библиотеки для обмена сообщениями между удаленными процессами (например, PVM). Однако среди всех библиотек, решающих похожие задачи, именно MPI получил наибольшее распространение. С технологической стороны, по-видимому, три причины сыграли в этом решающую роль:

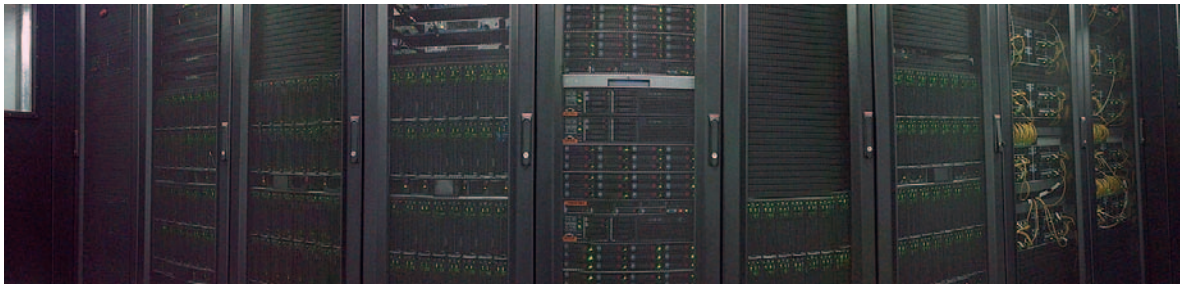
- высокая эффективность программ на MPI;
- высокая переносимость программ, написанных на MPI;
- наличие свободной реализации.

Эффективность является одним из важнейших свойств программ. Важнее только реализация необходимых функций и надежность работы и воспроизводимость результатов. В параллельном программировании, где программы отличаются большим объемом вычислений, эффективность приобретает особый смысл. Одна из самых главных причин, для чего создают параллельные программы – это уменьшение времени их работы, а оно напрямую зависит от эффективности.

Эффективность в MPI достигается в первую очередь за счет использования самого быстрого из доступных средств доставки сообщений. В мультипроцессоре используются окна разделяемой несколькими процессами памяти. В мультикомпьютерах используется самая быстрая сеть (Infiniband, Myrinet). Если она недоступна, то используется старый, добрый и надежный TCP/IP.

Обычный «сферический и в вакууме» прикладной программист, как правило, достаточно хорошо изолирован от специфики аппаратной платформы, на которой он пишет свои программы. Временами эта изолированность выходит боком, но такова жизнь. Прослойка из системного программного обеспечения и языков программирования высокого уровня позволяет ему игнорировать отличия между платформами. Следование общепринятым стандартам позволяет легко переносить программы с одной аппаратной платформы на другую.

В параллельном программировании языки и интерфейсы уже не скрывают новые, специфические для параллельных ЭВМ свойства архитектуры, которые связаны с координацией работы параллельно исполняющихся частей программы и обменом данных между ними. В результате получаются параллельные программы, ориентированные на конкретные машины. С одной стороны, свойства MPI позволяют строить его эффективные реализации на широком спектре параллельных ЭВМ. С другой стороны, набор операций удобен для написания самых разнообразных параллельных программ. В итоге, MPI стал самым распространенным стандартом для написания параллельных программ, для которого построены реализации на очень большом числе параллельных архитектур. Все это привело к тому, что параллельная программа, написанная с использованием MPI, обладает высокой переносимостью.



Гибридный кластер Сибирского суперкомпьютерного центра — типичный вычислительный центр.

Кроме переносимости и эффективности, другим существенным свойством MPI является языковая нейтральность. Реализации библиотеки MPI не привязаны к какому-либо одному языку. Вместо этого они описывают семантику операций, доступных MPI-программам через функциональный интерфейс. Как правило, MPI-программы пишут на C, Fortran или C++, но можно использовать и многие другие языки, например, C#, Java, Python или R.

Наличие свободной реализации позволяет легко установить MPI и начать обучать свои программы секретам взаимодействия:

```
> sudo aptitude install mpi-default-bin mpi-default-dev
```

Документация для вдумчивого изучения тоже не помешает:

```
> sudo aptitude install mpi-doc mpi-specs
```

mpi-doc, кроме html-справочника, добавляет в систему man-странички по многочисленным MPI-функциям, очень удобные для быстрого подглядывания.

Здравствуй, Мир!

Предполагается, что вы представляете, что такое программирование на C, ну или хотя бы написали на этом языке свой личный HelloWorld. Вот так выглядит простейшая MPI-программа:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  printf("Здравствуй, Мир!\n");
  MPI_Finalize();
  return 0;
}
```

В отличие от классического HelloWorld, для его MPI-реализации необходимо подключить **mpi.h** с определениями констант, типов и функций MPI. Также в коде появились две спецфункции:

- MPI_Init – инициализация MPI;
- MPI_Finalize – обязательное корректное завершение работы с MPI.

В этой простой программе для лаконичности мы не обрабатываем возвращаемые ошибки. В реальной же программе ошибки игнорировать не следует. После успешного вызова MPI_Init (когда возвращается значение MPI_SUCCESS) инфраструктура MPI приложения готова к работе, и мы можем пользоваться всеми интерфейсными функциями MPI.

Компиляция MPI-программы делается с помощью стандартизованной утилиты *mpicc*, которая входит в состав пакета *mpi-default-dev*. Программа *mpicc* представляет собой обертку поверх компилятора. Она скрывает от пользователя отличия между разными платформами. Компиляция MPI-модификации HelloWorld делается так:

```
> mpicc helloworld.c -o helloworld
```

Если эту программу теперь запустить, то она сделает то, что от нее ожидается – а именно, скажет

```
> ./helloworld
Здравствуй, Мир!
```

Нам же нужно больше, поэтому, воспользовавшись утилитой *mpirun* из пакета *mpi-default-bin*, выполним:

```
> mpirun -n 2 ./helloworld
Здравствуй, Мир!
Здравствуй, Мир!
```

Ключик **-n** задает число параллельно исполняемых процессов. В нашем случае с миром поздоровались две копии программы. Также вместо *mpirun* можно использовать *mpiexec* или *orterun*. Это синонимы.

Немного усложним учебный код:

```
#include <mpi.h>
#include <stdio.h>
main(int argc, char **argv){
  int rankNode, sizeCluster;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rankNode);
  MPI_Comm_size(MPI_COMM_WORLD, &sizeCluster);
  printf("Здравствуй, Мир! от процесса %d из %d\n", rankNode, sizeCluster);
  MPI_Finalize();
}
```

Функция MPI_Comm_size возвращает количество запущенных для исполнения MPI-программы процессов, функция MPI_Comm_rank возвращает номер процесса, вызвавшего функцию:

```
> mpicc helloworld2.c -o helloworld2
> mpirun -n 2 ./helloworld2
Здравствуй, Мир! от процесса 1 из 2
Здравствуй, Мир! от процесса 0 из 2
```

Если запустить больше процессов –

```
> mpirun -n 7 ./helloworld2
Здравствуй, Мир! от процесса 0 из 7
Здравствуй, Мир! от процесса 1 из 7
Здравствуй, Мир! от процесса 2 из 7
Здравствуй, Мир! от процесса 5 из 7
Здравствуй, Мир! от процесса 6 из 7
Здравствуй, Мир! от процесса 3 из 7
Здравствуй, Мир! от процесса 4 из 7
```

можно увидеть, что выполняются они не обязательно в порядке запуска (последним выполнился процесс 4, а не процесс 6).

Это проявление весьма неудобного свойства параллельных программ, а именно недетерминированного результата исполнения. Оно может приводить к редко воспроизводимым и трудно отлаживаемым ошибкам в параллельных программах. Возникает ситуация, подобная той, когда измерение физических величин влияет на состояние измеряемой системы. Отладочный код может повлиять на ход исполнения программы, и искомая ошибка перестанет проявляться.

В тех случаях, когда нужно установить некоторый четкий порядок, можно использовать операции MPI для синхронизации процессов. Платой за это является увеличение времени исполнения программ, так как при запуске операции синхронизации часть процессов попадает в состояние ожидания ее завершения.

Следует также учитывать, что, например, на процессоре Intel(R) Core(TM)2 Duo, как следует из названия, только два вычислительных ядра, поэтому довольно бессмысленно запускать больше двух процессов одновременно. Можно также потребовать, чтобы каждый процесс привязался к своему ядру:

```
> mpiexec -n 2 -bind-to-core -report-bindings ./helloworld2
fork binding child [[7067,1],0] to cpus 0001
fork binding child [[7067,1],1] to cpus 0002
Здравствуй, Мир! от процесса 1 из 2
```

»

```
Здравствуй, Мир! от процесса 0 из 2
> mpiexec -n 7 -bind-to-core -report-bindings ./helloworld2
Not enough processors were found on the local host to meet the
requested binding action
```

Игра «Жизнь»

Клеточный автомат «Жизнь [Game of Life, или просто Life]» был предложен английским математиком Джоном Конвеем [John Horton Conway] в 1979 году. Вряд ли он стал бы так широко известен, если бы не американский математик и исключительно энергичный популяризатор науки Мартин Гарднер [Martin Gardner]. По игре «Жизнь» есть масса популярной литературы, в том числе и на русском языке.

Игра «Жизнь» проводится на бесконечной клеточной доске, где каждой из клеток присвоена 1 (живая клетка) или 0 (ничего нет). При переходе из нулевого в единичное состояние говорят, что клетка рождается, а из единичного в нулевое – умирает. Новое значение клетки определяется ее прежним значением и суммой значений ее восьми соседей. Клетка рождается при сумме три, а умирает при сумме менее двух – от одиночества и более трех – от перенаселенности. При всех иных конфигурациях значений и сумм клетка сохраняет свое состояние. Несмотря на простоту формулировки, эволюции разных комбинаций живых клеток могут весьма отличаться, и многие из них имеют свои имена. Например, на рисунке на следующей странице изображено глайдерное ружье – пример бесконечно растущей колонии.

Состояния всех клеток вместе образуют состояние всего клеточного автомата. Состояние клеточного автомата меняется по тактам, в дискретном времени. При смене состояния клеточного автомата состояния всех образующих его клеток меняются одновременно. Для каждой клетки новое состояние вычисляется по одинаковому для всех клеток правилу. Входные данные для правила – состояния самой клетки и ее восьми соседей на i-м такте. Результатом будет состояние этой клетки на (i+1)-м такте. Подобные алгоритмы замечательно параллелятся. Также интересна их особенность, состоящая в том, что время на одну итерацию не зависит от состояния автомата.

Следует понимать, что клеточный автомат – это абстрактная модель. В ней предполагается бесконечное число клеток. Реальные компьютеры обладают конечными ресурсами памяти и времени. Поэтому при моделировании клеточного автомата клетки размещаются в массиве конечных размеров. Для всех клеток, не лежащих на границах массива, работают правила моделируемого клеточного автомата без каких-либо изменений. Обработка же граничных клеток производится несколько иначе. Два наиболее распространенных способа обработки – замыкание граней (из прямоугольного массива получается трехмерный тор) и использование нулевых значений для тех соседних клеток из окрестности клетки на границе, которые лежат за пределами массива.

Последовательная программа для «Жизни» (без процедур задания начальных значений и печати или сохранения результата) выглядит примерно следующим образом:

```
#define QTY_STEPS  8192
#define SIZE_ARRAY 1024
char ar[2][SIZE_ARRAY][SIZE_ARRAY];
void computeCell(unsigned dir, unsigned x, unsigned y){
  unsigned sum;
  sum = ar[dir][y-1][x-1]+ar[dir][y-1][x]+ar[dir][y-1][x+1]+
    ar[dir][y][x-1]+ar[dir][y][x+1]+
    ar[dir][y+1][x-1]+ar[dir][y+1][x]+d ar[dir][y+1][x+1];
  if((sum<2)||(sum>3))
    ar[1-dir][y][x]=0;
  else
    ar[1-dir][y][x] = (sum==3) ? 1: ar[dir][y][x];
}
void simulateStep(unsigned layer){
  unsigned x,y;
  for(y=1;y<SIZE_ARRAY-1;y++)
  for(x=1;x<SIZE_ARRAY-1;x++)
    computeCell(layer,x,y);
}
void simulate(unsigned stepQty){
  unsigned step;
  for(step=0;step<stepQty;step++){
    simulateStep(step & 1);
  }
}
main(){
  simulate(QTY_STEPS);
}
```

Классификация параллельных компьютеров

Современные параллельные компьютеры можно разделить на: параллельные векторные системы (PVP), симметричные мультипроцессорные системы (SMP), массивно-параллельные системы (MPP), системы с неоднородным доступом к памяти (NUMA) и кластеры.

Примером параллельной векторной системы (PVP) является первый суперкомпьютер CRAY-I, который был создан в 1975 году. Одно время понятия «параллельные векторные системы» и «суперкомпьютеры» были синонимами, но сейчас в TOP500 (http://www.top500.org/) классических PVP не осталось совсем.

Симметричные мультипроцессорные системы (SMP) содержат несколько одинаковых процессоров и общую память. Все процессоры имеют одинаковую скорость доступа к общей памяти. Имея последовательную реализацию некоторой программы, как правило, можно относительно легко построить ее параллельную реализацию для SMP. Основной недостаток SMP – плохая масштабируемость. Существуют доступные SMP-системы с десятками процессоров, но в списке TOP500 нет ни одной из них.

Компьютеры класса MPP (массивно-параллельные системы) состоят из узлов, объединенных высокоскоростным коммутатором. Узел содержит один или несколько процессоров и память. Узел не имеет доступа к памяти других узлов. Такие системы, в отличие от SMP, хорошо масштабируются. Число узлов в самых больших MPP-системах достигает сотен тысяч. За это приходится платить, так как параллельную программу для MPP-системы, в отличие от SMP, построить сложнее. Процессы, выполняющиеся на разных узлах, не имеют общей памяти, и все взаимодействие между ними делается с помощью посылки сообщений. На ноябрь 2012 г. в TOP500 находится 89 классических MPP-суперкомпьютеров, включая первые две позиции.

Системы с неоднородным доступом к памяти (NUMA) являются гибридом SMP и MPP. Физически вся память в NUMA распределена между узлами, как в MPP. Но при этом поддерживается единое пространство памяти, как в SMP. То есть из узла можно обратиться к памяти в любом другом узле. Время доступа к своей, локальной памяти меньше, чем к памяти другого узла.

Кластер – это дешевый аналог MPP-системы. В нем вместо скоростного коммутатора используется обычая сеть; таковые становятся все быстрее и быстрее. Если MPP пока еще выделяются мощью, то кластеры берут массовостью, ибо 411 суперкомпьютеров, остающиеся после вычитания MPP из TOP500, являются кластерными решениями.

» **Пропустили номер?** Узнайте на с. 108, как получить его прямо сейчас.

Функция ComputeCell вычисляет новое состояние одной клетки. Функция SimulateStep вычисляет новое состояние всего клеточного автомата, а Simulate имитирует работу клеточного автомата указанное число шагов.

Обратите внимание, что в массиве ar введено 2 слоя. Это требуется для того, чтобы имитировать одновременность задания новых значений клеток. Чтобы исключить копирование одного слоя в другой, на четном шаге текущие значения клеток берутся из нулевого слоя, новые пишутся в первый; а на нечетных шагах, наоборот, текущие значения берутся из первого слоя, новые пишутся в нулевой.

Для MPI-версии программы возьмем за основу последовательную реализацию. Для простоты ограничимся двумя процессами. Разделим массив, хранящий состояния клеток, поровну.

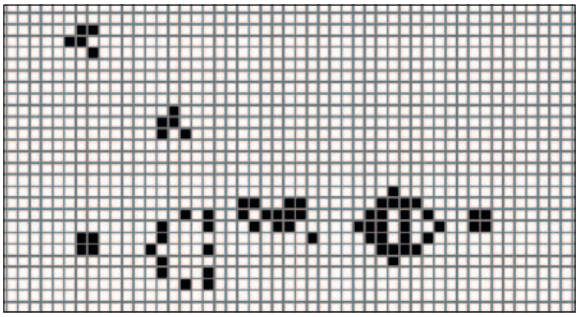
Разделение произведем по строкам. Пусть первая половина строк хранится в массиве ar первого процесса (rankNode равен нулю), вторая – в этом же массиве ar у второго процесса (rankNode равен 1).

При подсчете новых состояний клеток последней строки в нулевом процессе требуются текущие состояния трех клеток из строки снизу. Значения клеток этой строки подсчитываются в первом процессе. Аналогично, для первого процесса нужна строка сверху. Этой информацией о состоянии клеток на границе необходимо обменяться. Это единственное отличии MPI-программы от обычной последовательной. При росте числа одновременно выполняемых процессов также растет и число граничных элементов.

Строки с клетками, подсчитываемыми в одном процессе, но используемыми и в каком-либо другом процессе, называются теневыми гранями.

Функция ComputeCell не меняется, так что ниже мы ее не дублируем:

```
#define QTY_STEPS  8192
#define SIZE_ARRAY 1024
#define SIZE_HALFARRAY (SIZE_ARRAY/2+1)
char ar[2][SIZE_HALFARRAY][SIZE_ARRAY];
#define RES_OK         0
#define RES_ERROR      1
int mytag=99;
void simulateStep(unsigned layer){
  unsigned x,y;
  for(y=1;y<SIZE_HALFARRAY-1;y++)
  for(x=1;x<SIZE_ARRAY-1;x++)
  computeCell(layer,x,y);
}
void simulate(unsigned stepQty,int rankNode){
  unsigned step;
  MPI_Status status;
  for(step=0;step<stepQty;step++){
    simulateStep(step & 1);
    if(rankNode==0){
      MPI_Send(ar[1 - step & 1][SIZE_ARRAY/2 - 1], SIZE_ARRAY,
MPI_CHAR, rankNode + 1,  mytag, MPI_COMM_WORLD);
    MPI_Recv(ar[1 - step & 1][SIZE_ARRAY/2], SIZE_ARRAY, MPI_
CHAR, rankNode + 1, mytag,MPI_COMM_WORLD, &status);
    }
    else if(rankNode==1){
      MPI_Recv(ar[1 - step & 1][0], SIZE_ARRAY, MPI_CHAR,
rankNode - 1, mytag, MPI_COMM_WORLD, &status);
      MPI_Send(ar[1 - step & 1][1], SIZE_ARRAY, MPI_CHAR,
rankNode - 1, mytag, MPI_COMM_WORLD);
    }
  }
}
main(int argc, char **argv){
```



› Глайдерное ружье.

```
  int rankNode, sizeCluster;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rankNode);
  MPI_Comm_size(MPI_COMM_WORLD, &sizeCluster);
  if (sizeCluster!=2) {
    printf ("Только для двух ядер!");
    exit(1);
  }
  simulate(QTY_STEPS, rankNode);
  MPI_Finalize();
}
```

А вот оператор цикла в SimulateStep уже отличается от того, что был в последовательной версии, как и размер массива ar. Функция main уже содержит в себе команды инициализации и завершения MPI.

Наиболее сильным изменениям подверглась функция simulate. Теперь после каждого шага требуется передавать значения клеток всех теневых граней между соседними узлами.

Эту задачу выполняют операторы передачи MPI_Send и приема MPI_Recv. Они должны быть комплементарными, то есть каждый оператор передачи данных в одном процессе обязательно обязан иметь соответствующий ему оператор приема в другом процессе. В противном случае произойдет ненормальное завершение работы MPI программы. Подробности об использовании этих команд и описание их аргументов можно посмотреть с помощью команды man.

Собственно говоря, все. Кажется, ничего сложного, но если хочется увеличить число параллельных процессов, то придется аккуратно обрабатывать больше границ и следить за согласованием приема-передачи информации, то есть придется думать. Это и есть основная сложность! Но зато MPI-программа на двух процессах выполняется примерно за 110 секунд вместо 180 на сферическом в вакууме Intel(R) Core(TM)2 Duo одновременно с набором этого текста. Ускорение не в двойку, но достаточно значительное, чтобы за него побороться.

На примере программы моделирования игры «Жизнь» мы рассмотрели самые базовые возможности MPI. Из сотен функций, описанных в текущем стандарте MPI 3.0, мы использовали только шесть. Стандарт ориентирован на комфортную работу при распараллеливании широкого спектра задач, и для этого в нем присутствуют разнообразные группы функций для коммуникаций между двумя процессами, коммуникаций внутри группы процессов, работы с группами процессов, кэширования данных и многого другого. LXF

Обратная связь

Приглашаем высказаться потенциальных авторов статей по параллельным вычислениям – ценные предложения, критику и советы присылайте по электронной почте: E.M.Baldin@inp.nsk.su, ostap@ssd.sscc.ru.