Демьянович Юрий Казимирович

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ УРОК 3. О ПРОБЛЕМАХ РАСПАРАЛЛЕЛИВАНИЯ

Очередной урок по параллельным вычислениям начнем с обзора содержания предыдущих уроков (см. [1, 2]).

Прежде всего напомним, что термин «параллельные вычисления» применяется для одновременно проводимых вычислительных процессов с целью решения одной (часто суперсложной) задачи на параллельной вычислительной системе (ВС); процессы эти, вообще говоря, различны, четко отделены друг от друга, но в определенные моменты времени обмениваются необходимой информацией. Цель распараллеливания – ускорить решение суперсложных задач. Примерами таких задач служат задачи прогноза погоды, климата, землетрясений, цунами, ураганов, состояния оружия (особенно ядерного, так как его испытания запрещены), состояния мостов и других сооружений и т. д. К числу таких задач относятся отыскание эффективных лекарств, расшифровка генома человека, построение единой теории мироздания в физике и т. п. Как правило, такие задачи сводятся к сложным системам дифференциальных уравнений с соответствующими граничными и начальными условиями. Для отыскания численного решения этих задач требуется решить много систем линейных алгебраических уравнений с огромным количеством неизвестных (с сотней тысяч, а иногда и с миллионом неизвестных).

Параллельная ВС состоит из многих вычислительных модулей (процессоров, ядер), работающих параллельно; таких модулей в ВС может быть больше ста тысяч. Их работа должна быть синхронизирована: время от времени они обмениваются информацией. Ясно, что написать вручную программу работы каждого из них практически невозможно. Поэтому параллельная ВС в обязательном порядке снабжается хотя бы одним из средств автоматического распараллеливания. Однако упомянутые средства не могут действовать без подсказок программиста – специальных директив (команд), вставляемых в программу. С помощью этих директив программист решает основные задачи распараллеливания:

1) указывает группы подзадач (или операций), которые не зависят друг от друга и, следовательно, могут выполняться параллельно;

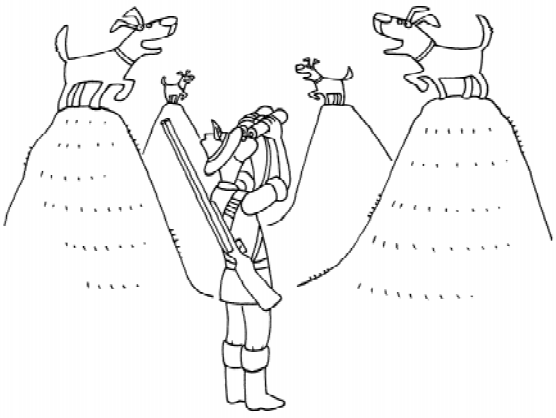
2) указывает последовательность выполнения этих групп.

Таким образом, в первую очередь средства распараллеливания должны позволять эффективно решать упомянутые задачи 1)

Цель распараллеливания — ускорить решение суперсложных задач. ... прогноза погоды...

и 2). Кроме того, желательна простота использования этих средств, применимость к уже имеющимся последовательным программам, согласованность с языками высокого уровня, а также возможность создавать программы, не зависящие от используемой параллельной системы. Последнее означает, что средство распараллеливания должно быть стандартным (одинаковым) для различных параллельных систем. В настоящее время приняты стандарты MPI, Open MP, DVM и некоторые другие. Первое из этих стандартных средств, а именно MPI (Message Passing Interface) мы начали изучать в предыдущей лекции (см. [2]). Здесь мы продолжим это изучение, но сначала остановимся на основных проблемах распараллеливания.

Исторически сложилось так, что при численном решении задач пользователи пишут программы на последовательном языке. Если оказалось, что без распараллеливания та или иная программа работает не эффективно, то ее можно попытаться распараллелить, с тем чтобы запустить задачу на параллельной системе. Кроме того, для многих пользователей психологически удобнее программирование начинать с последовательной программы. Поэтому естественно задаться вопросом о возможности автоматического распараллеливания программ транслятором. Однако решение задачи распараллеливания данной программы достаточно сложно.



Их работа должна быть синхронизирована: время от времени они обмениваются информацией...

Для распараллеливания программы требуется:

1) найти участки программы, где распараллеливание возможно,

2) распределить эти участки по вычислительным модулям,

3) обеспечить вычисления правильной синхронизацией.

Особенно трудно выполнить эти требования в случае системы с распределенной памятью. Но даже для системы с общей памятью это достаточно сложно сделать автоматически.

Об этом свидетельствует следующий простой пример (см. [3, 4]).

Задача 1. *Пусть имеется фрагмент программы (назовем его фрагмент (A)), состоящий из трех строк:*

```
C  программный фрагмент (A)
     DO  10  i=1,n
      DO  10  j=1,n
10      U(i+j)=U(2*n-i-j+1)
```

Вопрос заключается в следующем: какие итерации в этой конструкции независимы и можно ли ее распараллелить?

Попытаемся заменить фрагмент (A) фрагментом (B) вида

```
C  программный фрагмент (B)
   DO  10  i=1,n
    DO  20   j=1,n-i
20    U(i+j)=U(2*n-i-j+1)
    DO  30 j=n-i+1,n
30    U(i+j)=U(2*n-i-j+1)
10 continue
```

считая, что внутренние циклы 20 и 30 исполняются на двух параллельно работающих вычислительных модулях.

Можно ли рассматривать эту программу как результат распараллеливания внутреннего цикла в предыдущей программе?

Для ответа на поставленный вопрос проведем исследование предлагаемых фрагментов моделированием их исполнения. Ограничимся случаем $n = 3$, предполагая, что массив U(6) ранее был инициализирован в рассматриваемой программе в соответствии со следующей схемой (многоточие означает, что значения соответствующих элементов массива для дальнейшего несущественны):

U	c	...	b1	b	a	...
	1	2	3	4	5	6

1. Сначала рассмотрим работу программного фрагмента (A), демонстрируя результат работы каждой итерации внешнего цикла в виде схемы заполнения массива U.

```
n=3       U(i+j)=U(7-i-j)
  i=1     U(1+j)=U(6-1-j+1)=U(6-j)
    j=1       U(2)=U(5)
    j=2       U(3)=U(4)
    j=3       U(4)=U(3)
```

U	c	a	b	b	a	...
	1	2	3	4	5	6

```
i=2     U(2+j)=U(6-2-j+1)=U(5-j)
  j=1       U(3)=U(4)
  j=2       U(4)=U(3)
  j=3       U(5)=U(2)
```

U	c	a	b	b	a	...
	1	2	3	4	5	6

```
i=3     U(3+j)=U(6-3-j+1)=U(4-j)
  j=1       U(4)=U(3)
  j=2       U(5)=U(2)
  j=3       U(6)=U(1)
```

U	c	a	b	b	a	c
	1	2	3	4	5	6

2. Переходя к демонстрации работы фрагмента (B), повторим его для удобства дальнейших рассмотрений:

```
C   программный фрагмент (B)
    DO  10  i=1,n
     DO   20  j=1,n-i
20     U(i+j)=U(2*n-i-j+1)
     DO   30 j=n-i+1,n
30     U(i+j)=U(2*n-i-j+1)
10  continue
```

Исходя из первоначального состояния массива U

U	c	...	b1	b	a	...
	1	2	3	4	5	6

при работе фрагмента (B) получаем последовательно

```
n=3         U(i+j)=U(7-i-j)
  i=1     U(1+j)=U(6-j)
C         цикл 20  j=1,2
    j=1       U(2)=U(5)
    j=2       U(3)=U(4)
C         цикл 30  j=3
    j=3       U(4)=U(3)
```

U	c	a	b	b	a	...
	1	2	3	4	5	6

```
  i=2       U(2+j)=U(5-j)
C           цикл 20  j=1
    j=1       U(3)=U(4)
C           цикл 30  j=2,3
    j=2       U(4)=U(3)
    j=3       U(5)=U(2)
```

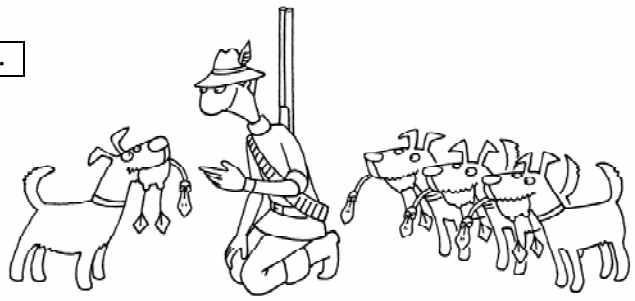
U	c	a	b	b	a	...
	1	2	3	4	5	6

```
  i=3   U(3+j)=U(4-j)
C        цикл 20  j=1,0 ==> пропуск-ся
C       цикл 30  j=1,3
    j=1     U(4)=U(3)
    j=2     U(5)=U(2)
    j=3     U(6)=U(1)
```

U	c	a	b	b	a	c
	1	2	3	4	5	6

На первый взгляд кажется, что результаты работы программного фрагмента (A) и результатов его распараллеливания идентичны.

Заметим, однако, что в данном случае рассмотрен лишь один из восьми возможных при распараллеливании вариантов работы параллельной системы: порядок исполнения внутренних циклов 20 и 30 параллельными модулями предсказать нельзя (этот порядок зависит от работы оборудования и распределения работы имеющегося программного окружения), поэтому для вывода о возможности замены фрагмента (A) распараллеливанием фрагмента (B), необходимо убедиться в совпадении результатов работы каждого из возможных при распараллеливании вариантов обработки фрагмента (B).



На первый взгляд кажется, что результаты работы программного фрагмента (A) и результатов его распараллеливания идентичны.

3. Следующий вариант обработки (сводящийся к перестановке порядка исполнения внутренних циклов 20 и 30 параллельными модулями лишь на первой итерации внешнего цикла) дает результат, отличный от результата работы фрагмента (A). Действительно, обрабатывая исходный массив U,

U	c	...	b1	b	a	...
	1	2	3	4	5	6

в этом случае имеем

```
n=3          U(i+j)=U(7-i-j)
  i=1      U(1+j)=U(6-j)
C          цикл 30 j=3
     j=3       U(4)=U(3)
C          цикл 20 j=1,2
     j=1       U(2)=U(5)
     j=2       U(3)=U(4)
```

U	c	a	b1	b1	a	...
	1	2	3	4	5	6

```
  i=2      U(2+j)=U(5-j)
C          цикл 20  j=1
      j=1      U(3)=U(4)
C          цикл 30  j=2,3
      j=2      U(4)=U(3)
      j=3      U(5)=U(2)
```

U	c	a	b1	b1	a	...
	1	2	3	4	5	6

```
  i=3    U(3+j)=U(4-j)
C          цикл 20  j=1,0 ==> пропуск-ся
C          цикл 30  j=1,3
      j=1     U(4)=U(3)
      j=2     U(5)=U(2)
      j=3     U(6)=U(1)
```

U	c	a	b1	b1	a	c
	1	2	3	4	5	6

Итак, полученный здесь результат отличается от результата работы исходного фрагмента (A). Это показывает невозможность замены фрагмента (A) указанным выше распараллеливанием фрагмента (B).

Вывод: Результат работы фрагмента (B) зависит от последовательности срабатывания вложенных циклов, поэтому без использования дополнительных средств синхронизации вычислений предложенный вариант распараллеливания применять нельзя.

Пример 2. Рассмотрим следующий фрагмент программы:

```
     DO  10 i=1,n
10     U(i)=Funct(i)
```

где Funct – функция пользователя. Для того чтобы ответить на вопрос, являются ли итерации цикла независимыми, нужно определить:

1) используются ли значения массива U в теле функции Funct и если нет, то:

2) нет ли там вызовов процедур или функций, прямо или косвенно использующих массив U.

В некоторых случаях такая проверка исключительно сложна, особенно в случаях, когда разные части массива используются в разных аспектах: одни для чтения, другие для записи. В таких случаях ответить на поставленный вопрос возможно лишь в процессе исполнения.

Для эффективного анализа ситуации транслятору (компилятору) нужны «подсказки», которые могут выражаться различным образом:

1) специальными директивами,

2) новыми языковыми конструкциями,

3) специальными библиотечными процедурами.

Для переносимости создаваемых программ с одной вычислительной системы на другую программные средства должны быть стандартными. В связи с этим напомним, что

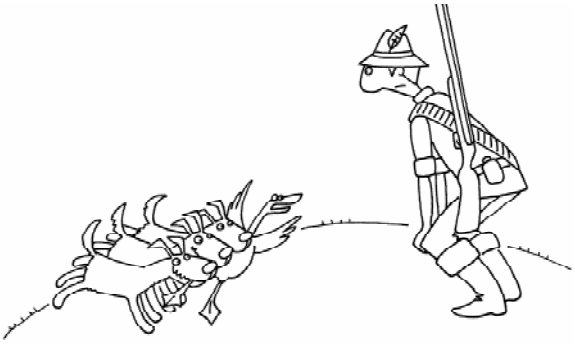
Листинг 1. Простой пример программы на языке Fortran, использующей стандарт MPI

```
envelope.f
1)  PROGRAMM envelope
2)  INCLUDE 'mpif.h'
3)  CALL MPI_INIT(ierr)
4)  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
5)  CALL MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
6)  PRINT *,'nprocs =',nprocs,'myrank =',myrank
7)  CALL MPI_FINALIZE(ierr)
8)  END
```

к настоящему времени разработаны стандарты на языки программирования высокого уровня, которые позволяют достаточно быстро освоить эффективное использование компьютера и поддерживают переносимость создаваемых программ. Разработаны стандарты на элементную базу, предназначенную для создания компьютеров. Аналогичным образом создаются стандарты на параллельные компьютерные системы и на параллельное программирование. Одним из наиболее распространенных стандартов параллельного программирования является стандарт MPI, которого мы касались в предыдущей лекции, а теперь продолжим его изучение.

Приведем рассмотренный ранее пример простой программы (листинг 1) (подробные комментарии к этой программе даны в упомянутой лекции, см. [2]).

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, то возможно сделать поведение различных процессов различным и наблюдать за их работой. Например, если после трансляции программы envelope (см. листинг 1) получается загрузочный модуль a.out, то после запуска последнего получим на (общем) принтере распечатку (листинг 2).



Это показывает невозможность замены фрагмента (A) указанным выше распараллеливанием

Рассмотрим теперь несколько более сложную программу (листинг 3).

Листинг 2. Распечатка результатов работы параллельной программы envelope, представленной на листинге 1

```
0: nprocs = 3 myrank = 0
1: nprocs = 3 myrank = 1
2: nprocs = 3 myrank = 2
```

Листинг 3. Пример программы на языке Фортран, использующей процедуру MPI_BCAST стандарта MPI (заметим, что символ $ символ продолжения строки программы)

```
1)  PROGRAMM bcast
2)  INCLUDE 'mpif.h'
3)  INTEGER imsg(4)
4)  CALL MPI_INIT(ierr)
5)  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
6)  CALL MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
7)  IF (myrank=0) THEN
8)   DO i=1,4
9)     imsg(i)=i
10)   END DO
11) ELSE
12)  DO i=1,4
13)    imsg(i)=0
14)  END DO
15) ENDIF
16) PRINT *,'Before:',imsg
17) CALL MPI_BCAST(imsg,4,MPI_GATHER,
18) $0,MPI_COMM_WORLD,ierr)
19) PRINT *,'after:',imsg
20) CALL MPI_FINALIZE(ierr)
21) END
```

Листинг 4. Распечатка результатов работы параллельной программы, представленной на листинге 3

```
0: Before: 1 2 3 4
1: Before: 0 0 0 0
2: Before: 0 0 0 0
0: after: 1 2 3 4
1: after: 1 2 3 4
2: after: 1 2 3 4
```

В соответсвии с принятой в MPI концепцией, процессы в MPI, вообще говоря, равноправны (нет заранее выделенного процесса): программист вправе считать любой из них основным. В программе bcast.f процесс с рангом rank=0 назовем корневым (ибо он в определенном отношении отличается от остальных: именно в нем будет получен окончательный результат). Корневой процесс заполняет целочисленный массив imsg ненулевыми значениями, в то время как остальные процессы заполняют его нулями. Процедура MPI_BCAST вызывается в строках 17, 18, при этом она рассылает четыре числа из корневого процесса в другие процессы коммуникатора MPI_COMM_WORLD. Заметим, что роль идентификатора imsg в корневом процессе и в некорневых различна: в корневом процессе он используется как обозначение посылающего буфера, а в некорневых процессах – как обозначение получающего буфера.

Результат работы только что приведенной программы будет следующим (листинг 4).

Возникает вопрос о том, как программировать вычисления на параллельной системе.

Для проведения вычислений с использованием MPI предназначена процедура MPI_REDUCE. Иллюстрацией ее применения для вычисления суммы элементов массива a(i) служит следующая программа (листинг 5).

В этой программе предполагается, что имеются три процесса, вовлеченные в вычисления, и каждому процессу поручается

Листинг 5. Пример программы на языке Фортран, использующей процедуру MPI_REDUCE стандарта MPI и вычисляющей сумму элементов массива a(i), i=1,2,: : :,9.

```
reduce.f
1)    PROGRAMM reduce
2)    INCLUDE 'mpif.h'
3)    REAL a(9)
4)    CALL MPI_INIT(ierr)
5)    CALL MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
6)    CALL MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
7)    ista = myrank*3+1
8)    iend = ista +2
9)    DO i = ista, iend
10)    a(i) = i
11)   END DO
12)   sum = 0.0
13)   DO i = ista, iend
14)    sum = sum + a(i)
15)   END DO
16)   CALL MPI_REDUCE(sum,tmp,1,MPI_REAL,MPI_SUM,0,
    $MPI_COMM_WORLD,ierr)
17)   sum=tmp
18)   IF (myrank=0) THEN
19)   PRINT *, 'sum=', sum
20)   ENDIF
21)   CALL MPI_FINALIZE(ierr)
22)   END
```

одна треть массива a(i). Каждый процесс вычисляет частичные суммы в строках 13–15, а в строках 16–17 эти частичные суммы складываются, и результат посылается в корневой процесс (в данном случае таковым является процесс с номером 0). Вместо девяти сложений, получается три сложения плюс одна глобальная сумма. В этом случае пятый аргумент в MPI_REDUCE указывает на характер проводимых операций, а четвертый – на тип данных.

Замечание 2. Поскольку порядок вычислений может быть изменен по сравнению с возможной последовательной программой, то из-за ошибок округления результат может быть различным на различных ВС.

Следующая лекция будет посвящена дальнейшему углублению знаний о распараллеливании программ, а сейчас предлагается ответить на следующие вопросы.

1. Какие задачи приводят к суперсложным вычислениям?
2. В чем состоят основные проблемы распаралелливания?
3. Зачем нужны программные средства распараллеливания MPI, Open MP и другие?
4. Каким требованиям должны удовлетворять средства распараллеливания?
5. Почему средства распараллеливания должны быть стандартными?
6. Какой смысл распараллеливать последовательную программу?
7. Сколько параллельных вычислительных модулей может иметь параллельня ВС?
8. Какие основные задачи решает программист с помощью директив распараллеливания?
9. С чего начинается распараллеливание последовательной программы?
10. Какому основному условию должны удовлетворять итерации цикла, для того чтобы цикл можно было распараллелить?
11. Какова роль оборудования параллельной ВС при автоматизированном распараллеливании?
12. Что требуется транслятору (компилятору) для эффективного распараллеливания?
13. Какие средства предоставляются пользователю (программисту, предметному специалисту) для «подсказок» транслятору?
14. Пишет ли пользователь MPI-программу для каждого вычислительного модуля параллельной ВС отдельно или он пишет одну MPI-программу для всех модулей?
15. Каким образом в MPI-программе предусматриваются различные варианты работы для различных модулей параллельной ВС?
16. Является ли единственно возможным вариант распечатки, приведенный на листинге 2?
17. Имеется ли равноправие между процессами в концепции MPI?
18. Может ли пользователь считать любой из процессов основным?
19. Можно ли предсказать порядок суммирования в программе reduce (см. листинг 5)?
20. В чем состоит работа процедур MPI_BCAST и MPI_REDUCE?

Литература

1. *Демьянович Ю.К.* Параллельные вычисления. Урок 1. Параллельная форма // Компьютерные инструменты в школе, 2011. № 1. С. 36–39.

2. *Демьянович Ю.К.* Параллельные вычисления. Урок 2. О средствах распараллеливания... Элементарная параллельная программа // Компьютерные инструменты в школе, 2011. № 2. С. 17–21.

3. *Воеводин В.В., Воеводин Вл.В.* Параллельные вычисления. СПб: БХВ-Петербург, 2002. 608 с.

4. *Бурова И.Г., Демьянович Ю.К.* Алгоритмы параллельных вычислений и программирование. Курс лекций. СПб: Изд-во СПб. ун-та, 2007. 200 с.

Демьянович Юрий Казимирович, доктор физико-математических наук, профессор, заведующий кафедрой параллельных алгоритмов математико-механического факультета СПбГУ.