А.П.Сапожников

Вокруг решета Эратосфена, или еще один опыт распараллеливания программ

АННОТАЦИЯ. Описывается программа Л.Караилиева, реализующая классический алгоритм т.н. решета Эратосфена для генерации простых чисел. Авторский интерфейс был модифицирован А.П.Сапожниковым для упрощения возможности использования нескольких процессоров в рамках технологии MPI. Исследована производительность этой программы. Показано, что она близка к оптимальной величине O(N*ln(N)). Приведены все тексты на языке Fortran.

ANNOTATION. Program of L.Karailiev for prime numbers generation is described. It uses a classical algorithm of Eratosthenes sieve. The author's interface was modified by A.P.Sapozhnikov to simplify the usage of several processors under MPI-technology. The productivity of program is explored. It proved to be near to its optimal value of O(N*ln(N)). All corresponding Fortran-sources are given.

1. Lumomir Alexandrov, D. B. Baranov, Plamen Yotov (JINR, BLTP, Dubna, Russia). Polynomial splines interpolating prime series // JINR-P5-2002-228. http://arxiv.org/abs/math/0212246v5

В работе /1/ опубликована программа, реализующая классический генератор простых чисел, то есть целых чисел, нацело делящихся только на 1 и самих себя. Вот эта программа:

```
   subroutine eratosthenes(n0,N,nprimes)
!
!  Lubomir Alexandrov Karailiev.  BLTP JINR. 1996
!  routine generates all prime numbers among [n0,N] into nprimes(1,...)
!  nprimes(0) will contain the quantity of prime numbers generated
!
      implicit real*8(a-h,o-z)
      dimension nprimes(0:N)
      np=1  ; nprimes(1)=2                    ! first prime=2
       if(n0.gt.2)  np=0                      ! if not from the beginning!
      n1=max(n0,3); if(mod(n1,2) == 0) n1=n1+1
      do nn=n1,N,2                            ! test odd numbers only!
        id=1
        isqrtn=dsqrt(dfloat(nn))
  1     id=id+1
        if(mod(nn,id) == 0) cycle             ! Eratosthenes sieve
        if(id < isqrtn) goto 1
         np=np+1; nprimes(np)=nn
      enddo
       nprimes(0)=np; return
      end subroutine eratosthenes
```

Как видно уже из ее названия, программа реализует классический алгоритм Эратосфена, в просторечии именуемый эратосфеновым решетом.

В этой программе мы позволили себе расширить авторский интерфейс, введя вместо одного параметра N диапазон [N0,N], что позволило легко использовать ее при параллельной работе нескольких процессов в рамках единой задачи.

Представляет интерес, во-первых, измерить производительность P этой программы на предмет соответствия известной из теории оптимальной оценке

Popt=N*O(ln(N)). /*/

Во-вторых, попытаться увеличить эту производительность за счет использования нескольких процессоров в рамках одной задачи, то есть распараллеливания.

Вот таблица времени вычислений T(sec.) при n0=1 и разных N:

I	1	2	3	4	5
N	1000000	2000000	4000000	8000000	16000000
T	7.7	21.5	61.0	181.5	558.8
T(i+1)/T(i)	-	2.7	2.8	3.0	3.1

Это отношение растет достаточно медленно. Сравните его с изменением значения функции F(N)=2N*ln(2N)/(N*ln(N)) при последовательном удвоении N:

F(N): 2.086 2.082 2.079 2.076 … 2.070 2.067

Здесь вдумчивый читатель вправе задать вопрос: F(N) медленно и монотонно убывает, сходясь в пределе к 2, а почему же P(N)=T(2N)/T(N) возрастает? Постараемся ответить.
Оптимальная оценка /*/, а следовательно, и функция F(N) – это строго определенная математическая функция, в то время, как T(i) – это результаты замеров времени счета ПРОГРАММЫ. В частности, в это время входит и время так называемой подкачки памяти, объем которой монотонно возрастает с ростом N.
Мы не поленились сделать тот же замер счетного времени того же теста, убрав из текста eratosthenes команды np=np+1 и nprimes(np)=nn записи получаемого простого числа в память, тем самым исключив влияние подкачки, и при больших N получили на 1-2% меньшие времена T, а следовательно и почти постоянное P(N) ~ 3.0 !

К тому же проведенные измерения показали, что производительность программы eratosthenes близка к теоретически оптимальной /*/.

Для распараллеливания мы написали программу Primus, которая по существу является надстройкой над eratosthenes, позволяющей задействовать заданное количество NP процессов. Распараллеливание в ней свелось к простому делению отрезка [2,N] числовой оси между процессами приблизительно поровну с последующим объединением полученных результатов в памяти главного процесса.

Для измерения мы использовали следующий тест:

```
Program Esieve
Data n/500000/, krep/5/
dimension np(0:10000000)
 n=500000
 do i=1,krep
   n=2*n
   call profile_start
   call primus(1,n,np)
   call profile_fin
```

```
      enddo
      end
```

Здесь Profile (http://www.jinr.ru/programs/jinrlib/profile/) – наш инструмент для засечки времени счета исследуемой программы. Приведем только выдачу этой тестовой программы при использовании единственного процесса:

```
  Time  0 =     0.4 sec.             /1/
 WallTime:       0.4  Total CPU_time:      0.5 sec.
  Time  0 =     0.8 sec.
 WallTime:       0.8  Total CPU_time:      1.3 sec.
  Time  0 =     2.2 sec.
 WallTime:       2.2  Total CPU_time:      3.5 sec.
  Time  0 =     5.6 sec.
 WallTime:       5.6  Total CPU_time:      9.0 sec.
  Time  0 =    14.5 sec.
 WallTime:      14.5  Total CPU_time:     23.6 sec.
```

При запуске “на 2 процесса” (mpirun –np 2 esieve.exe) получим следующую картину:

```
  Time  0 =     0.1 sec.             /2/
  Time  1 =     0.2 sec.
 WallTime:       0.2  Total CPU_time:      0.4 sec.
  Time  0 =     0.4 sec.
  Time  1 =     0.8 sec.
 WallTime:       0.5  Total CPU_time:      1.3 sec.
  Time  0 =     0.9 sec.
  Time  1 =     2.1 sec.
 WallTime:       1.4  Total CPU_time:      3.6 sec.
  Time  0 =     2.3 sec.
  Time  1 =     5.7 sec.
 WallTime:       3.6  Total CPU_time:      9.5 sec.
  Time  0 =     6.0 sec.
  Time  1 =    15.1 sec.
 WallTime:       9.4  Total CPU_time:     24.8 sec.
```

Для трех процессов (mpirun –np 3 esieve.exe) эта же выдача выглядит так:

```
  Time  0 =     0.1 sec.             /3/
  Time  1 =     0.1 sec.
  Time  2 =     0.2 sec.
 WallTime:       0.1  Total CPU_time:      0.4 sec.
  Time  0 =     0.2 sec.
  Time  1 =     0.5 sec.
  Time  2 =     0.5 sec.
 WallTime:       0.6  Total CPU_time:      1.4 sec.
  Time  0 =     0.5 sec.
  Time  1 =     1.2 sec.
  Time  2 =     1.5 sec.
 WallTime:       1.3  Total CPU_time:      3.7 sec.
  Time  0 =     1.3 sec.
  Time  1 =     3.3 sec.
  Time  2 =     4.0 sec.
 WallTime:       3.3  Total CPU_time:      9.6 sec.
  Time  0 =     3.4 sec.
```

```
  Time  1 =     8.8 sec.
  Time  2 =    10.5 sec.
 WallTime:       8.5  Total CPU_time:     24.9 sec.
```

А для четырех (mpirun –np 4 esieve.exe):

```
  Time  0 =     0.1 sec.             /4/
  Time  1 =     0.1 sec.
  Time  2 =     0.1 sec.
  Time  3 =     0.1 sec.
 WallTime:       0.2  Total CPU_time:      0.4 sec.
  Time  0 =     0.1 sec.
  Time  1 =     0.3 sec.
  Time  2 =     0.4 sec.
  Time  3 =     0.4 sec.
 WallTime:       0.4  Total CPU_time:      1.4 sec.
  Time  0 =     0.3 sec.
  Time  1 =     0.9 sec.
  Time  2 =     1.0 sec.
  Time  3 =     1.2 sec.
 WallTime:       1.1  Total CPU_time:      3.7 sec.
  Time  0 =     0.9 sec.
  Time  1 =     2.3 sec.
  Time  2 =     2.7 sec.
  Time  3 =     3.1 sec.
 WallTime:       3.2  Total CPU_time:      9.6 sec.
  Time  0 =     2.3 sec.
  Time  1 =     5.9 sec.
  Time  2 =     7.2 sec.
  Time  3 =     8.1 sec.
 WallTime:       8.1  Total CPU_time:     25.0 sec.
```

Мы уверены, что вдумчивый читатель уже понял, что в тесте нами использовался 2-процессорный компьютер!

Сопоставив теперь WallTiimes в /1/ и /2-4/, видим, что время решения задачи сократилось с 14 до 8-9 секунд, т.е. в 1.5 раза! Таким образом, проведенное распараллеливание можно считать успешным, т.е. полезным.

Наконец, для полноты картины приведем текст программы Primus, который хорошо иллюстрирует использование MPI-технологии, каковую мы уже давно и неустанно пропагандируем среди людей, занимающихся программированием:

```
      subroutine primus(n0,n,nprimes)  ! Параллельная версия eratosthenes
!
!  А.П.Сапожников.  ЛИТ ОИЯИ. 2013
!
      implicit real*8(a-h,o-z)
      dimension nprimes(0:n)
       Include 'mpif.h'
       integer st(MPI_Status_Size)
      logical iflag
      common/idents/idcomm,NProc,myProc

      call MPI_Initialized(iflag,ierr)        ! is MPI already initialized ?
       if(.not.iflag) call MPI_Init(ierr)      ! No - do initialization
       idcomm=MPI_COMM_WORLD
```

```
       master=0
      call MPI_Comm_Size(idcomm, NProc, ierr) ! How many processes we have?
      call MPI_Comm_Rank(idcomm, myProc, ierr)! Who am I ?

      nstart=iabs(n0)
      k=(n-nstart)/NProc; n1=nstart; n2=k    ! divide the whole job
      do i=1,myProc                           ! approximately equally
        n1=n2+1; n2=n1+k
      enddo
      if(myProc.eq.NProc-1) n2=n

      call eratosthenes(n1,n2,nprimes)   ! each process does its separate job
      k=nprimes(0)+1
      if(myProc.ne.master) then          ! and sends own results to master
         call MPI_Send(nprimes,k,MPI_Integer, master,0,idcomm,ierr)
      else
        do id=1,NProc-1                  ! master joins all generated primes
          np=nprimes(0) ; last=nprimes(np); np1=np
          call MPI_Recv(nprimes(np),N,MPI_Integer,id,0,idcomm,st,ierr)
          nprimes(0)=nprimes(0)+nprimes(np)
          nprimes(np1)=last
        enddo
      endif
!  propagate NPrimes  to all processes
      call MPI_BCast(nprimes,n+1,MPI_Integer, master, idcomm,ierr)
      if(.not.iflag) call MPI_Finalize(ierr)
      Return
      end subroutine primus
```

Видно, что Primus является простейшей MPI-надстройкой программы Eratosthenes. Обе программы включены в нашу библиотеку JINRLIB.

23.08.2013