А.П.Сапожников

# Второй опыт распараллеливания больших вычислительных программ. Параллельная версия программы FUMILI.

Мы продолжаем серию работ по распараллеливанию программ, начатую в [1]. Там уже были изложены общепринятые подходы к распараллеливанию и основные понятия инструментального пакета MPI [2], использованного и в данной работе. В качестве объекта для распараллеливания мы продолжаем выбирать наиболее значимые (по крайней мере для ОИЯИ) вычислительные программы, созданные выдающимися программистами XX века.

Игорь Николаевич Силин (1936-2006) был участником и соавтором большей части работ по фазовому анализу, выполненных в Объединенном Институте Ядерных Исследований. Успехи ОИЯИ в этой области во многом обусловлены созданным им алгоритмом минимизации нелинейных функционалов. Программа И.Н.Силина FUMILI [3-4], реализующая этот алгоритм, уже более 40 лет активно используется учеными многих стран.

Строго говоря, программа FUMILI, по нынешним меркам, уже не может называться большой (всего около 2000 строк на Фортране). Однако в те былинные времена (конец 60-х годов XX века) скудость вычислительных ресурсов заставляла авторов программ вкладывать много ума в небольшой объем компьютерной памяти. Это умение, не в обиду будь сказано, нынешними поколениями программистов утрачено абсолютно. Поэтому мы продолжаем считать FUMILI БОЛЬШОЙ программой.

В качестве исходного текста для распараллеливания мы использовали не основной авторский текст программы [4], а современную его модификацию FUMILIM [5], проделанную И.М.Ситником (ОИЯИ), где, наряду с бережным отношением к оригиналу, были заложены возможности для дальнейшего развития программы. Далее, чтобы не путать различные реинкарнации изначальной FUMILI, мы даем ее параллельной (лучше было бы сказать – параллелизованной!) версии имя PFUMILI, полностью сохраняя интерфейс FUMILIM, описанный И.М.Ситником в [5].

## Анализ проблемы.

Прежде всего, нас интересовали наиболее времяемкие места распараллеливаемой программы. Для этого исследования мы использовали следующую технику:
- в исходном тексте программы выделяется некоторое количество интервалов;
- в начале и конце каждого такого N-го интервала помещаются засечки времени;
- в самом конце тестовой программы делается печать накопленной статистики.

Здесь мы позволим себе опубликовать текст подпрограммы сбора временной статистики, ибо она безусловно представляет интерес для программирующих на Фортране.

```
      Subroutine Profile(n)  ! n=0 for dump all counters
      Parameter(NTC=100)     ! Max Number of Time-counters
        integer Ncall(NTC),NcallF
        real*4 t,tt,TimeCounters(NTC),t1
        Data TimeCounters/NTC*0.0/
        Data Ncall/NTC*0/              ! *2 !
        common /mpi_fum/ mpi_size, !number of MPI-processes  =1,2,...
     -                   mpi_rank, !current MPI-process rank =0,1,...,mpi_size-1
      Save
        if(n.lt.0) then
          t1=secnds(0.0)   ! астр.время старта теста
          return
        endif
        call cpu_time(t)
        if(n.gt.0.and.n.le.NTC) then
          Ncall(n)=Ncall(n)+1
          TimeCounters(n)=t-TimeCounters(n)
        endif
        if(n.ne.0) return
!   финальная выдача накопленной статистики
        do i=1,NTC
          tt=TimeCounters(i)
          j=NCall(i)/2
          TimeCounters(i)=0.0
          NCall(i)=0
          if(tt.gt.0) then
             write(*,1) mpi_rank,mpi_size,i,tt,j
  1   format(' Proc:',i2,' of ',i2,'. Time',i3,' = ',f7.2,' sec. Ncall=',i6)
          endif
        enddo
        write(*,2) secnds(t1),t,mpi_rank
  2   format(' Astime:',f10.2,'  CPU_time:',f9.2,'  for process ',i2)
        return
      End Subroutine Profile
```

Эта подпрограмма подсчитывает количество посещений заданного интервала и суммарное время, затраченное процессом на эти посещения. Ее достаточно удобно использовать и во внутренних циклах исследуемой программы. Здесь CPU_Time и Secnds – встроенные функции Compaq (DEC) Fortran.

Запустив программу FUMILI с достаточно типичным набором тестовых данных, использовавшихся И.М.Ситником, мы получили для нее следующую статистику:

```
      Proc: 0 of  1. Time  1 =    0.02 sec. Ncall=    45  callF=  3340001    /1/
      Proc: 0 of  1. Time  3 =    0.25 sec. Ncall=     3  callF=  3340001
      Proc: 0 of  1. Time  4 =   21.14 sec. Ncall=    45  callF=  3340001
      Astime:     21.48  CPU_time:    21.48  for process  0
```

Здесь callF – количество обращений к функции пользователя Funct, подсчитывавшееся отдельно.

Легко видеть, что наиболее времяемким местом программы FUMILI является интервал номер 4 (subroutine SGZ, вычисляющая производные минимизируемой функции по всем ее параметрам, а также так называемую Z-матрицу), занимающий львиную долю (98.5%) общего времени прогона теста. Таким образом, место приложения усилий по распараллеливанию определилось достаточно быстро. Кроме того, вспоминая известный закон Амдаля, можно надеяться, что при хорошем распараллеливании SGZ следует ожидать ускорение работы всей FUMILI, пропорциональное количеству параллельно работающих процессов. Как призывал в свое время известный коммунистический лозунг: «наши цели ясны, задачи определены, за работу, товарищи!»

**Реализация.**

Результат работы подпрограммы SGZ есть сумма по всем заданным NED экспериментальным точкам. Основная идея распараллеливания – поделить NED экспериментальных точек между всеми mpi_size процессами приблизительно поровну. Это делается так:

```
nn=ned/mpi_size       ! points for 1 process
n1=1+mpi_rank*nn      ! 1-st point
n2=n1+nn              ! last point
if(mpi_rank.eq.mpi_size-1) n2=ned
```

Здесь mpi_rank = 0,1,… mpi_size-1 – внутренний номер процесса, а последний оператор корректирует номер последней точки для последнего процесса, если NED не делится нацело на mpi_size. После чего цикл DO L1=1,NED заменяется в SGZ на более короткий цикл DO L1=n1,n2. Поскольку этот цикл является самым внешним циклом SGZ, распределение его работы между mpi_size процессами-исполнителями следует признать хорошим распараллеливанием.

Увы, кроме того, чтобы поделить вычислительную работу между собой, процессы обязаны сделать всеобщим достоянием полученные ими частные результаты, а это уже сложнее: надо совершать межпроцессные обмены информацией, которые в общем случае могут свести к нулю всю достигнутую экономию времени. В нашем случае программе SGZ приходится совершать одну операцию MPI_Send или MPI_Recv, а также 2 операции MPI_BCast. Приведем для полноты и этот фрагмент:

```
if(mpi_rank.eq.0) then
  do jj=1, mpi_size-1              ! receive from all others if any
    call mpi_Recv(gzbuf,nn,mpi_data,jj,0,mpi_comi,st,mpi_err)
    hisq=hisq+gzbuf(1)
    do i=1,nfrpar
        G(i)=G(i)+gzbuf(i+1)
    enddo
    do i=1,nzfr
        Z(i)=Z(i)+gzbuf(i+nfrpar+1)
    enddo
  enddo
  Z(nzfr+1)=hisq   ! to save 1 BroadCast
else
  gzbuf(1)=hisq
  call mpi_Send(gzbuf,nn,mpi_data,0,0,mpi_comi,mpi_err)        ! send data
endif
call MPI_BCast(G,nfrpar,mpi_data,0,mpi_comi,mpi_err)  ! propagate
call MPI_BCast(Z,nzfr+1,mpi_data,0,mpi_comi,mpi_err)  ! G and Z
```

**Достигнутые результаты:**

Запустив все тот же тест И.М.Ситника на исполнение коллективом из NP=2 процессов на двухъядерном компьютере, мы получили для него следующую статистику:

```
Proc: 0 of  2. Time  1 =    0.02 sec. Ncall=    45  callF=  1690166
Proc: 0 of  2. Time  3 =    0.33 sec. Ncall=     3  callF=  1690166
Proc: 0 of  2. Time  4 =   13.16 sec. Ncall=    45  callF=  1690166
Astime:     13.70  CPU_time:    13.55  for process  0

Proc: 1 of  2. Time  1 =    0.02 sec. Ncall=    45  callF=  1690001
Proc: 1 of  2. Time  3 =    0.33 sec. Ncall=     3  callF=  1690001
Proc: 1 of  2. Time  4 =   13.12 sec. Ncall=    45  callF=  1690001
Astime:     13.59  CPU_time:    13.50  for process  1
```

Здесь астрономического времени AsTime затрачено в 21.5/13.5 = 1.65 раза меньше, чем при NP=1, поскольку 2 процесса работали реально одновременно, а ускорение “не дотянуло” до идеальных 2 именно из-за необходимости совершать межпроцессные обмены данными. Напомним, что мы сознательно говорим о процессах, а не о процессорах, ибо, во-первых, технология MPI работает в терминах именно

процессов, во-вторых, в наших экспериментах мы используем простейшую модель кластера, состоящую из одного двухъядерного компьютера.

Приведем еще аналогичную финальную выдачу теста для NP=3:

```
Proc: 2 of  3. Time  3 =    0.31 sec. Ncall=     3  callF=  1140221
Proc: 2 of  3. Time  4 =    8.80 sec. Ncall=    45  callF=  1140221
Astime:     14.85  CPU_time:     9.10  for process  2

Proc: 1 of  3. Time  3 =    0.33 sec. Ncall=     3  callF=  1140056
Proc: 1 of  3. Time  4 =    8.20 sec. Ncall=    45  callF=  1140056
Astime:     14.85  CPU_time:     8.62  for process  1

Proc: 0 of  3. Time  3 =    0.33 sec. Ncall=     3  callF=  1140056
Proc: 0 of  3. Time  4 =    8.75 sec. Ncall=    45  callF=  1140056
Astime:     14.98  CPU_time:     9.14  for process  0
```

и для NP=4:

```
Proc: 2 of  4. Time  3 =    0.31 sec. Ncall=     3  callF=   865166
Proc: 2 of  4. Time  4 =    6.56 sec. Ncall=    45  callF=   865166
Astime:     14.23  CPU_time:     6.94  for process  2

Proc: 1 of  4. Time  3 =    0.31 sec. Ncall=     3  callF=   865166
Proc: 1 of  4. Time  4 =    6.42 sec. Ncall=    45  callF=   865166
Astime:     14.22  CPU_time:     6.78  for process  1

Proc: 0 of  4. Time  3 =    0.31 sec. Ncall=     3  callF=   865166
Proc: 3 of  4. Time  3 =    0.31 sec. Ncall=     3  callF=   865001
Proc: 0 of  4. Time  4 =    6.80 sec. Ncall=    45  callF=   865166
Proc: 3 of  4. Time  4 =    6.42 sec. Ncall=    45  callF=   865001
Astime:     14.34  CPU_time:     7.20  for process  0
Astime:     14.23  CPU_time:     6.81  for process  3
```

Обратите внимание на асинхронность выдачи, заметную в последнем случае. Она демонстрирует реальную независимость работы процессов. Кроме того, практически постоянная величина AsTime при всех NP>1 легко объясняется тем, что количество вычислительной работы комплекса из 2-х и более процессоров остается одинаковым, в случае же NP=1 половина комплекса просто простаивает!

Заметим также, что величина T0=CPU_Time для процесса 0 всегда чуть больше, чем для его соратников, потому что ему всегда приходится принимать данные от них всех, а им достаточно совершить единственную посылку. Именно поэтому отношение A=T0(NP>1)/T0(NP=1) хорошо описывает рост производительности кластера из NP процессоров по сравнению с однопроцессорным компьютером. Вот данные, полученные все на том же двухъядерном компьютере автора:

```
   NP  : 1   2     3     4     5     10     15    20     25    50    100
   A   : 1   1.65  2.3   3.0   3.6   7.0    10.0  12.5   14.0  23.5   26.0
AsTime: 21 13.7  14.9  14.3  14.8  15.5    16.6  17.7   18.9  24.0   33.0
```

Видно, что A(NP) растет все медленнее и медленнее с ростом NP. Причина – растущее количество межпроцессных коммуникаций. В то же время AsTime(NP) резко возрастает, начиная с NP>50. Это объясняется появляющейся при NP=50 “толкотней” в общей памяти компьютера. У настоящих, промышленных кластеров, где каждый процессор имеет собственную память, этого эффекта не ожидается.

Естественно, нет необходимости говорить, что результат работы теста для NP>1 совпадает с результатом для NP=1.

## Что же дальше?

На этом можно было бы и завершить работу, удовлетворившись достигнутыми, весьма впечатляющими результатами. Однако по совету И.М.Ситника мы обратили внимание на

процедуру MGCONV (интервал 1 в /1/), которая должна потреблять время, пропорциональное квадрату числа NPAR входных параметров FUMILI. Сильно уменьшив в тесте количество экспериментальных точек NED, дабы уменьшить доминирующее влияние процедуры SGZ, мы получили следующую временную статистику для NP=1 и NPAR=200:

```
Proc: 0 of  1. Time  1 =   12.05 sec. Ncall=   135  callF=  2296500
Proc: 0 of  1. Time  4 =    1.69 sec. Ncall=    45  callF=  2296500
Astime:     13.80  CPU_time:    13.81  for process  0
```

Увы, все попытки распараллелить MGCONV с целью уменьшить Time 1 не удались! Причина кроется в том, что на всем протяжении этой программы активно перевычисляется большое количество элементов матрицы Z, так что при любой логике разделения труда между процессами нам приходится каждый раз делать общим достоянием ВСЮ Z-матрицу, что влечет чрезмерно большое количество межпроцессных пересылок. Надо сказать, это явление весьма характерно для многих классических алгоритмов, оперирующих с матрицами.

Чтобы “не уйти, несолоно хлебавши”, нам придется исследовать сравнительный вклад SGZ и MGCONV в общее время работы FUMILI при фиксированном числе параметров NPAR и различных количествах экспериментальных данных NED. Вот эта таблица для достаточно солидного NPAR=100:

| NED | 500 | 1000 | 2000 | 5000 | 10000 | 20000 | 50000 | 100000 |
|---|---|---|---|---|---|---|---|---|
| SGZ/MGCONV | 0.5 | 0.8 | 1.5 | 4 | 8 | 18 | 45 | 90 |

Таким образом, проведенное распараллеливание следует считать эффективным для задач с числом экспериментальных точек 5000 или более. При малых NED время работы практически то же, что и для нераспараллеленной FUMILI.

**Заключение:**

Мы произвели очередную модернизацию программы FUMILI, допускающую ее эффективную эксплуатацию на современных промышленных вычислительных кластерах, объединяющих сотни однотипных процессоров. При этом интерфейс программы совершенно не изменился по сравнению с ее однопроцессорным вариантом. Это открывает возможности для распараллеливания более крупных вычислительных программ, использующих PFUMILI для выполнения отдельных этапов своей работы.

Приносим извинения читателю за обилие статистики машинных прогонов программы, но именно они акцентируют внимание на деталях ее поведения и позволяют аргументировать принятые решения.

Автор признателен И.М.Ситнику за полезные обсуждения.

**Литература:**

[1] А.П.Сапожников. Опыт распараллеливания больших вычислительных программ. Параллельная версия программы MINUIT. P11-2003-216, Дубна, ОИЯИ, 2003.

[2] MPI: The complete Reference. MIT Press, Cambridge, Massachusetts, 1997.

[3] S.N.Sokolov, I.N.Silin, Preprint JINR D-810, Dubna, 1961.

[4] И.Н.Силин. FUMILI (DFUMIL), JINRLIB, D510.
I.N.Silin, CERN Program Library, D510, FUMILI, 1983.

[5] I.M.Sitnik, Modification of the FUMILI Minimization Package.
In memory of Prof. I.N.Silin. E11-2008-43, JINR, Dubna, 2008.

Ноябрь 2009 г.
Дубна, ЛИТ, ОИЯИ.