

# Параллельная версия программы FUMILI

А.П. Сапожников

e-mail: sap@jinr.ru, Лаборатория информационных технологий, ОИЯИ, Дубна

В последнее десятилетие происходит бурное развитие технологий программирования параллельных вычислений, в частности, MPI [1]. Эта тенденция нашла свое отражение и в библиотеке JINRLIB [2]. Была сформулирована следующая стратегия распараллеливания: библиотечная программа, подготовленная для работы в среде MPI, должна успешно работать при любом количестве NP параллельных процессов, вовлекаемых в решение прикладной задачи, в частности и при NP=1 [3]. Таким образом, возникает единый исходный текст библиотечной программы, равно пригодный к эксплуатации как на традиционных последовательных вычислительных системах, так и на современных кластерах, состоящих их большого числа процессоров.

Необходимым условием для обеспечения такой унификации является наличие заранее подготовленных программ-заглушек (stubs) пакета MPI, что позволяет использовать библиотечные программы даже на тех машинах, где никакого MPI нет вообще. К счастью, базовые операции MPI сконструированы таким образом, что в качестве заглушек достаточно иметь пустые подпрограммы, лишь бы системный загрузчик смог реализовать все внешние ссылки. Эта идея была успешно опробована при распараллеливании программы минимизации функций многих переменных MINUIT (автор Ф.Джеймс, ЦЕРН), которая при работе на кластерах использует «настоящий» MPI, а при работе в однопроцессорной конфигурации – его заглушки [3], [4].

Продолжая серию работ по распараллеливанию программ, в качестве объекта для распараллеливания выбираем 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 [5]-[6], 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 [6], а современную его модификацию FUMILIM [7], проделанную И.М.Ситником (ОИЯИ), где, наряду с бережным отношением к оригиналу, были заложены возможности для дальнейшего развития программы.

## Анализ проблемы

Прежде всего, нас интересовали наиболее времязатратные 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,mpi_rank
! number of processes = 1,2,...
! current process rank = 0,1,...,mpi_size-1
common/ncallf/mpi_ncallf(2)
Save
if(n.lt.0) then
  t1=secsnds(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,mpi_ncallF(1)
1 format(' Proc:',i2,' of ',i2,'. Time',i3,' = ',f7.2,
- ' sec. Ncall=',i6,' callF=',i9)
    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'a.

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
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

```

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, вспоминная известный закон Амдаля, можно надеяться, что при хорошем распараллеливании SGZ следует ожидать ускорение работы всей FUMILI, пропорциональное количеству параллельно работающих процессов.

## 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.

## 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.

Приведем еще аналогичную финальную выдачу теста 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 и более процессоров остается одинаковым, в случае же  $NP = 1$  половина комплекса просто простаивает!

Заметим также, что величина  $T_0 = CPU\_Time$  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  $A = T_0(NP=1)/T_0(NP>1)$  хорошо описывает рост производительности кластера из  $NP$  процессоров по сравнению с однопроцессорным компьютером. Вот данные, полученные все на том же 2-ядерном 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  $NP$ . 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  $NP = 1$ .

## 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 FUMILI для выполнения отдельных этапов своей работы.

Автор признателен И.М.Ситнику за полезные обсуждения.

## Список литературы

- [1] *MPI: The complete Reference*. MIT Press, Cambridge, Massachusetts, 1997.
- [2] Л.В. Попкова, А.П. Сапожников, Т.Ф. Сапожникова, Р.Н. Федорова: *Библиотека программ JINRLIB*. RCDL'2008, Дубна, Россия, 2008. <http://www.jinr.ru/programs/jinrlib>.
- [3] А.П. Сапожников: *Опыт распараллеливания больших вычислительных программ. Параллельная версия программы MINUIT*. P11-2003-216, Дубна, ОИЯИ, 2003.
- [4] F.James: MINUIT, CERN Program Library Long Writeup, D506. CERN, Geneva, Switzerland.
- [5] S.N.Sokolov, I.N.Silin: Preprint JINR D-810, Dubna, 1961.
- [6] I.N.Silin: FUMILI, CERN Program Library, D510, 1983.
- [7] I.M.Sitnik: *Modification of the FUMILI Minimization Package. In memory of Prof. I.N.Silin*. E11-2008-43, JINR, Dubna 2008.