Р11-2003-216

А. П. Сапожников

ОПЫТ РАСПАРАЛЛЕЛИВАНИЯ БОЛЬШИХ ВЫЧИСЛИТЕЛЬНЫХ ПРОГРАММ. ПАРАЛЛЕЛЬНАЯ ВЕРСИЯ ПРОГРАММЫ **MINUIT**

Направлено в журнал «Программирование»

Если взглянуть на известный список самых мощных вычислительных систем Top-500 [1], легко заметить, что традиционных однопроцессорных машин там почти не осталось, вся верхняя часть списка оккупирована системами из большого числа однотипных и сравнительно недорогих процессоров. Это означает, что построение именно таких систем, так называемых кластеров, является в настоящее время общепризнанным путем повышения вычислительной мощности.

Естественно, что появление новых идей в области вычислительной техники самым непосредственным образом сказывается на программном обеспечении. В частности, возникает необходимость адаптации больших программ для численных расчетов к специфическим условиям существования в многомашинной и многопроцессорной среде. Такая адаптация, везде в дальнейшем именуемая распараллеливанием, обходится несоизмеримо дешевле повторной разработки программ, зачастую уникальных. Речь идет о программах, написанных на языках Фортран и С.

Существует два подхода к распараллеливанию программ. Первый из них, наиболее привлекательный, состоит в том, что проблема возлагается на компилятор, а пользователь только выдает компилятору указания: вот здесь попробуй распараллелить, а вот тут - не надо! Этот подход принят в системе OpenMP [2,3], довольно распространенной в настоящее время. Авторы [3] особо выделяют следующее достоинство технологии OpenMP: текст программы один и тот же как для ее последовательной версии, так и для параллельной. Недостаток здесь только один, но существенный: все это может работать только в вычислительных системах с общей памятью, на базе потоковой модели! А кластеры, наиболее распространенные ныне многопроцессорные системы, увы, к таковым не относятся.

Второй подход существенно труднее: необходимо вручную преобразовать текст программы, явно программируя как распределение работы между процессорами, так и все необходимые межпроцессорные коммуникации. А чтобы не думать о процессорах (кто его знает - сколько их всего!), вместо процессоров в ход идет понятие ПРОЦЕССОВ. За последние 30 лет мы все уже привыкли к тому, что даже на единственном процессоре могут успешно сосуществовать много ПРОЦЕССОВ.

Этот подход пригоден практически для всех вычислительных систем, программное обеспечение которых

имеет в своем составе библиотеку программ для организации МЕЖПРОЦЕССНЫХ коммуникаций.

Самый распространенный на сегодня инструментарий такого рода – это пакет "Message Passing Interface", или просто MPI [4], работающий практически везде и претендовавший, начиная с 1995 года, на роль стандарта при распараллеливании программ, а ныне благополучно таковым стандартом и ставший. Именно этот подход мы намерены использовать в настоящей работе.

Основные понятия MPI. Парадигма SPMD

При запуске задачи создается группа из P процессов. Группа идентифицируется целочисленным дескриптором (коммуникатором). Внутри группы процессы нумеруются от 0 до P-1. В ходе решения задачи исходная группа (ей присвоено имя MPI_COMM_WORLD) может делиться на подгруппы, подгруппы могут объединяться в новую группу, имеющую свой коммуникатор. Таким образом, процесс имеет возможность одновременно принадлежать нескольким группам процессов. Каждому процессу доступен свой номер myProc внутри любой группы, членом которой он является.

Поведение всех процессов описывается одной и той же программой. Межпроцессные коммуникации в ней программируются явно с использованием библиотеки MPI, которая и диктует стандарт программирования. Квазиодновременный запуск исходной группы процессов производится средствами операционной системы. При этом P определяется желанием пользователя, а отнюдь не количеством доступных процессоров!

Итак, все P процессов асинхронно выполняют одну и ту же программу. Но у каждого из них свой номер myProc. Поэтому в программе, естественно, будут такие фрагменты:

```
If (myProc.eq.0) then
   < делать что-то одно >
else if (myProc.eq.1) then
   < делать что-то другое >
    .  .  .
else
   < делать что-то P-е >
endif
```

Таким образом, в программе "под одной крышей" закодировано поведение всех процессов. В этом и заключена парадигма программирования Single Program – Multiple Data (SPMD).

Обычно поведение процессов одинаково для всех, кроме одного, который выполняет координирующие функции, не отказываясь, впрочем, взять на себя и часть общей работы. В качестве координатора обычно выбирают процесс с номером 0.

MINUIT как типичный объект для распараллеливания

Классическим примером больших вычислительных программ является MINUIT [5] – программа минимизации функции многих переменных, написанная в начале 70-х годов прошлого века Фредериком Джеймсом (ЦЕРН) и весьма популярная до сих пор. MINUIT состоит более чем из 60 подпрограмм, общая длина которых более 7000 строк. Это универсальная программа, пригодная для минимизации функций любого вида, но чаще она используется для функций вида χ^2. Минимизируемая функция оформляется пользователем как фортрановская подпрограмма FCN. MINUIT используется для решения широкого круга задач: от математической обработки результатов физического эксперимента до фундаментальных проблем теоретической физики. Такая популярность этой программы и побуждает выбрать именно ее как объект распараллеливания прежде других.

Основная цель настоящей работы – получить единый текст MINUIT, пригодный для эксплуатации при любом числе процессоров P, в том числе и при P=1, сохранив таким образом присущую программе универсальность. Причем при P=1 "параллельная" версия MINUIT должна работать ненамного медленнее оригинальной. Такая постановка задачи была обусловлена тем, что работа является частью более широкого замысла – создания новой версии библиотеки JINRLIB, легко адаптируемой для максимально разнообразных компьютерных платформ: от персональных компьютеров до больших вычислительных кластеров. Кроме того, естественным было стремление к минимальному изменению авторского текста.

Наиболее трудоемкое дело – многократное вычисление минимизируемой функции FCN. Удалось набрать статистику, какие подпрограммы MINUIT сколько раз вызывают FCN. На них и было обращено основное внимание. Таких подпрограмм немного: SEEK, MIGRAD, SIMPLEX.

Подпрограмма SEEK

Поиск минимума методом Монте-Карло производится в области, ограниченной значениями +/- ошибки по каждому параметру, причем точки выбираются по нормальному закону с дисперсией, равной величине ошибки. Напомним, что по каждому параметру программа отслеживает его значение x(i) в данный момент, его ошибку werr(i) и текущий шаг дифференцирования по этому параметру step(i).

Вот упрощенная схема процедуры SEEK. Здесь и далее NProc - общее число процессов, участвующих в работе, MyProc - номер текущего процесса, N - число независимых переменных, Nseek - количество монте-карловских точек.

```
   do 1 k=1,Nseek
     do 2 i=1,N
2    x(i)=xbest(i)+0.5*(rndm(1)+rndm(2)-1)*werr(i)
  →    if(Mod(k,Nproc).ne.MyProc) goto 1  !parallelization
     y=FCN(x)
     if(y.lt.ymin) then
       do 3 i=1,N
3      xbest(i)=x(i)
       ymin=y
     endif
1  continue
  →  call MN_Best(xbest,ymin)
```

Здесь стрелочками помечены две строки фортранного текста, добавленного в авторский текст. Подпрограмма MN_Best, основанная на стандартных MPI-примитивах MPI_Send, MPI_Recv, MPI_Bcast, выбирает наименьшее во всех Nproc процессах ymin и соответствующий ему вектор параметров xbest, а затем рассылает все N+1 чисел во все процессы. Оператор

```
if(Mod(k,Nproc).ne.MyProc) . . .          (1)
```

реализует основную идею разделения работы между процессами:

- все работы каким-то способом пронумерованы;
- каждый процесс исполняет только работы, соответствующие его номеру;
- полученные результаты объединяются и становятся общим достоянием.

Заметим, что все процессы здесь генерируют одинаковую последовательность случайных чисел, но каждый из них использует свою ее часть. В принципе, каждый процесс мог бы использовать свой, независимый датчик случайных чисел, но это явилось бы слишком серьезным изменением авторского текста!

Видно, что в основном цикле вообще нет межпроцессных коммуникаций, поэтому он распараллелен максимально эффективно, то есть время работы SEEK обратно пропорционально количеству задействованных процессов.

Подпрограмма MIGRAD

В подпрограмме MIGRAD используется градиентный метод Давидона. Здесь распараллеливанию подвергнут внутренний цикл вычисления частных производных по всем N независимым переменным. Вот его упрощенная схема:

```
   Do 1 i=1,N
 →   if(Mod(i,Nproc).ne.MyProc) goto 1   !parallelization
    x(i)=x(i)-step(i)
    y1=FCN(x)
    x(i)=x(i)+2*step(i)
    y2=FCN(x)
    grad(i)=(y2-y1)/2*step(i)
 →   call MN_SendGrad(i)
1  continue
 →   call MN_SendGrad(0)
```

Здесь так же стрелочками отмечены вставки в авторский текст. Подпрограмма MN_SendGrad либо передает нулевому процессу свою только что вычисленную производную по i-й переменной, либо получает от него все N производных. На самом деле значению производной сопутствует еще и величина шага по соответствующей переменной и ряд других (всего до 6) вспомогательных величин, существенно необходимых программе. Для этого удобным оказалось использование программно-конструируемых типов данных, активно пропагандируемых в пакете MPI. Конечно, это разумно, только если время вычисления функции существенно больше времени пересылки массива из 6 чисел.

Легко видеть, что вся процедура содержит ровно N+1 межпроцессных коммуникаций. Далее стоит отметить, что эта схема логично работает даже при Nproc > N, что, впрочем,

малоосмысленно: зачем привлекать слишком много исполнителей для небольшой работы.

Примерно по той же схеме делается распараллеливание цикла вычисления ковариантной матрицы V(x) (она в MINUIT треугольная). В этом случае основной цикл делается не N, а N*(N+1)/2 раз.

Внимательно анализируя (1), можно заметить, что при Nproc > 1 нулевой процесс никогда не получает большего количества работы, чем прочие. Это существенно, так как на него дополнительно возложены функции координатора.

Таким образом, процедуру вычисления всех частных производных можно считать распараллеленной достаточно эффективно. Однако сам итерационный процесс контроля сходимости градиентного метода, как почти любой итерационный процесс, распараллелить не удается, все процессы вынуждены дублировать эту работу, что несколько снижает общую эффективность распараллеливания.

Подпрограмма SIMPLEX

Здесь используется симплекс-метод Нелдера и Мида. Начальный симплекс образуют текущая точка минимума FCN и N точек, соответствующих локальным минимумам по каждой из N переменных. Очевидно, что этап построения начального симплекса можно распараллелить точно так же, как и вычисление частных производных в MIGRAD:

```
      Do 1 i=1,N
  →     if(Mod(i,Nproc).ne.MyProc) goto 1 ! parallelization
!           определение i-й точки симплекса
  →     call MN_SendPoint(i)
1   continue
  →     call MN_SendPoint(0)
```

Здесь MN_SendPoint либо передает нулевому процессу свою только что вычисленную точку, либо получает от него все N точек.

Далее на каждой итерации определяется отражение наихудшей точки симплекса относительно центра тяжести остальных точек. Эта новая точка или сама заменяет наихудшую в текущем симплексе, или служит отправной точкой для более сложного критерия подбора заменяющей. Все это требует от 1 до 3 вычислений FCN в каждой итерации. Этот итерационный процесс распараллелить не удается, поэтому при

плохой его сходимости экономия от параллельного вычисления начального симплекса сводится на нет.

Итак, мы видим, что различные блоки MINUIT удалось распараллелить в разной степени: стопроцентно (SEEK), приемлемо (MIGRAD), частично (SIMPLEX). Однако MINUIT, являясь весьма функционально мощной программой, позволяет пользователю применять эти 3 метода в разнообразных сочетаниях и с различной интенсивностью, а значит – учитывать и различный выигрыш в быстродействии при работе на нескольких процессорах!

Типовые этапы работы над MINUIT

Выше мы подробно описали основной этап работы по распараллеливанию MINUIT – разделение вычислительных операций между отдельными процессами. В ходе этой деятельности наметился еще ряд достаточно типичных для любой большой вычислительной программы этапов.

- Организация обрамления программы. Добавлены подпрограммы MN_Start и MN_Finish для захвата и освобождения MPI-ресурсов, каковыми являются временная группа процессов и вспомогательные типы данных. Временная группа процессов необходима, чтобы скрыть внутренние коммуникации MINUIT от использующей его прикладной программы, которая, в свою очередь, может быть распараллелена независимо!

- Проработка набора базовых коммуникационных операций, специфичных именно для MINUIT. Их получилось немного, всего 6. Это две только что упомянутые обрамляющие операции MN_Start и MN_Finish, три основные операции (MN_Best, MN_SendGrad, MN_SendPoint), используемые в центральной части программы, и одна вспомогательная операция MN_Bcast, о которой речь пойдет чуть ниже. Реализованы все они как отдельные подпрограммы, в основной же текст добавляются только вызовы этих подпрограмм. Именно это обеспечивает как локальность изменений авторского текста, так и возможность работы при любом числе процессов, в частности и при P=1. Здесь принципиальным должно являться требование, чтобы вызовы пакета MPI находились только внутри этих базовых операций, а не были "размазаны" по всей

прикладной программе. Более того, все базовые операции используют внутренний, скрытый от пользователя, коммуникатор mn_comm, конструируемый в MN_Start, что является типичной техникой реализации библиотечных MPI-программ.

- Подготовка программ-заглушек пакета MPI - для обеспечения возможности эксплуатации этой версии MINUIT на тех машинах, где никакого MPI нет вообще. Базовые операции сконструированы таким образом, чтобы в качестве заглушек достаточно было иметь пустые подпрограммы, лишь бы загрузчик сумел реализовать все внешние ссылки. Наличие MPI-заглушек позволяет иметь единый текст как для последовательной, так и для распараллеленной программы, разница только в том, что распараллеленная программа использует настоящий MPI, а нераспараллеленная - его заглушки (stubs).

- Реорганизация операций ввода и вывода. Их в MINUIT много, программа имеет весьма развитый интерфейс с пользователем. Печать, и вообще весь вывод, естественно, должен выполнять только один из P процессов, участвующих в совместной работе, и, очевидно, это должен быть процесс 0: ведь программа должна правильно работать при любом P, в частности при P=1, а только нулевой процесс есть всегда, при любом составе группы процессов, вовлеченных в решение задачи! Поэтому все операторы вывода должны предваряться проверкой номера процесса:

```
If (MyProc.eq.0) Write(*,*) Data
```

Что же касается ввода, то тут чуть сложнее. Ясно, что файлы с входной информацией должны принадлежать кому-то одному, стало быть нулевому процессу. Прочитав из файла, он должен поделиться прочитанным со своими партнерами:

```
If (MyProc.eq.0) read(1,*) (data(i),i=1,count)
call MN_Bcast(data,count) !Propagate for all processes
```

Это типичная техника организации ввода начальных данных при распараллеливании программы. Добавляются две вещи: условный переход и MN_Bcast - последняя из шести базовых коммуникационных операций. MN_Bcast

практически идентична MPI_Bcast, но маскирует от внешнего окружения как используемый коммуникатор mn_comm, так и номер координирующего процесса.

В принципе, представляется возможным автоматизация ручного труда на этом этапе, то есть построение программы, выполняющей все необходимые преобразования исходного текста для работы в стандарте MPI. Это может явиться предметом нашей следующей работы.

Некоторые результаты тестирования

Параллельная версия MINUIT была протестирована как на системе с общей памятью SPP-2000, так и на PC-Linux кластере. В обоих случаях работа происходила под управлением ОС типа UNIX. Отмечено совпадение результатов счета при P>1 и при P=1 с результатом счета по нераспараллеленной программе. Кроме того, этот же текст MINUIT был подробно тестирован на 2-процессорной IBM PC, работавшей под Windows-2000 с использованием MPICH NT 1.2.2. Здесь удалось получить достаточно достоверные замеры счетного времени процессов. В качестве минимизируемой функции FCN наряду с тривиальной тестовой функцией

$$FCN = \sum_{i=1}^{50} (x(i)-i)**2 ,$$

дающей минимум, равный 0 при x(i)=i, использовалась и тестовая FCN, предложенная автором MINUIT.

Variant	AsTime	Processes CPU-times	Total	FCN-calls	F/AT
0	8.2	8.17	8.17	10069	1228
1	8.9	8.94	8.94	10069	1191
2	7.8	7.20+7.27	14.47	15942	2144
3	10.4	6.66+6.72+6.72	20.10	22013	2116
4	19.8	9.77+9.69+9.64+9.64	38.74	42490	2099

Здесь вариант 0 соответствует оригинальному, нераспараллеленному тексту MINUIT. Варианты 1÷4 – прогону параллельной версии при P=1, 2, 3, 4.

Напомним, что на IBM PC время определяется с точностью около 0.05 секунды. В тестах использовалась машина с двумя процессорами. Кроме того, не удалось добиться, чтобы во всех пяти вариантах совершалось одинаковое количество

вычислительной работы. Тем не менее результаты теста демонстрируют реально достигнутый параллелизм.

Представляет интерес разница в 0.7 между вариантами 0 и 1. Она обусловлена накладными расходами на инициализацию пакета MPI и является постоянной величиной, не зависящей от общего объема работы.

Далее, сравнение вариантов 2, 3 и 4 с вариантом 1 показывает, что за совокупное время в 1.5, 2 и 4 раза большее (Total) совершается соответственно в 1.5, 2 и 4 раза большее количество (FCN-calls) полезной вычислительной работы, при этом астрономическое время (AsTime) изменяется как 1, 1.5 и 2, что характеризует оптимальное использование именно 2-процессорной вычислительной системы.

Показатель F/AT характеризует количество совершенной полезной работы в единицу астрономического времени. Его медленное снижение с ростом P=2, 3, 4 обусловлено растущими накладными расходами на межпроцессные коммуникации.

Работа выполнена в рамках проекта, поддержанного грантом РФФИ 03-07-90347.

Приложение

Базовые коммуникационные операции MINUIT. Мы публикуем здесь эти фортранные тексты, так как они демонстрируют типичную технику использования коммуникационного пакета MPI.

```
 common /mn_mpi/ mpi_size, ! number of processes   = 1,2,...
2                mpi_rank, ! current process rank
                           ! = 0,1,...,mpi_size-1
3                mpi_come, ! external communicator
                           ! (usually MPI_COMM_WORLD)
4                mpi_comi, ! internal communicator
5                mpi_data, ! MPI kind of DataType (REAL or DOUBLE)
6                mpi_gtype,! Derived Type for Gradient (6 numbers)
7                mpi_stype,! Derived Type for Gradient
                           ! (6*NPAR numbers)
8                mpi_npar, ! previous NPAR in mn_Send
9                mpi_time, ! used CPU-time
*                LOUT      ! flag for I/O permitted (mpi_rank=0)
 real*8 mpi_time
 logical LOUT



 Function MultiMinuit(job) ! Parallel version of MINUIT via MPI
 include 'minuit.cmn'
 include 'mpif.h'
 if(job.ne.0) call mn_start(MPI_COMM_WORLD)
                           ! MPI init for global communicator
 MultiMinuit=mpi_rank
 if(job.eq.0) call mn_finish(MPI_COMM_WORLD)
                           ! MPI finalization
 return
 end

 subroutine mn_Start(comm) !MPI initialize for arbitrary communicator
 include 'minuit.cmn'
 include 'mpif.h'
 integer comm

 call MPI_INITIALIZED(Lout,ierr)
 if(.not.Lout) call MPI_INIT(ierr)      ! else init was done outside!
 mpi_data=MPI_DOUBLE_PRECISION
 if(Precision.eq.1) mpi_data=MPI_REAL
 mpi_time=0
 mpi_come=comm                          ! remember external communicator
 call MPI_COMM_DUP(comm,mpi_comi,ierr)  ! for internal exchanges
 mpi_npar=1
 call MPI_TYPE_VECTOR(6,1,MNI,mpi_data,mpi_gtype,ierr)
 call MPI_TYPE_COMMIT(mpi_gtype,ierr)
                                        ! conctruct own types for mn_Send
```

```
      call MPI_TYPE_VECTOR(6,mpi_npar,MNI,mpi_data,mpi_stype,ierr)
      call MPI_TYPE_COMMIT(mpi_stype,ierr)

      call MPI_COMM_SIZE(comm,mpi_size,ierr)
      call MPI_COMM_RANK(comm,mpi_rank,ierr)
      Lout=(mpi_rank.eq.0)
      if (Lout) write(*,*) ' MPI-run using ',mpi_size,' processes.'
      if (.not.Lout) isw(5)=-999! block printing !!!
      if (.not.Lout) isw(6)=0    ! block interactive mode !!!
      return
C
      entry mn_Finish(comm)      ! MPI finalization (comm does not matter!)
      call MPI_COMM_FREE(mpi_comi,ierr)
      call MPI_TYPE_FREE(mpi_gtype,ierr)
      call MPI_TYPE_FREE(mpi_stype,ierr)
      mpi_size=1
      mpi_rank=0
      mpi_npar=-1
      return
      end

      subroutine mn_BCast(a,n)  ! Breeding just read input data
      include 'minuit.cmn'
      include 'mpif.h'
      if(mpi_size.eq.1) Return  ! nothing to do
      mt=mpi_data
      if(n.lt.0) mt=MPI_CHARACTER
      call MPI_BCast(a,iabs(n),mt,0,mpi_comi,ierr)
      return
      end

      subroutine mn_SendGrad(i)
!     Send i-th Gradient Info to Main Process ( i>0 )
!     Join all Gradients and BCast them to all Processes ( i=0 )
!     A Gradient Info is:
!           MNDERI :  GSTEP(i)+GRD(i)+G2(i)
!           MNHES1 :  GSTEP(i)+GRD(i)+DGRG(i)
!           MNHESS :  GSTEP(i)+GRD(i)+G2(i)+DIRIN(i)+YY(i)
!     But we always will send all these 6 values.
!     It's a self-constructed Type mpi_gtype
!        of 6 Reals starting from GRD with strand=MNI
      include 'minuit.cmn'
      include 'mpif.h'
      integer st(MPI_STATUS_SIZE)
      if(mpi_size.eq.1) Return              ! nothing to do
      if(i.gt.0) then
        if(mpi_rank.eq.0) return
        call mpi_Send(grd(i),1,mpi_gtype,0,i,mpi_comi,ierr)
        return
      endif
      if(mpi_rank.eq.0) then
        do 1 j=1, NPAR                      ! receive from all others
          if(Mod(j,mpi_size).eq.0) goto 1   ! these were calculated by us
          call mpi_Probe(MPI_ANY_SOURCE,MPI_ANY_TAG,mpi_comi,st,ierr)
```

```
      is=st(MPI_SOURCE)
      it=st(MPI_TAG)
      call mpi_Recv(grd(it),1,mpi_gtype,is,it,mpi_comi,st,ierr)
1   continue
  endif
  if(mpi_npar.ne.NPAR) then
    call MPI_TYPE_FREE(mpi_stype,ierr)
                          !Reconstruct because NPAR maybe changed!
    call MPI_TYPE_VECTOR(6,NPAR,MNI,mpi_data,mpi_stype,ierr)
    call MPI_TYPE_COMMIT(mpi_stype,ierr)
    mpi_npar=NPAR
  endif
  call mpi_BCast(grd(1),1,mpi_stype, 0,mpi_comi,ierr)
  return
  end

  Function mn_SendPoint(i)  ! called from MNSIMP
!  Send (P(k,i),k=1,npar),Pbar(i),YY(i),?DirIn(i)
!       to Main Process ( i>0 )
!  Join all these arrays and BCast them to all Processes ( i=0 )
!  Returns index of simplex vertex with minimal value of FCN
  include 'minuit.cmn'
  include 'mpif.h'
  integer st(MPI_STATUS_SIZE)
  index=NPAR+1
  if(mpi_size.eq.1) goto 9  ! nothing to do
  if(NPAR+2.gt.MAXINT) then
    if(Lout) write(*,*) ' Too many params. SIMPLEX stopped.'
    stop
  endif
  if(i.ne.0) then
    p(NPAR+1,i)=pbar(i)
    p(NPAR+2,i)=yy(i)        !      dimension P(MNI,MNI+1)
    if(mpi_rank.ne.0)
*     call mpi_Send(P(1,i),NPAR+2,mpi_data,0,i,mpi_comi,ierr)
      goto 9
    endif
    if(mpi_rank.eq.0) then  ! main process
      do j=1, NPAR          ! receives from all except itself
        if(Mod(j,mpi_size).ne.0) then
          call mpi_Probe(MPI_ANY_SOURCE,MPI_ANY_TAG,mpi_comi,st,ierr)
          k=st(MPI_SOURCE)
          m=st(MPI_TAG)
          call mpi_Recv(p(1,m),NPAR+2,mpi_data,k,m,mpi_comi,st,ierr)
        endif
      enddo
    endif
    call mpi_BCast(P,MNI*NPAR,mpi_data, 0,mpi_comi,ierr)
    absmin=yy(index)
    do j=1,NPAR
      pbar(j) =p(NPAR+1,j)
      yy(j)   =p(NPAR+2,j)
      if(absmin.lt.yy(j)) then
        absmin=yy(j)
```

```
        index=j
      endif
    enddo
9   continue
    mn_SendPoint=index
    return
    end

    subroutine mn_Best(n,a)   ! called from MNSEEK
    include 'minuit.cmn'
    include 'mpif.h'
    integer st(MPI_STATUS_SIZE)
    dimension a(n),t(MNI)     ! n=NPAR+1   a=args(NPAR)+func(1)
    if(mpi_size.eq.1) Return ! nothing to do
    if(mpi_rank.ne.0) then
      call mpi_Send(a,n,mpi_data,0,mpi_rank,mpi_comi,ierr)
    else
      do i=2,mpi_size
        call mpi_Recv(t,n,mpi_data,
   &             MPI_ANY_SOURCE,MPI_ANY_TAG,mpi_comi,st,ierr)
        if(t(n).lt.a(n)) then
          do j=1,n
            a(j)=t(j)
          enddo
        endif
      enddo
    endif
    call mpi_BCast(a,n,mpi_data,0,mpi_comi,ierr)
    return
    end
```

Литература

1. http://parallel.ru
2. http://www.openmp.org
3. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. БХВ-Петербург, 2002.
4. MPI: The complete Reference. MIT Press, Cambridge, Massachusetts, 1997.
5. CERN Program Library Long Writeup D506. James. F. MINUIT. CERN, Geneva, Switzerland.

Получено 3 декабря 2003 г.

Сапожников А. П.
P11-2003-216

Опыт распараллеливания
больших вычислительных программ.
Параллельная версия программы MINUIT

На примере популярной программы MINUIT — минимизации функции многих переменных — обсуждаются проблемы распараллеливания больших вычислительных программ. Предлагается версия MINUIT для широкого класса многопроцессорных вычислительных систем с использованием коммуникационного пакета MPI. Приведены результаты тестирования, демонстрирующие реально достигнутый параллелизм.

Работа выполнена в Лаборатории информационных технологий ОИЯИ.

Препринт Объединенного института ядерных исследований. Дубна, 2003

Перевод автора

Sapozhnikov A. P.
P11-2003-216

The Experience of Large Computational
Programs Parallelization.
Parallel Version of MINUIT Program

The problems around large computational programs parallelization are discussed. As an example a parallel version of MINUIT, widely used program for minimization, is introduced. The given results of MPI-based MINUIT testing on multiprocessor system demonstrate really reached parallelism.

The investigation has been performed at the Laboratory of Information Technologies, JINR.

Preprint of the Joint Institute for Nuclear Research. Dubna, 2003

Редактор *О. Г. Андреева*
Макет *Е. В. Сабаевой*

Подписано в печать 28.12.2003.
Формат 60 × 90/16. Бумага офсетная. Печать офсетная.
Усл. печ. л. 1,0. Уч.-изд. л. 1,04. Тираж 310 экз. Заказ № 54260.

Издательский отдел Объединенного института ядерных исследований
141980, г. Дубна, Московская обл., ул. Жолио-Кюри, 6.
E-mail: publish@pds.jinr.ru
www.jinr.ru/publish/