# Как нам распараллелить программу и запустить ее на кластере HybriLIT

*А.П.Сапожников (sap@jinr.ru), Т.Ф.Сапожникова (tsap@jinr.ru)*

Эта статья была написана более 10 лет назад [1]. Но сейчас, следуя читательским просьбам, мы обновим ее применительно к современным техническим реалиям, благо идеологические основы практически не изменились, а актуальность все растет.

Если взглянуть на современную редакцию известного списка Top500 самых мощных вычислительных систем в мире, можно сделать любопытное открытие: там нет ни одной привычной нам машины с одним процессором! Подавляющее преимущество здесь имеют кластеры, состоящие из тысяч процессоров.

Однако обычные наши фортранные программы просто так не в состоянии использовать одновременно более одного процессора. Их необходимо подвергнуть так называемому распараллеливанию, то есть как-то преобразовать. Настоящий опус и призван показать, как можно это делать, используя инструментальный пакет MPI, ныне входящий в состав практически любой вычислительной системы.

Почему именно MPI, и что это за штука?

Вообще говоря, существует два подхода к распараллеливанию программ. Первый из них, и самый привлекательный, состоит в том, что проблема возлагается на компилятор, а пользователь только выдает компилятору ценные указания типа того: вот здесь попробуй распараллелить, а вот тут – не моги! Этот подход принят в системе OpenMP, довольно распространенной в настоящее время. Недостаток здесь только один, но существенный: все это может работать только в вычислительных системах с общей памятью! А кластеры, увы, к таковым не относятся.

Второй подход существенно труднее: пользователь сам, вручную, преобразует текст своей (или чужой!) 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 работоспособен практически на всех вычислительных системах, программное обеспечение которых имеет в своем составе библиотеку программ для организации МЕЖПРОЦЕССНЫХ коммуникаций. Самый продвинутый на сегодня инструментарий такого рода – это пакет «Message Passing Interface», или просто MPI, работающий практически везде, претендовавший, начиная с 1995 года на роль стандарта при распараллеливании программ, а ныне благополучно таковым стандартом и ставший.

На базе MPI существует несколько десятков технологий параллельного программирования. Наиболее интересной из них, на наш взгляд, является DVM – Distributed Virtual Memory, разработанная в ИПМ РАН. В чем-то она пытается сочетать оба описанных выше подхода. Здесь так же, как в OpenMP, пользователь имеет возможность указать компилятору, где распараллеливать, а где нет. С другой стороны, аналогично явному программированию межпроцессных коммуникаций, пользователь явно программирует распределение памяти между виртуальными процессорами. Причины, по которым мы все же начинаем с MPI, просты:

a) это базовый инструмент, все прочие – это навороты над ним;
b) это стандартный инструмент, он работает везде, а главное – на наших машинах;
c) это простой и по-человечески привычный инструмент.

А теперь уместно задаться основным вопросом: а надо ли заниматься этим самым распараллеливанием? Каков будет предел наших грядущих достижений? Предел этот описывается так называемым законом Амдаля. Тех немногочисленных читателей, которые еще не знают этого имени (Gene Amdahl), отсылаем в Википедию.

Итак, пусть $0 \le S \le 1$ - доля вычислительных операций нашей программы, которые должны совершаться сугубо последовательно. Тогда при одновременном использовании P процессоров мы можем ускорить свою программу максимум в

$$K = 1/(S+(1-S)/P) \text{ раз.}$$

Устремляя P к бесконечности, получим $K < 1/S$. В частности, если $S > 0.1$, то $K < 10$ при любом P. Если же $S = 0$ (чего в природе, вообще говоря, не наблюдается), то $K = P$. Напомним, что в 1998 году с появлением SPP-2000 у нас стало P = 8. Сейчас кластер HybriLIT [2] имеет $P > 300$. Так что решайте сами - окупятся Ваши труды или нет! Впрочем, пакет MPI позволяет распараллеливать программу для выполнения на любом количестве даже разнотипных компьютеров, соединенных в локальную сеть.

Тех, кто хочет получить более подробную информацию о распараллеливании вычислений или подробное описание пакета MPI, мы отсылаем к специализированному серверу МГУ [3]. Здесь же Вас ждут только основы идеологии MPI и несколько конкретных примеров.

**Основные понятия MPI. Парадигма SPMD.**

При запуске задачи создается группа из NProc процессов. Группа идентифицируется целочисленным дескриптором (коммуникатором). Внутри группы процессы нумеруются от 0 до NProc-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 может одновременно принадлежать нескольким группам процессов. Каждый процесс может узнать свой номер myProc внутри любой группы, к которой он принадлежит.

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 NProc определяется желанием пользователя, а отнюдь не количеством доступных процессоров!

Итак, все NProc 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 третье >
Endif
```

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

Не надо путать аббревиатуру SPMD с SIMD. Последняя обозначает специфическую компьютерную архитектуру в классификации Флинна, когда несколько процессоров СИНХРОННО исполняют одну и ту же программу, а о процессах и речи не идет!

Словарь пакета MPI состоит из некоторого количества поименованных целочисленных констант. Все эти имена начинаются с префикса MPI_. Описания всех этих констант хранятся в системном файле mpif.h для Фортрана и mpi.h для языка C. Вот важнейшие из них:

MPI_Comm_World - предопределенная группа из всех запущенных процессов;
MPI_Integer, MPI_Real, MPI_Byte, … - имена различных типов данных;
MPI_Any_Source, MPI_Any_Tag - джокеры (нечто вроде *.* в MS-DOS);
MPI_Source, MPI_Tag, MPI_Error - поля статуса принятого сообщения.

**Основные операции MPI**

MPI_Init(ierr), MPI_Finalize(ierr) - обрамление головной программы.
MPI_Abort(comm,ierr) - завершение работы всей группы при ошибке.

MPI_Comm_Size(comm,NProc,ierr) - сколько всего процессов в группе comm?
MPI_Comm_Rank(comm,myProc,ierr) - каков лично мой номер внутри группы comm?

Пожалуй, уже пора изваять нашу первую MPI-программу. Как запустить ее «на счет», мы расскажем попозже.

```
Program Hello
Include 'mpif.h'
call MPI_Init(ierr)
call MPI_Comm_Size(MPI_Comm_World, NProc, ierr)      ! Сколько нас?
call MPI_Comm_Rank(MPI_Comm_World, myProc, ierr)     ! Кто я?
write(*,*) ' Hello, I am ',myProc, ' process of ',NProc,
```

```
&        ' in a group ',MPI_COMM_WORLD
      call MPI_Finalize(ierr)
      End
```

Забегая вперед, скажем, что выходной файл, полученный этой программой, будет содержать NProc строк. Например, при NProc=3 Вы увидите следующее:

```
Hello, I am 0 process of 3 in a group 91
Hello, I am 2 process of 3 in a group 91
Hello, I am 1 process of 3 in a group 91
```

Порядок строк может быть и иным, ведь процессы трудились асинхронно!

**"Point-to-Point" операции MPI (самые главные)**

```
MPI_Send(Buf,Cnt,Type,Whom,Tag,comm,ierr)
MPI_Recv(Buf,Cnt,Type,From,Tag,comm,status,ierr)
```

Процесс myProc (тот, кто выполняет операцию!) посылает процессу Whom (принимает от процесса From) Cnt штук данных типа Type в массив Buf с тегом Tag. В качестве From и Tag могут фигурировать джокеры MPI_Any_Source, MPI_Any_Tag. Сие означает «принимаю от кого угодно» и «принимаю с любым тегом».

Кстати, почему размер сообщения измеряется в штуках, а не в байтах, словах или в прочих привычных нам единицах? Да просто потому, что один и тот же тип данных в общем случае требует разного количества памяти на разных машинах! Более того, мы избавлены и от забот по преобразованию чисел из одного внутримашинного представления в другое.

Далее, comm – целочисленный дескриптор группы (в терминологии MPI – коммуникатор), к которой принадлежит процесс Whom (From). Напомним, что внутри каждой группы нумерация процессов независимая. Обычно в качестве коммуникатора comm используют MPI_Comm_World – группу, содержащую все запущенные процессы.

Status – это массив из 3 целочисленных полей ( integer status(3) ), содержащих основные атрибуты последнего принятого процессом сообщения:

```
Status(MPI_Source)   - от кого оно принято;
Status(MPI_Tag)      - с каким Tag-ом оно передавалось;
Status(MPI_Error)    - были ли ошибки при передаче.
```

В переменную ierr MPI запишет 0, если операция выполнена нормально, иначе там будет ненулевой код ошибки. Вообще-то этот параметр присутствует практически во всех MPI-операциях и всегда является последним в списке параметров. Поскольку программистам вообще свойственно игнорировать проверки кодов ошибок, и дабы упростить изложение, мы в дальнейшем про него забудем. Тем более, что любители языка С этого параметра не видят вообще, он спрятан в выходное значение соответствующей функции. Так, аналогом фортранного MPI_Send будет С - функция

```
int MPI_Send(void* buf, int cnt, MPI_Datatype type, int whom, int tag, MPI_Comm comm)
```

**Коллективные операции MPI**

MPI_Barrier(comm)
MPI_Bcast(Buf,Cnt,Type,root,comm)
MPI_Gather(Sbuf,Scnt,Styp,Rbuf,Rcnt,Rtyp,root,comm)
MPI_Scatter(Sbuf,Scnt,Styp,Rbuf,Rcnt,Rtyp,root,comm)
MPI_Reduce(Sbuf,Rbuf,Cnt,Type,Op,root,comm)
MPI_AllReduce(Sbuf,Rbuf,Cnt,Type,Op,comm)

Здесь процесс с номером root инициирует операцию, в то время как остальные процессы «работают на подхвате».

Op = (MPI_Min, MPI_Max, MPI_Sum, MPI_Prod,...)

Важно отметить, что коллективные операции исполняются ВСЕМИ ПРОЦЕССАМИ ГРУППЫ! Но исполняются по-разному, в зависимости от того, является процесс главным в коллективе или нет!

Вообще-то очевидно, что эти операции можно построить самому из двух базовых, SEND и RECV, но для нашего удобства в MPI есть более 200 различных операций. В назидательных целях изобразим самодельную программу MPI_Bcast:

```
Subroutine MPI_Bcast(Buf,Cnt,Type,root,comm)   ! Передает Cnt штук данных
Dimension Buf(*)                               ! типа Type, лежащих в BUF,
Integer Cnt,Type,root,comm,newcomm,status(3)   ! от процесса root всем прочим
Include 'mpif.h'
call MPI_Comm_Dup(comm,newcomm)                ! Создаем временную группу
call MPI_Comm_Size(newcomm,NProc)              ! и работаем внутри нее!
call MPI_Comm_Rank(newcomm,myProc)
if (myProc.eq.root) then
  do k=0,NProc-1                               ! Наиболее тупым способом,
    call MPI_Send(Buf,Cnt,Type,k,0,newcomm)
  enddo                                        ! сам себе - тоже посылал!
endif
call MPI_Recv(Buf,Cnt,Type, root, 0,newcomm, status)
call MPI_Comm_Free(newcomm)                    ! Уничтожим временную группу
end
```

Обратите внимание на то, что мы пользуемся временным коммуникатором newcomm вместо исходного коммуникатора comm. Это необходимо для того, чтобы Send, испущенный внутри этой программы, не нарвался на Recv, испущенный вне ее. Такая техника использования временных, рабочих групп процессов характерна для библиотечных программ, желающих скрыть от внешнего мира свои внутренние межпроцессные коммуникации.

Далее мы увидим пример использования операции MPI_AllReduce, а за разъяснением смысла других коллективных операций отсылаем читателя в Интернет.

Вот этот пример: коллективное интегрирование функции одной переменной F(x).

```
Program Cooperative_Integration
Include 'mpif.h'
external F                                          ! Интегрируемая функция
data A/0/,  B/1/                                    ! Пределы интегрирования
call MPI_Init
call MPI_Comm_Size(MPI_Comm_World, NProc)           ! Сколько нас?
call MPI_Comm_Rank(MPI_Comm_World, myProc)          ! Кто я?
dx=(B-A)/NProc                                      ! Делим интервал поровну
a1=A+myProc*dx                                      ! между всеми процессами группы
b1=a1+dx
s1=Common_Integration( F, a1, b1 )                  ! - обычная библиотечная программа
call MPI_AllReduce(s1,S,1,MPI_Real, MPI_Sum, MPI_Comm_World)
if(myProc.eq.0) write(*,*)' Integral=',S
call MPI_Finalize
End
```

Здесь каждый процесс интегрирует, как умеет, F(x) на своем подинтервале, используя обычную, не распараллеленную процедуру Common_Integration. А операция MPI_AllReduce делает главную работу – суммирует эти подинтегральчики, живущие в переменной S1 каждого процесса, помещая результат суммирования в переменную S. Откуда это видно – да потому, что параметр OP в ней равен MPI_Sum.

Печать, и вообще весь вывод, естественно выполняет кто-то один, и очевидно это должен быть процесс 0, ведь наша программа должна правильно работать при любом NProc, в частности, при NProc=1, а только 0-й процесс есть всегда, при любом составе группы процессов, вовлеченных в решение задачи!

Что же касается ввода, то тут чуть сложнее. Ясно, что файлы с входной информацией должны принадлежать кому-то одному, стало быть 0-му процессу в основной группе MPI_COMM_WORLD. Прочитав из файла, он должен поделиться прочитанным со своими партнерами:

```
Real*4  array(100)
. . .
if(myProc.eq.0) then
  open(1,file='input.dat',...)
  read(1) array
  close(1)
endif
call MPI_Bcast(array,100,MPI_REAL,0,comm)
```

Это типичная техника организации ввода начальных данных при распараллеливании программы. Добавляются две вещи: условный переход и MPI_Bcast.

Столь же доступным примером является программа умножения матриц, где все элементы матрицы-произведения можно вычислять параллельно, т.е. S из закона Амдаля, так же как и в предыдущем примере, очевидно близко к 0.
Вот "нераспараллеленный" вариант:

```
      program mumu                    ! matrix multiplication  C = A * B
      parameter (N=400)               ! matrix dimension
      real*8 A(N,N),B(N,N),C(N,N)
      real*8 t
C...  мы опустили формирование исходных матриц A и B
      do i=1,N
        do j=1,N
          t=0.0
          do k=1,N
            t=t+A(i,k)*B(k,j)
          enddo
          C(i,j)=t
        enddo
      enddo
      end
```

Теперь попытаемся проделать эту же работу коллективом из нескольких процессов. Процессы, пронумерованные от 0 до P-1, исполняют один и тот же программный код, используя независимо работающие процессоры. Процесс 0 распределяет работу между всеми исполнителями, пересылая им обе исходные матрицы A и B. Каждый исполнитель (в том числе и сам процесс 0) вычисляет "свои" столбцы матрицы C, после чего пересылает результат своей работы обратно процессу 0.

```
      program MUMU                    ! matrix multiplication (parallel version)
      parameter (N=400)               ! matrix dimension
      include 'mpif.h'                ! Здесь описаны нужные нам MPI-объекты
      integer status(3)               ! Важно сказать, что это массив!
      integer comm,typ,tag,myProcess,P
      data tag/0/, typ/MPI_DOUBLE_PRECISION/, comm/MPI_COMM_WORLD/
      real*8 A(N,N),B(N,N),C(N,N)
      real*8 t

C...  инициализация MPI: запрос номера "своего" процесса

      call MPI_Init
      call MPI_Comm_rank(comm,myProcess)      ! Кто я?
      call MPI_Comm_size(comm,P)              ! Сколько всего нас?

C...  Процесс 0 рассылает остальным обе исходные матрицы
C...       (мы опустили их формирование)
      call MPI_Bcast(A,N*N,typ, 0,comm)
      call MPI_Bcast(B,N*N,typ, 0,comm)

      nc=N/P                          ! Сколько столбцов матрицы C должен я посчитать?
      nrest=mod(N,P)                  ! (а надо поделить их приблизительно поровну)
      if(myProcess.lt.nrest) then
        nc1=1+(nc+1)*myProcess        ! Это номер первого столбца
        nc2=nc1+nc                    ! А это номер последнего
      else                                    ! Выглядит крутовато,
```

```
      nc1=1+(nc+1)*nrest+nc*(myProcess-nrest)      ! зато матрица делится
      nc2=nc1+nc-1                                 ! почти поровну,
    endif                                          ! с точностью до одного столбца!

C...  Все начали трудиться, каждый над своей частью матрицы C ...

    write(*,*) ' Process',myProcess,'  of',P,
   -       '  started for columns from ',nc1,' till ',nc2

    do i=1,N
      do j=nc1,nc2                   ! Каждый вычисляет только свою часть матрицы C
        t=0.0
        do k=1,N
          t=t+A(i,k)*B(k,j)
        enddo
        C(i,j)=t
      enddo
    enddo

C...  Объединяем результаты в памяти процесса 0
C     (поскольку матрицы хранятся в памяти по столбцам,
C     подряд идущие столбцы можно пересылать за одно обращение!)

    if(myProcess.eq.0) then
      do i=1,P-1                     ! 0-й процесс обращается ко всем остальным
        if(i.lt.nrest) then
          nc1=1+(nc+1)*i             ! Ему надо вспомнить, кто какие столбцы считал
          k=nc+1
        else
          nc1=1+(nc+1)*nrest+nc*(i-nrest)
          k=nc                       ! (Этот фрагмент уже фигурировал выше)
        endif
        call MPI_Recv(C(1,nc1),N*k,typ,i,tag,comm,status)
      enddo
    else                             ! А ненулевые процессы это и так знают
      call MPI_Send(C(1,nc1),N*(nc2-nc1+1),typ,0,tag,comm)
    endif

    write(*,*) ' Process',myProcess,'  finished.'
    call MPI_Finalize                ! Ну, слава богу, все...
    end
```

Итак, мы видим, что даже в простейших случаях распараллеливание программы требует изрядных усилий. Более того, в ходе вычислений, как правило, необходимы межпроцессные коммуникации, которые могут вообще "съесть" весь эффект от распараллеливания. Здесь все зависит от соотношения цены программы и стоимости Вашего труда:

Если программа легко распараллеливается - почему бы не сделать это?

Если программа не очень нужная - стоит ли мучиться?

Если программа просто незаменима - может быть, стоит потрудиться?

Еще одно замечание. Интуиция подсказывает, что достаточно легко могут быть распараллелены так называемые Монте-Карловские программы, где вычислительной обработке подвергаются независимые события, сгенерированные с помощью датчика случайных чисел. Здесь важно обеспечить, чтобы каждый из параллельно работающих процессов получал свою, независимую от остальных процессов, серию случайных чисел. Для этого каждый из процессов, начиная свою работу, должен как-то по-своему инициализировать датчик.

На наш взгляд, идеальным датчиком для использования в распараллеленной программе является датчик, предложенный G.Marsaglia, способный выдавать до 32000 независимых серий равномерно распределенной на [0,1] случайной величины. Лучше всего инициализировать серию номером своего процесса:

```
. . .
call MPI_Comm_rank(MPI_COMM_WORLD,myProcess)    ! Кто я?
call RandomInitiate(myProcess,myProcess)        ! Начинаем свою серию
. . .
R = Random(1)              ! Real*8  uniform-distributed number on [0,1]
```

Датчик входит в состав нашей библиотеки JINRLIB. Немаловажным обстоятельством является то, что это самый быстрый из известных нам датчиков: всего 5 сложений и ни одного умножения с плавающей запятой!

Кстати: использование пакета MPI вовсе не уменьшит мобильность Вашей программы. На машинах, где нет MPI, Вы можете использовать заглушку:

файл mpif.h:

```
parameter(MPI_COMM_WORLD=0)
parameter(MPI_DOUBLE_PRECISION=0)
parameter(MPI_STATUS_SIZE=10)
```

файл mpi.for:

```
subroutine MPI_Comm_rank(comm,myProcess,ierr)
myProcess=0          ! наш процесс имеет номер 0
return
subroutine MPI_Comm_size(comm,NProc,ierr)
NProc=1              ! и он единственный
return
. . .                ! все остальные "MPI-программы" - пустые!!!
```

Применение этой заглушки позволит запускать Вашу программу в однопроцессном режиме, не меняя ее текста.

**Как заставить работать MPI-программу на нашем кластере HybriLIT?**

Во-первых, Вы должны получить доступ к пакету MPI. Для динамического изменения переменных окружения на HybriLIT установлен пакет MODULES, который позволяет

пользователю изменять свой список доступных компиляторов. Для работы с GNU или Intel-компиляторами добавьте модули:

GNU:

module add openmpi/1.6.5

или:

module add openmpi/1.8.1

Intel:

module add intel-2013.1.046

Во-вторых, вместо трансляторов f77 и cc вызывайте:

GNU:

mpif77 example.f mpicc example.c

Intel:

mpiifort example.f mpiicc example.c

В-третьих, при запуске программы указывайте, на сколько процессов Вы хотите ее распараллелить.

mpiexec –n 3 a.out - в данном случае на троих

Если вызовете без параметров - будет работать в одиночку.

На HybriLIT команду mpiexec можно использовать в интерактивном режиме только для запуска тестовых задач (не больше 5 мин. счетного времени). В счетном режиме запуск задачи на счет осуществляется планировщиком SLURM (Simple Linux Utility for Resource Management) с помощью команды:

sbatch s

где s – имя заранее заготовленного Вами script-файла, содержащего «паспорт задачи». Вот пример простейшего паспорта задачи, в котором из превеликого множества возможных slurm-директив фигурируют только самые необходимые:

```
#!/bin/sh
#SBATCH -p cpu
#SBATCH -t 60
#SBATCH -n 3
mpiexec a.out
```

Здесь исполняемый файл a.out, изготовленный компилятором, отправляется во входную очередь задач, предназначенных для счета на cpu-подмножестве нашего кластера. Задача требует 3 процессора (точнее – она запускается на 3 процесса). Задача получит уникальный номер во входной очереди. Пусть это будет 1234. Тогда листинг задачи будет оформлен как файл slurm-1234.out в той же Вашей директории, где находился ее исполняемый файл a.out.

Кстати - ничто не мешает Вам указывать число процессов большее, чем количество имеющихся в наличии процессоров! Просто Ваши процессы будут простаивать в очереди к процессорам, напрасно расходуя ресурсы системы.

Наши эксперименты с программой MUMU при N=3000 и разным числом процессов P показали, что при P>24 уменьшения времени уже нет. Конечно, эти цифры характерны только для этой программы.

Действительно, в этой программе не требуется межпроцессных коммуникаций во время счета, обмен информацией требуется только в начале и в конце работы. Поэтому, пока системе хватает ресурсов памяти, ускорение и должно линейно зависеть от числа процессов (три землекопа выкопают ту же самую яму втрое быстрее, чем один. Если же все землекопы сразу в одну яму не поместятся - они будут только мешать друг другу!)

Ниже приведены результаты проведенных экспериментов с программой умножения матриц на кластере HybriLIT. Компилятор Intel Fortran вызывался с опциями оптимизации O0, O1, O2, O3 (Рис.1a) и дополнительными опциями march=core-avx-i, march=core2 и Ofast (Рис.1b). Сравнение результатов показывает, что для данной программы наименьшее время счета получается при опции –O3 и числе параллельных процессов 24. Использование дополнительных опций оптимизации к уменьшению времени счета не приводит.

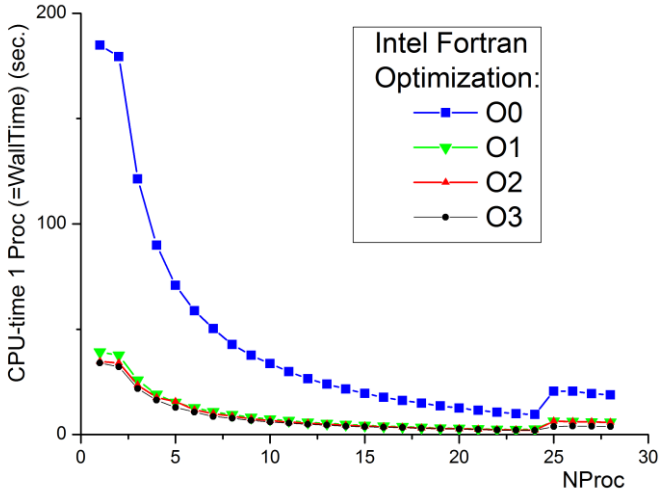


Рис.1a. Зависимость времени работы программы от числа параллельных процессов и уровней оптимизации O1-O3 для Intel Фортрана.

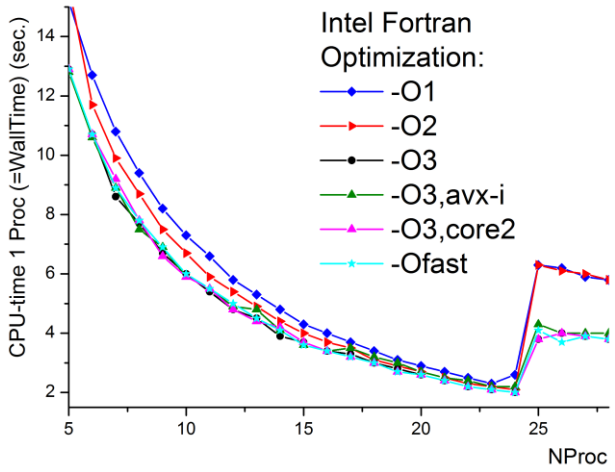


Рис.1b. Зависимость времени работы программы от числа параллельных процессов и дополнительных опций оптимизации для Intel Фортрана.

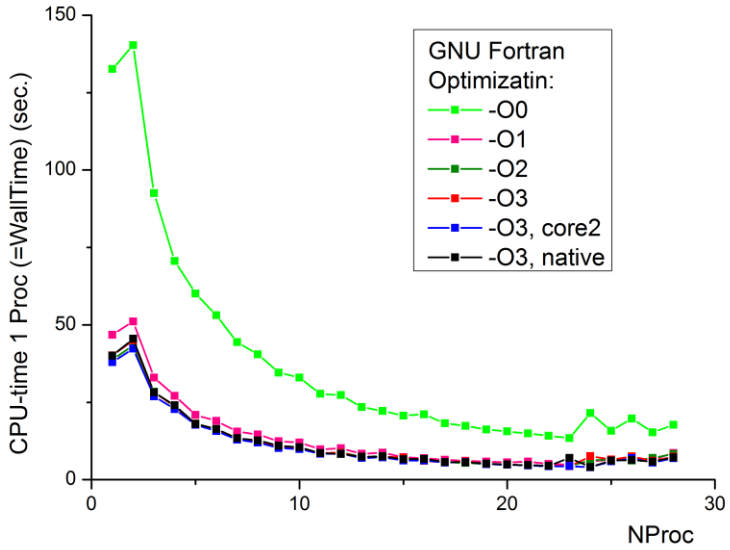


Рис.2. Зависимость времени работы программы от числа параллельных процессов и опций оптимизации для GNU Фортрана.

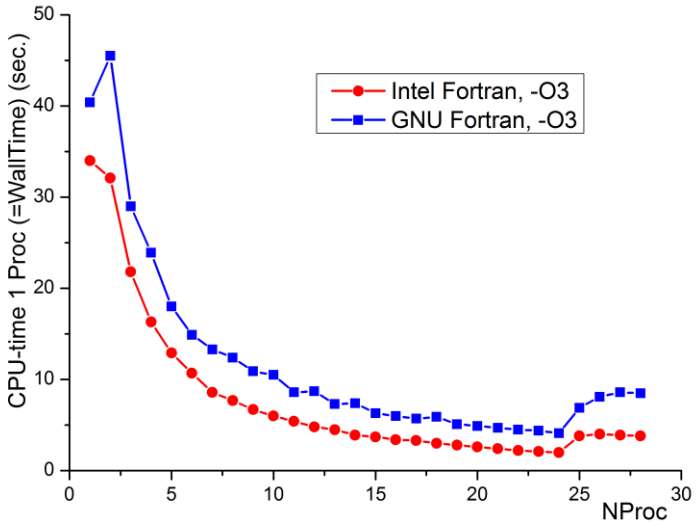


Рис.3. Зависимость времени работы программы от числа параллельных процессов для GNU и Intel Фортрана с опцией оптимизации -O3.

Аналогичные результаты получены для GNU-фотрана (Рис.2). На Рис.3 можно сравнить время работы программы для этих двух компиляторов при одинаковой опции оптимизации (-O3).

Некоторое увеличение времени счета при числе параллельных процессов больше 24 объясняется архитектурой HybriLIT: число ядер в вычислительных узлах равно 24, и пересылка данных между процессами, выполняющимися на разных узлах кластера, требует больше времени.

**Литература**


1. А.П.Сапожников. Как нам распараллелить программу. Информационный бюллетень ЛИТ №2 [43], 4-8160, Дубна, ОИЯИ, 2003, с.37-50.
2. http://hybrilit.jinr.ru
3. http://parallel.ru
4. Open MPI - свободно распространяемая реализация MPI для гетерогенных кластеров из UNIX-машин: https://www.open-mpi.org.
5. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. БХВ-Петербург, 2002.
6. MPI: The complete Reference. MIT Press, Cambridge, Massachusetts, 1997.