Министерство образования и науки Российской Федерации
Федеральное агентство по образованию
Томский государственный университет

Утверждаю
Декан Механико-математического
факультета, профессор
_____ Старченко А.В.
“___” _____ 2014 г.

Методические указания

# Распараллеливание явных и неявных разностных схем, эффективность параллельных и последовательных программ

Томск 2014

РАССМОТРЕНЫ И УТВЕРЖДЕНЫ
Методической комиссией механико-математического факультета

Протокол от _____
Председатель комиссии __О.П. Федорова

В методических указаниях рассматриваются методы численного решения уравнения теплопроводности на многопроцессорной ЭВМ. Подробно описывается распараллеливание явных и неявных разностных схем. Для случая неявной разностной схемы описывается распараллеливание и применение для решения систем линейных алгебраических уравнений двух методов: метода Зейделя и метода сопряженных градиентов. Так же обсуждаются другие возможности ускорения вычислений, учитывающие особенности языка программирования и используемой вычислительной техники.

Методические указания разработаны для студентов и аспирантов механико-математического факультета.

**Оглавление:**

**Распараллеливание явных и неявных разностных схем.**

Задачи математического моделирования требуют большого количества компьютерного времени. Одним из способов преодоления этой трудности является использование многопроцессорных вычислительных систем. В данном методическом пособии будут рассматриваться многопроцессорные вычислительные системы с распределенной памятью, так как в настоящее время супер-ЭВМ, которыми мы располагаем, построены именно по такому принципу. Если говорить о списке самых мощных компьютеров Top-500, то 72% от общего числа суперкомпьютеров составляют именно кластерные системы. Это связано с сравнительно небольшой стоимостью высокопроизводительных систем, построенных по данному принципу.

Наиболее общим подходом распределения вычислительной загрузки между вычислительными узлами является распределение вычислительных областей на подобласти. Так называемый принцип геометрической декомпозиции. Причем самым простым способом решения этих разделенных на подобласти задач является применение явных разностных схем. Однако такие схемы отличаются медленной скоростью сходимости для стационарных краевых задач и малым шагом интегрирования – для нестационарных задач, ограничения на который накладывает условие устойчивости.

Неявные схемы относительно свободны от этого недостатка. Однако распараллеливание неявных схем с глобальной пространственной связанностью данных и требованием решения линейных систем большой размерности становиться весьма нетривиальной задачей. Для решения системы линейных алгебраических уравнений необходимо выбрать алгоритм, который при распараллеливании сохранит свойства последовательного и покажет хорошие результаты (высокую эффективность). Конечно решение СЛАУ требует больших вычислительных затрат, однако выигрыш проявляется в свойствах неявных схем. Как правило, все неявные схемы абсолютно устойчивы и поэтому шаг по времени можно взять довольно большой. В то время как при использовании явных схем встречаются серьезные ограничения на шаг по времени.

Распараллеливание и само знакомство с явными и неявными схемами будет осуществляться на примере численного решения уравнения теплопроводности в единичном квадрате с граничными условиями первого рода и заданными начальными условиями. Задача состоит в следующем: определить распределение скалярной величины $T(t,x,y)$ для случая установления процесса теплопередачи $(t \to \infty)$:

$$\frac{\partial T}{\partial t} = \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right); \quad (1)$$

$$T\big|_G = T_b(x,y);\ T\big|_{t=0} = 0;$$
$$0 \le x, y \le 1;\ 0 \le t \le T; \quad (2)$$

Численное решение данной задачи будем искать с использованием метода конечных разностей. Для области исследования построим равномерную сетку. Будут использоваться следующие обозначения $T(t_n, x_i, y_j) = T^n_{i,j}$. Верхний индекс определяет принадлежность временному слою, а нижние принадлежность к $(i, j)$-ому узлу сетки [5].

Приведем конечно-разностные формулы для аппроксимации дифференциальных операторов:

$$\left(\frac{\partial T}{\partial t}\right)^n_{i,j} \approx \frac{T^{n+1}_{i,j} - T^n_{i,j}}{\tau};$$

$$\left(\frac{\partial^2 T}{\partial x^2}\right)^n_{i,j} \approx \lambda\left(\frac{T^{n+1}_{i+1,j} - 2\cdot T^{n+1}_{i,j} + T^{n+1}_{i-1,j}}{h_x^2}\right) + (1-\lambda)\left(\frac{T^n_{i+1,j} - 2\cdot T^n_{i,j} + T^n_{i-1,j}}{h_x^2}\right);$$

$$\left(\frac{\partial^2 T}{\partial y^2}\right)^n_{i,j} \approx \lambda\left(\frac{T^{n+1}_{i,j+1} - 2\cdot T^{n+1}_{i,j} + T^{n+1}_{i,j-1}}{h_y^2}\right) + (1-\lambda)\left(\frac{T^n_{i,j+1} - 2\cdot T^n_{i,j} + T^n_{i,j-1}}{h_y^2}\right);$$

Здесь $\lambda$ - коэффициент определяющий вид схемы ($0 \le \lambda \le 1$), если $\lambda = 0$ схема явная, если $\lambda = 1$ схема неявная. Это два крайних случая, при $0 < \lambda < 1$ получаем смешенные схемы. Особого внимания заслуживает схема Кранка-Николсона, для которой $\lambda = 0.5$ [6].

Далее применив, эти конечно разностные формулы к дифференциальному уравнению получим конечно-разностный аналог дифференциального уравнения – систему линейных алгебраических уравнений. Начальные и граничные условия на используемой равномерной сетке аппроксимируются точно.

$$
\begin{cases}
\dfrac{T_{i,j}^{n+1}-T_{i,j}^{n}}{\tau}=\alpha\left(\begin{array}{l}
\lambda\left(\dfrac{T_{i+1,j}^{n+1}-2\cdot T_{i,j}^{n+1}+T_{i-1,j}^{n+1}}{h_x^2}\right)+\\
(1-\lambda)\left(\dfrac{T_{i+1,j}^{n}-2\cdot T_{i,j}^{n}+T_{i-1,j}^{n}}{h_x^2}\right)+\\
\lambda\left(\dfrac{T_{i,j+1}^{n+1}-2\cdot T_{i,j}^{n+1}+T_{i,j-1}^{n+1}}{h_y^2}\right)+\\
(1-\lambda)\left(\dfrac{T_{i,j+1}^{n}-2\cdot T_{i,j}^{n}+T_{i,j-1}^{n}}{h_y^2}\right)
\end{array}\right)\\
\qquad i=1,Nx-1;\;\; j=1,Ny-1\\
T_{0,j}^{n}=T_b(x_0,y_j);\quad T_{Nx,j}^{n}=T_b(x_{Nx},y_j);\quad j=0,Ny;\\
T_{i,0}^{n}=T_b(x_i,y_0);\quad T_{i,Ny}^{n}=T_b(x_i,y_{Ny});\quad i=0,Nx;\\
T_{i,j}^{0}=0;\quad i=0,Nx;\;\; j=0,Ny;
\end{cases}
\tag{3}
$$

Далее необходимо определиться, какую схему применять для решения задачи. Если используется грубая сетка с большим шагом по координатам и ограничение на шаг по времени накладываемое условием устойчивости позволяет делать достаточно крупные шаги по времени, то выгодно применять явную разностную схему, которая очень проста в своей реализации. Если же сетка мелкая и из условной устойчивости требуется небольшой шаг по времени, то выгодно применять неявные схемы [6].

Какую разностную схему получим после аппроксимации, явную неявную или смешанную, зависит от того, чему равен коэффициент $\lambda$ при аппроксимации производных по пространству [6].

Далее будет рассмотрена параллельная реализация явной разностной схемы для решения задачи теплопроводности.

**Явная схема.**

Пусть $\lambda = 0$. То есть для аппроксимации производных по пространству используются значения переменной $T_{i,j}$ с $n$-го временного слоя. Тогда из (3):

$$\begin{cases} T_{i,j}^{n+1} = T_{i,j}^{n} + \alpha \left( \dfrac{T_{i+1,j}^{n} - 2 \cdot T_{i,j}^{n} + T_{i-1,j}^{n}}{h_x^2} \right) + \left( \dfrac{T_{i,j+1}^{n} - 2 \cdot T_{i,j}^{n} + T_{i,j-1}^{n}}{h_y^2} \right) \\ T_{0,j}^{n} = T_b(x_0, y_j); \quad T_{Nx,j}^{n} = T_b(x_{Nx}, y_j); \quad j = 0, Ny; \\ T_{i,0}^{n} = T_b(x_i, y_0); \quad T_{i,Ny}^{n} = T_b(x_i, y_{Ny}); \quad i = 0, Nx; \\ T_{i,j}^{0} = 0; \quad i = 0, Nx; \quad j = 0, Ny; \end{cases} \qquad (4)$$

Таким образом, получена явная формула для вычислений значений неизвестной функции на новом временном слое. Для дальнейшего понимания перепишем полученную формулу в следующем виде:

$$T_{i,j}^{n+1} = (1 + ap) T_{i,j}^{n} + ae \cdot T_{i+1,j}^{n} + aw \cdot T_{i-1,j}^{n} + an \cdot T_{i,j+1}^{n} + as \cdot T_{i,j-1}^{n}; \qquad (5)$$

Коэффициенты $ap$, $ae$, $aw$, $an$, $as$ легко определяются из формулы (4).

$$ae = \frac{\lambda \cdot \tau}{h_x^2}; \; aw = \frac{\lambda \cdot \tau}{h_x^2}; \; an = \frac{\lambda \cdot \tau}{h_y^2}; \; as = \frac{\lambda \cdot \tau}{h_y^2};$$

$$ap = -\left( ae + aw + an + as \right); \quad i = 1, Nx - 1; \quad j = 1, Ny - 1;$$

Эта формула легко программируется и очень проста для применения:

```
Do i=1,Nx-1
   Do j=1,Ny-1
      Tnew(i,j)=(1+ap)*T(i,j)+ae*T(i+1,j)+aw*T(i-1,j)+an*T(i,j+1)+as*T(i,j-1)
   End do
End do
```

Далее обсудим возможности распараллеливания решения поставленной задачи. В рассматриваемом примере возможны два различных способа разделения данных – *одномерная* или *ленточная* схема или *двухмерное* или *блочное* разбиение вычислительной сетки. Из практики параллельных вычислений хорошо известно, что, в целом, более эффективная двумерная декомпозиция расчетной области теряет свои преимущества перед одномерной в том случае, если число процессоров параллельного компьютера не велико. А так как рассматривается случай кластерной системы с малым числом процессоров, то внимание уделим одномерной декомпозиции по одному из координатных направления. Для определенности декомпозицию расчетной области проведем по индексу $j$ (рис. 1), хотя на практике выбор координаты для разбиения определяется особенностями языка программирования или способом хранением данных в оперативной памяти. Совокупность узлов расчетной сетки, попавших на один процессорный элемент, будем называть полосой.

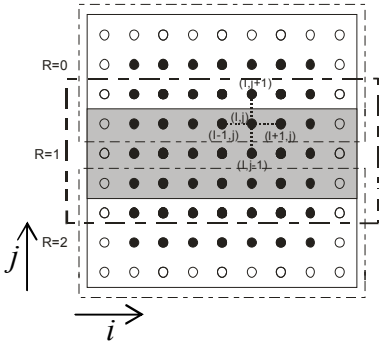



Рис. 1. Одномерная декомпозиция расчетной области (кружки представляют граничные узлы сетки).

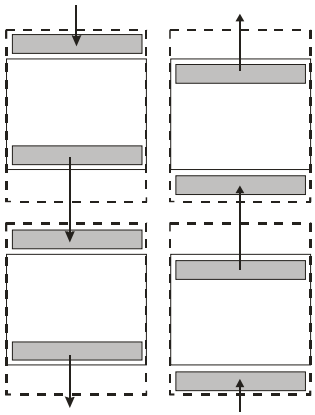


Рис. 2. Схема обменов между вычислительными узлами.

Данные распределены, следующий этап построения параллельной программы – проектирование коммуникаций. При аппроксимации дифференциальной задачи использовался шаблон крест (рис. 1), поэтому для организации вычислений в приграничных узлах каждого процессорного элемента требуется продублировать граничные строки предшествующей и следующей полос вычислительной сетки, которые по определению полосы располагаются на соседних вычислительных узлах. Продублированные граничные строки полос используются только при проведении расчетов, пересчет же

этих строк происходит в полосах своего исходного месторасположения. Тем самым дублирование граничных строк должно осуществляться перед началом выполнения каждой очередной итерации метода. [1]

Процедура обмена сеточных значений граничных строк между соседними процессорами может быть разделена на две последовательные операции, во время первой из которых каждый процессор передает свою нижнюю граничную строку следующему процессору и принимает такую же строку от предыдущего процессора (рис. 2). Вторая часть передачи строк выполняется в обратном направлении: процессоры передают свои верхние граничные строки своим предыдущим соседям и принимают переданные строки от следующих процессоров [1].

```
      PROGRAM Example1
c  Решение двумерного уравнения теплопроводности в единичном
c  квадрате с использованием явной разностной схемы.
       Implicit none
       Include 'mpif.h'
c  Параметры:
c  Nx,Ny - количество точек области в каждом направлении,
c  m - количество точек на процессор.
c  T,Tnew - численное решение задачи.
       Integer i,j,Ny,Nx,m
       Integer comm,ierr,size,rank,left,right,tag,status(100)
       Parameter (Nx=720,Ny=Nx)
       Double precision ap,ae,aw,an,as,hx,hy,Lx,Ly,
      1                 T(0:Nx,0:Ny),Tnew(0:Nx,0:Ny),alfa,
      2                 tau,time,timefin,timeStart,timeStop
      3                 timeStart1,timeStop1
       Parameter (timefin=1.0d3,Lx=1.0d0,Ly=1.0d0,
      1           hx=Lx/Nx,hy=Ly/Ny,alfa=1.0d-5)
c  Инициализация, определение числа выделенных процессоров (size)
c  и номера текущего процесса (rank)
       Call MPI_INIT(ierr)
       Call MPI_comm_size(mpi_comm_world,size,ierr)
       Call MPI_comm_rank(mpi_comm_world,rank,ierr)
       comm = mpi_comm_world
c  Определение номеров процессоров (соседей) сверху и снизу.
c  Если таковые отсутствуют то им присваивается значение
c  MPI_PROC_NULL (для них коммуникационные операции игнорируются)
       If (rank.eq.0) then
         left=MPI_PROC_NULL
       Else
         left=rank-1
       End if
       If (rank.eq.size-1) then
         right=MPI_PROC_NULL
       Else
         right=rank+1
```

```
        End if
c  Определение числа элементов обрабатываемых одним процессором.
        m=Ny/size
        If (rank.lt.(Nx-size*m)) then
          m=m+1
        End if
c  Задание начальных значений.
        Do i=0,Nx
          Do j=0,m
            T(i,j)=100.0d0
            Tnew(i,j)=100.0d0
          End do
        End do
c  Задание граничных условий.
        If (rank.eq.0) then
          Do i=0,Nx
            T(i,0)=10.0d0
          End do
        End if
        If (rank.eq.size-1) then
          Do i=0,Nx
            T(i,m)=10.0d0
          End do
        End if
        Do j=0,m
          T(0,j)=1.0d0
          T(Nx,j)=10.0d0
        End do
c  Подготовка к расчетам.
        tau = 0.2*(2*alfa*(1/hx**2+1/hy**2))
        ae=alfa*tau/(hx*hx)
        aw=alfa*tau/(hx*hx)
        an=alfa*tau/(hy*hy)
        as=alfa*tau/(hy*hy)
        ap=-(ae+aw+an+as)
        time = 0
        If (rank.eq.0) then
          open (2,file='rezult1.dat')
          timeStart = MPI_Wtime()
          timeStop1=0.0
        End if
c  Движение по временной оси.
        Do while (time.lt.timefin)
c  Расчет значений с нового временного слоя.
          Do i=1,Nx-1
            Do j=1,m-1
              Tnew(i,j) = T(i,j)*(1+ap) + T(i+1,j)*ae + T(i-1,j)*aw
     1                                  + T(i,j+1)*an + T(i,j-1)*as
            End do
          End do
          time =time + tau
c  Переприсваивание внутренних узлов области.
          Do i=1,Nx-1
```

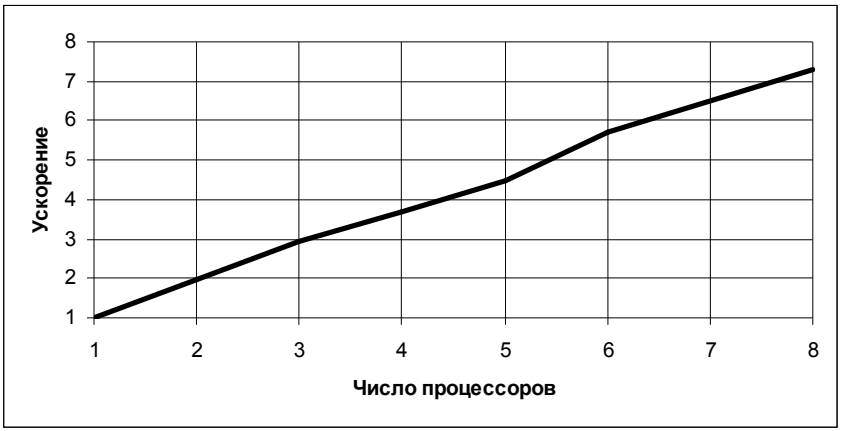
```
            Do j=1,m-1
              T(i,j)=Tnew(i,j)
            End do
          End do
c  Пересылка граничных значений на соседние процессоры.
          tag =100
          If (rank.eq.0) timeStart1 = MPI_Wtime()
          Call MPI_SENDRECV(T(0,1),Nx+1,MPI_double_precision,
      1                     left,tag,
      2                     T(0,0),Nx+1,MPI_double_precision,
      3                     left,tag,comm,status,ierr)
          Call MPI_SENDRECV (T(0,m-1),Nx+1,MPI_double_precision,
      1                      right,tag,
      2                      T(0,m),Nx+1,MPI_double_precision,
      3                      right, tag,comm,status,ierr)
          If (rank.eq.0) timeStop1= timeStop1+MPI_Wtime()-timeStart1
        End do
c  Фиксация времени окончания расчетов.
        If (rank.eq.0) timeStop=MPI_Wtime()-timeStart
c  Сбор результатов на нулевом процессоре.
        Call MPI_GATHER( T(0,1),(m)*(Nx+1),MPI_DOUBLE_PRECISION,
      1                  T(0,1),(m)*(Nx+1),MPI_DOUBLE_PRECISION,
      2                  0,comm,ierr)
        If (rank.eq.0) then
          Do i=0,Nx
            write (2,'(560f15.4)') (T(i,j),j=0,Ny)
          End do
          Write (2,'(a,i15,a,f15.9)') 'Size=',size,'  Tame=',TimeStop
          Write (2,'(a,f15.9,a,f15.9)') ' FTame=',time, ' tau=',tau
          Write (2,'(a,f15.9,a,f15.9)') '  Tsch=',TimeStop-TimeStop1,
      1                                 '  Tobm=',TimeStop1
        End if
        Call MPI_FINALIZE(ierr)
        End
```

В таблице приведено время работы программы, для разного числа процессоров при значении коэффициента теплопроводности $\alpha = 10^{-5}$. Расчеты проводились на равномерной сетке 720 на 720 (время приведено в секундах).

| **Число проц-в** / **Коэф-т тепл-ти** | **1** | **2** | **3** | **4** | **5** | **6** | **8** |
|---|---|---|---|---|---|---|---|
| $\alpha = 10^{-5}, \tau = 0.039$ | **7136** | **3644** | **2434** | **1943** | **1591** | **1249** | **981** |
| **Время счета** | **7135** | **3541** | **2365** | **1881** | **1504** | **1129** | **834** |
| **Время пересылок** | **0,66** | **102** | **67** | **61** | **86** | **119** | **146** |

Тестовые расчеты, проведенные для различных значений коэффициента теплопроводности, показали значительное увеличение времени счета при увеличении $\alpha$. Увеличение коэффициента теплопроводности на порядок влечет за собой десятикратное увеличение времени счета. Это объясняется уменьшением шага по времени, который вычисляется по формуле $\tau = \dfrac{0.2}{2 \cdot \alpha \left( \dfrac{1}{h_x^2} + \dfrac{1}{h_y^2} \right)}$, которая выбрана так, что бы всегда выполнялось условие устойчивости.

Далее приводиться график ускорения для значения коэффициента теплопроводности $\alpha = 10^{-5}$.




Видно, что ускорение меняется по линейному закону, что присуще всем явным разностным схемам.

**Неявная схема.**

Пусть $\lambda = 1$. То есть для аппроксимации производных по пространству используются значения переменной $T_{i,j}$ с $n+1$-го временного слоя. Тогда получаем:

$$\begin{cases} T_{i,j}^{n+1} = T_{i,j}^{n} + \dfrac{\alpha}{\tau}\left( \dfrac{\left(T_{i+1,j}^{n+1} - 2T_{i,j}^{n+1} + T_{i-1,j}^{n+1}\right)}{hx^2} + \dfrac{\left(T_{i,j+1}^{n+1} - 2T_{i,j}^{n+1} + T_{i,j-1}^{n+1}\right)}{hy^2} \right); \\ T_{0,j}^{n} = T_b(x_0, y_j); \quad T_{Nx,j}^{n} = T_b(x_{Nx}, y_j); \quad j = 0, Ny; \\ T_{i,0}^{n} = T_b(x_i, y_0); \quad T_{i,Ny}^{n} = T_b(x_i, y_{Ny}); \quad i = 0, Nx; \\ T_{i,j}^{0} = 0; \quad i = 0, Nx; \ j = 0, Ny; \end{cases} \tag{6}$$

Для большей наглядности перепишем полученные формулы в следующем виде

$$(1 - ap)T_{i,j}^{n+1} = ae \cdot T_{i+1,j}^{n+1} + aw \cdot T_{i-1,j}^{n+1} + an \cdot T_{i,j+1}^{n+1} + as \cdot T_{i,j-1}^{n+1} + b_{i,j}; \tag{7}$$

Коэффициенты $ap, ae, aw, an, as, b_{i,j}$ легко определяются из формулы (6).

$$ae = \frac{\lambda \cdot \tau}{h_x^2}; \ aw = \frac{\lambda \cdot \tau}{h_x^2}; \ an = \frac{\lambda \cdot \tau}{h_y^2}; \ as = \frac{\lambda \cdot \tau}{h_y^2}; b_{i,j} = T_{i,j}^{n}$$

$$ap = -\left(ae + aw + an + as\right); \quad i = 1, Nx - 1; \quad j = 1, Ny - 1;$$

$$i = 0; \quad ap = 1; \quad b_{i,j} = T_b(x_0, y_j); \quad ae = aw = as = an = 0;$$

$$i = Nx; \quad ap = 1; \quad b_{i,j} = T_b(x_{Nx}, y_j); \quad ae = aw = as = an = 0;$$

$$j = 0; \quad ap = 1; \quad b_{i,j} = T_b(x_i, y_0); \quad ae = aw = as = an = 0;$$

$$j = Ny; \quad ap = 1; \quad b_{i,j} = T_b(x_i, y_{Ny}); \quad ae = aw = as = an = 0;$$

В случае неявной разностной схемы вместо организации вычислений по готовой формуле необходимо решение системы линейных

алгебраических уравнений большой размерности с разреженной матрицей. Преимущества этого подхода были описаны выше, теперь непосредственно перейдем к выбору метода решения системы линейных алгебраических уравнений.

Прямые методы, как правило, не используются для решения систем линейных алгебраических уравнений, полученных после аппроксимации дифференциальной задачи, так как они имеют ряд существенных недостатков: вынуждают целиком хранить матрицу в оперативной памяти, что делает ее использование нерациональным, не уменьшают влияние погрешности округления, что становиться неприемлемым при решении плохообусловленных задач большой размерности. Поэтому, будем рассматривать итерационные методы решения СЛАУ. А именно, будет рассмотрено два метода, относящихся к различным классам, метод Зейделя и метод сопряженных градиентов [3,5].

**Метод Зейделя.**

Для рассматриваемой задачи формулы метода Зейделя примут следующий вид

$$\left(T_{i,j}^{n+1}\right)^{k+1} = \frac{1}{ap}\left(\begin{array}{l} b_{i,j} - ae\cdot\left(T_{i+1,j}^{n+1}\right)^{k+1} - aw\cdot\left(T_{i-1,j}^{n+1}\right)^{k} \\ \qquad - an\cdot\left(T_{i,j-1}^{n+1}\right)^{k} - as\cdot\left(T_{i,j+1}^{n+1}\right)^{k+1} \end{array}\right);$$

$$i = 0, \ldots, Nx; \quad j = 0, \ldots, Ny; \quad k = 0, 1, 2, \ldots$$

Где $n$ - номер временного слоя, а $k$ -номер итерации метода на $n$ -ом временном слое.

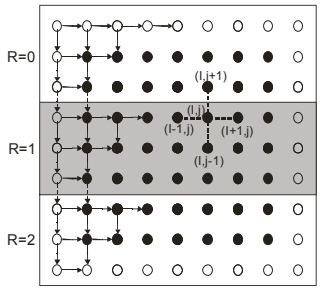



Рис. 3. Связи между узлами сетки в методе Гаусса-Зейделя.

Этот метод относиться к классу треугольных методов. В [5] показано, что метод Зейделя всегда сходится, если $A$ - симметричная, положительно определенная матрица. Причем можно утверждать, что метод Зейделя сходится со скоростью геометрической прогрессии со знаменателем $q < 1$, если выполняется условие диагонального преобладания, т.е.

$$|ae| + |aw| + |an| + |as| \le q\,|ap|, \quad q < 1;$$

Использование последних рассчитанных значений в методе Гаусса-

Зейделя увеличивает скорость сходимости, но ограничивает параллелизм алгоритма и требует синхронизации параллельных вычислений. На рис. 3 показаны связи, наложенные на узлы сетки в методе Гаусса-Зейделя. Получается, что второй процессор $(R=1)$ будет простаивать, пока не получит необходимые ему значения от первого, а последний процессор будет дожидаться пока все предыдущие не вычислят значения неизвестных из первого столбца (рис. 3). Та же неравномерность загрузки проявится и в конце итерации, когда первый процессор раньше остальных закончит вычислительную работу. Описанный процесс носит название волновой обработки данных [1].

Альтернативным подходом при расчете следующей итерации является ситуация, когда каждый процесс использует последующие значения $\left(T_{i,j}^{n+1}\right)^{k+1}$, которыми он сам располагает, и не дожидается когда ему будут присланы последние рассчитанные значения. Такой подход называют асинхронным. Он часто бывает эффективен, но при его использовании не представляется возможным проведение анализа сходимости численного метода. Если отвлечься от упорядоченного обхода узлов и выбрать какой-либо другой способ обхода, то также будет реализовываться метод Зейделя. Перспективным стал способ обхода узлов, который носит название красно-черное упорядочивание [1]. Он предполагает разделение всех узлов на 2 вида – красные и черные в шахматном порядке. Выбор такого способа разделения обусловлен выбором используемого шаблона, в соответствии с которым для расчета в красных узлах нужны значения в черных узлах и наоборот. И второе, этот вид упорядочивания дает наиболее сильные результаты сходимости [2]. На идейном уровне красно-черное упорядочивание обеспечивает перевод рекуррентных формул метода Зейделя в двухшаговое использование формул Якоби.

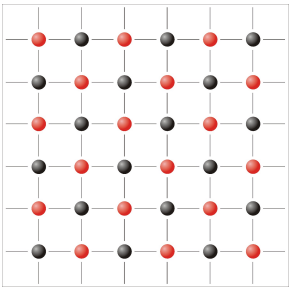


Рис. 4. Красно-черное упорядочивание.

```
      PROGRAM Example2
c  Решение двумерного уравнения теплопроводности в единичном
c  квадрате по методу Зейделя с красно-черным упорядочиванием.
       Implicit none
       Include 'mpif.h'
```

```
c  Параметры:
c  Nx,Ny - количество точек области в каждом направлении,
c  m - количество точек на процессор.
c  T,Tnew - численное решение задачи.
       Integer i,j,k,Ny,Nx,m
       Integer comm,ierr,size,rank,left,right,tag,status(100)
       Parameter (Nx=720,Ny=Nx)
       Double precision ap,ae,aw,an,as,b(0:Nx,0:Ny),err,gerr,re
      2             T(0:Nx,0:Ny),Tnew(0:Nx,0:Ny),alfa,hx,hy,Lx,Ly,
      3             tau,time,timefin,timeStart,timeStop,rez,grez,
      4             timeStart1,timeStop1
       Parameter (timefin=1.0d3,Lx=1.0d0,Ly=1.0d0,
      1           hx=Lx/Nx,hy=Ly/Ny,alfa=1.0d-5)
c  Инициализация, определение числа выделенных процессоров (size)
c  и номера текущего процесса (rank).
       Call MPI_INIT(ierr)
       Call MPI_comm_size(mpi_comm_world,size,ierr)
       Call MPI_comm_rank(mpi_comm_world,rank,ierr)
       comm = mpi_comm_world
c  Определение номеров процессоров (соседей) сверху и снизу.
c  Если таковые отсутствуют то им присваивается значение
c  MPI_PROC_NULL (для них коммуникационные операции игнорируются)
       If (rank.eq.0) then
         left=MPI_PROC_NULL
       Else
         left=rank-1
       End if
       If (rank.eq.size-1) then
         right=MPI_PROC_NULL
       Else
         right=rank+1
       End if
c  Определение числа элементов обрабатываемых одним процессором.
       m=Ny/size
       If (rank.lt.(Nx-size*m)) then
         m=m+1
       End if
c  Задание начальных значений.
       Do i=0,Nx
         Do j=0,m
           T(i,j)=100.0d0
           Tnew(i,j)=100.0d0
           b(i,j)=0.0d0
         End do
       End do
c  Задание граничных условий.
       If (rank.eq.0) then
         Do i=0,Nx
           T(i,0)=10.0d0
         End do
       End if
       If (rank.eq.size-1) then
         Do i=0,Nx
```

```
            T(i,m)=10.0d0
          End do
        End if
        Do j=0,m
          T(0,j)=1.0d0
          T(Nx,j)=10.0d0
        End do
c  Подготовка к расчетам.
        tau=2.0d0
        Do i=1,Nx-1
          Do j=1,m-1
            b(i,j)=T(i,j)
          End do
        End do
        ae=alfa*tau/(hx*hx)
        aw=alfa*tau/(hx*hx)
        an=alfa*tau/(hy*hy)
        as=alfa*tau/(hy*hy)
        ap=ae+aw+an+as

        If (rank.eq.0) then
          open (2,file='rezult1.dat')
          timeStart = MPI_Wtime()
          timeStop1 = 0.0
        End if
c  Движение по временной оси.
        Do while (time.lt.timefin)
          If (rank.eq.0) timeStart1 = MPI_Wtime()
          time =time + tau
          k=0
          gerr=10
          grez=10
c  Расчет значений с нового временного слоя.
c  На каждом временном шаге для уменьшения погрешности численного
c  метода реализуется несколько итераций.
          Do While ((gerr.gt.1.0d-5).or.( grez.gt.1.0d-5))
            k=k+1
            gerr=0.0d0
            err=0.0d0
            gerr=0.0d0
            err=0.0d0
c  Вычисление новых значений функции в "красных" узлах
c  по пятиточечной схеме.
          Do i=1,Nx-1
           Do j=1,m-1
            If (mod(i+j,2).eq.0) then
              Tnew(i,j) = (b(i,j)+ae*T(i+1,j)+aw*T(i-1,j)
     1                           +an*T(i,j+1)+as*T(i,j-1))/(1.0+ap)
            End if
           End do
          End do
c  Переприсваивание для внутренних "красных" узлов области.
          Do i=1,Nx-1
```

```
          Do j=1,m-1
           If (mod(i+j,2).eq.0) then
             err=err+abs(T(i,j)-Tnew(i,j))
             re=((1.0+ap)*Tnew(i,j)-ae*Tnew(i+1,j)-aw*Tnew(i-1,j)
     1                    -b(i,j) -an*Tnew(i,j+1)-as*Tnew(i,j-1))
             T(i,j)=Tnew(i,j)
             rez=re*re
           End if
          End do
         End do
c  Пересылка граничных значений на соседние процессоры.
         If (rank.eq.0) timeStart1 = MPI_Wtime()
         tag=10
         Call MPI_SENDRECV (T(0,1),Nx+1,MPI_double_precision,
     1                      left,tag,
     2                      T(0,0),Nx+1,MPI_double_precision,
     3                      left,tag,comm,status,ierr)

         Call MPI_SENDRECV ( T(0,m-1),Nx+1,MPI_double_precision,
     1                       right,tag,
     2                       T(0,m),Nx+1,MPI_double_precision,
     3                       right,tag,comm,status,ierr)
         If (rank.eq.0) timeStop1=timeStop1+MPI_Wtime()-timeStart1
c  Вычисление новых значений функции в "черных" узлах
c  по пятиточечной схеме.
         Do i=1,Nx-1
          Do j=1,m-1
           If (mod(i+j,2).ne.0) then
             Tnew(i,j) = (b(i,j)+ae*T(i+1,j)+aw*T(i-1,j)
     1                          +an*T(i,j+1)+as*T(i,j-1))/(1.0+ap)
           End if
          End do
         End do
c  Переприсваивание для внутренних "красных" узлов области.
         Do i=1,Nx-1
          Do j=1,m-1
           If (mod(i+j,2).ne.0) then
             err=err+abs(T(i,j)-Tnew(i,j))
             re=((1.0+ap)*Tnew(i,j)-ae*Tnew(i+1,j)-aw*Tnew(i-1,j)
     1                    -b(i,j) -an*Tnew(i,j+1)-as*Tnew(i,j-1))
             T(i,j)=Tnew(i,j)
             rez=re*re
           End if
          End do
         End do
         If (rank.eq.0) timeStart1 = MPI_Wtime()
         Call MPI_ALLREDUCE( err,gerr,1,MPI_DOUBLE_PRECISION,
     1                       MPI_SUM,comm,ierr )
         Call MPI_ALLREDUCE( rez,grez,1,MPI_DOUBLE_PRECISION,
     1                       MPI_SUM,comm,ierr )
c  Пересылка граничных значений на соседние процессоры.
         Call MPI_SENDRECV (T(0,1),Nx+1,MPI_double_precision,
     1                      left,tag,
```

```
     2                       T(0,0),Nx+1,MPI_double_precision,
     3                       left,tag,comm,status,ierr)

         Call MPI_SENDRECV (T(0,m-1),Nx+1,MPI_double_precision,
     1                       right,tag,
     2                       T(0,m),Nx+1,MPI_double_precision,
     3                       right,tag,comm,status,ierr)
         If (rank.eq.0) timeStop1=timeStop1+MPI_Wtime()-timeStart1
         End do
c  Пересчет правой части.
         Do i=1,Nx-1
           Do j=1,m-1
             b(i,j)=T(i,j)
           End do
         End do
       End do
c  Фиксация времени окончания расчетов.
       If (rank.eq.0) timeStop=MPI_Wtime()-timeStart
c  Сбор результатов на нулевом процессоре.
       Call MPI_GATHER( T(0,1),(m)*(Nx+1),MPI_DOUBLE_PRECISION,
     1                  T(0,1),(m)*(Nx+1),MPI_DOUBLE_PRECISION,
     2                  0,comm,ierr)
       If (rank.eq.0) then
         Do i=0,Nx
           write (2,'(560f15.4)') (T(i,j),j=0,Ny)
         End do
         Write(2,'(a,i15,a,f15.9)') 'Size=',size,'Tame=',timeStop
         Write(2,'(a,f15.9,a,f15.9)') ' FTame=', time, 'tau=',tau
         Write (2,'(a,f15.9,a,f15.9)')' Tsch=',timeStop-TimeStop1,
     1                                 ' Tobm=',TimeStop1
        End if
        Call MPI_FINALIZE(ierr)
   End
```

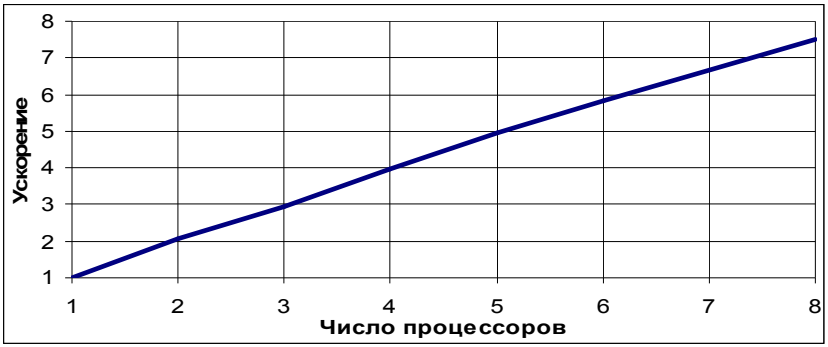
В таблице приведено время работы программы, для разного числа процессоров и коэффициента теплопроводности $a = 10^{-5}$. Расчеты проводились на равномерной сетке 720 на 720 (время приведено в секундах).

| **Число проц-в** / **Коэф-т тепл-ти** | **1** | **2** | **3** | **4** | **5** | **6** | **8** |
|---|---|---|---|---|---|---|---|
| $\alpha = 10^{-5}, \tau = 2.0$ | **24890** | **13153** | **8732** | **6612** | **5376** | **4555** | **3550** |
| **Время счета** | **24884** | **12927** | **8497** | **6316** | **5056** | **4195** | **3155** |
| **Время пересылок** | **6** | **208** | **235** | **296** | **320** | **360** | **395** |

Тестовые расчеты, проведенные для различных значений коэффициента теплопроводности, показали значительное увеличение времени счета при увеличении $\alpha$. Это объясняется увеличением числа итераций, требуемых для сходимости итерационного метода на каждом временном шаге, при уменьшении коэффициента теплопроводности $\alpha = 10^{-5}, \alpha = 10^{-4}, \alpha = 10^{-3}$. Следующий график показывает, сколько итераций метода необходимо для сходимости на каждом временном шаге. для $\alpha = 10^{-5}$


Метод Зейделя распараллеленный по методу красно-черного упорядочивания превращается в два шага метода Якоби, а этот метод эквивалентен вычислению с использованием явной схемы.

Далее приводиться график ускорения для $a = 10^{-5}$.




Ускорение, как и в случае с явной схемой, близко к линейному.

**Метод сопряженных градиентов.**

Альтернативными к треугольным методам [7] решения систем $Ax = b$ являются алгоритмы ускорения итерационных процессов на основе выбора итерационных параметров из условия минимизации функционалов, определяющих точность текущих последовательных приближений.

В дальнейшем будут использованы следующие обозначения: $r^k = b - Ax^k$ - невязка, $\omega^k = B^{-1}r^k$ - поправка и $z^k = x^k - x$ - погрешность (ошибка). Здесь $x$ точное решение системы $Ax = b$, $x^k$ - $k$-ое приближение к точному решению. $B$ в терминологии [3] предобуславливающая матрица.

К открытию нового метода независимо пришли М. Хестенес и Э. Штифель [3]. Алгоритм называется методом сопряженных градиентов (CG). Он является наиболее предпочтительным для симметричных положительно определенных систем. Формулы классического метода сопряженных градиентов имеют следующий вид:

$p = x;$
$v = Ap;$
$p = b - v;$
$r = p;$
$\alpha = \left(\|r\|_2\right)^2;$
*for* $((i < i\_\max).and.(\alpha > \varepsilon))$
$v = Ap;$
$\lambda = \alpha \Big/ (v, p)_2;$
$x = x + \lambda p;$
$r = r - \lambda v;$
$\alpha_{new} = \left(\|r\|_2\right)^2;$
$p = r + \left(\alpha_{new} \Big/ \alpha\right) p;$
$\alpha = \alpha_{new};$
*end do.*

Достаточным условием сходимости метода сопряженных градиентов является симметричность и положительная определенность матрицы $A$, при этом на спектр матрицы накладывается условие $0 < m \le \gamma(A) \le M$. При этом скорость сходимости можно определить по формуле

$$\phi\left(x^n\right) \le \left(\frac{2 \cdot \gamma^n}{1 + \gamma^{2n}}\right)\phi\left(x^0\right);$$

$$\gamma = \frac{1 - \sqrt{m/M}}{1 + \sqrt{m/M}};$$

Скорость сходимости метода сопряженных градиентов выше, чем скорость сходимости метода Зейделя [5].

Мы видим, что CG имеет то особенно привлекательное свойство, что в его реализации предусмотрено одновременное хранение в памяти лишь четырех векторов. Кроме того, в его внутреннем цикле, помимо матрично-векторного произведения, вычисляются только два скалярных

произведения, три операции типа «saxpy» (сложение вектора, умноженного на число, с другом вектором) и небольшое количество скалярных операций. Таким образом, и необходимая память, и вычислительная работа в методе не очень велики. В алгоритме $x^{k+1}$ вычисляется используя рекуррентные соотношения для трех групп векторов. Одновременно в памяти требуется хранить лишь самые последние векторы их каждой группы, записываемые на места своих предшественников [1]. Первая группа векторов – это приближенные решения $x^k$. Вторая группа – это невязки $r^k = b - Ax^k$. Третья группа – это сопряженные градиенты $p^k$. Эти вектора называют градиентами по следующей причине: шаг метода CG можно рассматривать как выбор числа $a_{new}/a$ из условия, чтобы новое приближенное решение $x^{k+!} = x^k + a_{new}/a \cdot p^k$ минимизировало норму невязки $\|r_k\|_{A^{-1}} = \left(r_k^T A^{-1} r_k\right)^{1/2}$. Иными словами, $p^k$ используется как направление градиентного поиска. Они называются сопряженными, или, точнее, А-сопряженными, потому что $p_k^T A p_j = 0$ при $j \neq k$.

Параллельную реализацию рассмотрим на примере неявного метода решения уравнения теплопроводности. Чтобы яснее представить себе процесс распараллеливания и связанные с ним изменения в алгоритме, разберем отдельно каждую алгебраическую операцию, которая имеет место быть в изложенном методе.

Первая операция это матрично-векторное произведение $y = Ax$.

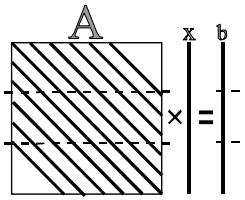



Рис. 5. Строчное распределение.

Известно, что в общем случае при умножение матрицы на вектор возможны два способа распределения данных. В первом случае каждому процессору назначается определенное количество строк матрицы $A$ и целиком вектор $x$ (рис. 5), каждый процессор реализует умножение распределенных ему строк матрицы $A$ на вектор $x$ (число вычислительных узлов меньше или равно числу строк). Получаем абсолютно не связанные подзадачи, однако предложенный алгоритм не лишен и недостатков. Дело в том, что компоненты результирующего вектора $y$ оказывается разбросанными по всем процессорам и для старта следующей итерации необходимо собрать вектор $y$ на каждом процессорном элементе.

Другой вариант – каждому процессорному элементу распределяется определенное количество столбцов матрицы $A$, тогда отпадает необходимость хранить на каждом процессоре целиком вектор $x$ и он распределяется по всем вычислительным узлам (схему распределения данных представлена на рис. 6). Тогда в процессе вычислений результирующий вектор $y$ оказывается разобранным на слагаемые, а каждое слагаемое находится на отдельном процессоре. То есть опять требуются пересылки данных для того, что бы собрать результирующий вектор и разослать каждому процессору честь вектора $y$, необходимую ему для начала следующей итерации.

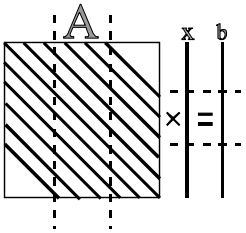


Рис. 6. Столбцовое распределение.

Для случая системы с ленточной матрицей более эффективным будет способ столбцового распределения данных по процессорным элементам, так как слагаемые, из которых складываются компоненты результирующего вектора $y$, будут отличны от нуля лишь на одном процессоре, то есть отпадает необходимость в пересылках.

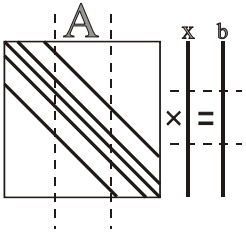


Рис. 7. Случай разреженной матрицы.

Следующие две операции – это вычисление скалярного произведения и векторная операция saxpy. При сложении векторов, каждый процессор выполняет действия над распределенными ему компонентами векторов, эта операция не требует обменов данными, если нет необходимости в использовании вектора целиком.

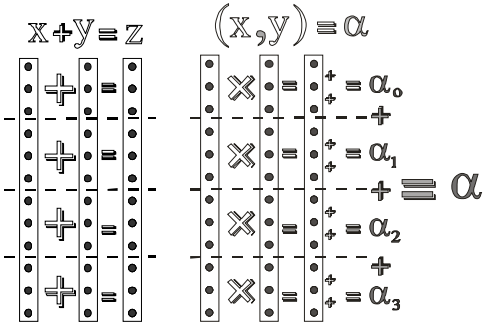


Рис. 8. Схема операций saxpy и скалярного произведения.

Вычисление скалярного произведения – принципиально другая операция, ее результатом является не вектор, а одно число, которое в последствии должно быть разослано на все вычислительные узлы. При вычислении скалярного

произведения первый этап – покомпонентное умножение векторов, в результате которого получается вектор – происходит независимо на каждом процессоре подобно сложению векторов. Затем необходимо просуммировать все компоненты полученного вектора. Сначала на каждом процессорном элементе выполняется суммирование принадлежащих ему компонент полученного вектора, затем результаты суммирования на каждом процессоре собираются и суммируются на одном из процессоров, полученный результат рассылается остальным вычислительным узлам.

Таким образом рассмотрена параллельная реализация каждой процедуры метода сопряженных градиентов, далее приводиться параллельная программа реализующая этот метод.

```
      PROGRAM Example3
c  Решение двумерного уравнения Пуассона в единичном квадрате
c  по методу сопряженных градиентов.
       Implicit none
       Include 'mpif.h'
c      Параметры:
c      Nx,Ny - количество точек области в каждом направлении,
c      m - количество точек на процессор.
c      T,Tnew - численное решение задачи.
       Integer i,j,k,Nx,Ny,m
       Integer comm,ierr,size,rank,left,right,tag,status(100)
       Parameter (Nx=720,Ny=Nx)
       Double precision ap,ae,aw,an,as,b(0:Nx,0:Ny),err,gerr,
      1             T(0:Nx,0:Ny),Tnew(0:Nx,0:Ny),alfa,hx,hy,Lx,Ly,
      2             tau,time,timefin,timeStart,timeStop,rez,grez
      3             timeStart1,timeStop1
       Parameter (timefin=1.0d3,Lx=1.0d0,Ly=1.0d0,
      1           hx=Lx/Nx,hy=Ly/Ny,alfa=1.0d-5)
c  Параметры для метода сопряженных градиентов
       Double precision r(0:Nx,0:Ny),p(0:Nx,0:Ny),al,be,lay,alNew,
      1                 v(0:Nx,0:Ny),scalmult
       Common /setka / m,comm,rank,size,left,right
       Common /tim / timeStart1,timeStop1
c  Иницилизация, определение числа выделенных процессоров (size)
c  и номера текущего процесса (rank).
       Call MPI_INIT(ierr)
       Call MPI_comm_size(mpi_comm_world,size,ierr)
       Call MPI_comm_rank(mpi_comm_world,rank,ierr)
       comm = mpi_comm_world
c  Определение номеров процессоров (соседей) сверху и снизу.
c  Если таковые отсутствуют то им присваивается значение
c  MPI_PROC_NULL (для них коммуникационные операции игнорируются)
       If (rank.eq.0) then
         left=MPI_PROC_NULL
       Else
```

```
          left=rank-1
        End if
        If (rank.eq.size-1) then
          right=MPI_PROC_NULL
        Else
          right=rank+1
        End if
c  Определение числа элементов обрабатываемых одним процессором.
        m=Ny/size
        If (rank.lt.(Nx-size*m)) then
          m=m+1
        End if
c  Задание начальных значений.
        Do i=0,Nx
          Do j=0,m
            T(i,j)=100.0d0
            Tnew(i,j)=100.0d0
            b(i,j)=0.0d0
          End do
        End do
c  Задание граничных условий.
        If (rank.eq.0) then
          Do i=0,Nx
            T(i,0)=10.0d0
          End do
        End if
        If (rank.eq.size-1) then
          Do i=0,Nx
            T(i,m)=10.0d0
          End do
        End if
        Do j=0,m
          T(0,j)=1.0d0
          T(Nx,j)=10.0d0
        End do
c  Подготовка к расчетам.
        tau=2.0d0
        Do i=1,Nx-1
          Do j=1,m-1
            b(i,j)=T(i,j)
          End do
        End do
        ae=alfa*tau/(hx*hx)
        aw=alfa*tau/(hx*hx)
        an=alfa*tau/(hy*hy)
        as=alfa*tau/(hy*hy)
        ap=ae+aw+an+as
        If (rank.eq.0) then
          open (2,file='rezult1.dat')
          timeStart = MPI_Wtime()
          timeStop1 = 0.0
        End if
c  Движение по временной оси.
```

```
        Do while (time.lt.timefin)
           time =time + tau
c  Расчет значений с нового временного слоя.
c  На каждом временном шаге для уменьшения погрешности численного
c  метода релизуется несколько итераций.
          Do i=0,Nx
            Do j=0,Ny
              v(i,j)=0.0d0
              p(i,j)=0.0d0
              r(i,j)=0.0d0
            End do
          End do
          Call sumvv(T,T,0.0d0,p)
          Call multmv(ap,ae,aw,an,as,p,v)
          Call sumvv(b,v,-1.0d0,p)
          Call sumvv(p,T,0.0d0,r)
          al=scalmult(r,r)
          gerr=1.0d0
          grez=1.0d0
          Do While ((gerr.gt.1.0d-5).or.(gerr.gt.1.0d-5))
            err=0.0d0
            rez=0.0d0
            Call multmv(ap,ae,aw,an,as,p,v)
            lay=al/scalmult(v,p)
            Call sumvv(T,p,lay,Tnew)
            Call sumvv(r,v,-lay,r)
            alNew=scalmult(r,r)
            Call sumvv(r,p,alNew/al,p)
            al=alNew
            Do i=0,Ny
              Do j=0,Ny
                err=err+abs(T(i,j)-Tnew(i,j))
                T(i,j)=Tnew(i,j)
              End do
            End do
            rez=sqrt(al)
            If (rank.eq.0) timeStart1 = MPI_Wtime()
            call MPI_ALLREDUCE(err,gerr,1,MPI_DOUBLE_PRECISION,
     1                         MPI_SUM,comm,ierr )
            call MPI_ALLREDUCE(rez,grez,1,MPI_DOUBLE_PRECISION,
     1                         MPI_SUM,comm,ierr )
          If (rank.eq.0) timeStop1=timeStop1+MPI_Wtime()-timeStart1
          End do
c  Пересчет правой части.
          Do i=1,Nx-1
            Do j=1,m-1
              b(i,j)=T(i,j)
            End do
          End do
        End do
c  Фиксация времени окончания расчетов.
        If (rank.eq.0) timeStop=MPI_Wtime()-timeStart
c  Сбор результатов на нулевом процессоре.
```

```
       Call MPI_GATHER(T(0,1),(m)*(Nx+1),MPI_DOUBLE_PRECISION,
      1                T(0,1),(m)*(Nx+1),MPI_DOUBLE_PRECISION,
      2                0,comm,ierr)
       If (rank.eq.0) then
         Do i=0,Nx
           write (2,'(560f15.4)') (T(i,j),j=0,Ny)
         End do
         Write(2,'(a,i15,a,f15.9)') ' Size=',size,'Tame=',timeStop
         Write(2,'(a,f15.9,a,f15.9)') ' FTame=', time, ' tau=',tau
         Write (2,'(a,f15.9,a,f15.9)')' Tsch=',TimeStop-TimeStop1,
     1                                 ' Tobm=',TimeStop1
       End if
       Call MPI_FINALIZE(ierr)
       End
c      ------- 1111 -------------------
       Function scalmult(x,y)
         Implicit none
         Include 'mpif.h'
         Integer i,j,Nx,Ny,m,ierr,comm,rank,
      1          size,jbeg,jend,left,right
         Parameter (Nx=720,Ny=Nx)
         Double precision z,gz,scalmult,
      1                   x(0:Nx,0:Ny),y(0:Nx,0:Ny)
      2                   timeStart1,timeStop1
         Common /setka / m,comm,rank,size,left,right
         Common /tim / timeStart1,timeStop1
         z=0.0d0
         gz=0.0d0
         If (rank.eq.0) then
           jbeg=0
         Else
           jbeg=1
         End if
         If (rank.eq.size-1) then
           jend=m
         Else
           jend=m-1
         End if
         Do j=jbeg,jend
           Do i=0,Nx
             z=z+x(i,j)*y(i,j)
           End do
         End do
         If (rank.eq.0) timeStart1 = MPI_Wtime()
         call MPI_ALLREDUCE( z,gz,1,MPI_DOUBLE_PRECISION,
      1                      MPI_SUM,comm,ierr )
         If (rank.eq.0) timeStop1=timeStop1+MPI_Wtime()-timeStart1
         scalmult=gz
         return
         End
c      ------- 2222 -------------------
       Subroutine sumvv(x,y,sig,z)
         Implicit none
```

```
          Include 'mpif.h'
          Integer i,j,Nx,Ny,m,rank,size,comm,left,right
          Parameter (Nx=720,Ny=Nx)
          Double precision sig,z(0:Nx,0:Ny)
          Double precision x(0:Nx,0:Ny),y(0:Nx,0:Ny)
          Common /setka / m,comm,rank,size,left,right
          Do j=0,m
            Do i=0,Nx
              z(i,j)=x(i,j)+sig*y(i,j)
            End do
          End do
          return
          End
c      ------ 3333 --------------------
        Subroutine multmv(ap,ae,aw,an,as,x,y)
          Implicit none
          Include 'mpif.h'
          Integer i,j,Nx,Ny,m,tag,rank,size,comm,
      1           left,right,status(100),ierr
          Parameter (Nx=720,Ny=Nx)
          Double precision x(0:Nx,0:Ny),y(0:Nx,0:Ny),
      1                     z(0:Nx,0:Ny),sum
          Double precision ap(0:Nx,0:Ny),ae(0:Nx,0:Ny),
      1                    aw(0:Nx,0:Ny),an(0:Nx,0:Ny),
      2                    as(0:Nx,0:Ny),b(0:Nx,0:Ny),
      3                    timeStart1,timeStop1
          Common /setka / m,comm,rank,size,left,right
          Common /tim / timeStart1,timeStop1
c  Пересылка граничных значений на соседние процессоры.
        If (rank.eq.0) timeStart1 = MPI_Wtime()
        tag=10
        Call MPI_SENDRECV(x(0,1),Nx+1,MPI_double_precision,
      1                    left,tag,
      2                    x(0,0),Nx+1,MPI_double_precision,
      3                    left,tag,comm,status,ierr)
       Call MPI_SENDRECV(x(0,m-1),Nx+1,MPI_double_precision,
      1                  right,tag,
      2                  x(0,m),Nx+1,MPI_double_precision,
      3                  right,tag,comm,status,ierr)
        If (rank.eq.0) timeStop1=timeStop1+MPI_Wtime()-timeStart1
        Do j=0,m
          Do i=0,Nx
             If ((i.gt.0).and.(j.gt.0).and.(i.lt.Nx).and.(j.lt.m))
             then
                y(i,j)=(1.0+ap)*x(i,j)-ae*x(i+1,j)
      1                                -aw*x(i-1,j)
      2                                -an*x(i,j+1)
      3                                -as*x(i,j-1)
             Else
          End do
        End do
        return
        End
```

В таблице приведено время работы программы, для разного числа процессоров и для коэффициента теплопроводности $a = 10^{-5}$. Расчеты проводились на равномерной сетке 720 на 720 (время приведено в секундах).

| Число проц-в / Коэф-т тепл-ти | 1 | 2 | 3 | 4 | 5 | 6 | 8 |
|---|---|---|---|---|---|---|---|
| $\alpha = 10^{-5}, \tau = 2.0$ | **3378** | **2463** | **2137** | **1922** | **1787** | **1555** | **1353** |
| **Время счета** | **3376** | **2217** | **1970** | **1884** | **1739** | **1488** | **1262** |
| **Время пересылок** | **2** | **246** | **167** | **38** | **48** | **67** | **91** |

Тестовые расчеты, проведенные для различных значений коэффициента теплопроводности, показали незначительное увеличение времени счета при увеличении $\alpha$ (в сравнении с методом Зейделя, где время счета увеличивалось на порядок). Это объясняется уменьшением числа итераций, требуемых для сходимости итерационного метода на каждом временном шаге. Следующий график показывает, сколько итераций метода необходимо для сходимости на каждом временном шаге (для метода Зейделя эта величина составляла 250-310 итераций на один шаг по времени).

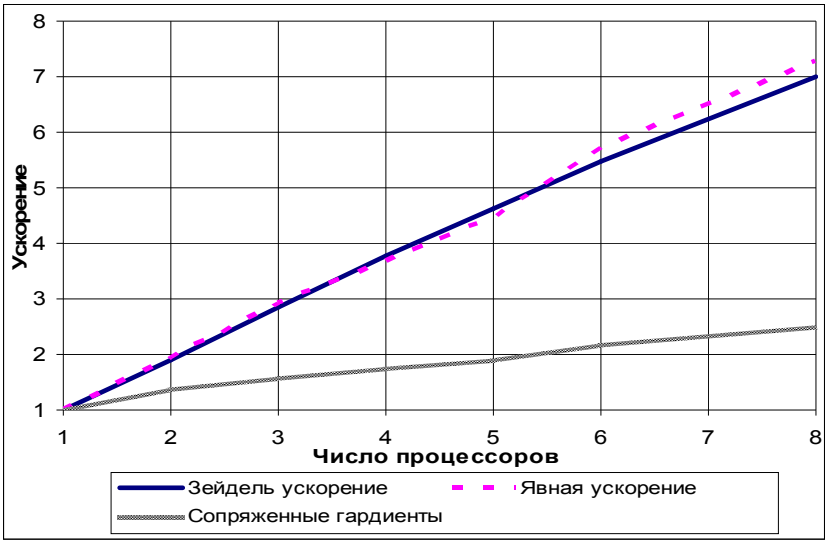

Далее приводиться график ускорения для значения $\alpha = 10^{-5}$.

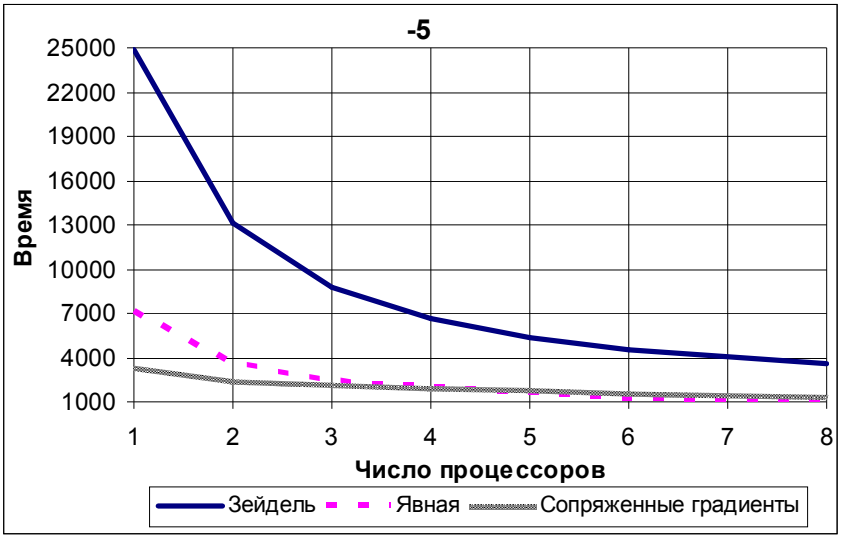

Видно, что ускорение на восьми процессорах не превышает 3.

**Выводы.**

При аппроксимации дифференциальной задачи используется один и тот же пятиточечный шаблон, поэтому при организации параллельных вычислений, как в случае явной схемы, так и в случае неявной требуется организация обменов частями решения. Из приведенных примеров видно, что обмены реализуются одинаково и требуются после каждой итерации метода. А если шаг по времени при использовании неявной схемы много больше шага по времени использующегося при реализации явной схемы, то количество обменов для случая неявной схемы будут много меньше. И как следствие при увеличении числа процессоров использование неявной разностной схемы даст лучшие результаты.

Разберем полученные результаты. Для этого сравним ускорение, которое дают изученные методы для $\alpha = 10^{-5}$. Видно, что ускорение для явной схемы и для метода Зейделя совпадает и близко к линейному. Что касается метода сопряженных градиентов, то он даже на восьми процессорах дает ускорение не более чем в три раза. Но ускорения является весьма относительной оценкой и большее значение имеет сравнить время работы вычислительной машины для каждого метода для разного числа процессоров.




Из этого графика видно, что по времени самой экономичной является явная схема, затем идет метод сопряженных градиентов и самый плохой результат у метода Зейделя.

Из приведенных примеров видно, что для данной задачи (для данной сетки) наиболее подходит неявная схема в сочетании с методом сопряженных градиентов, демонстрирующие минимальное время работы.

Распараллеливание метода Зейделя хоть и дает близкое к линейному ускорение, но с точки зрения временных затрат является неэффективным.

Далее попробуем представить явные и неявные методы в матрично-векторном виде. Явная схема, может быть переписана в виде матрично-векторного умножения. Для этого необходимо изменить нумерацию узлов сетки, и посмотреть на матрицу неизвестных $T_{i,j}$ как на вектор-столбец $T_{I=0,Nx*Ny}$ следующим образом:

$$i,j = 0,0;\ \ 0,1;\ \ \ldots\ \ 0,n-1;\ \ 1,0;\ \ 1,1;\ \ \ldots\ \ m-1,0;\ \ \ldots\ \ m-1,1;\ \ \ldots\ \ m-1,n-1;$$

$$I\ =\ 1;\ \ \ 2;\ \ \ldots\ \ n-1;\ \ n;\ \ n+1;\ \ldots\ \ 2n-1;\ \ \ldots\ (m-1)n;\ \ \ldots\ \ \ mn-1;$$

Далее перепишем все в матричном виде $T^{n+1} = (I + A)T^n$. Где матрица

$$A = \begin{pmatrix}
ap_{0,0} & an_{0,0} & & ae_{0,0} & & & & & & & & \\
as_{0,1} & ap_{0,1} & an_{0,1} & & ae_{0,1} & & & & & & & \\
 & as_{0,2} & ap_{0,2} & an_{0,2} & & ae_{0,2} & & & & & & \\
 & & & & & & & & & & & \\
 & & & & & & & & & & & \\
aw_{1,0} & & as_{1,0} & ap_{1,0} & & & ae_{1,0} & & & & & \\
 & aw_{1,1} & & & ap_{1,1} & an_{1,1} & & ae_{1,1} & & & & \\
 & & aw_{1,2} & & & as_{1,2} & ap_{1,2} & an_{1,2} & & & & \\
 & & & & & & & & & & & \\
 & & & & & & & & & & & \\
 & & & & aw_{n,n-1} & & as_{n,n-1} & ap_{n,n-1} & an_{n,n-1} & & & \\
 & & & & & aw_{n,n} & & as_{n,n} & ap_{n,n} & & &
\end{pmatrix};$$

$A$ имеет следующий вид. Таким образом, применение явных схем сводит нахождение решения задачи к последовательному применению к приближенному решению операции матрично-векторного произведения. Эта операция легко распараллеливается, а учет структуры матрицы позволяет максимально сократить необходимое число обменов. Распараллеливание этой операции было рассмотрено в методе сопряженных градиентов. В случае неявной разностной схемы мы приходим к системе линейных алгебраических уравнений следующего вида $(I - A)T^{n+1} = T^n$, где матрица $A$ имеет то же вид, что и для явной разностной схемы. При решении системы на каждом временном шаге необходимо проводить достаточное для сходимости число итераций метода, в зависимости от числа обусловленности системы.

Важным является следующие, при распараллеливании численного метода, будь то явная разностная схема или параллельная реализация итерационного метода, все сводиться к распараллеливанию таких операций как матрично-векторное умножение, скалярное произведение, сложение векторов.

**2D декомпозиция**.

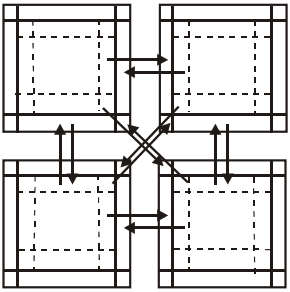


Рис. 9. Схема обменов при двумерной декомпозиции.

При большом числе процессоров необходимо использовать двумерную декомпозицию. При этом по-прежнему необходимо на каждом процессоре создавать фиктивные ячейки. Операции обменов схематически изображены на рисунке 9. На первый взгляд может показаться, что обменов стало больше. Проверим, что происходит на самом деле. Посчитав число пересылаемых элементов при каждом способе декомпозиции.

- 1d декомпозиция, число обменов есть величина $2n$, где $n$-размерность задачи.

- 2d декомпозиция, число обменов есть величина $\frac{4n}{\sqrt{p}}$, где $n$-размерность задачи, $p$- число вычислительных узлов. (формулы приведены для решения двумерной задачи).

Таким образом, видим, что при $p > 5$ двумерная декомпозиция является более перспективной для распараллеливания. Реализация 2d декомпозиции является более трудоемким процессом в сравнении с реализацией 1d декомпозиции и в данных методических указаниях, приводиться не будет. Основная возникающая сложность при ее реализации состоит в организации межпроцессорных обменов. Так как изложенные численные методы, сведены к набору матрично-векторных операций, которые последовательно применяются к каждому новому полученному приближению и не требуют не чего кроме знания значения содержимого соседний ячейки, то есть правильной организации обменов.

**Эффективность параллельных и последовательных программ.**

Параллельные компьютеры используют для получения выигрыша во времени за счет использования дополнительных узлов. При этом помимо минимизации обменов между процессорами необходимо, что бы максимальную производительность показывал каждый вычислительный узел. Эта глава будет обсуждать вопросы организации эффективных вычислений на каждом отдельном вычислительном узле. Оказывается, для этого необходимо [2]:

- учитывать организацию подсистем памяти (доступ к элементам матриц, эффекты КЭШ памяти).
- использовать оптимизирующие компиляторы(o1, o2, o3, intel).
- использовать специализированные библиотеки (blas).

Все эти вопросы будут рассмотрены на примере задачи умножения матриц. Хотя математически эта задача очень проста, она таит немало богатств с математической точки зрения.

Рассмотрим следующий вопрос, как организованны вычисления в компьютере? Прежде чем компьютер выполнит операцию, операнды должны находиться «в нужном месте», а за доставку в нужное место приходиться платить. Проблема в том, что память обычно организованна иерархическим образом. Между функциональными устройствами, выполняющими арифметические операции, и основной памятью, в которой лежат данные, может располагаться относительно небольших размеров высокопроизводительная кэш-память. Детали могут меняться от машины к машине, но вот две вещи, о которых о которых необходимо постоянно помнить при работе в условиях многоуровневой памяти:

- объем памяти на каждом уровне иерархии ограничен по экономическим причинам и уменьшается при повышении уровня иерархии.
- обмен данными между уровнями иерархии сопряжен с затратами, иногда сравнительно большими.

Выходы из этого очевидны. Если мы хотим эффективные матричные алгоритмы для типичного компьютера, мы должны постоянно держать в поле зрения передвижение данных по уровням иерархии памяти. Многоуровневая память налагает на разработчика алгоритмов специфические требования, побуждая его мыслить в определенном ключе. Писать программы так, что бы данные, попавшие в КЭШ-память, использовались с максимальной эффективностью. В современных компьютерах при вычислениях в КЭШ-память помещаются наиболее часто используемые участки кода, плюс происходит прогнозирование

возможно используемых, в дальнейшим данных, которые непосредственно во время вычисления переносятся в КЭШ.

В качестве иллюстрационного примера часто используется задача перемножения матриц большой размерности. Имеются 6 вариантов упорядочивания тройного цикла в алгоритме умножения матриц, какой вариант предпочтительней [2]. Ответ на этот вопрос зависит от архитектуры компьютера, на котором будет реализован алгоритм.

```
Do i=1,N
  Do j=1,N
    Do k=1,N
      C(i,j) = C(i,j) +A(i,k)*B(k,j)
    End do
  End do
End do
```

Будем исходить из того, что мы программируем на фортране. Массивы в фортране храниться по столбцам. Это означает, что элементы матрицы, находящиеся в одном и том же столбце, занимают последовательные ячейки памяти. Так, если для хранения матрицы $A \in R^{4\times 6}$ было бы отведено 24 ячейки памяти, то при программировании по фортране элементы матрицы выстраиваются в памяти, как показано на рис. 10.

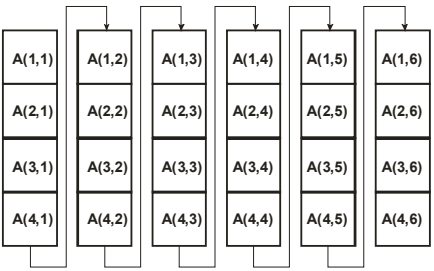



Рис. 10. Представление данных в языке фортран.

Рассмотрим подробно один из вариантов циклов, например $jki$. C(i,j) = C(i,j) +A(i,k)*B(k,j) видно, что доступ к элементам всех матриц осуществляется по столбцам. Что и даст на практике существенный выигрыш во времени при использовании данной последовательности циклов. Далее приведены тестовые расчеты для различных последовательностей циклов. Для различных компиляторов, архитектур и размерностей.

Но сначала несколько слов о компиляторах (ключах компилятора) Release и Debug. Компилятор Debug оптимизирует программный код с точки зрения максимального экономия использования оперативной памяти, компилятор Release оптимизирует программный код с точки зрения достижения максимального быстродействия.

| Время перемножения матриц 1400x1400 Debug компилятор | | | | | | |
|---|---|---|---|---|---|---|
| Процессор | ***(i,j,k)*** | ***(i,k,j)*** | ***(j,i,k)*** | ***(j,k,i)*** | ***(k,i,j)*** | ***(k,j,i)*** |
| AMD Sempron 2500+ | 159 | 311 | 540 | 131 | 322 | 132 |
| AMD AthlonXP 1800+ | 172 | 269 | 516 | 110 | 275 | 110 |
| Intel Celron 3000 | 300 | 514 | 844 | 60 | 514 | 61 |
| Intel Pentium 3000 | 114 | 130 | 283 | 62 | 133 | 62 |

| Время перемножения матриц 700x700 Debug компилятор | | | | | | |
|---|---|---|---|---|---|---|
| Процессор | ***(i,j,k)*** | ***(i,k,j)*** | ***(j,i,k)*** | ***(j,k,i)*** | ***(k,i,j)*** | ***(k,j,i)*** |
| AMD Sempron 2500+ | 17 | 27 | 53 | 16 | 28 | 16 |
| AMD AthlonXP 1800+ | 19 | 31 | 61 | 13 | 31 | 13 |
| Intel Celron 3000 | 16 | 49 | 70 | 7 | 50 | 7 |
| Intel Pentium 3000 | 13 | 15 | 34 | 7 | 16 | 7 |

| Время перемножения матриц 1400x1400 Release компилятор | | | | | | |
|---|---|---|---|---|---|---|
| Процессор | ***(i,j,k)*** | ***(i,k,j)*** | ***(j,i,k)*** | ***(j,k,i)*** | ***(k,i,j)*** | ***(k,j,i)*** |
| AMD Sempron 2500+ | 65 | 184 | 263 | 14 | 197 | 31 |
| AMD AthlonXP 1800+ | 74 | 172 | 254 | 34 | 172 | 40 |
| Intel Celron 3000 | 115 | 459 | 577 | 6 | 462 | 14 |
| Intel Pentium 3000 | 30 | 65 | 100 | 6 | 74 | 11 |

| Время перемножения матриц 700x700 Release компилятор | | | | | | |
|---|---|---|---|---|---|---|
| Процессор | ***(i,j,k)*** | ***(i,k,j)*** | ***(j,i,k)*** | ***(j,k,i)*** | ***(k,i,j)*** | ***(k,j,i)*** |
| AMD Sempron 2500+ | 4 | 16 | 22 | 2 | 17 | 4 |
| AMD AthlonXP 1800+ | 7 | 18 | 26 | 4 | 18 | 5 |
| Intel Celron 3000 | 3 | 34 | 38 | 1 | 34 | 1 |
| Intel Pentium 3000 | 3 | 7 | 10 | 1 | 7 | 1 |

Видно, что вне зависимости от архитектуры процессора, используемого компилятора и размерности, последовательность циклов *jki* является наиболее выигрышной. Это объясняется, что при данной последовательности циклов доступ к элементам каждой из матриц осуществляется по столбцам.

Далее будут представлены результаты использования специализированных библиотек, на примере библиотеки BLAS (Basic Linear Algebra Subprograms). Эта библиотека реализует операции трех уровней:

1 уровень: это векторные операции.

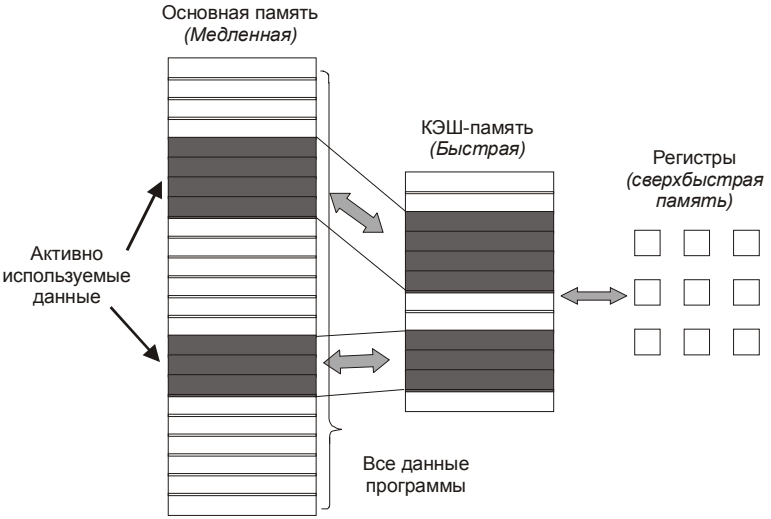
2 уровень: это матрично-векторные операции.

3 уровень: это процедуры реализующие умножение матриц.

При реализации этих процедур используются все возможности оптимизации расчетов.

Следующий важный вопрос это влияние размера КЭШа.

Прежде чем обсуждать влияние размера КЭШ памяти на эффективность вычислений остановимся на вопросе блочного умножения матриц. Умение непринужденно работать с блочными матрицами крайне необходимо для матричных вычислений. Использование блочных обозначений упрощает вывод многих важных алгоритмов. Кроме того, все более возрастает роль блочных алгоритмов в вычислениях на суперкомпьютерах. Здесь под блочными алгоритмами мы подразумеваем по существу такие алгоритмы, которые содержат много операций умножения матрицы на матрицу.




Можно рассматривать два варианта блочного произведения матриц. Первое, когда блоки создаются по средству перемены индексов, при этом матрицы остаются неизменными и меняется лишь способ обхода элементов матрицы

```
Do ib=0,m
   Do  i=(ib-1)*a+1,ib*a
      Do jb=0,m
          Do  j=(jb-1)*a+1,jb*a
              Do kb=1,m
                  Do  k=(kb-1)*a+1,kb*a
                        C(i,j) = C(i,j) +A(i,k)*B(k,j)
                  End do
              End do
          End do
       End do
    End do
```

Связи на элементы матрицы представлены на рисунке 10.

И второй способ, когда от двумерных матриц осуществляется переход к четырехмерным матрицам. При этом алгоритм принимает следующий вид

```
Do ib=0,m
  Do  i=(ib-1)*a+1,ib*a
    Do jb=0,m
      Do  j=(jb-1)*a+1,jb*a
        Do kb=1,m
          Do  k=(kb-1)*a+1,kb*a
            C(ib,ib,i,j) = C(ib,jb,i,j) +A(ib,kb,i,k)*B(kb,jb,k,j)
          End do
        End do
      End do
    End do
  End do
End do
```

Тогда элементы матрицы в памяти компьютера хранятся следующим образом (см. рис. 11).

Теперь переедем непосредственно к обсуждению эффекта КЭШ-памяти. Под этим понимают следующие, так оптимально подобрать размер блока матрицы, так что бы он целиком уместился в КЭШ памяти. То есть случай максимально рационального использования КЭШ-памяти.

Если для предыдущего случая было 6 $(3!)$ вариантов последовательностей циклов, то теперь мы имеем набор из 6 индексов $i,\ j,\ k,\ ib,\ jb,\ kb$ и $6! = 720$ вариантов последовательностей циклов, перебрать все эти варианты довольно проблематично, и не имеет глубокого смысла. Поэтому, уже обладая некоторым набором знаний,

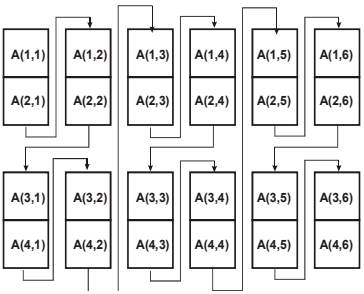



Рис. 11 Представление данных при блочном распределении.

полученных при простом перемножении матриц, можно сделать предположение о наиболее выигрышных вариантах последовательностей циклов.

Наиболее естественными кажутся следующие последовательности:

Для двумерных матриц $jb, kb, ib, j, k, j$ где меняется последовательность 3 последних индексов.

Для четырех мерных матриц $j, k, i, jb, kb, ib$ где так же меняется последовательность 3-х последних индексов.

Следующий тест проводился на интеловском процессоре для размерностей матриц 1400 на 1400 и 700 на 700. Здесь сравнивались два компилятора f77 и Intel, и изучалось влияние использования блочных алгоритмов на время выполнения поставленной задачи. Так же приведен результат работы процедуры Dgemm при использовании различных компиляторов.

| Время перемножения матриц 1400x1400 (16 блоков) f77 компилятор. | | | | | | | |
|---|---|---|---|---|---|---|---|
| Процессор Intel Pentium III 650 | *(i,j,k)* | *(i,k,j)* | *(j,i,k)* | *(j,k,i)* | *(k,i,j)* | *(k,j,i)* | dgemm |
| без блоков | 132 | 339 | 454 | 118 | 358 | 117 | 130 |
| двумерные матрицы | 113 | 136 | 233 | 122 | 148 | 123 | ----- |
| четырехмерные матрицы | 67 | 95 | 192 | 78 | 95 | 79 | ----- |
| Время перемножения матриц 1400x1400 (16 блоков) Intel компилятор. | | | | | | | |
| Процессор Intel Pentium III 650 | *(i,j,k)* | *(i,k,j)* | *(j,i,k)* | *(j,k,i)* | *(k,i,j)* | *(k,j,i)* | dgemm |
| без блоков | 80 | 195 | 278 | 57 | 266 | 79 | 57 |
| двумерные матрицы | 62 | 102 | 165 | 59 | 155 | 81 | ----- |
| четырехмерные матрицы | 22 | 29 | 55 | 30 | 37 | 40 | ----- |
| Время перемножения матриц 1400x1400 (100 блоков) f77 компилятор. | | | | | | | |
| Процессор Intel Pentium III 650 | *(i,j,k)* | *(i,k,j)* | *(j,i,k)* | *(j,k,i)* | *(k,i,j)* | *(k,j,i)* | dgemm |
| без блоков | 132 | 339 | 454 | 118 | 358 | 117 | 130 |
| двумерные матрицы | 55 | 96 | 165 | 61 | 99 | 63 | ----- |
| четырехмерные матрицы | 90 | 108 | 218 | 109 | 107 | 109 | ----- |

| Время перемножения матриц 1400x1400 (100 блоков) Intel компилятор. | | | | | | | |
|---|---|---|---|---|---|---|---|
| Процессор Intel Pentium III 650 | *(i,j,k)* | *(i,k,j)* | *(j,i,k)* | *(j,k,i)* | *(k,i,j)* | *(k,j,i)* | dgemm |
| без блоков | 80 | 195 | 278 | 57 | 266 | 79 | 57 |
| двумерные матрицы | 63 | 104 | 169 | 61 | 154 | 79 | ----- |
| четырехмерные матрицы | 21 | 29 | 51 | 30 | 37 | 40 | ----- |

Есть мнение, что блочное умножение дает больший выигрыш во времени при использовании четырехмерных матриц. Так как лишь в этом случае мы получаем связанность, показанную на рисунке 11,. И лишь в этом случае в КЭШ-память будут загружаться именно блоки, в то время как для двумерных матриц меняется только порядок обхода элементов матрицы.

Тестовые расчеты показали, что использование блочных алгоритмов является более предпочтительным с точки зрения экономии времени. Компилятор Intel оказывается более эффективным в сравнении с компилятором f77. Особенно наглядно это выражается при умножении четырех мерных матриц.

| Время перемножения матриц 1400x1400 (16 блоков) Debug компилятор. | | | | | | |
|---|---|---|---|---|---|---|
| Процессор Intel Pentium 3000 | *(i,j,k)* | *(i,k,j)* | *(j,i,k)* | *(j,k,i)* | *(k,i,j)* | *(k,j,i)* |
| без блоков | 114 | 130 | 283 | 62 | 133 | 62 |
| двумерные матрицы | 113 | 114 | 249 | 61 | 113 | 61 |
| четырехмерные матрицы | 129 | 129 | 307 | 129 | 129 | 129 |
| Время перемножения матриц 1400x1400 (16 блоков) Release компилятор. | | | | | | |
| Процессор Intel Pentium 3000 | *(i,j,k)* | *(i,k,j)* | *(j,i,k)* | *(j,k,i)* | *(k,i,j)* | *(k,j,i)* |
| без блоков | 30 | 65 | 100 | 6 | 74 | 11 |
| двумерные матрицы | 30 | 63 | 94 | 4 | 63 | 5 |
| четырехмерные матрицы | 4 | 3 | 9 | 3 | 4 | 4 |

| Время перемножения матриц 1400x1400 (16 блоков) Debug компилятор. | | | | | | |
|---|---|---|---|---|---|---|
| Процессор AMD Sempron 2500+ | *(i,j,k)* | *(i,k,j)* | *(j,i,k)* | *(j,k,i)* | *(k,i,j)* | *(k,j,i)* |
| без блоков | 159 | 311 | 540 | 131 | 322 | 132 |
| двумерные матрицы | 132 | 182 | 384 | 133 | 188 | 123 |
| четырехмерные матрицы | 213 | 218 | 501 | 218 | 217 | 215 |

| Время перемножения матриц 1400x1400 (16 блоков) Release компилятор. | | | | | | |
|---|---|---|---|---|---|---|
| Процессор AMD Sempron 2500+ | *(i,j,k)* | *(i,k,j)* | *(j,i,k)* | *(j,k,i)* | *(k,i,j)* | *(k,j,i)* |
| без блоков | 65 | 184 | 263 | 14 | 197 | 31 |
| двумерные матрицы | 19 | 78 | 113 | 16 | 99 | 36 |
| четырехмерные матрицы | 5 | 5 | 11 | 5 | 6 | 5 |

Из представленных результатов видно, что использование четырех мерных матриц дает потрясающие результаты в случае использования компилятора Release, однако при использовании этого компилятора необходимо проявлять особую осторожность, так как велика возможность возникновения ошибки вычислений, за счет высокой скорости выполнения алгебраических операций.

## Литература


1. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. – Нижний Новгород.: издательство Нижегородского госуниверситета, 2003. http://www.software.unn.ac.ru/ccam/files/HTML_Version/index.html#
2. Деммель Дж. Вычислительная линейная алгебра. Теория и приложения. – М.: Мир, 2001. – 430 с.
3. Ильин В.П.. Методы неполной факторизации для решения алгебраических систем. – М.: Наука, 1995. – 287 с.
4. Ортега Дж. Введение в параллельные и векторные методы решения линейных систем. – М.: Мир, 1991. – 364 с.
5. Самарский А.А. Введение в численные методы. – М.: Наука, 1987. – 288 с.
6. Самарский А.А. Теория разностных схем. – М.: Наука, 1989. – 614 с.
7. Самарский А.А., Николаев Е.С.. Методы решения сеточных уравнений. – М.: Наука, 1987. – 601 с.