Подходы к распараллеливанию алгоритмов решения задач для уравнений в частных производных

Пример одномерной по пространству начально-краевой задачи для уравнения в частных производных параболического типа.

Сегодня мы продолжим рассматривать различные подходы к распараллеливанию алгоритмов, которые возникают при решении задач для уравнений в частных производных. Обратимся к уже рассмотреной задаче:

$$\begin{cases} \varepsilon \dfrac{\partial^2 u}{\partial x^2} - \dfrac{\partial u}{\partial t} = -u\dfrac{\partial u}{\partial x} - u^3, \quad x \in (a,b), \quad t \in (t_0, T], \\ u(a,t) = u_{left}(t), \quad u(b,t) = u_{right}(t), \quad t \in (t_0, T], \\ u(x,t_0) = u_{init}(x), \quad x \in [a,b]. \end{cases} \tag{9.1}$$

Последовательный алгоритм решения, основанный на реализации неявной схемы.

На этой лекции мы будем распараллеливать неявную схему $ROS1$ для данного нахождения сеточных значений $u_n^m = u(x_n, t_m))$ уравнения:

$$\left[E - \alpha\tau \vec{f}_{\vec{y}}(\vec{y}_m, t_m)\right]\vec{w}_1 = \vec{f}(\vec{y}_m, t_m + \frac{\tau}{2}), \qquad m = \overline{0, M-1}, \vec{y}_{m+1} = \vec{y}_m + \tau Re\vec{w}_1. \quad (9.2)$$

Где $\vec{y}_m = \left(u_1^m, u_2^m, \ldots, u_{N-2}^m, u_{N-1}^m\right)$. Теперь мы можем используя эти формулы по набору сеточных значений на текущем временном слое найти значения на следующем временном слое. Продолжая этот процесс для каждого M можно получить решение задачи. Отличительная черта схемы в этом занятии заключается в отсутствии возможности нахождения решения по отдельности. И получить сеточные решения можно только целиком, потому что нужно решить СЛАУ. Однако эта схема, хотя и требует более сложную программную реализацию, но имеет более высокую точность и устойчива, при $\alpha = \frac{1+}{2}$. И мы можем получать лучший результат на менее густых сетках. Также и явная и неявная схема по вычислительной трудоемкости одинаковы. При переходе со одного временного слоя на следующий нам требуется N^1 операций для обоих схем, так как матрица системы для неявной схемы имеет трехдиагональный вид.

$$
\begin{aligned}
f[0] &= \varepsilon\frac{y[1]-2y[0]+u_{left}(t)}{h^2}+y[0]\frac{y[1]-u_{left}(t)}{2h}+y[0]^3,\\
f[n] &= \varepsilon\frac{y[n+1]-2y[n]+y[n-1]}{h^2}+y[n]\frac{y[n+1]-y[n-1]}{2h}+y[n]^3, \quad n=\overline{2,N-2},\\
f[N-2] &= \varepsilon\frac{u_{right}(t)-2y[N-2]+y[N-3]}{h^2}+y[N-2]\frac{u_{right}(t)-y[N-3]}{2h}+y[N-2]^3.
\end{aligned}
$$

$$
\begin{aligned}
\vec{f_{\vec{y}}}[0,0] &= 1-\alpha\tau\Big(\frac{-2\varepsilon}{h^2}+\frac{y[1]-u_{left}(t)}{2h}+3y[0]^2\Big),\\
\vec{f_{\vec{y}}}[0,1] &= -\alpha\tau\Big(\frac{\varepsilon}{h^2}+\frac{y[0]}{2h}\Big),\\
\vec{f_{\vec{y}}}[n,n-1] &= -\alpha\tau\Big(\frac{\varepsilon}{h^2}-\frac{y[n]}{2h}\Big), \qquad n=\overline{1,N-3},\\
\vec{f_{\vec{y}}}[n,n] &= 1-\alpha\tau\Big(\frac{-2\varepsilon}{h^2}+\frac{y[n+1]-y[n-1]}{2h}+3y[n]^2\Big), \qquad n=\overline{1,N-3},\\
\vec{f_{\vec{y}}}[n,n+1] &= -\alpha\tau\Big(\frac{\varepsilon}{h^2}+\frac{y[n]}{2h}\Big), \qquad n=\overline{1,N-3},\\
\vec{f_{\vec{y}}}[N-2,N-3] &= -\alpha\tau\Big(\frac{\varepsilon}{h^2}-\frac{y[N-2]}{2h}\Big),\\
\vec{f_{\vec{y}}}[n,n] &= 1-\alpha\tau\Big(\frac{-2\varepsilon}{h^2}+\frac{u_{right}(t)-y[N-3]}{2h}+3y[N-2]^2\Big).
\end{aligned}
$$

где τ - шаг сетки по времени, h - шаг сетки по пространству, ε - параметр задачи, α определяет схему.

Программная реализация параллельного алгоритма решения.

```
from mpi4py import MPI
from numpy import empty, array, int32, float64 linespace, sin, pi
from matplotlib.pyplot import style, figure, axes, show

comm = MPI.COMM_WORLD
numprocs = comm.Get_size()

comm_cart = comm.Create_cart(dims = [numprocs], periods = [False],
reorder=True)
rank_cart = comm_cart.Get_rank()

def u_init(x):
    u_init = sin(3*pi*(x-1/6))
    return u_init

def u_left(t):
    u_left = -1.
    return u_left

def u_right(t):
    u_right = 1.
    return u_right
...
```

В отличие от наших прошлых программ, не хватает строчки **rank = comm.Get_rank(),** ее мы заменили, потому что на основе нашего коммуникатора **COMM_WORLD** мы создаем декартову топологию вида линейки, в которой разрешена перенумерация процессов, чтобы заданная нами топология как можно лучше ложилась на физическую топологию компьютера. Однако нужно иметь в виду, что такое не всегда возможно, и иногда результата у такого действия может не быть. Функции для формирования правых частей линейных алгебраических уравнений будут показаны позже.

Программа начинается с установки отправной точки по времени. Затем определяются параметры задачи и численного решения. После идут два привычных шага.

Массивы **rcounts** содержат информацию о числе узлов, за вычисление которых ответственен каждый MPI процесс. Также на каждом процессе выделяем место в памяти под число узлов, за которое ответственен каждый процесс. И в конце отправляем эти массивы на все процессы.

```
184     if rank_cart == 0:
185         state_time = MPI.Wtime()
186 
187     a = 0; b = 1
188     t_0 = 0; T = 2/0
189     eps = 10**(-1.5)
190 
191     N = 200; M = 300; alpha = 0.5
192 
193     h = (b - a)/N; x = linespace(a, b, N + 1)
194     tau = (T -t_0)/M; t = linespace(t_0, T, M + 1)
195 
196     if rank_cart == 0:
197         ave, res = divmod(N + 1, numprocs)
198         rcounts = empty(numprocs, dtype=int32)
199         displs = empty(numprocs, dtype = int32)
200         for k in range(0, numprocs):
201             if k < res:
202                 rcounts = ave + 1
203             else:
204                 rcounts = ave
205             if k == 0:
206                 displs[k] = 0
207             else:
208                 displs[k] = displs[k - 1] + rcount[k - 1]
209     else:
210         rcounts = None; displs = None
211 
212     N_part = array(0, dtype=int32)
213 
214     comm_cart.Scatter([rcounts, 1, MPI.INT], [N_part, 1, MPI.INT], root =0
```

```
216     if rank_cart == 0:
217         rcounts_aux = empty(numprocs, dtype=int32)
218         displs_aux = empty(numprocs, dtype = int32)
219         rcounts_aux[0] = rcounts[0] + 1
220         displs_aux[0] = 0
221         for k in range(1, numprocs-1):
222             rcounts[k] = rcounts[k] + 2
223             displs_aux[k] = displs[k] - 1
224         rcounts_aux[numprocs-1] = rcounts[numprocs-1] + 1
225         displs_aux[numprocs-1] = displs[numprocs-1] - 1
226     else:
227         rcounts_aux = None; displs_aux = None
228 
229     N_part_aux = array(0, dtype=int32)
230 
231     comm_cart.Scatter([rcounts_aux, 1, MPI.INT], [N_part_aux, 1, MPI.INT],
    root = 0)
232     comm_cart.Scatter([displs_aux, 1, MPI.INT], [displs_aux, 1, MPI.INT],
   root = 0)
233 
234     u_part_aux = empty((M +1, N_part_aux), dtype = float64)
235 
236     for n in range(N_part_aux):
237         u_part_aux[0, n] = u_init(x[displs_aux + n])
238 
239     y_part = u_part_aux[0, 1:N_part_aux-1]
240 
```

Эти массивы содержат в себе информацию о наборе узлов, с помощью которого делаются вычисления на следующем временном слое. Соответственно, на первом и последнем слое требуется один дополнительный узел, а на промежуточных — по два дополнительных узла. Аналогично создается переменная для хранения числа узлов, с помощью которых производятся вычисления. Массив **u_part_aux** — это множество узлов пространственно-временной сетки, за расчет которых ответственен соответствующий MPI процесс. Далее заполняем с помощью начального условия массив **u_part_aux** и формируем вспомогательный массив y. Для него мы исключали индексы 0 и N, что и учитывается в программе.

Рассмотрим реализацию одностадийной **схемы Розенброка**. Сначала создаем части диагоналей, которые затем вместе с частью правой части уравнения используются для расчета вектора w1. Затем рассчитываем части вспомогательного вектора y_part и заполняем соответствующие сеточные значения функции. После заполняем функции с помощью известных граничных и начальных условий. После этого идет передача сообщений между процессорами для дополнения векторов значениями, которые требуются для расчета на следующем временном слое.

Важное замечание: в нашем случае, несмотря на громоздкость выражения, с ростом количества процессоров количество действий не увеличивается, потому что каждый процесс, кроме первого и последнего, фактически передает и принимает два числа вне зависимости от числа процессов.

```
241     for m in range(M):
242         codiagonal_down_part, diagonal_part, codiagonal_up_part =
243             diagonal_preparation(u_part_aux[m], t[m], h,
244                 N_part_aux, u_left, u_right, eps, tau, alpha)
245         w_1_part = parallel_tridiagonal_matrix_algorithm(
246             codiagonal_down_part, diagonal_part, codiagonal_up_part,
247             f_part(u_part_aux[m], t[m]+tau/2, h, N_part_aux, u_left,
248             u_right, eps))
249         y_part = y_part + tau*w_1.real
250 
251         u_part_aux[m + 1, 1:N_part_aux - 1] = y_part
252         if rank_cart == 0:
253             u_part_aux[m+1, 0] = u_left(t[m+1])
254         if rank_cart == numprocs - 1:
255             u_part_aux[m+1, N_part_aux-1] = u_right(t[m+1])
256 
257         if rank_cart ==0:
258             comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, N_part_aux-2],
259                 1, MPI.DOUBLE], dest=1, sendtag=0, recvbuf=
260                 [u_part_aux[m+1, N_part_aux-1:], 1, MPI.DOUBLE],
261                 source=1, recvtag=MPI.ANY_TAG, status=None)
262         if rank_cart in range(1, numprocs-1):
263             comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, 1],
264                 1, MPI.DOUBLE], dest=rank_cart-1, sendtag=0, recvbuf=
265                 [u_part_aux[m+1, 0:], 1, MPI.DOUBLE],
266                 source=rank_cart-1, recvtag=MPI.ANY_TAG, status=None)
267             comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, N_part_aux-2],
268                 1, MPI.DOUBLE], dest=rank_cart+1, sendtag=0, recvbuf=
269                 [u_part_aux[m+1, N_part_aux-1:], 1, MPI.DOUBLE],
270                 source=rank_cart+1, recvtag=MPI.ANY_TAG, status=None)
271         if rank_cart == numprocs-1:
272             comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, 1],
273                 1, MPI.DOUBLE], dest=numprocs-2, sendtag=0, recvbuf=
274                 [u_part_aux[m+1, 0:], 1, MPI.DOUBLE],
275                 source=numprocs-2, recvtag=MPI.ANY_TAG, status=None)
276 
277 
```

```
276     if rank_cart == 0:
277         u_T = empty(N+1, dtype=float64)
278     else:
279         u_T = None
280 
281     if rank_cart == 0:
282         comm_cart.Gatherv([u_part_aux[M, 0:N_part_aux-1],N_part,
283             MPI.DOUBLE], [u_T, rcounts, displs, MPI.DOUBLE], root=0)
284     if rank_cart in range(1, numprocs-1):
285         comm_cart.Gatherv([u_part_aux[M, 1:N_part_aux-1],N_part,
286             MPI.DOUBLE], [u_T, rcounts, displs, MPI.DOUBLE], root=0)
287     if rank_cart == numprocs-1:
288         comm_cart.Gatherv([u_part_aux[M, 1:N_part_aux],N_part,
289             MPI.DOUBLE], [u_T, rcounts, displs, MPI.DOUBLE], root=0)
290 
291     if rank_cart == 0:
292         end_time = MPI.Wtime()
293         print('N={}, M={}'.format(N,M ))
294         print('Number of MPI process is {}'.format(numprocs))
295         print('Elapsed time is {:.4f} sec'.format(end_time-start_time))
296 
297         style.use('dark_background')
298         fig = figure()
299         ax = axes(xlim(a,b), ylim=(-2.0, 2.0))
300         ax.set_xlabel('x'); ax.set_ylabel('u')
301         ax.plot(x, u_T, color='y', ls='-', lw=2)
302         show()
303 
304 
```

Для проверки программы мы выделяем память под массив под сеточные значения функции в финальный момент времени T, затем собираем со всех процессов наш финальный вектор. И в конце выводим время работы и график решения в финальный момент времени.

```
133     def f_part(y, t, h, N_part_aux, u_left, u_right, eps):
134         N_part = N_part_aux - 2
135         f_part = empty(N_part, dtype=float64)
136         if rank_cart == 0:
137             f_part[0] = f[0] = eps * (y[1]  -2 * y[0]  + u_left(t))/h**2 +
    \
138             y[0]*(y[1] - u_left(t))/(2 *h) + y[0]**3
139             for n in range(2, N_part_aux-1):
140                 f_part[n-1] = eps * (y[n + 1] - 2*y[n] + y[n - 1])/h**2 +
    \
141                     y[n]*(y[n + 1] - y[n - 1])/(2*h)+y[n]**3
142         if rank_cart in range(1, numprocs-1):
143             for n in range(2, N_part_aux-1):
144                 f_part[n-1] = eps * (y[n + 1] - 2*y[n] + y[n - 1])/h**2 +
    \
145                     y[n]*(y[n + 1] - y[n - 1])/(2*h)+y[n]**3
146         if rank_cart == numprocs-1:
147             for n in range(1, N_part_aux-2):
148                 f_part[n-1] = eps * (y[n + 1] - 2*y[n] + y[n - 1])/h**2 +
    \
149                     y[n]*(y[n + 1] - y[n - 1])/(2*h)+y[n]**3
150             f_part[N_part-1] = eps*(u_right(t) - 2*y[N_part_aux-2]
151                 + y[N_part_aux-3])/h**2 + y[N_part_aux-2]*(u_right(t) -
152                 y[N_part_aux-2])/(2*h)+y[N_part_aux-2]**3
153
```

Важно понимать, что передается не весь вектор y, а только его часть длины для соответствующего процесса. Для первого и нулевого процессора учитывается тот факт, что нам известны левое и правое граничное условие. Функция, определяющая диагонали матрицы, переписывается полностью аналогично и представлена в полной программе, которая находится в материалах коллекции.

Последовательный метод прогонки совпадает с написанным ранее, а параллельный метод прогонки скопирован из 7 лекции, только comm заменен на comm_cart, а rank на rank_cart.

Обсуждение эффективности программной реализации.

Время работы последовательной версии программы на одном ядре составило 560с., при параметрах $N = 20000, M = 5000$. Видно, что с увеличением числа ядер время уменьшается, при том примерно до 126 ядер график получился весьма монотонным, доходит до 24с., а после начинает монотонно расти. Ранее передача сообщений между процессорами была организована таким образом, что время передачи сообщений не зависила от числа участвующих в вычислениях процессов. Получалось, что с ростом числа процессов время почти стремится к нулю и остается только время на прием/передачу сообщений. Рост после 126 ядер обусловлен тем, что внутри функции решающей алгоритм с трехдиагональной матрицей несколько раз используются команды $Gather$ и $Scatter$ и накладные расходы по их использованию возрастают с числом используемых для вычислений процессов. Ускорение рассчитывается как время работы на n процессорах деленное на время работы на одном процессоре. График нарисованный пунктиром это случай линейного ускорения. В нашем случае сначала ускорение (красны график) ниже линейного случая, а затем начинает уменьшаться, т.к. начинают сказываться накладные расходы на прием/передачу сообщений. Сначала же мы не достигаем линейного ускорения, потому что наш алгоритм требует конкретно $17N$ операций при переходе со одного временного слоя на другой, а до этого рассмотренный алгоритм требовал $8N$ операции. Таким образом умножив значения на $\frac{17}{8}$ мы получаем зеленый график, и видно что ускорение в нашем алгоритме более существенное. Эффективность распараллеливания мы считаем как $\frac{S_n}{n}$, в случае линейного ускорения эффективность получается равной 1. Есть условность считать эффективность порядка 0.8 очень хорошей. И так как нас интересует опять именно эффективность работы нашей программы, и то сколько времени процессоры простаивают и не делают рассчетов, мы должны снова домножить наш график на $\frac{17}{8}$. И зеленый график показывает эффективность именно нашей программы. Очень важно понимать что именно мы подразумеваем под эффективностью и для чего это нам нужно.

Спасибо!