Подходы к распараллеливанию алгоритмов решения задач для уравнений в частных производных

Продолжаем рассматривать построение и программную реализацию параллельных алгоритмов, которые могут понадобиться при решении задач для уравнений в частных производных.

На предыдущих двух лекциях мы рассматривали задачи в частных производных, одномерные по пространству. На этой лекции нам предстоит рассмотреть обобщение на многомерный случай.

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

$$
\begin{cases}
\varepsilon \left(\dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} \right) - \dfrac{\partial u}{\partial t} = -u \left(\dfrac{\partial u}{\partial x} + \dfrac{\partial u}{\partial y} \right) - u^3, \quad (x, y) \in (a, b) \times (c, d), \quad t \in (t_0, T], \\
u(a, y, t) = u_{left}(y, t), \quad u(b, y, t) = u_{right}(y, t), \quad y \in (c, d), \quad t \in (t_0, T], \\
u(x, c, t) = u_{bottom}(x, t), \quad u(x, d, t) = u_{top}(x, t), \quad x \in [a, b], \quad t \in (t_0, T], \\
u(x, y, t_0) = u_{init}(x, y), \quad (x, y) \in [a, b] \times [c, d].
\end{cases}
\tag{10.1}
$$

Из постановки видно, что задача нелинейная, а значит аналитически её решить не представляется возможным. На помощь приходят численные методы. Подробно решение конкретно этой задачи было разобрано в курсе ”Численные методы лекция 27. Напомним лишь общую идею.

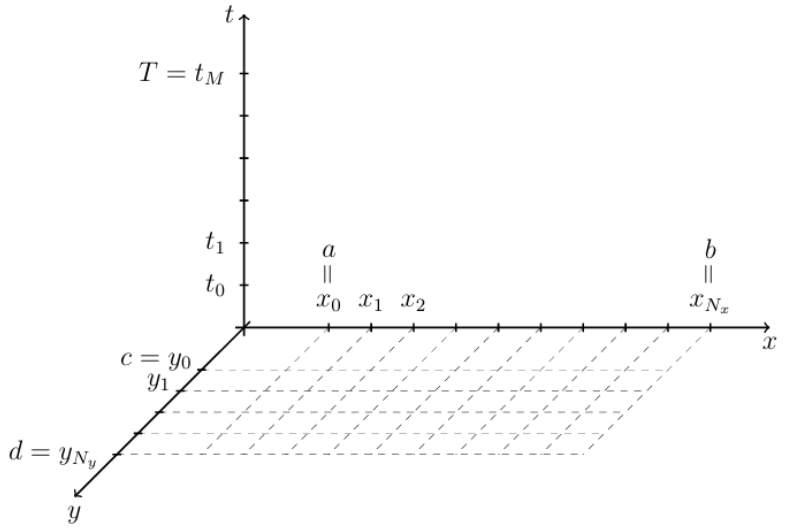

Рис. 10.1: Пространственно-временная сетка для задачи 10.1

На нулевом слое по времени все значения известны абсолютно точно из начальных условий. На последующих слоях абсолютно точно известны только значения на границе пространственной области.

Для построения разностной схемы будем использовать шаблон, изображённый на рис. 10.2.

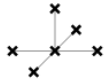

Рис. 10.2: Шаблон для явной схемы алгоритма решения задачи 10.1

Применяя указанный шаблон, получим:

$$u_{i,j}^{m+1} = u_{i,j}^m + \tau \left(\varepsilon \left(\frac{u_{i+1,j}^m - 2u_{i,j}^m + u_{i-1,j}^m}{h_x^2} + \frac{u_{i,j+1}^m - 2u_{i,j}^m + u_{i,j-1}^m}{h_y^2} \right) + \right.$$
$$\left. + u_{i,j}^m \left(\frac{u_{i+1,j}^m - u_{i-1,j}^m}{2h_x} + \frac{u_{i,j+1}^m - u_{i,j-1}^m}{2h_y} \right) + \left(u_{i,j}^m \right)^3 \right).$$

Программная реализация последовательного алгоритма решения

```
from numpy import empty, linspace, tanh, savez
import time

def u_init(x, y):
    return 0.5*tanh(1/eps*((x-0.5)**2 + (y-0.5)**2 - 0.35**2)) - 0.17

def u_left(y, t):
    return 0.33

def u_right(y, t):
    return 0.33

def u_top(x, t):
    return 0.33

def u_bottom(x, t):
    return 0.33

start_time = time.time()

a = -2; b = 2; c = -2; d = 2
t_0 = 0; T = 4; eps = 10**(-1.5)

N_x = 50; N_y = 50; M = 500
```

Такая большая разница между параметрами сетки по пространству и по времени взята не просто так, мы делаем так для того, чтобы численный счёт не разваливался. Это – особенность явной схемы.

Далее заполняем значения на нулевом временном слое и значения на границе:

```
x, h_x = linspace(a, b, N_x+1, retstep=True)
y, h_y = linspace(c, d, N_y+1, retstep=True)
t, tau = linspace(t_0, T, M+1, retstep=True)
```

```
28
29  u = empty((M+1, N_x+1, N_y+1))
30
31  for i in range(N_x+1):
32      for j in range(N_y+1):
33          u[0, i, j] = u_init(x[i], y[j])
34
35  for m in range(M+1):
36      for j in range(1, N_y):
37          u[m, 0, j] = u_left(y[j], t[m])
38          u[m, N_x, j] = u_right(y[j], t[m])
39
40      for i in range(0, N_x+1):
41          u[m, i, N_y] = u_top(x[i], t[m])
42          u[m, i, 0] = u_bottom(x[i], t[m])
```

Наконец, пробегаем по всем слоям и вычисляем значения функции внутри пространственной области:

```
43  for m in range(M):
44      for i in range(1, N_x):
45          for j in range(1, N_y):
46              d2x = (u[m, i+1, j] - 2*u[m, i, j] + u[m, i-1, j]) / h_x**2
47              d2y = (u[m, i, j+1] - 2*u[m, i, j] + u[m, i, j-1]) / h_y**2
48
49              d1x = (u[m, i+1, j] - u[m, i-1, j]) / (2*h_x)
50              d1y = (u[m, i, j+1] - u[m, i, j-1]) / (2*h_y)
51
52              u[m+1, i, j] = u[m, i, j] + tau*(eps*(d2x + d2y) +
53                             u[m, i, j]*(d1x + d1y) + u[m, i, j]**3)
54
55  print(f"Elapsed time is {time.time() - start_time:.4f} sec")
56  savez("results", x=x, y=y, t=t, u=u)
```

После завершения работы программы можно построить график:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

frames = 100
results = np.load("results.npz")

x = results["x"]
y = results["y"]
t = results["t"]
u = results["u"]

fig, ax = plt.subplots()

xx, yy = np.meshgrid(x, y, indexing="ij")

def plot_frame(s):
    ax.clear()

    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_aspect("equal")

    m = s * (u.shape[0] // frames)

    ax.pcolor(xx, yy, u[m], shading="auto")
    ax.set_title(f"m = {m}")

anim = FuncAnimation(fig, plot_frame, interval=60, frames=frames)
anim.save("results.gif")
```

Параллельный алгоритм решения в случае одномерного деления пространственной области на блоки

Рассмотрим один временной слой и разобьём его на части по процессам (см. рис. 10.3). Бледно-серым показаны значения, которые необходимы нескольким процессам, то есть их необходимо будет пересылать.

Несложно заметить, что длина пересылаемых сообщения будет равна $N_y - 1$ (при этом граничные процессы передают и получают по одному сообщению на каждой итерации по времени, а все остальные – по два).

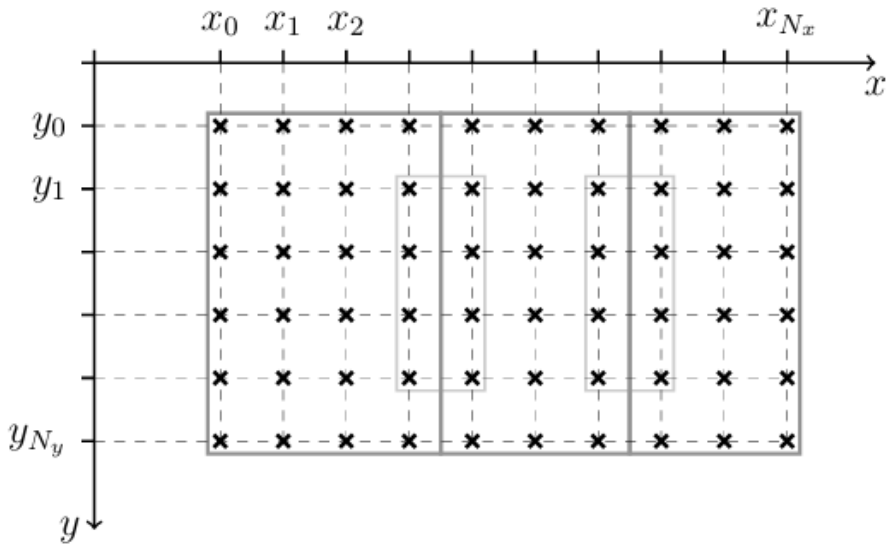

Рис. 10.3: Схема разделения данных по MPI процессам

Программная реализация параллельного алгоритма решения

Полная версия программы выложена в разделе ”Материалы к лекции”под номером 10-2.

```
1  from mpi4py import MPI
2  from numpy import empty, int32, float64, linspace, tanh, savez
3
4  comm = MPI.COMM_WORLD
5  numprocs = comm.Get_size()
```

Вводим виртуальную декартову топологию типа ”линейка”(т.к. одномерная). Аргумент `reorder=True` позволяет расположить физически близкие процессоры под близкими номерами.

```
6  comm_cart = comm.Create_cart(dims=[numprocs],
7                               periods=[False], reorder=True)
8  rank_cart = comm_cart.Get_rank()
9
10 ...
```

После этого снова задаём все параметры задачи (строки 4-17, 21-27 в последовательной программе). Изменится только измерение времени:

```
11 if rank_cart == 0:
12     start_time = MPI.Wtime()
```

Снова введём функции для вычисления вспомогательных массивов `rcounts` и `displs` и применим её для вычисления длин и сдвигов для частей массива `u`.

```
def auxiliary_arrays_determination(M, num):
    ave, res = divmod(M, num)

    rcounts = empty(num, dtype=int32)
    displs = empty(num, dtype=int32)

    for k in range(num):
        rcounts[k] = ave + 1 if k < res else ave
        displs[k] = 0 if k == 0 else displs[k-1] + rcounts[k-1]

    return rcounts, displs

rcounts_N_x, displs_N_x = auxiliary_arrays_determination(N_x+1, numprocs)

N_x_part = rcounts_N_x[rank_cart]
```

Помним, что нужно также хранить дополнительные столбцы слева и справа.

```
if rank_cart in [0, numprocs-1]:
    N_x_part_aux = N_x_part + 1
else:
    N_x_part_aux = N_x_part + 2

displs_N_x_aux = displs_N_x - 1
displs_N_x_aux[0] = 0

displ_x_aux = displs_N_x_aux[rank_cart]

u_part_aux = empty((M+1, N_x_part_aux, N_y+1), dtype=float64)
```

Заполняем вспомогательный массив значениями в начальный момент времени и на границах:

```
for i in range(N_x_part_aux):
    for j in range(N_y+1):
        u_part_aux[0, i, j] = u_init(x[displ_x_aux+i], y[j])

for m in range(M):
    for j in range(1, N_y):
        if rank_cart == 0:
            u_part_aux[m, 0, j] = u_left(y[j], t[m])
        if rank_cart == numprocs-1:
            u_part_aux[m, 0, j] = u_right(y[j], t[m])

    for i in range(N_x_part_aux):
        u_part_aux[m, i, N_y] = u_top(x[displ_x_aux+i], t[m])
        u_part_aux[m, i, 0] = u_bottom(x[displ_x_aux+i], t[m])
```

Переходим к основной части программы.

```
for m in range(M):

    for i in range(1, N_x_part_aux-1):
        for j in range(1, N_y):
            d2x = (u_part_aux[m, i+1, j] -
                   2*u_part_aux[m, i, j] +
                   u_part_aux[m, i-1, j]) / h_x**2
            d2y = (u_part_aux[m, i, j+1] -
                   2*u_part_aux[m, i, j] +
                   u_part_aux[m, i, j-1]) / h_y**2

            d1x = (u_part_aux[m, i+1, j] -
                   u_part_aux[m, i-1, j]) / (2*h_x)
            d1y = (u_part_aux[m, i, j+1] -
                   u_part_aux[m, i, j-1]) / (2*h_y)

            u_part_aux[m+1, i, j] = u_part_aux[m, i, j] + \
                                    tau*(eps*(d2x + d2y) +
                                    u_part_aux[m, i, j]*(d1x + d1y) +
                                    u_part_aux[m, i, j]**3)
```

Остаётся только передать значения соседним процессам. Напомним, что в данной реализации простаивающих процессов нет, а размер пересылаемых сообщений не зависит от количества процессов.

```
    if rank_cart > 0:
        comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, 1],
                                    N_y+1, MPI.DOUBLE],
                           dest=rank_cart-1,
                           sendtag=0,
                           recvbuf=[u_part_aux[m+1, 0],
                                    N_y+1, MPI.DOUBLE],
                           source=rank_cart-1,
                           recvtag=MPI.ANY_TAG,
                           status=None)

    if rank_cart < numprocs-1:
        comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, N_x_part_aux-2],
                                    N_y+1, MPI.DOUBLE],
                           dest=rank_cart+1,
                           sendtag=0,
                           recvbuf=[u_part_aux[m+1, N_x_part_aux-1],
                                    N_y+1, MPI.DOUBLE],
                           source=rank_cart+1,
                           recvtag=MPI.ANY_TAG,
                           status=None)
```

```
94
95  if rank_cart == 0:
96      end_time = MPI.Wtime()
```

Не всегда разумно собирать все полученные данные на одном процессе, так как для них может просто не хватить памяти. Можно собрать данные только на некоторых временных слоях. Мы, для простоты, просто соберём данные в финальный момент времени.

```
97  if rank_cart == 0:
98      u_T = empty((N_x+1, N_y+1), dtype=float64)
99
100 if rank_cart == 0:
101     comm_cart.Gatherv([u_part_aux[M, :-1], N_x_part*(N_y+1), MPI.DOUBLE],
102                       [u_T, rcounts_N_x*(N_y+1), displs_N_x*(N_y+1),
103                        MPI.DOUBLE], root=0)
104
105 elif rank_cart in range(1, numprocs-1):
106     comm_cart.Gatherv([u_part_aux[M, 1:-1], N_x_part*(N_y+1), MPI.DOUBLE],
107                       None, root=0)
108
109 elif rank_cart == numprocs-1:
110     comm_cart.Gatherv([u_part_aux[M, 1:], N_x_part*(N_y+1), MPI.DOUBLE],
111                       None, root=0)
112
113 if rank_cart == 0:
114     print(f"Elapsed time is {end_time - start_time:.4f} sec")
115     savez("results", x=x, y=y, u_T=u_T)
```

Обратите внимание, что мы спланировали алгоритм таким образом, что передаваемые массивы расположены в памяти непрерывно (индекс `j` в массиве `u_part_aux` – последний). В противном случае массивы `rcounts` и `displs` были бы намного сложнее.

Параллельный алгоритм решения в случае двумерного деления пространственной области на блоки

Предыдущий алгоритм был лишь обобщением одномерного по пространству алгоритма, рассмотренного в лекции 8. Попробуем более строго учесть то, что область по пространству двумерная. Будем делить область на двумерные блоки. В дальнейшем это позволит нам использовать декартову топологию типа ”сетка”.

Схему разбиения можно посмотреть на рис. 10.4, обмен данными между соседними процессами показан бледно-серыми квадратами.

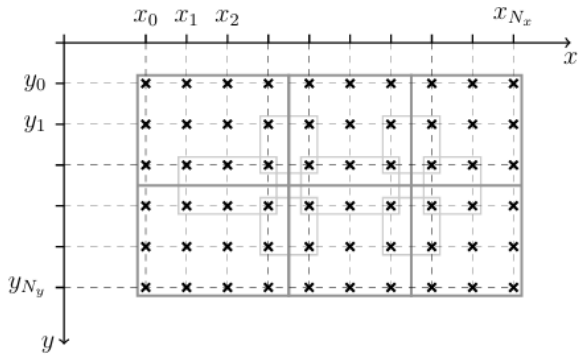

Рис. 10.4: Схема разделения данных по MPI процессам, двумерный случай

Оценим объём пересылаемой информации на каждой итерации по времени. Пусть $N_x = N_y = N$, n – количество процессов. Вводим топологию одинаково по x и по y, то есть предполагаем, что корень из n извлекается. Тогда длина каждого блока – $\frac{N}{\sqrt{n}}$. Так как мы на каждой итерации передаём и получаем по 4 сообщения, полученный объём – $\frac{8N}{\sqrt{n}}$. Напомним, что в предыдущем алгоритме мы передавали по 2 сообщения длины N, то есть объём передаваемой информации равнялся $4N$.

В новом подходе объём передаваемых сообщений будет уменьшаться при увеличении числа процессов.

Программная реализация параллельного алгоритма решения

Будем считать, что число процессов – квадрат натурального числа больше единицы.

```
1  from mpi4py import MPI
2  from numpy import empty, int32, float64, linspace, tanh, sqrt, savez
3
4  comm = MPI.COMM_WORLD
5  numprocs = comm.Get_size()
6
7  num_row = num_col = int32(sqrt(numprocs))
8
9  comm_cart = comm.Create_cart(dims=(num_row, num_col),
10                               periods=(False, False), reorder=True)
11 rank_cart = comm_cart.Get_rank()
12
13 ...
```

После введения виртуальной топологии снова задаём все параметры задачи (строки 4-17, 21-27 в последовательной программе), а также сохраняем начальное время и задаём функцию для вычисления вспомогательных массивов (строки 11-23 в предыдущей параллельной программе).

Создаём вспомогательные массивы, в этот раз уже по двум пространственным переменным:

```
14 rcounts_N_x, displs_N_x = auxiliary_arrays_determination(N_x+1, num_col)

15 rcounts_N_y, displs_N_y = auxiliary_arrays_determination(N_y+1, num_row)
```

Далее определим координаты текущего процесса внутри топологии и опередим размер и сдвиги частей массива u.

```
my_row, my_col = comm_cart.Get_coords(rank_cart)

N_x_part = rcounts_N_x[my_col]
N_y_part = rcounts_N_y[my_row]

if my_col in [0, num_col-1]:
    N_x_part_aux = N_x_part + 1
else:
    N_x_part_aux = N_x_part + 2

if my_row in [0, num_row-1]:
    N_y_part_aux = N_y_part + 1
else:
    N_y_part_aux = N_y_part + 2

displs_N_x_aux = displs_N_x - 1
displs_N_x_aux[0] = 0

displs_N_y_aux = displs_N_y - 1
displs_N_y_aux[0] = 0

displ_x_aux = displs_N_x_aux[my_col]
displ_y_aux = displs_N_y_aux[my_row]

u_part_aux = empty((M+1, N_x_part_aux, N_y_part_aux), dtype=float64)
```

Вычисляем и задаём начальные и граничные условия:

```
for i in range(N_x_part_aux):
    for j in range(N_y_part_aux):
        u_part_aux[0, i, j] = u_init(x[displ_x_aux+i], y[displ_y_aux+j])

for m in range(1, M+1):
```

```
    for j in range(1, N_y_part_aux-1):
        if my_col == 0:
            u_part_aux[m, 0, j] = u_left([displ_y_aux+j], t[m])
        if my_col == num_col-1:
            u_part_aux[m, -1, j] = u_right([displ_y_aux+j], t[m])

    for i in range(N_x_part_aux):
        if my_row == 0:
            u_part_aux[m, i, 0] = u_bottom([displ_x_aux+i], t[m])
        if my_row == num_row-1:
            u_part_aux[m, i, -1] = u_top([displ_x_aux+i], t[m])
```

Приступаем к основной части программы. Вычисления на каждой итерации почти не поменяются (изменится только диапазон индексов по y):

```
for m in range(M):

    for i in range(1, N_x_part_aux-1):
        for j in range(1, N_y_part_aux-1):
            d2x = (u_part_aux[m, i+1, j] -
                   2*u_part_aux[m, i, j] +
                   u_part_aux[m, i-1, j]) / h_x**2
            d2y = (u_part_aux[m, i, j+1] -
                   2*u_part_aux[m, i, j] +
                   u_part_aux[m, i, j-1]) / h_y**2

            d1x = (u_part_aux[m, i+1, j] -
                   u_part_aux[m, i-1, j]) / (2*h_x)
            d1y = (u_part_aux[m, i, j+1] -
                   u_part_aux[m, i, j-1]) / (2*h_y)

            u_part_aux[m+1, i, j] = u_part_aux[m, i, j] + \
                                    tau*(eps*(d2x + d2y) +
                                    u_part_aux[m, i, j]*(d1x + d1y) +
                                    u_part_aux[m, i, j]**3)
```

С операциями пересылки вверх и вниз могут возникнуть сложности. При извлечении среза по среднему индексу пересчёт индексов считается неочевидно. Чтобы избежать трудностей, будем просто копировать данные во временные массивы и пересылать их содержимое. Если этого не сделать, данные не будут храниться непрерывно в памяти.

```
98      if my_row > 0:
99          temp_array_send = u_part_aux[m+1, :, 1].copy()
100         temp_array_recv = empty(N_x_part_aux, dtype=float64)
101
102         comm_cart.Sendrecv(sendbuf=[temp_array_send,
103                                     N_x_part_aux, MPI.DOUBLE],
104                            dest=(my_row-1)*num_col + my_col,
105                            sendtag=0,
106                            recvbuf=[temp_array_recv,
107                                     N_x_part_aux, MPI.DOUBLE],
108                            source=(my_row-1)*num_col + my_col,
109                            recvtag=MPI.ANY_TAG,
110                            status=None)
111
112         u_part_aux[m+1, :, 0] = temp_array_recv
113
114     if my_row < num_row-1:
115         temp_array_send = u_part_aux[m+1, :, -2].copy()
116         temp_array_recv = empty(N_x_part_aux, dtype=float64)
117
118         comm_cart.Sendrecv(sendbuf=[temp_array_send,
119                                     N_x_part_aux, MPI.DOUBLE],
120                            dest=(my_row+1)*num_col + my_col,
```

После основной части измеряем финальное время и собираем значения функции в финальный момент времени. Чтобы отправляемые данные располагались в памяти непрерывно, отправляем столбцы с каждого процесса по очереди. Вообще говоря, можно реализовать этот блок по-другому, более эффективно.

```
129 if rank_cart == 0:
130     end_time = MPI.Wtime()
131     print(f"Elapsed time is {end_time - start_time:.4f} sec")
132 
133     u_T = empty((N_x+1, N_y+1), dtype=float64)
134 
135     for row in range(num_row):
136         for col in range(num_col):
137             if row == col == 0:
138                 for i in range(N_x_part):
139                     u_T[i, :N_y_part] = u_part_aux[M, i, :N_y_part]
140             else:
141                 for i in range(rcounts_N_x[col]):
142                     comm_cart.Recv([u_T[displs_N_x[col]+i,
143                                         displs_N_y[row]:],
144                                     rcounts_N_y[row], MPI.DOUBLE],
145                                    source=row*num_col+col,
146                                    tag=0,
147                                    status=None)
148 
149     savez(x=x, y=y, u_T=u_T)
150 
151 else: # rank_cart != 0
152     for i in range(N_x_part):
153         if my_row == 0:
154             comm_cart.Send([u_part_aux[M, 1+i], N_y_part, MPI.DOUBLE],
155                            dest=0, tag=0)
156 
157         else:
158             if my_col == 0:
159                 comm_cart.Send([u_part_aux[M, i, 1:],
160                                 N_y_part, MPI.DOUBLE],
161                                dest=0, tag=0)
```

Обсуждение эффективности программной реализации.

Тестирование проводилось на суперкомпьютере ”Ломоносов-2”. Время работы последовательной программы на 1 ядре – 791 с. (для параметров $N_x = N_y = 200$, $M = 4000$, также было изменено значение параметра $T = 4$).

На рис. 10.5 приведён график времени работы двух подходов, на рис. 10.6 – график ускорения. Видно, что первый подход на 100 ядрах выходит на плато, так как время на коммуникацию остаётся постоянным. Во втором подходе, в отличие от первого, время продолжает убывать.

Также можно заметить, что при малом количестве ядер ускорение близко к линейному, а потом заметно проседает. Так происходит из-за того, что мы выходим за границы одного узла.

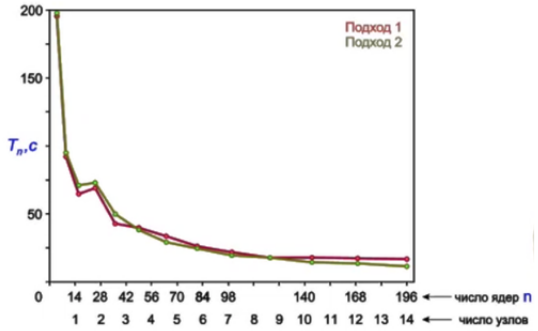

Рис. 10.5: График времени работы параллельных программ

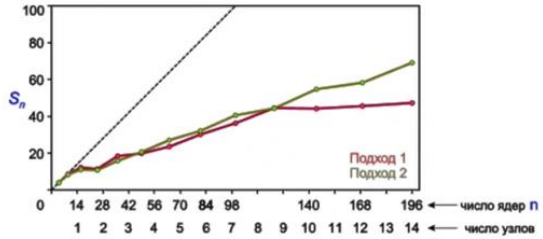

Рис. 10.6: График ускорения параллельных программ

На рис. 10.7 можно увидеть график эффективности. Напомним, что значения эффективности распараллеливания в интервале $[0.8, 1]$ говорят о удачной программной реализации. В нашем случае эффективность близка к 0.4, что для программы, написанной за одну лекцию, довольно неплохо.

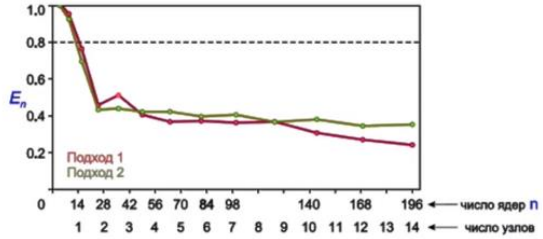

Рис. 10.7: График эффективности работы параллельных программ

Спасибо!