Учим процессор расходовать ресурсы

Альберт Нагапетян* Семен Пьянков†

1. Постановка задачи

В современном мире мы постоянно встречаем и используем вычислительные устройства, такими как телефоны, компьютеры, мейнфремы оснащенные процессорами, которые обрабатывают команды, процессы посылаемые операционной системой или гипервизором. В каждый момент процессор времени может делать заданное число операций, что является его скорость. Это число операций делится в некоторой пропорции между доступными для выполнения в текущий такт процессами. Процессор может корректировать скорость своей работы в каждый из T моментов времени. Скорость процессора в каждый момент времени должна лежать в некотором отрезке допустимых значений, а также скорости в два последующих момента времени не могут отличаться по абсолютной величине сильнее чем на некоторое число R. Энергия, потребляемая процессором в момент времени t, зависит от его скорости в этот момент времени, по закону $\alpha + \beta x + \gamma x^2$.

Задача минимизировать суммарную энергию, потребляемую процессором, и найти распределение ресурсов по процессам и скоростей по времени. Энергия в каждый момент времени вычисляется по закону

$$W(\mathbf{x}) = \sum_{i=0}^{T-1} \alpha + \beta x_i + \gamma x_i^2,$$

где x_i - скорость процессора в момент времени i. Следовательно нужно минимизировать сумму энергий, затраченных за все время работы процессора:

$$\min_{\mathbf{x}} \sum_{i=0}^{T-1} \alpha + \beta x_i + \gamma x_i^2$$

На скорость x_i накладываются ограничения:

$$S_{\min} \leq x_i \leq S_{\max},$$

где $S_{\min}, S_{\max}$ - заданные минимальная и максимальная скорости. Кроме того, ограничения касаются и приращения скорости, то есть скорости выполнения двух последовательных задач отличаются не более, чем на заданную константу R:

$$|x_i - x_{i-1}| \leq R \quad \forall i = 1 \dots T-1,$$

*albert.nagapetyan@phystech.edu

†pyankov.sa@phystech.edu

где T - суммарное время работы процессора.

Скорость в каждый момент времени зависит от того, сколько ресурсов нужно затратить на процессы в каждый момент времени:

$$x_i = \sum_{j=0}^{n-1} X_{ij},$$

где n - число jбщее число процессов, X_{ij} - объем ресурсов, затраченный в i-ый момент времени j-ым процессом.

Кроме того, есть ограничение на суммарную работу, которую должен выполнить процессор: суммарный объём вычислений для каждой работы должен быть не меньше требуемого для её выполнения

$$\sum_{i=0}^{T-1} X_{ij} \geq W_j,$$

где W_j - работа, которую надо затратить для j-го процесса, обозначим $W \in \mathbf{R^n}, W = \|W_j\|_{j=1}^{n-1}$

Получаем задачу:

$$\begin{aligned}
&\min_{\mathbf{x}} \alpha T + \boldsymbol{\beta}_*^{\mathbf{T}} \mathbf{x} + \gamma \mathbf{x}^\top \mathbf{x}, \text{ где } \boldsymbol{\beta}_* \in \mathbf{R^T}, \ \boldsymbol{\beta}_{*i} = \beta, \quad i = 0..n-1 \\
&s.t. \ \mathbf{x} \succeq \mathbf{S_{min}} \\
&\mathbf{x} \preceq \mathbf{S_{max}} \\
&|x_i - x_{i-1}| \leq R \\
&\mathbf{y} \succeq \mathbf{W}, \text{ где } y_j = \sum_{i=0}^{T-1} X_{ij} \\
&x_i = \sum_{i=0}^{n-1} X_{ij} \\
&x_{ij} \geq 0
\end{aligned} \tag{1}$$

2. Анализ задачи и описание алгоритмов её решения

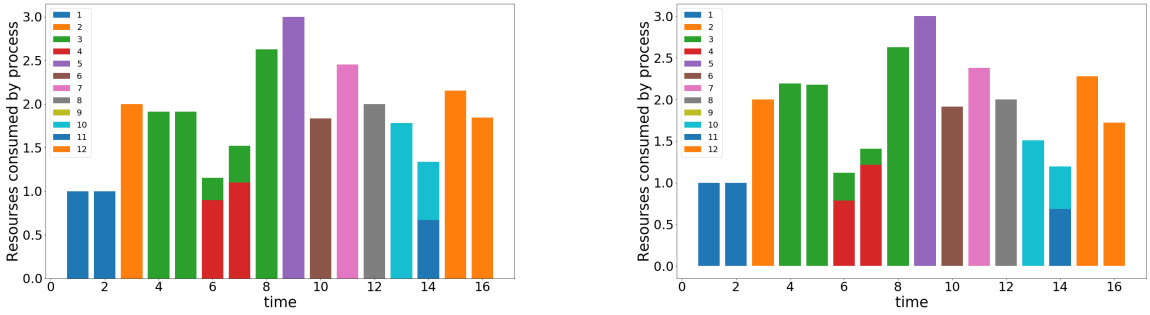
Недостатки постановки - не реализовано в матричной форме. Для краткости записи (да и простоты вычисления) было бы удобнее реализовать в виде матриц и векторов, но не все ограничения позволяют это сделать. Возможно, введя дополнительные переменные, можно было бы реализовать, но между экономией времени работы человека и машины мы решили сделать выбор в свою пользу.

Достоинства постановки - составлена задача квадратичного программирования с ограничениями типа равенства и неравенства. Эта задача выпуклая, а значит ее можно решать такими методами как: метод внутренней точки (в нашем случае использовался ECOS [1]), ADMM [2] (в нашем случае использовался SCS). В общем случае, эти методы решают выпуклые конические задачи, нашу задачу можно свести к задаче

SOCP (second order cone problem), это делает cvxpy (функция get_problem_data). Плюсы этих методов в универсальности и отсутствии необходимости подключать какие-либо дополнительные библиотеки, однако некоторые методы не масштабируемы на задачи целочисленного программирования.

3. Результаты экспериментов

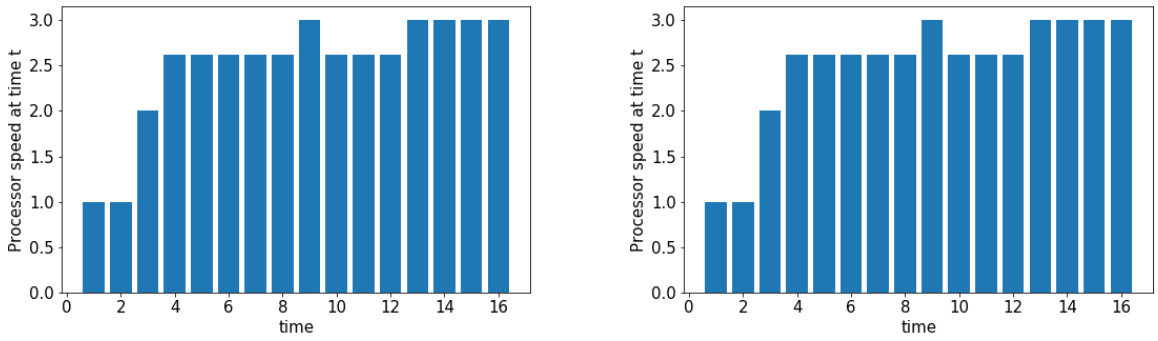
В результате вычислений методами ECOS и SCS мы получили распределение ресурсов по процессам(Рис. 1 и Рис. 2), которое отличается при решении разными методами, однако оба решения удовлетворяют ограничениям и дают одинаковое суммарное значение энергии. Решения получаются различными, так как они лежат на одной линии уровня, отвечающей минимальному значению функции.



(a) ECOS (b) SCS

Рис. 1: Распределение ресурсов по процессам

И значение скорости процессора во времени:



(a) ECOS (b) SCS

Рис. 2: Распределение скорости процессора

Также была оценена зависимость точности вычисления от числа итераций, см. Рис. 3

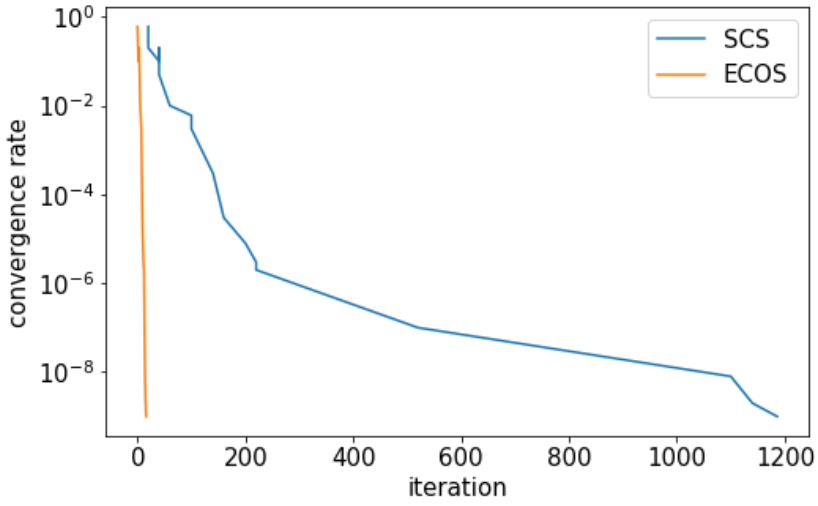


Рис. 3: Зависимость точности от числа итераций

Сравнение SCS и ECOS при решении задачи с точностью 10^{-9}:

	iter	Total time,ms	Dual gap
ECOS	16	110	$9 \cdot 10^{-7}$
SCS	1186	129	$2.4 \cdot 10^{-12}$

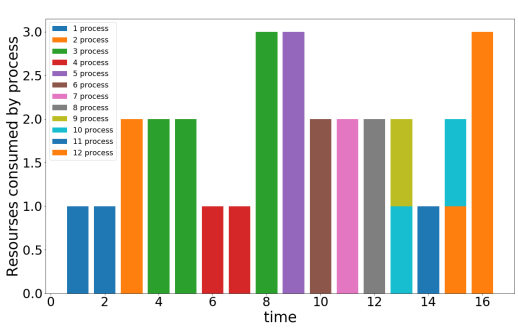
Получили, что ECOS нужно сделать гораздо меньше итераций, чем SCS для достижения опеределенной точности (как это видно из графика на Рис. 3), также скорость выполнения SCS выше, то есть выгоднее пользоваться ECOS (метод внутренней точки). Также получили оценки зазора двойственности: $9 \cdot 10^{-7}$ для ECOS и $2.4 \cdot 10^{-12}$ для SCS, т.е. при решении методом SCS зазор двойственности меньше.

Однако, после некоторых раздумий, было принято предположение, что нагрузка (равно как и скорость) процессора должна быть целочисленная. Тогда была найдена модификация ECOS - ECOS_BB, результат можно видеть на Рис. 4a и Рис. 4b:

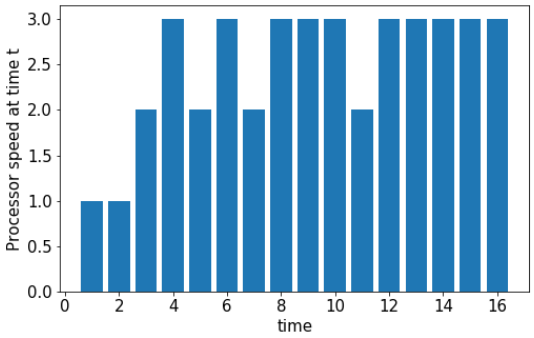
4. Обсуждение

Мы решили задачу, нашли распределения энергии и скорости процессора в каждый момент времени (в том числе и для целочисленного случая - более приближенного к реальности).

При решении возникла проблема с составлением задачи математического программирования. Чтобы упростить выражение, хотелось получить векторно-матричную форму записи. Но это не будет иметь особого смысла ($\succeq$ реализуется все равно поэлементно, да и не очень понятно, как оформить ограничение $|x_i - x_{i-1}| \leq R$



(a) Распределение нагрузки



(b) Скорость

Рис. 4: ECOS_BB

без введения дополнительных переменных).

Еще одна трудность - предположение целочисленности задачи. Из-за этого пришлось использовать ECOS_BB, который, конечно, дал решение, отличающееся от непрерывного случая: отличается как распределение нагрузки (это можно видеть, если сравнить график на Рис. 4a с графиками на Рис. 1a и Рис. 1b), так и распределение скоростей по времени (видно при сравнении графика на Рис. 4b с графиками на Рис. 2a и Рис. 2b). Решение, полученное для непрерывной задачи, дает меньшие затраты энергии 162.1 в сравнении с 163.9, полученными при решении целочисленной задачи, но на практике такое распределение вряд ли будет столь экономично. Поэтому для реального применения целочисленное решение выглядит более приемлемым. Скорее всего, при достаточно больших затратах, непрерывный случай будет точнее отображать реальные условия и отличие дискретного решения от непрерывного не будет отличаться так сильно.

При первом прочтении условия показалось, что задача достаточно узкая, поскольку нужно знать точное расписание занятости процессора (то есть нужно знать, какие требуется запустить процессы в какое время и на какое время). Но как высяснилось, бывают и такие системы[3]. К тому же, если не вдаваться в подробности, то планировщик операционной системы примерно так и составляет расписание (конечно, задача усложняется тем, что мы не знаем, какие к нам придут задачи в будущем, но те задачи, которые известны, распределяются с минимальным потреблением ресурсов). В целом это сильно упрощенная задача, но можно попытаться её обобщить на многопоточную архитектуру, учесть ограничение на число процессов и попробовать реализовать решение задачи для реального планировщика (хотя в действительности проще пользоваться готовыми алгоритмами планирования, например из невытесняющих это FIFO, из вытесняющих - циклическая).

Кроме того, в многопоточных системах при использовании пула потоков возникает необходимость в статическом планировании. Потоки для фрагментов задания будут определяться заранее, наше решение поможет оптимизировать это распределение.

Еще одно применение - распределение задач в проекте внутри команды. Только необходимо ограничить число одновременно-запущенных процессов числом членов команды (или какой-то величиной, зависящей от этого числа).

5. Список литературы

[1] A. Domahidi, E. Chu, and S. Boyd. ECOS: An SOCP solver for embedded systems. In *European Control Conference (ECC)*, pages 3071–3076, 2013.

[2] S. Boyd. Alternating direction method of multiplier. In *Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers (Boyd, Parikh, Chu, Peleato, Eckstein)*, 2011.

[3] Herbert Bos Andrew S. Tanenbaum. Scheduling. In *Modern operating systems*, 2015.

6. Роли

6.1 Формализация задачи (составление задачи математического программирования)

— Семен Пьянков, Альберт Нагапетян

6.2 Реализация на Python:

6.2.1 Поиск алгоритмов, которые позволяют решить эту задачу

— Альберт Нагапетян, Семен Пьянков

6.2.2 Написание кода, решающего задачу с использованием этих алгоритмов

— Альберт Нагапетян

6.2.3 Отладка

— Альберт Нагапетян

6.3 Обоснование решения

— Семен Пьянков, Альберт Нагапетян

6.4 Оформление отчета

— Семен Пьянков

6.5 Внесение правок в отчет

— Выполнялось поровну каждым участником

6.6 Создание корректно оформленного репозитория на Github

— Семен Пьянков

На самом деле разделение очень условное, поскольку вносились изменения разного плана всеми участниками во все части проекта.