ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ № ?, 2014, с. ??–??

УДК 004.75

С. А. Смирнов

Облачная система подстройки параметров алгоритмов

Аннотация. В статье описана система, предназначенная для поиска лучших (в том или ином смысле) настроек алгоритма. Система работает и с дискретными, и с непрерывными параметрами, а также использует параллелизм, предоставляемый публичными облаками. В работе представлено общее описание системы, метод оценки производительности алгоритма в облаке и численные результаты использования системы на нескольких наборах тестовых задач.

Ключевые слова и фразы: оптимизация параметров алгоритмов, облачные вычисления.

Введение

Наблюдаемый в настоящее время активный рост числа поставщиков инфраструктуры как услуги (IaaS) — это прямое следствие удешевления стоимости вычислений и увеличения уровня автоматизации инфраструктуры. Облачные сервисы позволяют сильнее автоматизировать работу программиста, делая его более продуктивным. Это можно рассматривать как еще один шаг в непрерывном процессе добавления дополнительных уровней абстракции к компьютерной системе: языки программирования высокого уровня, интерактивная отладка, автоматические системы сборки и т.д. Облака позволяют разработчикам быстро создавать изолированные среды для разработки и тестирования, быстро вводить в эксплуатацию виртуальные машины с требуемым ПО, производить тестирование на нагрузку и масштабируемость.

Существует множество задач, автоматизируемых с помощью облаков. Одной из таких задач является тонкая настройка алгоритма, чтобы он работал лучше в том или ином смысле, например, выполнялся бы за меньшее время. Эта задача может быть решена многими

способами: изменением жестко закодированных внутри программы параметров, анализом исходного кода программы, настройкой параметров внутри конфигурационных файлов программы. В нашем исследовании мы выбрали последний из перечисленных способов: тонкую настройку конфигурационных параметров солвера SCIP (Solving Constraint Integer Programs). В настоящее время SCIP [1] представляет собой один из самых быстрых некоммерческих солверов для смешанного целочисленного программирования (MIP) и целочисленного нелинейного программирования (MINLP). Это также и фреймворк для целочисленного программирования ограничениями и для метода branch-cut-and-price. Фреймворк обеспечивает полный контроль над процессом решения и доступ к состоянию солвера и его компонентов. Не смотря на то что SCIP — очень быстрый солвер, даже с параметрами по умолчанию, как правило возможно подстроить его параметры для решения конкретных классов задач. Сделать это было бы довольно просто, если бы и параметров, и тестовых задач было немного. Однако SCIP очень хорошо настраивается, имея более тысячи параметров. Очевидно, с таким огромным числом параметров, точная их настройка займет немало времени. Именно поэтому мы решили автоматизировать этот процесс, сделав систему, выбирающую оптимальные настройки для набора пробных задач. Из-за огромного числа запусков SCIP, необходимого для подбора параметров, для ускорения процесса нам прошлось использовать облако.

В статье используется следующая терминология. У программы есть *конфигурационные параметры*, контролирующие работу программы. У каждого параметра есть *значение*. Набор значений параметров называется *настройками*. При запуске солвер получает настройки в виде конфигурационного файла, а также исходные данные задачи (*экземпляр задачи*). Задача нашей системы состоит в том, чтобы найти настройки, обеспечивающие наименьшее время работы для фиксированного набора экземпляров задачи.

Другие работы в области оптимизации параметров алгоритмов включают инструмент выбора параметров оптимизации (Selection Tool for Optimization Parameters, STOP) [2], основанный на интеллектуальной проверке отдельных настроек из множества всевозможных настроек. Данный инструмент работает с небольшим числом дискретных параметров. Другой пример — это фреймворк OPAL [3], основанный на адаптивном прямом поиске по сетке. Наш подход позволяет работать с большим числом параметров и их значений.

1. Реализация

Для начала приведем краткое описание использования системы. Для начала работы необходимо задать нужное число хостов в конфигурационном файле Vagrant (http://www.vagrantup.com), а затем запустить систему скриптом `init-virtualbox.sh` или `init-digitalocean.sh`. Через некоторое время мы получаем кластер из виртуальных машин, состоящий из одного хоста-мастера и указанного в конфигурационном файле числа хостов-работников. В кластере установлен и настроен планировщик задач SLURM [4], а также другое необходимое для работы ПО. Затем можно подключиться к хосту-мастеру командой `vagrant ssh` и управлять системой подбора параметров с помощью команды `optctl.py`.

Процесс оптимизации настроек состоит из трех этапов: «проверки времени», «большого шага» и «промежуточного шага». На этапе проверки времени каждый экземпляр задачи решается с настройками по умолчанию на каждом из вычислительных узлов. Главная цель этого этапа состоит в нахождении максимально допустимого времени выполнения каждого из экземпляров, используемого в качестве ограничения времени выполнения экземпляров задач в SLURM. Этап большого шага наиболее сложен с точки зрения объема вычислений. На этом этапе проверяется множество настроек, где лишь одно из значений параметров отличается от значения по умолчанию. Для параметров с небольшим числом дискретных значений проверяются все возможные значения. Для параметров, выражаемых числами с плавающей точкой и целыми числами, проверяются специальные значения и несколько значений из диапазона допустимых значений. В результате, после большого шага, появляется возможность ранжировать параметры (с их конкретными значениями) по степени их влияния на время решения задачи. Далее, на этапе промежуточного шага, берутся четыре лучших параметра с большого шага и проверяются всевозможные их комбинации. В результате этого этапа мы получаем лучшую комбинацию четырех лучших параметров. Этап промежуточного шага можно повторить несколько раз, добавляя к текущему набору лучших параметров комбинацию из следующей четверки с большого шага, пока результат выполнения не перестанет улучшаться.

1.1. Измерение времени выполнения в облаке

В описываемой нами системе желательно иметь возможность измерять время выполнения алгоритма с конкретными настройками за один запуск, причем время должно быть измерено достаточно точно, чтобы сравнивать его с временем работы остальных настроек для того же экземпляра задачи. При запуске алгоритма в виртуальной машине в облаке, измерять время выполнения с помощью таких распространенных источников времени, как физическое время (например, вызов `gettimeofday()`) или число циклов процессора (например, вызов `clock()`) не совсем корректно. Это связано с тем, что суммарно потребляемые виртуальными машинами ресурсы время от времени оказываются больше, чем доступные на физическом сервере-гипервизоре и тогда производительность виртуальной машины может подать. Это, в свою очередь, приводит к большому разбросу времен выполнения приложения. Неплохой альтернативой физическому времени и циклам процессора является число инструкций, выполненных центральным процессором за время выполнения программы. Разумеется, разные инструкции выполняются за разное число циклов процессора, так что может быть не просто сопоставить время выполнения программы числу выполненных процессором инструкций. Однако для сравнения производительности одной и той же программы с одними и теми же начальными данными, но с разными настройками, этот источник логического времени должен подходить хорошо.

В процессорах архитектуры x86 число выполненных инструкций измеряется на аппаратном уровне блоком мониторинга производительности (Performance Monitoring Unit, PMU). Чтобы получить доступ к аппаратным счетчикам, в Linux можно использовать PAPI или приложение perf. Не все гипервизоры поддерживают виртуализацию PMU. Например, она не поддерживается в VirtualBox, но поддерживается в современных версиях KVM.

В нашей системе мы воспользовались приложением perf для измерения числа инструкций, выполненных в режиме пользователя (без учета режима ядра). Данный счетчик дает крайне стабильные результаты вне зависимости от нагрузки на физический сервер.

Представим пример запуска SCIP под контролем perf-stat. Было выполнено шесть запусков SCIP, представлены средние значения и их стандартные отклонения для нескольких разных счетчиков:

```
$ perf stat -r 6 -e cpu-clock,task-clock,cycles,instructions,\
instructions:u,instructions:k scipampl TSP_Uniform_50_10.nl
```

```
Performance counter stats for 'scipampl TSP_Uniform_50_10.nl' (6 runs):
     81154.175629 cpu-clock                                 ( +-  2.32% )
     81154.063870 task-clock       # 0.999 CPUs utilized   ( +-  2.32% )
  175,626,392,898 cycles           # 2.164 GHz             ( +-  0.13% )
  267,235,503,611 instructions     # 1.52  insns per cycle ( +-  0.00% )
  265,101,243,265 instructions:u   # 1.51  insns per cycle ( +-  0.00% )
    2,134,260,346 instructions:k   # 0.01  insns per cycle ( +-  0.10% )

     81.224668399 seconds time elapsed                      ( +-  2.32% )
```

Одновременный запуск двух экземпляров SCIP на той же машине:

```
Performance counter stats for 'scipampl TSP_Uniform_50_10.nl':
     82580.457064 cpu-clock
     82579.334255 task-clock           # 0.493 CPUs utilized
  181,566,274,355 cycles               # 2.199 GHz
  267,300,821,128 instructions         # 1.47  insns per cycle
  265,099,385,783 instructions:u       # 1.46  insns per cycle
    2,201,435,345 instructions:k       # 0.01  insns per cycle

    167.578326122 seconds time elapsed

Performance counter stats for 'scipampl TSP_Uniform_50_10.nl':
     82581.195083 cpu-clock
     82580.104484 task-clock           # 0.493 CPUs utilized
  181,589,302,031 cycles               # 2.199 GHz
  267,299,923,846 instructions         # 1.47  insns per cycle
  265,099,381,995 instructions:u       # 1.46  insns per cycle
    2,200,541,851 instructions:k       # 0.01  insns per cycle

    167.589704033 seconds time elapsed
```

Как можно видеть, счетчик `instaructions:u` выдает наиболее стабильные значения. Также видно, что с его помощью можно надежно измерять время работы алгоритма одновременном выполнении нескольких экземпляров алгоритма на хосте.

2. Результаты численных экспериментов

Разработанная система была проверена на двух наборах экземпляров задач. При этом один из наборов настраивался сразу для двух версий SCIP: 3.0.2 и 3.1.0. При тестировании использовался кластер из 48 идентичных виртуальных машин.

Первый набор задач состоял из десяти созданных случайным образом задач коммивояжера одинакового размера. Настраивался SCIP версии 3.1.0. Большой шаг занял шесть с половиной часов. За это

Таблица 1. Задача о коммивояжере, до и после оптимизации параметров

Экземпляр задачи	Время до, с.	Время после, с.
TSP_Uniform_50_1	3,0	2,5
TSP_Uniform_50_2	10,4	6,4
TSP_Uniform_50_3	29,4	16,1
TSP_Uniform_50_4	6,8	8,7
TSP_Uniform_50_5	39,9	36,6
TSP_Uniform_50_6	37,9	7,6
TSP_Uniform_50_7	105,8	31,3
TSP_Uniform_50_8	102,8	13,6
TSP_Uniform_50_9	31,4	10,9
TSP_Uniform_50_10	80,7	13,2

время было произведено 28810 запусков солвера, занявших 296 часов процессора (измеренных солвером с помощью функции `clock()`), что примерно эквивалентно 46 (а не 48) вычислительным узлам, непрерывно работающим шесть с половиной часов. После одного промежуточного шага были получены оптимальные настройки. Второй промежуточный шаг не показал улучшений. Если сравнивать суммы времен работы солвера с настройками по умолчанию и с оптимизированными настройками, мы обнаруживаем трехкратное (3x) ускорение с последними (см. Таблицу 1). Оптимизированные настройки состояли всего лишь из одного значения параметра:

```
lp/scaling = FALSE
```

Ради интереса было случайным образом создано еще несколько экземпляров той же задачи, которые затем были запущены с параметрами по умолчанию и с ранее найденными оптимизированными настройками (см. Таблицу 2). В данном случае ускорение от использования тех же настроек оказалось равным 1,41x.

Второй набор задач состоял из пяти экземпляров задачи о балансировке нагрузки, быстро решаемых SCIP 3.0.2 и очень медленно решаемых SCIP 3.1.0.

Мы попытались подобрать параметры, позволяющие SCIP 3.1.0 работать на уровне параметров по умолчанию SCIP 3.0.2. Для этого были выбраны только те параметры, которые изменили значения по умолчанию, были переименованы или добавлены в 3.1.0. Таких

Таблица 2. Второй набор экземпляров задачи о коммивояжере

Экземпляр задачи	Время до, с.	Время после, с.
TSP_Uniform_50_11	52,6	27,3
TSP_Uniform_50_12	11,5	9,2
TSP_Uniform_50_13	1,4	2,2
TSP_Uniform_50_14	7,0	5,2
TSP_Uniform_50_15	270,8	230,6
TSP_Uniform_50_16	64,0	10,5
TSP_Uniform_50_17	4,0	3,7
TSP_Uniform_50_18	27,1	14,4
TSP_Uniform_50_19	5,0	12,0

параметров оказалось 186 (против 1547 параметров).

Один из экземпляров (w6_t19_test_8) был автоматически отброшен на этапе проверки времени из-за превышения лимита памяти виртуальной машины (512 MB). Большой шаг занял пять с половиной часов, а состоял он из 1260 заданий. Общее затраченное процессорное время, измеренное в солвере, оказалось равным 237 часом, что примерно соответствует непрерывной работе 43 хостов в течение пяти с половиной часов. Оптимизированные значения параметров были получены после первого промежуточного шага, а второй не дал дальнейшего улучшения. В данном случае от использования оптимизированных настроек было получено ускорение в 1,65x. Более того, данные настройки также ускорили и отброшенный из-за недостатка памяти пример. Как можно видеть (см. Таблицу 3), полученные настройки не позволили достичь производительности SCIP 3.0.2, однако заметное ускорение все же было получено. Вероятно, оптимизация на всем множестве параметров позволила бы добиться лучших результатов. Полученные оптимизированные настройки:

```
heuristics/rins/minnodes = 25
lp/checkdualfeas = FALSE
lp/disablecutoff = 1
```

Для этого же набора экземпляров задач была проведена оптимизация на всех параметрах SCIP 3.0.2. На большом шаге за тринадцать с половиной часов было обработано 13200 заданий. По оценке солвера было потрачено 594 часа процессорного времени, что примерно соответствует работе 44 вычислительных узлов. Оптимальные настройки были получены после двух промежуточных шагов, далее улучшения

Таблица 3. Задача балансировки нагрузки

Задача	3.1.0, def., с.	3.1.0, опт., с.	3.0.2, def., с.	3.0.2, опт., с.
w6_t15_test_4	5,95	3,11	1,97	0,92
w6_t18_test_4	586,18	186,8	70,3	32,5
w6_t19_test_4	1420,4	823,9	223,9	178,3
w6_t19_test_5	941,7	737,6	138,7	119,3
w6_t19_test_8	11596,3	7077,7	382,6	319,7

не было. В данном случае было получено ускорение 1,26x. Оптимизированные значения параметров после первого промежуточного шага:

```
constraints/linear/upgrade/setppc = FALSE
lp/solvefreq = 0
```

После второго промежуточного шага:

```
constraints/linear/upgrade/setppc = FALSE
lp/solvefreq = 0
conflict/preferbinary = TRUE
heuristics/fracdiving/freqofs = 1
heuristics/veclendiving/freq = -1
```

3. Заключение

В результате описанной работы была создана система, обеспечивающая подстройку параметров алгоритма. Для управления виртуальными машинами использовался Vagrant, для обработки очереди заданий был применен SLURM, язык программирования Python [5] использовался для автоматизации, а VirtualBox — для локальной отладки. Оптимизировались параметры солвера SCIP. Система была протестирована на нескольких наборах задач, где было получено ускорение не менее 1,26x.

В дальнейшем возможно включить в систему поддержку других популярных солверов, например, CBC или Ipopt. Другой потенциальной возможностью расширения системы может быть обеспечение доступа к ней через Web. Также планируется публикация исходных кодов на GitHub.

Основной вывод работы состоит в том, что дешевизна и удобство облачных вычислений позволяют использовать новые и нестандартные подходы к решению задач.

Список литературы

[1] T. Achterberg. *Constraint Integer Programming*, Technische Universität Berlin, (2007), http://opus.kobv.de/tuberlin/volltexte/2007/1611/. ↑2, 8

[2] M. Baz, B. Hunsaker, P Brooks, A. Gosavi. *Automated tuning of optimization software parameters*, 2007. Vol. **7**. ↑2

[3] C. Audet, K.-C. Dang, D. Orban. *Optimization of algorithms with OPAL* // Mathematical Programming Computation, 2012, p. 1–22. ↑2

[4] A. B Yoo, M. A Jette, M. Grondona. *SLURM: Simple linux utility for resource management* // Job Scheduling Strategies for Parallel Processing Springer, 2003, p. 44–60. ↑3, 8

[5] M. F Sanner et al. *Python: a programming language for software integration and development* // J Mol Graph Model, 1999. Vol. **17**, no. 1, p. 57–61. ↑8

Об авторе:

Сергей Андреевич Смирнов

И.о. научного сотрудника лаборатории № Ц-1 ИППИ РАН, к.т.н. Область научных интересов включает распределенные вычисления, облачные вычисления, а также использование пакетов оптимизации.

e-mail: sasmir@gmail.com

Образец ссылки на эту публикацию:

С. А. Смирнов. *Облачная система подстройки параметров алгоритмов* // Программные системы: теория и приложения: электрон. научн. журн. 2014. Т. ??, № ?, с. ??–??.

URL: http://psta.psiras.ru/read/

Sergey Smirnov. *Cloud system for program parameters fine tuning* .

Abstract. The paper presents a software system aimed at finding best (in some sense) parameters of an algorithm. The system handles both discrete and continuous parameters and employs massive parallelism offered by public clouds. The paper presents an overview of the system, a method to measure algorithm's performance in the cloud and numerical results of system's use on several problem sets. (*in Russian*).

Key Words and Phrases: algorithmic parameter optimization, parameter tuning, cloud computing.