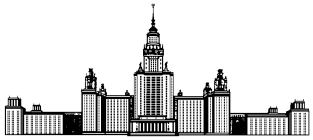
Московский Государственный Университет им. М.В. Ломоносова



Факультет Вычислительной Математики и Кибернетики

Отчёт о выполнении практического задания по курсу СКиПОД

Выполнил: Мурат Апишев, гр.417 (ММП)

октябрь 2014

Содержание

1 Постановка задачи

1.1 Оценки качества тематического моделирования

Одной из активно развивающихся областей машинного обучения является тематическое моделирование коллекций текстовых документов. Алгоритмы тематического моделирования позволяют выделять из текстов темы, представленные в виде векторов слов, характеризующих данную тему.

Для оценки качества получаемых моделей существуют различные функционалы. Одним из них является т.н. *когерентность тем*. Оценивание этой величины требует информацию о попарной встречаемости самых популярных в каждой теме слов.

Счётчики могут быть различными, в данной работе будет рассчитываться попарная встречаемость в пределах документа, просуммированная по всем документам коллекции.

1.2 Концептуальная схема решения

Решение задачи состоит из нескольких последовательных шагов:

1. Пройти по коллекции первый раз, собрав информацию о том, какие слова содержаться в коллекции, и сколько раз они в ней встречаются.

2. Отсортировав получившийся результат, выделить n самых часто встречающихся слов (которые с большой долей вероятности войдут в число самых популярных терминов в будущих темах) [1]

3. Пройти по коллекции второй раз и посчитать, сколько раз каждая пара из самых популярных слов (далее `top-слова`) встречается совместно в каждом документе.

4. Просуммировать полученные результаты по всем документам.

Параллелизм обработки организовывается за счёт распределения файлов с данными между процессами. Шаги обхода коллекции производятся рабочими процессами (далее `обработчик`), шаги агрегации — процессом-мастером (далее `мастер`).

2 Реализация

2.1 Фреймворк и библиотеки

Программа изначально создавалась и тестировалась на ОС Linux Ubuntu 13.04. Язык программирования — C++. Используется фреймворк Open MPI 1.6.5, интерфейс для которого предоставляла библиотека Boost.MPI. Для вспомогательных целей использовались ещё несколько библиотек, входящих в состав Boost 1.56. Однако в следствие отсутствия необходимого ПО на кластере «Ломоносов», а также невозможности его установки, от Boost.MPI пришлось отказаться в пользу адаптированного интерфейса, написанного вручную.

2.2 Схема распределения вычислений

Процесс работы приложения описан ниже в виде алгоритма. Мастер рассылает частями списки файлов обработчикам, обработчики, получив списки, выполняют нужную работу

[1] подобное предположение допустимо только в том случае, если коллекция является очищенной от фоновых слов, которые не несут особого смысла, но встречаются постоянно.

(в зависимости от итерации — #1 или #2) и возвращают результаты мастеру. Мастер, собрав все результаты итерации #1, инициирует итерацию #2. После второй итерации он записывает агрегированные финальные данные в файл.

Мастер

```
повторять в «вечном» цикле
  если есть непросмотренные файлы то
    итерация #1 продолжается
  иначе
    итерация #1 завершена
  для всех обработчиков
    разослать сообщение о продолжении/завершении итерации #1
  если итерация #1 продолжается то
    для всех обработчиков
      отправить обработчику набор заданного числа имён файлов для обработки
    для всех обработчиков
      получить результаты обработки коллекции
    агрегировать все результаты в общий
  иначе
    выйти из цикла
выбрать заданное число top-слов
для всех обработчиков
  отправить вектор top-слов
повторять в «вечном» цикле
  если есть непросмотренные файлы то
    итерация #2 продолжается
  иначе
    итерация #2 завершена
  для всех обработчиков
    разослать сообщение о продолжении/завершении итерации #2
  если итерация #2 продолжается то
    для всех обработчиков
      отправить обработчику набор заданного числа имён файлов для обработки
    для всех обработчиков
      получить результаты обработки коллекции
    агрегировать все результаты в общий
  иначе
    выйти из цикла
```

Обработчик

```
создать объект обработчика файлов
повторять в «вечном» цикле
  получить сообщение о продолжении/завершении итерации #1
  если итерация #1 продолжается то
    получить список файлов для обработки
    вызвать метод обработчика, регистрирующий слова и их вхождения для указан-
```

ных в списке файлов
7: отправить полученные результаты мастеру
8: **иначе**
9: выйти из цикла
10: получить вектор top-слов
11: **повторять** в «вечном» цикле
12: получить сообщение о продолжении/завершении итерации #2
13: **если** итерация #2 продолжается **то**
14: получить список файлов для обработки
15: вызвать метод обработчика, считающий частоты попарной встречаемости переданных слов
16: отправить полученные результаты мастеру
17: **иначе**
18: выйти из цикла

2.3 Алгоритм вычислений

У объекта класса «обработчик файлов» (не путать с процессом `обработчиком`) есть два метода. Первый из них находит число вхождений всех слов, просто просматривая по одному разу каждый файл. Схема работы второго метода, считающего число попарных вхождений в документ `top-слов`, описана в следующем алгоритме:

1: **повторять** для каждого имени файла из полученного списка
2: открыть файл с этим именем
3: считывая по одному слову, получить счётчики встречаемости top-слов
4: вызвать метод обработчика, регистрирующий слова и их вхождения для указанных в списке файлов
5: отсортировать результаты в порядке возрастания счётчиков
6: **для всех** слов w от первого до предпоследнего
7: **для всех** слов u от следующего после w до последнего
8: **если** w $>$ u[2] **то**
9: key = 'w u'
10: **иначе**
11: key = 'u w'
12: value = min{частота w, частота u}
13: занести пару «key, value» в глобальное для этого процесса-обработчика хранилище (поле объекта-обработчика)
14: вернуть все собранные данные мастеру

2.4 Описание программы

2.4.1 Сборка и запуск

Сборка программы производится запуском команды `make` из корневой директории проекта. В результате будет создана директория `build`, содержащая исполняемый файл

[2] в лексикографическом смысле — нужно для того, чтобы избежать одновременного появления пар 'w u' и 'w u'

srcmain. Для запуска программы нужно перейти build и выполнить следующую команду:

<mpi_cmd> -n <N> ./srcmain <A> <S> <T>

где:

- mpi_cmd — mpirun для Ubuntu 13.04 с Open MPI 1.6.5 и sbatch для кластера «Ломоносов»;
- N — число создаваемых процессов (≥ 2);
- A — адрес директории с данными;
- S — число файлов, обрабатываемых обработчиком за один раз; [3];
- T — число top-слов;

Замечание: Команда make создаёт директорию build, после чего производит в ней построение исполняемого файла при помощи компилятора mpic++ с флагами -W -Wall -std=c++11 -lboost_system -lboost_filesystem.

2.4.2 Список предусмотренных исключительных ситуаций

Все описанные в списке исключения являются наследниками класса std::runtime_error.

- IncorrectPath — имя директории является некорректным или такая директория не существует;
- FailedOpenFile — обработчику не удалось открыть файл;
- FileReadingError — обработчику не удалось корректно считать слово из файла;
- InvalidNumberOfArguments — программе передано некорректное число аргументов;
- InvalidNumberOfProcessors — программа запущена на числе процессоров < 2;

2.4.3 Формат входных и выходных данных

Входные данные должны представлять из себя набор файлов в формате .txt, расположенных в одной директории (без вложенных директорий).

Выходные данные — файл build/results.txt, содержащий строки в формате:

слово_1 слова_2 совместная_встречаемость_слов_1_2

3 Эксперименты

3.1 Критерии качества параллелизма

Программа запускалась на текстовых коллекциях объёмом 4 и 10 Гб (2000 и 5000 файлов соответственно). Количество обрабатываемых top-слов — 100. Для каждого запуска на обработчик подавалось такое количество файлов, чтобы вся коллекция просматривалась за один проход на каждой из итераций #1 и #2. Критерием качества масштабирования являлось время, общее и затрачиваемое на каждую из двух итераций. Эксперименты

[3] Если размеры коллекции не слишком огромные, рекомендуется назначать равным «число файлов» / (N - 1)

производились на суперкомпьютере «Ломоносов» с набором числа x86-ядер {3, 5, 9, 17, 33, 65}[4]

3.2 Результаты и выводы

На графиках ниже приведены результаты запусков вычислений для разных числа процессов и объёмов текстовых коллекций. Для пояснения графиков под ними приведены таблицы с данными, на основании которых они строились.

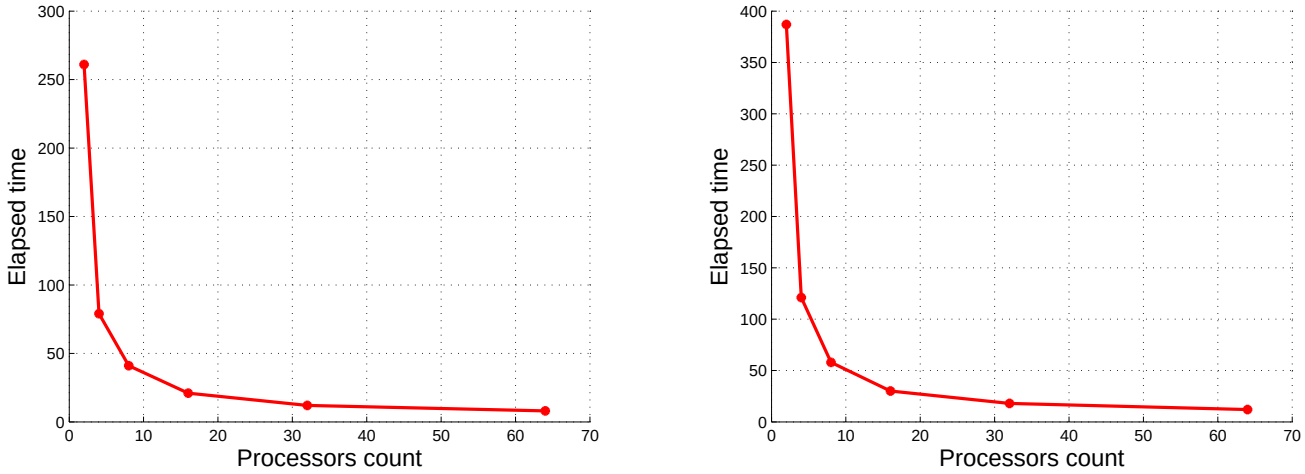


Рис. 1: Графики «Число ядер — время» для первой и второй итераций работы программы на 2000 документах.

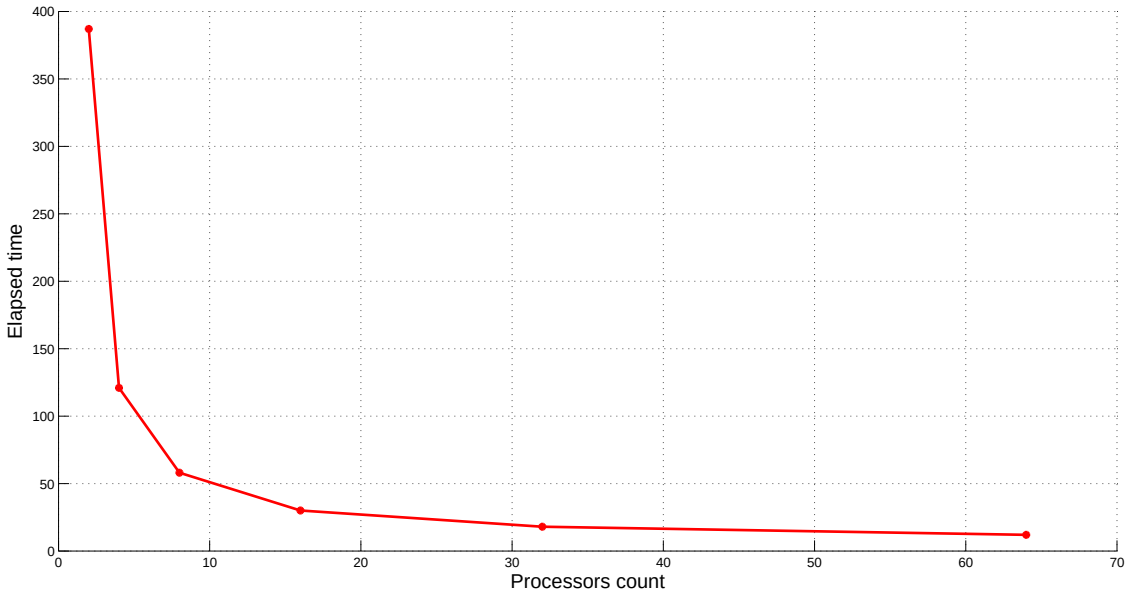


Рис. 2: График «Число ядер — время» для всей программы на 2000 документах.

[4] 1 мастер + 2^n обработчиков.

Число ядер / период	итерация #1	итерация #2	общее время
2	165	247	412
4	79	121	201
8	41	58	99
16	21	30	51
32	12	18	31
64	8	12	20

Видно, что каждое удвоение числа ядер приводит к пропорциональному уменьшению времени счёта. Масштабируемость ухудшается, начиная с 32 ядер, на 64 это становится особенно заметно — коэффицинт роста оказался равен ≈ 1.5. Данные результаты характерны для обеих итераций и, соответственно, для общего результата.

Замедление роста производительности объясняется увеличением значимости накладных расходов по пересылке данных и их синхронизации при маленьком числе документов, обрабатываемых каждым ядром. Для проверки этой гипотезы экспенимент был проведён повторно на большем наборе документов. Результаты рассматриваются далее.

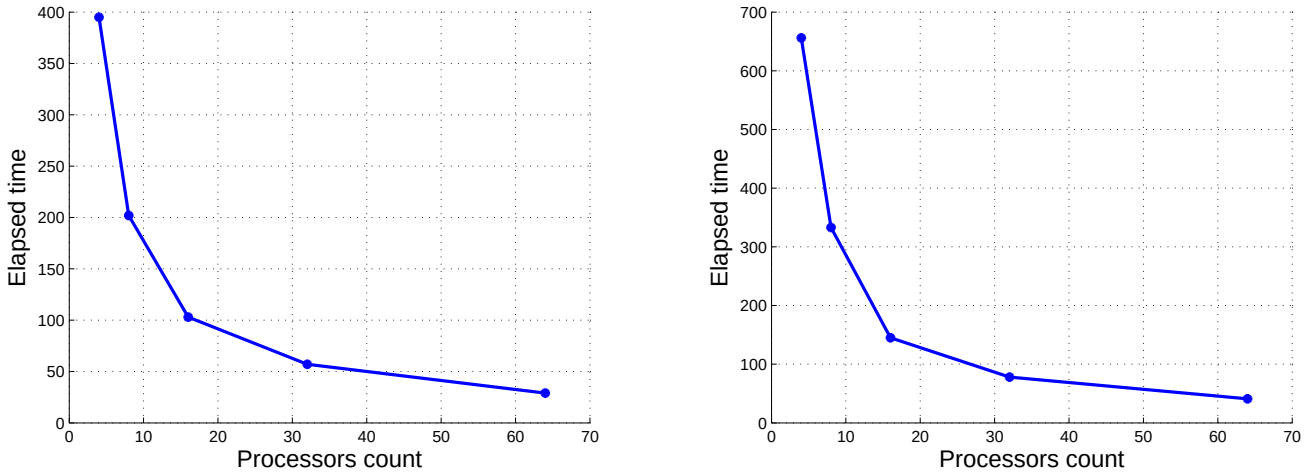


Рис. 3: Графики «Число ядер — время» для первой и второй итераций работы программы на 5000 документах.

Число ядер / период	итерация #1	итерация #2	общее время
4	395	656	1051
8	202	333	535
16	103	145	249
32	57	78	135
64	29	41	70
128	18	24	42
256	12	16	28

Как и ожидалось, гипотеза подтвердилась — с увеличением числа данных производительность программы увеличилась и сохранила линейную масштабируемость и на 32, и на 64 ядрах. Таким образом, эффективная распределённая обработка данных в данном

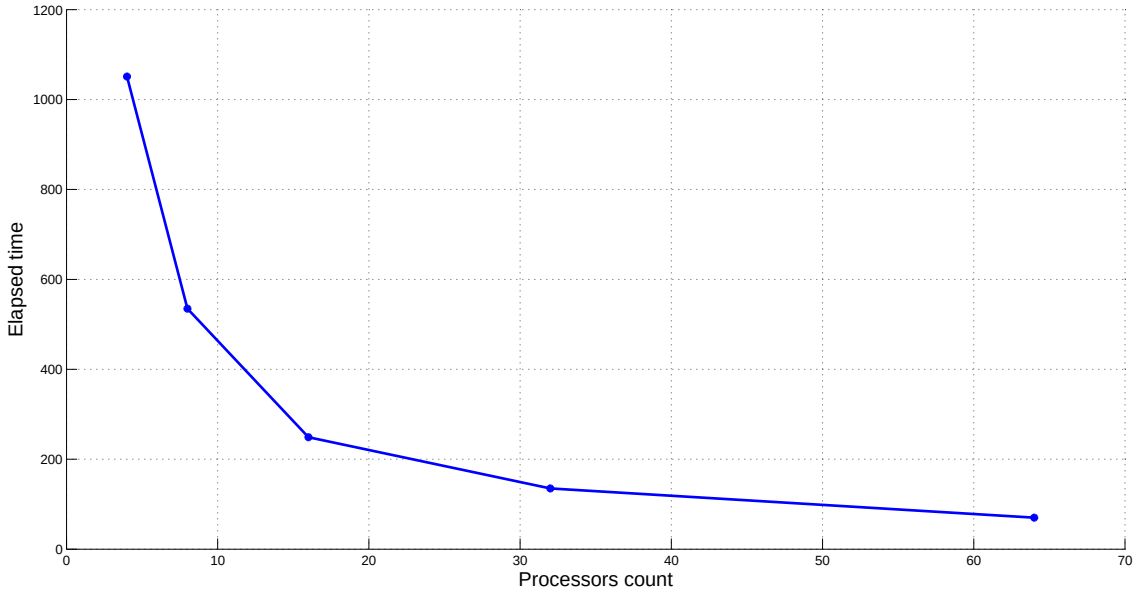


Рис. 4: График «Число ядер — время» для всей программы на 5000 документах.

приложении достигается в том случае, когда нагрузка на каждое рабочее ядро достаточно сильно превосходит затраты на накладные расходы и агрегацию данных на мастере[5]. В то же время, запуск алгоритма на 128 и 256 ядрах явственно показывает, что для этого объёма данных «предел» производительности почти достигнут — на 128 ядрах коэффициент роста составил около 1.8, на 256 — всего 1.5.

[5]Последнее замечение можно устранить, заменив синхронную отправку данных и единоразовую агрегацию на асинхронную и многократную, но в рамках данной работы этот подход не рассматривается