САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ – ПРОЦЕССОВ УПРАВЛЕНИЯ

КАФЕДРА КОМПЬЮТЕРНОГО МОДЕЛИРОВАНИЯ И МНОГОПРОЦЕССОРНЫХ СИСТЕМ

Федосеев Георгий Александрович

Курсовая работа

Развертывание и настройка системы непрерывной интеграции для проекта MPD Root в среде GitLab

Направление 010400

Прикладная математика и информатика

Научный руководитель,
доктор. техн. наук,
профессор,
Дегтярёв А. Б.

Санкт-Петербург

2016

Содержание

Введение

Системы непрерывной интеграции (англ. Continuous Integration, CI) на сегодняшний день являются неотъемлемой частью крупных проектов по разработке программного обеспечения. Такие системы отлеживают каждое изменение в системе контроля версий и обеспечивают незамедлительную сборку, развертывание и тестирование приложения. [1]

По сравнению с ручной сборкой и тестированием автоматические системы имеют ряд неоспоримых преимуществ:

- Весь процесс автоматизирован. Таким образом исключено влияние человека, а, следовательно, и связанные с этим фактором ошибки. Также у разработчика освобождается дополнительное время, которое он может потратить на разработку продукта.
- Однотипное окружение - каждая сборка происходит в выделенном контейнере, где создается окружение, максимально приближенное к тому, в котором будет развернут готовый продукт.
- Быстрота - программе не нужно тратить время на набор команд и сверку с инструкциями.
- Автоматически генерируемые отчеты о сборках, которые разработчик может проверить в любое время и найти место возникновения ошибки.

В этой работе будет рассмотрен процесс развертывания и настройки системы Непрерывной интеграции для проекта MPD Root, а также представлены эксперименты по оптимизации скорости выполнения сборок.

Проект MPD Root — это программные фреймворки симуляции и анализа данных, полученных на Многоцелевом Детекторе (Multi Purpose Detector, MPD), созданном для изучения ядерной материи при экстремальных значениях плотности и температуры. Эксперимент MPD будет проводиться на создаваемом в ОИЯИ на основе Нуклотрона коллайдерном комплексе для ускорения тяжелых ионов (Nuclotron-based Ion Collider fAcility, NICA) [2].

Определения

Непрерывная интеграция (Continous Integration, CI) — практика частой сборки и тестирования программного проекта с целью выявления ошибок на ранней стадии.

Система контроля версий — ПО для облегчения работы с изменяющейся информацией, позволяет хранить несколько версий одного документа и, при необходимости, возвращаться к более ранним, определять, кто и когда сделал то или иное изменение, а также многое другое.

Репозиторий — хранилище в системе контроля версий, отведенное под конкретный проект.

Коммит — объект, характеризующий изменения в репозитории.

Docker-контейнер — изолированная оболочка для приложения, запущенная на базе операционной системы Linux.

Образ (image) docker-контейнера — объект, хранящий состояние docker-контейнера. При запуске контейнера указывается образ, на базе которого его нужно запустить.

Программное окружение — совокупность доступных для запуска программ в данной ОС.

Сборка — процесс компиляции ПО.

Развертывание ПО — процесс настройки и запуска ПО.

Постановка задачи

Пусть имеется виртуальная машина VM Microsoft Azure с установленной на ней ОС с ядром Linux.

Характеристики сервера:

- **CPU**: Octa core Intel Xeon CPU E5-2673 v3 (-HT-MCP-)
- **Количество ядер:** 8
- **Потоков на ядро:** 1
- **SSD:** 91.6GB
- **ОС:** Ubuntu 14.04

(для получения характеристик использовались утилиты `lscpu` *и* `inxi` *)*

Также на сервере установлена система менеджмента Git репозиториев GitLab, где расположен репозиторий с MPD Root, для которого необходимо развернуть систему непрерывной интеграции.

Конечная система должна удовлетворять следующим требованиям:

- сборка и тестирование ПО должно проводится при каждом коммите
- процесс сборки и тестирования не должен занимать по времени более 10 минут [3]
- каждая сборка должна производиться в выделенном контейнере, для обеспечения независимости от окружения и обеспечения среды, максимально приближенной к рабочей среде готового продукта
- по каждой сборке должен генерироваться отчет с выводом информации о процессе сборки и результатах тестирования
- весь процесс сборки должен описываться в текстовом файле(-ах), который включен в репозиторий и может меняться от коммита к коммиту
- система должна легко развертываться на описанном выше выделенном сервере и производить сборку и тестирование ПО проекта

Глава 1. Обзор использованных технологий

Для решения поставленной задачи потребовались следующие инструменты:

1.1 GitLab

Это приложение с открытым исходным кодом — менеджер Git репозиториев, используемый в проекте MPD Root. С приложением удобно работать через предоставляемый веб-интерфейс.

1.2 GitLab CI Runner

Это приложение с открытым исходным кодом, созданное для запуска задач непрерывной интеграции. На рынке существует множество решений для непрерывной интеграции, например: Jenkins, Travis-CI, Bamboo и др. Для реализации задачи был выбран именно GitLab CI Runner, так как этот инструмент легко интегрируется с менеджером репозиториев GitLab, который применяется в проекте MPD Root.

К GitLab-репозиторию можно привязать множество удаленных серверов с установленным GitLab CI Runner, которые будут выполнять задачи построения и тестирования. Для данной работы Runner был установлен локально: на том же сервере, где запущен менеджер репозиториев.

1.3 Docker

Это проект с открытым исходным кодом, созданный для развертывания приложений внутри программных контейнеров. Docker предоставляет дополнительный уровень абстракции от других процессов на уровне операционной системы. Docker также позволяет запускать несколько независимых контейнеров на одной операционной системе одновременно и, в отличии от виртуальных машин, не теряет время на старт полноценной гостевой операционной системы. Docker — это фреймворк виртуализации, сфокусированный не вокруг эмуляции аппаратных средств, а вокруг запуска приложений [4].

В данной работе Docker используется как инструмент для изоляции параллельно запущенных задач непрерывной интеграции друг от друга.

Docker на данный момент является одной из самых популярных и развивающихся технологий контейнеризации приложений, что гарантирует наличие подробной документации и поддержки сообщества, поэтому данная технология и была выбрана для решения поставленной задачи.

Глава 2. Установка и настройка

На выделенный сервер с помощью команды `apt-get install` были установлены пакеты `gitlab-ce`, `gitlab-ci-multi-runner` и `docker.io`.

2.1 GitLab и GitLab Runner

В конфигурационном файле `/etc/gitlab-runner/config.toml` для GitLab Runner были заданы следующие параметры:

- `concurrent = 8` максимальное число одновременных построений для данного раннера было увеличено с 1 до 8 (в целях эксперимента)
- `executor = "shell"` CI скрипты будут выполняться локально в среде shell из-под пользователя gitlab-runner. Также есть варианты запуска раннера по ssh, из контейнера docker или из виртуальной машины Parallels, но вариант shell дает больше свободы, что будет понятно далее. Это даст возможность пересобирать образ для docker прямо из CI скрипта.
- `timeout = 5400` максимальное время сборки в секундах. Сборка “с нуля” занимает около 40 минут, поэтому было выставлено максимальное время в 1.5 часа на случай параллельно запущенных сборок.
- `output_limit = 15000` лимит лога для одной сборки в килобайтах. Значения, установленного по умолчанию в 4 МБ, было мало для сборки “с нуля”, поэтому был установлен лимит в 15 МБ.

Остальные значения были осталены исходными.

С помощью веб-интерфейса GitLab был клонирован репозиторий проекта mpdroot https://git.jinr.ru/nica/mpdroot.git. А из терминала к GitLab репозиторию командой

`gitlab-ci-multi-runner register`

был подключен GitLab Runner.

2.2 Docker

В корне репозитория проекта был создан файл Dockerfile, где описано построение образа контейнера, в котором будет происходить построение проекта mpdroot.

В начале файла, с помощью команды `FROM`, можно задать другой образ, на котором будет основываться данный. В нашем случае мы зададим образ ОС Ubuntu 14.04, который будет скачен с сервера готовых образов Docker Hub. Далее в файле описываются команды `RUN` загрузки и установки всех необходимых для mpdroot зависимостей, в том числе и установка зависимости FairSoft. В конце файла, с помощью команды `ADD`, мы копируем содержание корня репозитория в независимую файловую систему docker-образа.

Очень важной особенностью построения docker-образов является кэширование команд. В первый раз при построении образа будут выполнены все команды. Далее, при последующих запусках, если мы не будем менять Dockerfile, результат большинства команд будет получен из кэша (существуют команды, которые не кэшируются, например, команда `ADD`). Если же мы изменим какую-либо команду, то при следующем построении результаты измененной и всех последующих команд будут вычисляться “с нуля”.

Поэтому в нашем случае очень удобно вынести построение всех зависимостей в построение docker-образа. В первый раз построение займет более получаса, но зато в последующие разы (при условии неизменности зависимостей) будет происходить за минуты.

Содержание файла Dockerfile см. в приложении.

2.3 CI-скрипт

CI-скрипт в системе GitLab хранится в корне репозитория под именем **.gitlab-ci.yml** и имеет формат данных YAML. В скрипте могут быть описаны задачи CI-процесса (jobs). Каждая задача имеет обязательную секцию script, где описываются команды задачи. Также в начале можно перечислить различные этапы (stages) CI-процесса и определить их последовательность, а потом отнести каждую из задач к определенному этапу. Задачи одного этапа

выполняются параллельно. Задачи следующего этапа не выполняются, пока не выполнены задачи текущего этапа. В нашем случае будет 3 этапа: build, test и cleanup. Соответствующие им задачи: build_mpd, test_mpd, clean_allocated_container.

Приведем основные этапы нашего CI-процесса:

- Происходит изменение в системе контроля версий проекта (регистрируется коммит)
- GitLab связывается со свободным раннером и отдает ему задачу сборки версии проекта, заданной коммитом
- Раннер клонирует к себе в локальную папку репозиторий проекта

Далее управление передается CI-скрипту со стартовой точкой в корне клонированного репозитория.

- Начинается этап построения, задача build_mpd
  - Перед запуском построения проекта mpdroot в секции before_script мы описываем пересборку docker-образа с помощью команды

    ```
    sudo docker build -t ci-env-$CI_PROJECT_ID  .
    ```

    В команде используется переменная среды, которой присвоен id проекта GitLab. Также в конце команды указана директория (текущая) в которой лежит Dockerfile. Данное построение, как было описано выше, займет немного времени, если есть возможность получить результаты выполнения команд в Dockerfile из кэша.
  - В основной секции script мы запускаем построенный образ и передаем в параметре все команды, которые надо выполнить внутри контейнера (команды построения mpdroot).

    ```
    sudo docker run -i ci-env-$CI_PROJECT_ID /bin/bash -s <<EOF
      <команды>
    EOF
    ```

    Будет создан отдельный контейнер. После успешного выполнения команд в контейнере нам нужно сохранить этот результат в новый образ, который мы будем использовать на этапе тестирования:

    ```
    sudo docker commit $CONTAINER_ID ci-build-$CI_BUILD_REF
    ```

    Название нового образа теперь уже связано с текущим коммитом.
- Этап тестирования, задача test_mpd

    - Запускаем образ, построенный на предыдущем этапе, с помощью команды docker run и передаем команды тестирования проекта
- Этап отчистки, задача clean_allocated_container — на этом этапе с помощью команд docker rm и docker rmi удаляются контейнеры и образы, связанные с текущим коммитом.

На данном этапе CI-скрипт закончил свою работу. Напротив коммита в веб-интерфейсе GitLab появится зеленая галочка, если построение и тестирование прошло успешно, и красный крестик, если возникли ошибки. Доступ ко всем логам процесса CI можно получить из веб-интерфейса GitLab.

Полный CI-скрипт см. в приложении.

Глава 3. Эксперименты

В экспериментах менялось 3 параметра:

- число одновременно запущенных CI-задач,
- число потоков для каждой задачи
- и сдвиг во времени каждой следующей параллельно выполняющейся задачи.

Большинство экспериментов проводились с заранее собранным docker-образом. Также в конце рассмотрена задача в случае сборки docker-образа “с нуля”.

Для запуска задач в репозитории изменялся единственный файл, в который записывалась текущая дата и время, после чего создавался коммит. В случае запуска нескольких сборок одновременно, все команды по отправке коммита записывались в терминале в одну строку и исполнялись с задержкой максимум в 2-3 секунды. В экспериментах со сдвигом во времени между командами создания коммитов вставлялась команда sleep с соответствующим числом секунд сдвига.

Для снятия показаний с ядер процессора использовалась утилита `dstat`, данные записывались в файл формата csv для последующего графического представления.

В качестве результатов экспериментов будем рассматривать профили сборок — это графики зависимости загрузки каждого из 8 ядер процессора от времени. Загрузка ядра может меняться от 0% до 100%. Время указано в секундах.

График загрузки каждого ядра отрисован линией своего цвета, средняя загрузка по всем ядрам отрисована более жирной синей линией.

Для начала приведем профиль загрузки ядер при отсутствии задач сборки и тестирования:

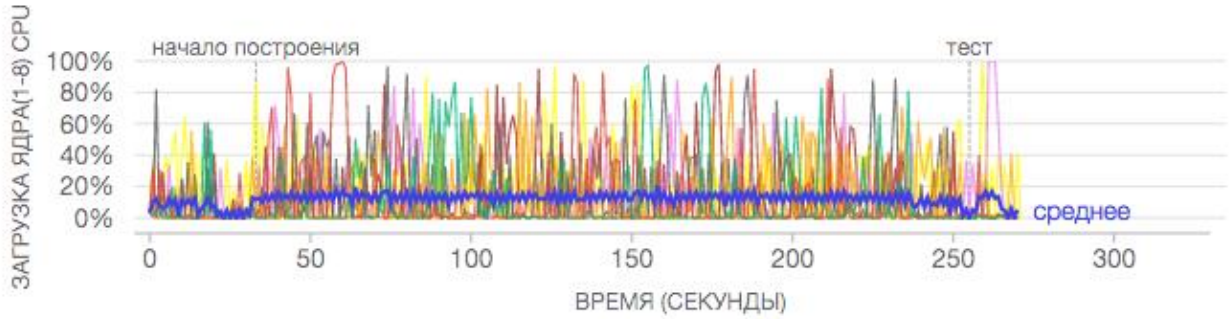
Среднее значение колеблется в близи нуля. Резкие скачки отдельных ядер связаны с другими процессами, запущенными на сервере, в частности, с процессом веб-представления GitLab. Скачки вызваны обновлением веб-страницы статуса сборок каждые 2 секунды в целях наблюдения.

Далее рассмотрим различные варианты сборок.

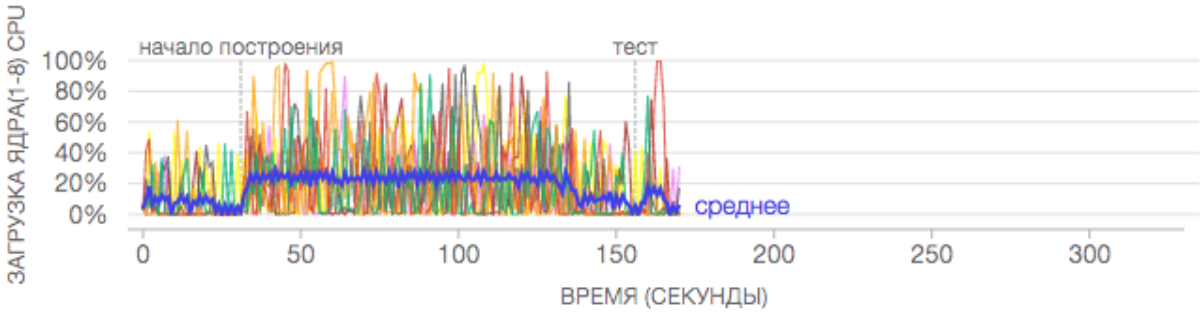
3.1 Одна CI-задача с различным числом потоков

На графиках также отмечены моменты во времени, где начинались задачи построения основного проекта (mpdroot) и тестирования. Как видно из графиков, больше всего времени занимает процесс сборки проекта, тогда как тест занимает около 10-30 секунд.

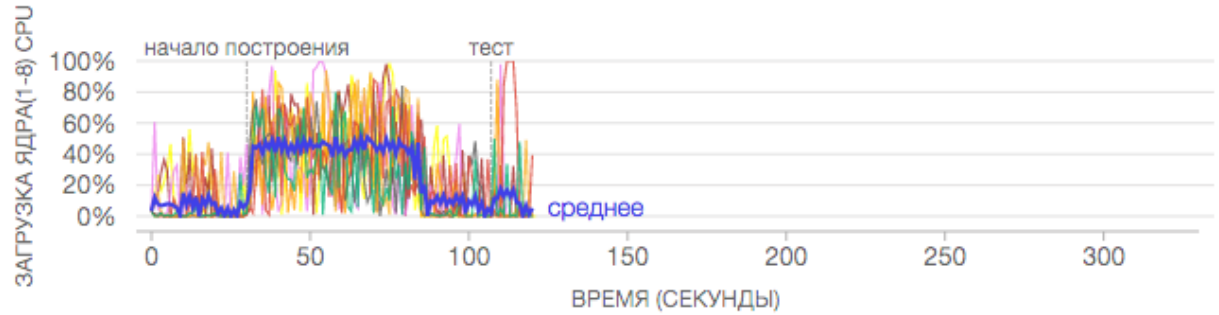
1 задача; 1 поток = 271 сек.



1 задача; 2 потока = 171 сек.

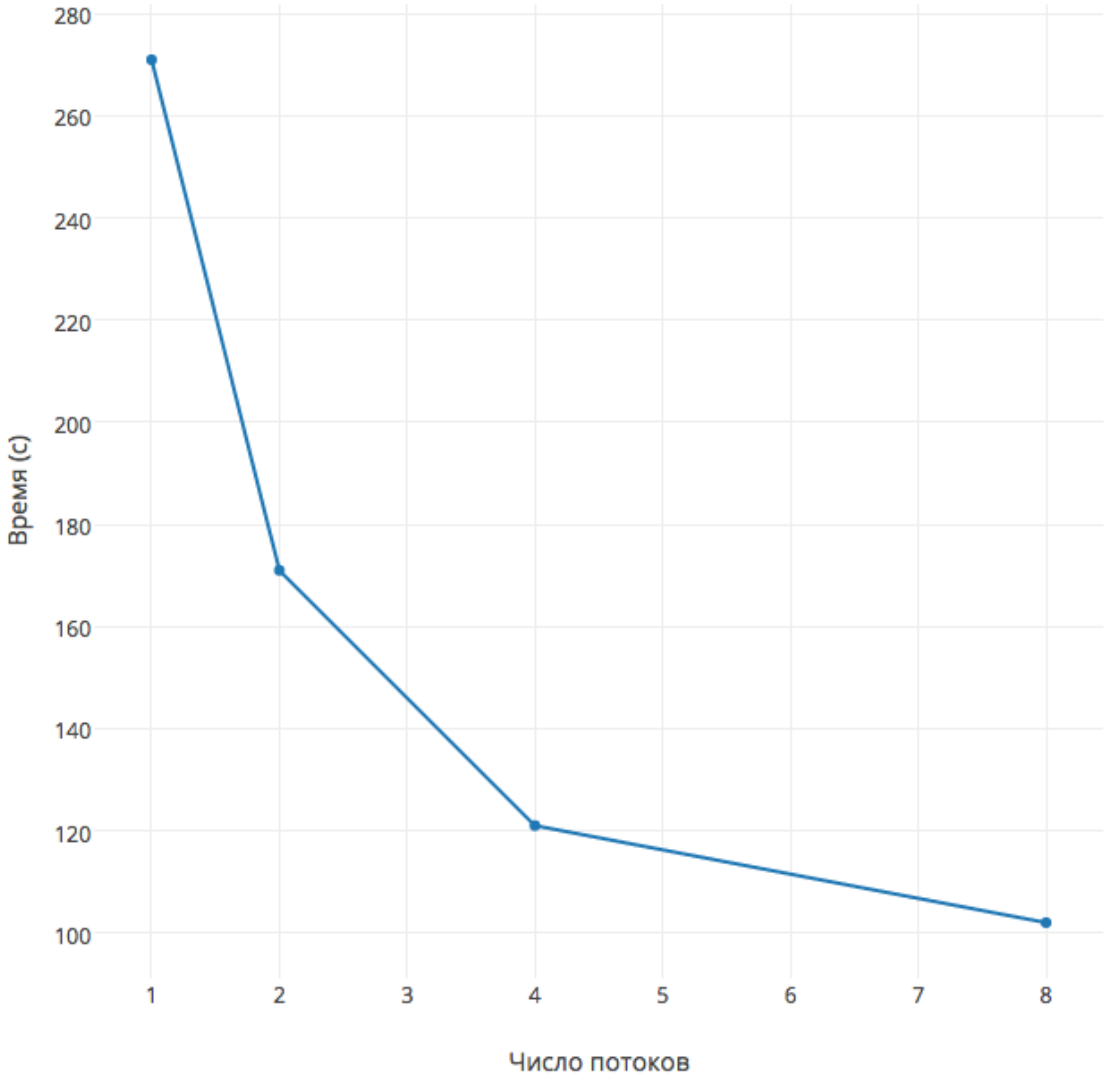


1 задача; 4 потока = 121 сек.



1 задача; 8 потоков = 102 сек.

Условия	**Время (с)**
1 задача; 1 поток	271
1 задача; 2 потока	171
1 задача; 4 потока	121
1 задача; 8 потоков	102



Как можно видеть на рисунке выше, при увеличении числа потоков снижается время выполнения задачи. При увеличении числа потоков с 1 до 8 можно выиграть около 3 минут.

3.2 Две CI-задачи параллельно с различным числом потоков и сдвигами во времени

Условия	Время (с)
2 задачи; 4 потока на каждую	175
2 задачи; 8 потоков на каждую	174
2 задачи; 8 потоков на каждую; сдвиг 20 сек.	160
2 задачи; 8 потоков на каждую; сдвиг 30 сек.	165
2 задачи; 8 потоков на каждую; сдвиг 50 сек.	187

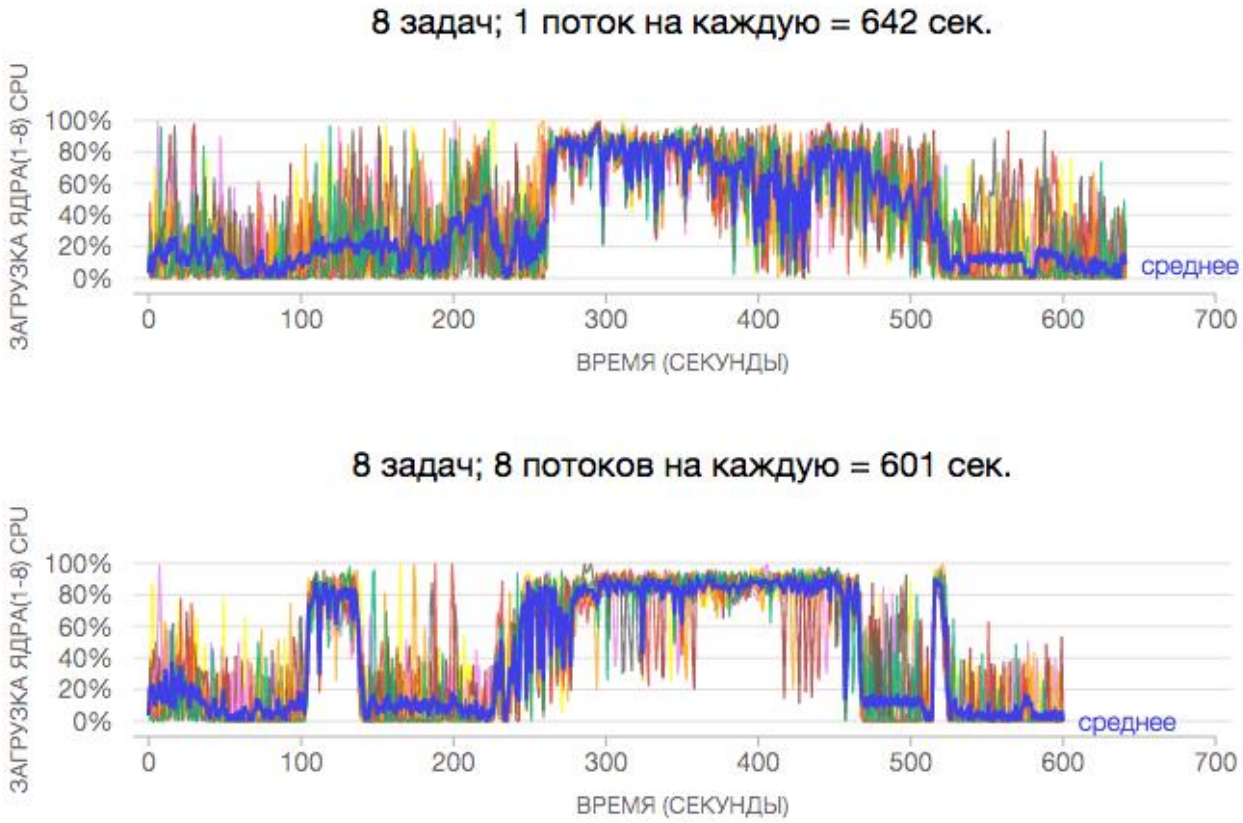
Как можно видеть из таблицы, увеличение числа потоков с 4 до 8 при одновременном выполнении 2-х CI-задач не дает видимого эффекта. Сокращения времени построения можно добиться сдвигом на пару десятков секунд второй задачи относительно первой.

3.3 Четыре CI-задачи параллельно с различным числом потоков и сдвигами во времени

Условия	**Время (с)**
4 задачи; 4 потока на каждую	332
4 задачи; 8 потоков на каждую	329
4 задачи; 8 потоков на каждую; сдвиг 10 сек.	291
4 задачи; 8 потоков на каждую; сдвиг 20 сек.	310
4 задачи; 8 потоков на каждую; сдвиг 30 сек.	292

Как и в прошлом эксперименте, при параллельном выполнении нескольких задач, увеличение числа потоков не дает заметных результатов, но выигрыша в пару десятков секунд можно добиться сдвигом во времени задач друг относительно друга.

3.4 Восемь CI-задач параллельно с различным числом потоков



Условия	**Время (с)**
8 задач; 1 поток на каждую	642
8 задач; 8 потоков на каждую	601

3.5 Одна CI-задача - сборка “с нуля”

3.6 Вывод

Количество одновременных задач [лучшие условия]	**Лучшее время (с)**
1 [8 потоков]	102
2 [8 потоков; сдвиг 20 сек.]	160
4 [8 потоков; сдвиг 10 сек.]	291
8 [8 потоков]	601

Таким образом во всех приведенных ситуациях самым быстрым вариантом оказывается вариант с использованием 8 потоков. При таком использовании наилучшим образом задействуются все 8 физических ядер процессора.

Также прирост в 20-30 секунд дает сдвиг во времени сборок относительно друг друга. Это объясняется тем, что в процессе построения существуют отрезки времени, когда CPU задействован по-минимуму. Это ситуации, когда происходит скачивание или копирование файлов, а также выполнение других операций, не требующих больших мощностей процессора. При сдвиге сборок во времени удается заполнить эти свободные отрезки задачами построения, которые обычно максимально задействуют процессор.

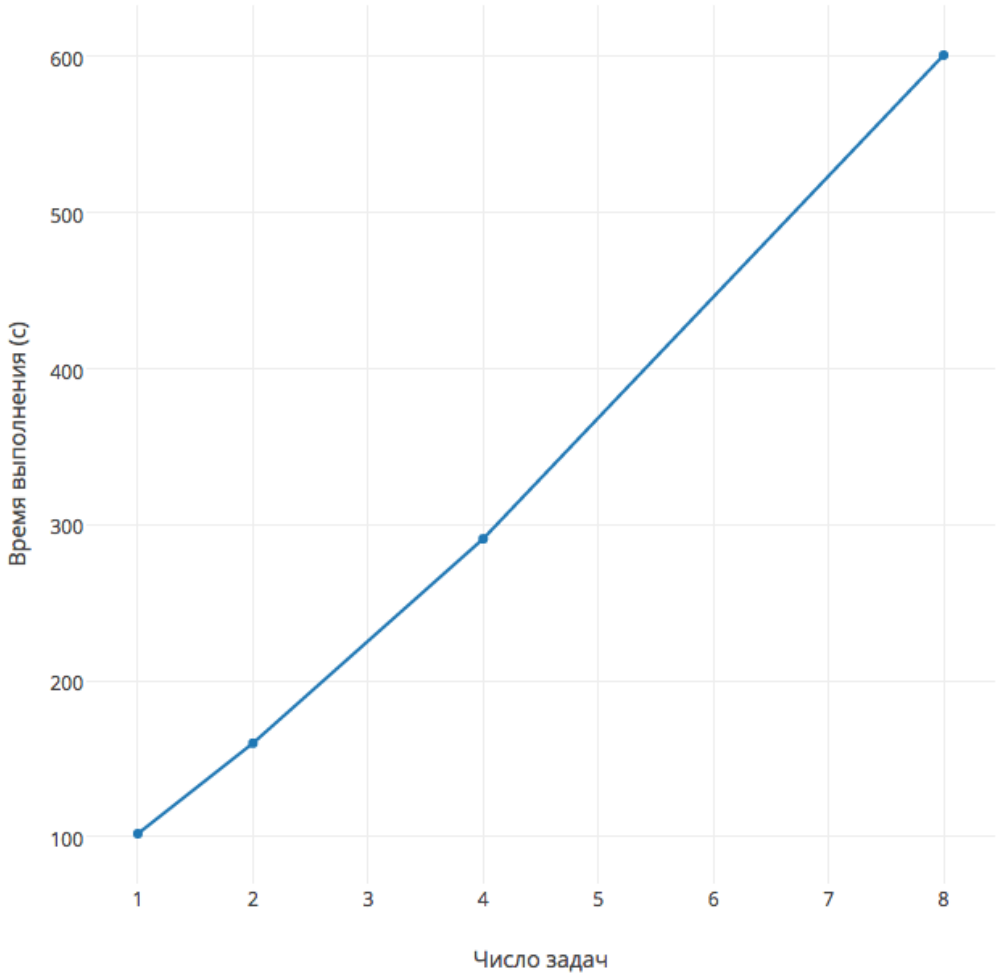


График зависимости времени выполнения от числа параллельных задач (каждая задача запущена на 8 потоков). Наблюдается линейная зависимость.

Заключение

В ходе работы удалось настроить систему непрерывной интеграции для построения и тестирования проекта MPD Root.

Одним из основных принципов систем CI является принцип быстроты сборки — разработчик должен мгновенно убедится в работоспособности своих изменений и, в случае ошибок, внести исправления на ранней стадии. Поэтому время сборки является основным параметром, который нужно оптимизировать.

В этом плане самое заметное сокращение времени сборки дает кэширование зависимостей проекта. В данной работе кэширование сборки зависимостей позволило сократить время общей сборки с 30 до 2-3 минут. На втором месте по эффективности оптимизации идет метод распараллеливания CI-задач, запущенных на одном сервере. Из проведенных экспериментов следует, что чем больше физических ядер процессора и чем меньше отрезков времени, где ядра процессора загружены не полностью, тем эффективнее результат.

Также, очевидно, что увеличение количества CI-Runner-серверов (при вынесении их на отдельные машины) даст возможность выполнять большее число CI-задач одновременно.

Сам процесс непрерывной интеграции требует постоянной настройки и редактирования как CI-скрипта, так и образа контейнера сборки. При добавлении новых блоков кода в проект, нужно покрывать их тестами и добавлять вызов этих тестов в CI-скрипт.

Список литературы

1. Fowler M., Foemmel M. Continuous integration //Thought-Works) http://www.thoughtworks.com/Continuous Integration.pdf. – 2006. – С. 122.
2. Многоцелевой Детектор - MPD (концептуальный дизайн-проект) http://nica.jinr.ru/files/CDR_MPD/MPD_CDR_ru.pdf
3. Shore J. Continuous Integration on a Dollar a Day. http://www.jamesshore.com/Blog/Continuous-Integration-on-a-Dollar-a-Day.html
4. Fink J. Docker: A software as a service, operating system-level virtualization framework //Code4Lib Journal. – 2014. – Т. 25.

Приложение

Файл Dockerfile

```
### CACHE NOTE
# command results are being cached
# everything starting from changed RUN commands or ADD commands will NOT use cache

FROM ubuntu:14.04

### apt-get installs

RUN apt-get update
RUN sudo apt-get install cmake cmake-data g++ gcc gfortran \
  build-essential make patch sed libx11-dev libxft-dev \
  libxext-dev libxpm-dev libxmu-dev libglu1-mesa-dev \
  libgl1-mesa-dev ncurses-dev curl bzip2 gzip unzip tar \
  subversion git xutils-dev flex bison lsb-release \
  python-dev libxml2-dev wget libssl-dev \
  libcurl4-openssl-dev automake autoconf libtool -y

### install fairsoft

# clone from Fairsoft Repo
RUN cd /opt; \
  git clone https://github.com/FairRootGroup/FairSoft.git fairsoft; \
  cd fairsoft; \
  git checkout mar15p6; \

# create install config file
  sudo rm -rf fs_inst_settings; \
  echo "compiler=gcc" >> fs_inst_settings; \
  echo "debug=no" >> fs_inst_settings; \
  echo "optimize=yes" >> fs_inst_settings; \
  echo "geant4_download_install_data_automatic=yes" >> fs_inst_settings; \
  echo "geant4_install_data_from_dir=no" >> fs_inst_settings; \
  echo "build_python=yes" >> fs_inst_settings; \
  echo "install_sim=yes" >> fs_inst_settings; \
  echo "SIMPATH_INSTALL=$(pwd)/install" >> fs_inst_settings; \
  echo "platform=linux" >> fs_inst_settings; \

# install FairSoft
  ./configure.sh fs_inst_settings;

# add repo files to docker image
ADD ./ /repo_root/

```

Файл .gitlab-ci.yml

```
stages:
  - build
  - test
  - cleanup


# Please edit Dockerfile to change build environment
build_mpd:
  stage: build
  before_script:
    # Build image using Dockerfile
    #  (everything starting from changed RUN commands or ADD commands will not use cache)
    - sudo docker build -t ci-env-$CI_PROJECT_ID .
  script:
    # start container using built image
    - |
      sudo docker run -i ci-env-$CI_PROJECT_ID /bin/bash -s <<EOF

      cd repo_root
      mkdir build
      . SetEnv.sh
      cd build
      cmake ..
      #build framework
      make -j8
      . config.sh

      EOF
    - echo "Project ID is $CI_PROJECT_ID"
    - CONTAINER_ID=$(sudo docker ps -a | awk '{ print $1,$2 }' | grep ci-env-$CI_PROJECT_ID | awk '{print $1}' | head -1)
    - echo $CONTAINER_ID;
      # save changes in container to image
    - sudo docker commit $CONTAINER_ID ci-build-$CI_BUILD_REF
    - echo "ci-build-$CI_BUILD_REF"


test_mpd:
  stage: test
  script:
    - |
      # run another container using image created on prev stage
      sudo docker run -i ci-build-$CI_BUILD_REF /bin/bash -s <<EOF
      -e;
      cd repo_root/build
      . config.sh
      root -b ../macro/mpd/runMC.C
      #ctest -VV
      #make test
      EOF

```

```
clean_allocated_container:
  stage: cleanup
  script:
    - for i in `sudo docker images|grep ci-build-$CI_BUILD_REF |awk '{print $3}'`;do sudo docker rmi -f $i;done
    # remove none images if they are not in use
    - for i in `sudo docker images|grep none |awk '{print $3}'`;do sudo docker rmi $i;done || true;
    # remove Exited and Dead containers, ignore errors
    # (errors can be caused by another thread trying to remove container)
    - sudo docker rm `sudo docker ps -a | grep Exited | awk '{print $1 }'` || true;
    - sudo docker rm `sudo docker ps -a | grep Dead | awk '{print $1 }'` || true;
  when: always
```