

Университет ИТМО

Кафедра вычислительной техники

Отчет по прохождению практики

Студента

Р3311 группы

Морозова С.Д.

Руководитель
Соснин В.В.

Санкт-Петербург

2016

Содержание

1	Введение	3
2	Система компьютерной верстки \TeX(\LaTeX)	4
2.1	Краткое описание	4
2.2	Сравнение \LaTeX и MS Word	5
2.3	Выбор инструмента редактирования	6
3	Системы контроля версий	7
3.1	Краткое описание	7
3.2	Git	7
3.2.1	Особенности	7
3.2.2	Основные команды	10
3.3	GitHub	11
4	Паралельные вычисления	12
4.1	Немного теории	12
4.2	Характеристики параллельных вычислений	14
4.2.1	Ускорение	14
4.2.2	Эффективность	14
4.2.3	Упущенная эффективность	14
4.3	Основные проблемы параллельного программирования	15
4.3.1	Синхронизация	15
4.3.2	Гонка данных	15
4.3.3	Взаимная блокировка (Deadlock)	16
4.4	Распарелеливание цикла	17
5	Функции замера времени	18
5.1	Категории функций?	18
5.1.1	Календарное время	18
5.1.2	Время процессора	19

5.2	Функции	19
5.2.1	Кроссплатформенные функции	19
5.2.2	Windows	21
5.2.3	Linux	21
5.3	Проблемы и сложности замеров времени при параллельный вычислениях	23
6	Практическая часть?	24
6.1	Описание экспериментальной программы	24
6.2	Результаты работы программы	25
6.3	Выводы	27
7	Вывод по производственной практике	28
8	Список литературы	29

1 Введение

Тема прохождения практики — параллельные вычисления. Цель задания — сравнить различные функции в языке C, которые можно использовать для измерения времени работы параллельных программ.

Однако требования руководителя практики таковы, что перед тем как приступить к выполнению основного задания нужно ознакомиться с системой компьютерной вёрстки TeX (LaTeX), которая должна использоваться для написания отчёта, и ознакомиться с системой контроля версий Git, с последующим созданием учетной записи на сайте GitHub или аналогичном.

2 Система компьютерной верстки \TeX (\LaTeX)

2.1 Краткое описание

\TeX — система компьютерной вёрстки с формулами, разработанная американским профессором информатики Дональдом Кнутом. Название происходит от греческого слова $\tau\epsilon\chi\upsilon\eta$ — «искусство», «мастерство», поэтому последняя буква читается как русская Х. Хотя \TeX является системой набора и верстки, развитые возможности макроязыка \TeX делают его Тьюринг-полным языком программирования.

\TeX работает с боксами (box) и клеем (glue). Бокс — двумерный объект прямоугольной формы, характеризуется тремя величинами (высота, ширина, глубина). Элементарные боксы — это буквы, которые объединяются в боксы-слова, которые в свою очередь сливаются в боксы-строки, боксы-абзацы и т.д.

Между боксами располагается клей, который имеет некоторую ширину по умолчанию и степени увеличения/уменьшения этой ширины. Объединяясь в бокс более высокого порядка, боксы могут шевелиться, но после того как найдено оптимальное решение, это состояние закрепляется, и полученный бокс выступает как единое целое.

Интересный факт. На версии 3.0 дизайн был заморожен, поэтому в новых версиях не будет добавления новой функциональности, только исправление ошибок. Версия \TeX 'а асимптотически приближается к числу π . Это факт говорит о том, что последняя версия 3.14159265 (январь 2014) является крайне стабильной и возможны лишь мелькие исправления. Дональд Кнут заявил, что последнее обновление (сделанное после его смерти) сменит номер версии на π , и с этого момента все ошибки станут особенностями.

\LaTeX — созданный Лесли Лэмпортом набор макрорасширений (или макропакет) системы компьютерной вёрстки \TeX , который облегчает набор сложных документов. Стоит отметить, что как и любой другой макропа-

кет¹ \LaTeX не может расширить возможности \TeX (все, что можно сделать в одном пакете можно сделать и в любом другом). Пакет позволяет автоматизировать многие задачи набора текста и подготовки статей, включая набор текста на нескольких языках, нумерацию разделов и формул, размещение иллюстраций и таблиц на странице, ведение библиографии и др. Все это делает \LaTeX крайне удобным инструментом для написания научных статей, диссертаций и т.п..

2.2 Сравнение \LaTeX и MS Word

В качестве сравнения — перечислим плюсы и минусы \LaTeX перед MS Word(а так же всеми его аналогами).

Плюсы \LaTeX :

- Кроссплатформенность
- Язык международного обмена по математике и физике (большинство научных издательств принимают тексты в печать только в этом формате)

Минусы \LaTeX :

- Не является системой типа WYSIWYG ²
- При серьезных отклонениях от стандартных стилей документов требуется достаточно сложное программирование

То есть, выбирая между \LaTeX и MS Word, стоит обратить внимание на то,какой текст вы собираетесь печатать, насколько нестандартный будет стиль текста, на его примерный объем. В некоторых случаях достаточно использовать MS Word, в других — использование \LaTeX может заметно упростить работу.

¹ Так же существуют Plain \TeX , AMS- \TeX , AMS- \LaTeX и т.д.

²What You See Is What You Get(Что видишь, то и получишь). Стоит отметить, что существуют дистрибутивы \TeX в которых есть попытки реализовать WYSIWYG. Например платный дистрибутив BaKoMa \TeX + текстовый редактор BaKoMa \TeX Word.

2.3 Выбор инструмента редактирования

В ходе изучения всех возможных вариантов работа с \LaTeX для создания данного отчета, была выбрана программа Textmaker ³.

Выбор Textmaker'а обусловлен следующими его особенностями:

- Автоматическая подсветка синтаксиса
- Функция автодополнения команд \LaTeX
- Соккрытие блоков кода (Code folding)
- Быстрая навигация по структуре документа
- Указание на строку с ошибкой, для быстрой отладки
- Интегрированный просмотр PDF

³Официальный сай Textmaker: <http://www.xmlmath.net/texmaker/>

3 Системы контроля версий

3.1 Краткое описание

Система контроля версий (СКВ) — это система, регистрирующая изменения в одном или нескольких файлах с тем, чтобы в дальнейшем была возможность вернуться к определённым старым версиям этих файлов.

СКВ широко используются при разработке программного обеспечения, для хранения кодов разрабатываемых программ. Однако данные системы подходят не только программистам. Художники, которые хотят сохранять каждое изображение/эксиз своей работы, писатели пишущие книги или научные статьи, бухгалтеры, которые хранят разные версии отчетов и т.д., все они могут использовать СКВ для достижения своих целей.

Иначе говоря СКВ можно применять в любых областях в которых ведётся работа с большим количеством непрерывно изменяющихся электронных документов.

3.2 Git

Git — созданная Линусом Торвальдсом, распределенная система контроля версий.

3.2.1 Особенности

Одной из основных особенностей Git состоит в способе хранения данных. В принципе, большинство других систем хранит информацию как список изменений (патчей) для файлов. Эти системы (CVS, Subversion, Perforce, Vazaar и другие) относятся к хранимым данным как к набору файлов и изменений, сделанных для каждого из этих файлов во времени, как показано на Рис. 1

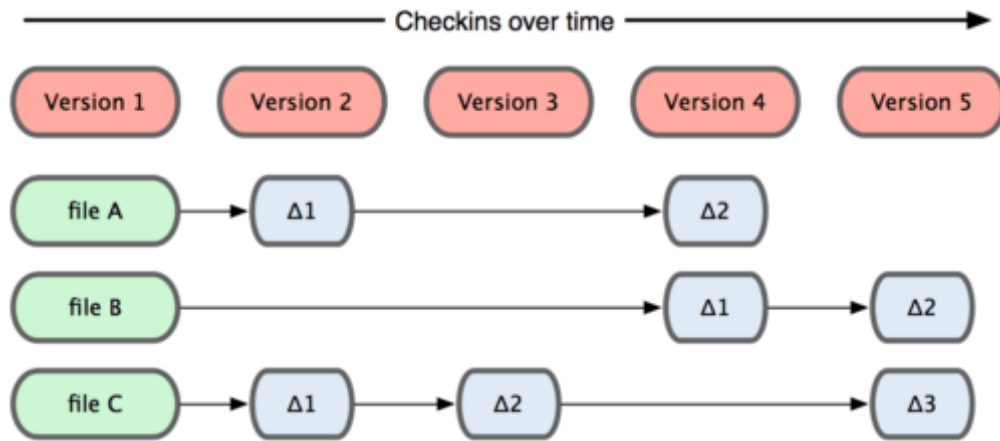


Рис. 1: Другие системы хранят данные как изменения к базовой версии для каждого файла.

Git не хранит свои данные в таком виде. Вместо этого Git считает хранящиеся данные набором слепков небольшой файловой системы. Каждый раз, когда вы фиксируете текущую версию проекта, Git, по сути, сохраняет слепок того, как выглядят все файлы проекта на текущий момент. Ради эффективности, если файл не менялся, Git не сохраняет файл снова, а делает ссылку на ранее сохранённый файл. То, как Git подходит к хранению данных, похоже на Рис. 2

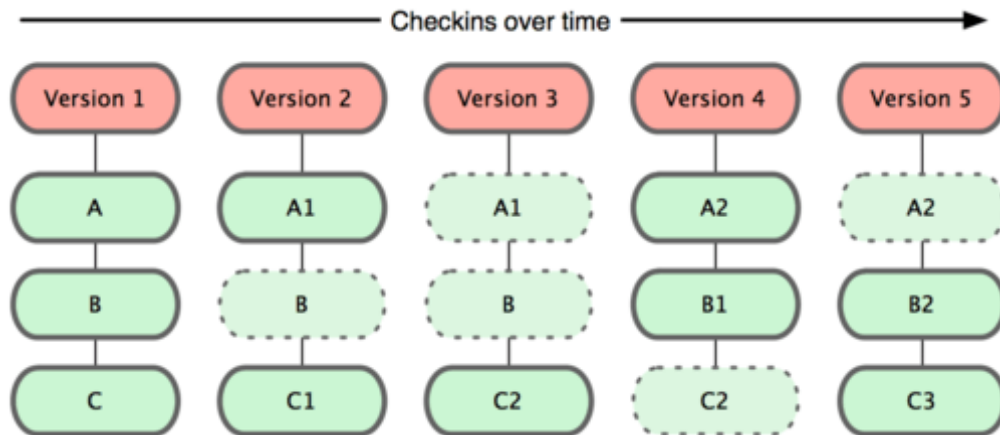


Рис. 2: Git хранит данные как слепки состояний проекта во времени.

За счет этого, для большинства операций в Git нужны только локальные ресурсы и файлы. Что в свою очередь определяет два основных преимущества Git перед остальными СКВ.

- Быстродействие. Поскольку вся история проекта хранится локально у вас на диске, большинство операций кажутся практически мгновенными (в отличии от централизованных систем, где практически на каждую операцию накладывается сетевая задержка).
- Возможность работать (делать коммиты) без доступа к сети или VPN.

3.2.2 Основные команды

В целом, следующая картинка (Рис. 3) наглядно демонстрирует основные команды Git, знание которых достаточно, чтобы начать им пользоваться.

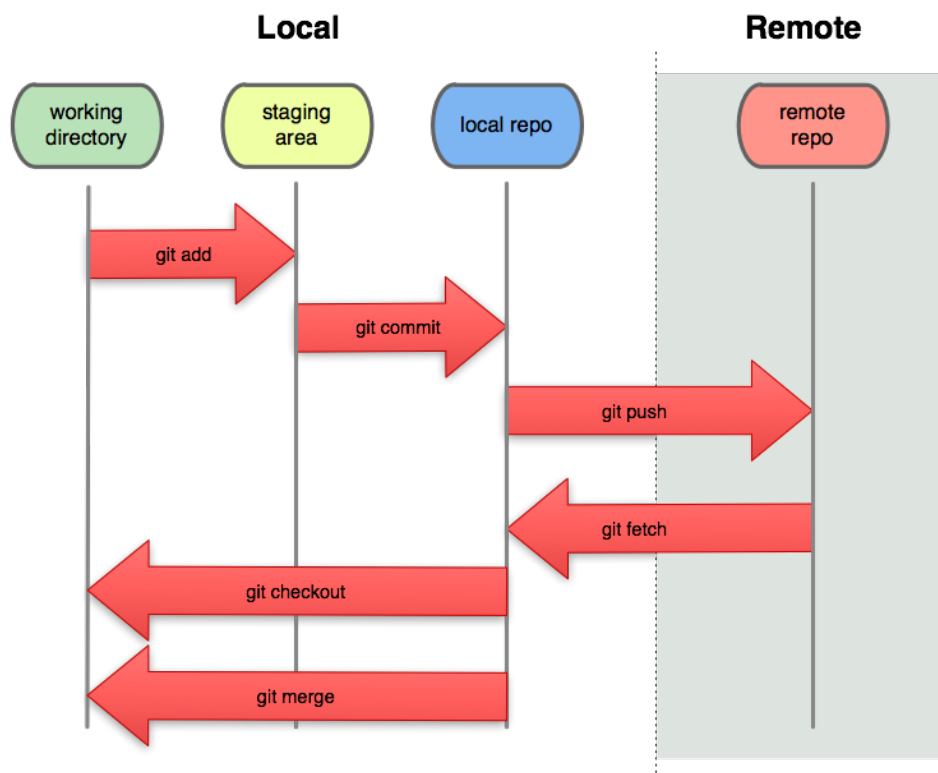


Рис. 3: Основные команды при работе с Git.

3.3 GitHub

[GitHub](#) — крупнейший веб-сервис для хостинга IT-проектов и их совместной разработки. Основан на системе контроля версий Git и разработан на Ruby on Rails и Erlang компанией GitHub, Inc (ранее Logical Awesome).

В ходе прохождения практики, на сайте GitHub была создана учетная запись [MorozovSD](#). В данной учетной записи был создан репозиторий [Practice-2016](#) по которому можно легко отследить процесс прохождения практической работы.

4 Параллельные вычисления

Т.к. практика предполагает не доскональное изучение параллельного программирования, а лишь сравнение функций замера времени в программах, работающих на основе параллельных вычислений, то данная глава носит более ознакомительных характер, содержащий тот минимум знаний, необходимый для работы с этой области.

4.1 Немножко теории

Мультипрограммирование — параллельное выполнение нескольких программ. Мультипрограммирование позволяет уменьшить общее время их выполнения.

Под параллельными вычислениями понимается параллельное выполнение одной и той же программы. Параллельные вычисления позволяют уменьшить время выполнения одной программы. Чаще всего, хороший последовательный алгоритм не является таковым для параллельного выполнения. (а параллельные алгоритмы могут не являться эффективными при работе с одним процессором), поэтому одна из задач параллельных вычислений это разработка эффективных алгоритмов, полностью использующие количество предоставленных процессоров.

Т.к. тема практики — измерение времени работы параллельно работающей программы, то необходимо получить оценку времени выполнения программы одним процессором T_1 для идеализированного случая, когда число процессоров не ограничивается — T_∞ . А так же оценить верхнюю и нижнюю границы времени выполнения конечным число процессоров T_p ⁴.

Для введенных характеристик очевидно следующее соотношение:

$$T_\infty \leq T_p \leq T_1 \quad (1)$$

Для T_p справедлива следующая оценка снизу:

⁴В отчете будут представлены только конечные формулы, без доказательств

$$T_p \geq \frac{T_1}{p} \quad (2)$$

Для T_p справедлива следующая оценка сверху:

$$T_p \leq \frac{T_1}{p} + T_\infty \quad (3)$$

Исходя из вышеперечисленных неравенств, неравенство (1) может быть заменено более точным:

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty \quad (4)$$

На графике это выглядит следующим образом (Рис. 4):

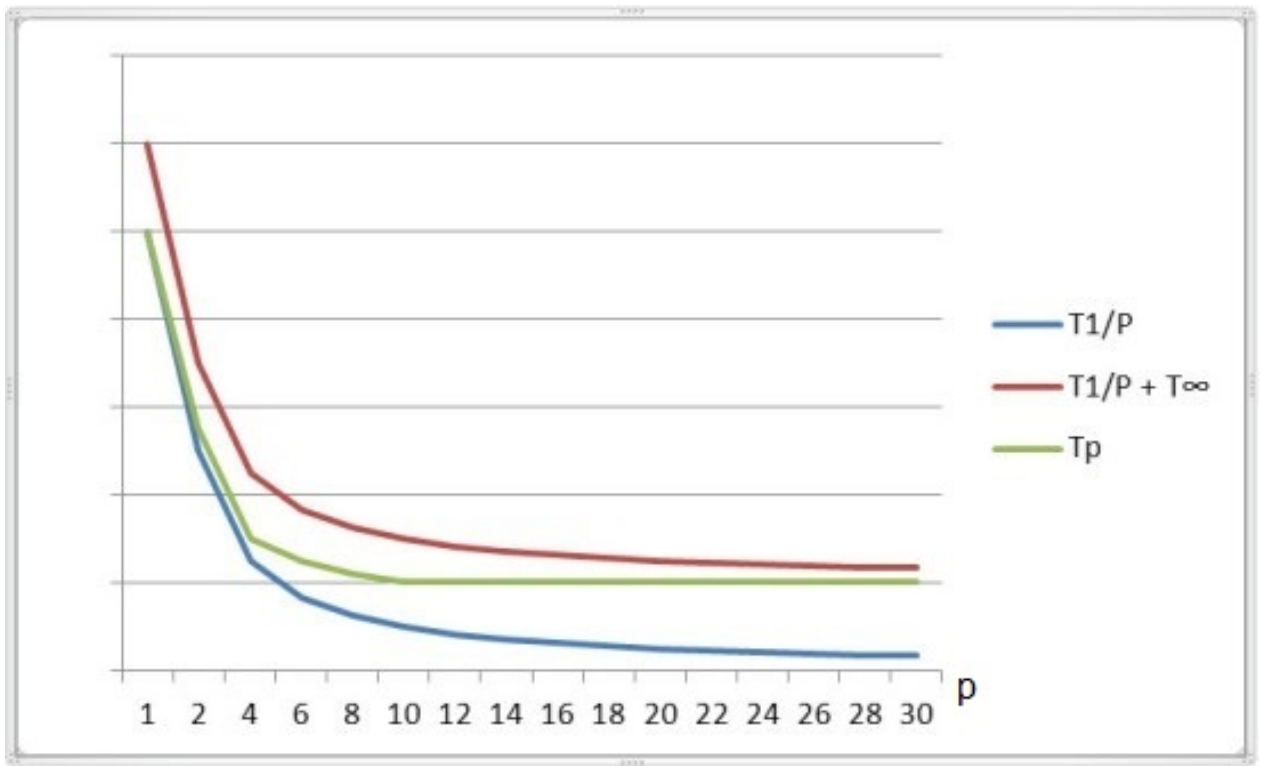


Рис. 4: Графическая интерпретация поведения функции T_p

Очевидно, что при $p = 1$ графики функций T_p и $\frac{T_1}{p}$ совпадают. Так же происходит совпадение графиков T_p и $\frac{T_1}{p} + T_\infty$ при $p \rightarrow \infty$.

4.2 Характеристики параллельных вычислений

4.2.1 Ускорение

Ускорение $S_p(n)$ ⁵ определяют как отношение:

$$S_p(n) = \frac{T_1(n)}{T_p(n)} \quad (5)$$

Интерпритировать данную формулу следует как отношение время наилучшего алгоритма, для которого достаточно одного процессора, и время наилучшего параллельного алгоритма, который может использовать p имеющихся процессоров.

4.2.2 Эффективность

Эффективность $E_p(n)$ определяют как отношение:

$$E_p(n) = \frac{S_p(n)}{p} \quad (6)$$

При оптимальном ускорении⁶ эффективность равна 1. Если же эффективность существенно ниже 1, то часто число процессоров целесообразно уменьшить, используя их более эффективно.

4.2.3 Упущенная эффективность

Мера неиспользованных возможностей — упущенной выгоды — $U(n)$, определяют следующим образом:

$$U(n) = \frac{T_p(n)}{T_{p_{opt}}} - 1 \quad (7)$$

Оптимальное время, которое можно достичь, используя p процессоров, дается нижней оценкой для T_p , поэтому получаем:

⁵Все вводимые характеристики рассматриваются как функции параметра n , характеризующего сложность решаемой задачи. Обычно n понимается как объем входных данных.

⁶Оптимальное ускорение достигается когда $T_p = \frac{T_1}{p}$

$$U(n) = p \frac{T_p(n)}{T_1} - 1 \quad (8)$$

Если для компьютера с p ядрами время решения задачи оптимально и сокращается в $\sim p$ раз в сравнении с решением задачи на одноядерном компьютере, то наши потери равны нулю, возможности компьютера полностью используются. Если же задача решается за время T_1 — столь же долго, как на одноядерном компьютере, то потери пропорциональны числу неиспользованных ядер.

4.3 Основные проблемы параллельного программирования

4.3.1 Синхронизация

Синхронизация нужна для того, чтобы согласовать обмен информацией между модулями (между параллельно выполняемыми множествами операций). Синхронизация может привести к простое процессора, т.к. после достижения точки синхронизации он должен ждать, пока другие задания достигнут точки синхронизации. Задержка с подачей в процессор необходимых данных ведет к простое процессора и снижению эффективности параллельной обработки

4.3.2 Гонка данных

Проблема «гонки данных» возникает для мультипроцессорных компьютеров с общей памятью. В одни и те же моменты времени процессоры могут получать доступ к одним и тем же данным, хранимым в общей памяти, как для чтения, так и для записи.

Если с чтением данных проблем не возникает, то одновременная запись двух разных значений в одну и ту же ячейку памяти не возможна. Запись всегда идет последовательно, следовательно в памяти останется храниться значение, пришедшее последним (причем не известно какое значение каким

придет). Конкурирование процессоров за запись в одну и ту же ячейку памяти и есть «гонка данных».

Один из способов справиться с этой проблемой — это закрытие доступа к ресурсу первым пришедшим процессором. Остальные процессоры прерывают выполнение и становятся в очередь за обладание ресурсом. Обладатель ресурса спокойно выполняет свою работу, а по ее окончании открывает ресурс, с которым теперь начинает работать тот, кто первым стоит в очереди.

4.3.3 Взаимная блокировка (Deadlock)

Блокировка — хороший механизм решения проблемы «гонки данных». Однако блокировка может прервать выполнение всей программы, когда наступает ситуация, называемая взаимной блокировкой, клинчем, смертельным объятием или deadlock'ом.

В качестве примера рассмотрим простейшую ситуацию, приводящую к возникновению клинча. Пусть есть два конкурента A и B , претендующие на два ресурса x и y . Пусть гонку за ресурс x выиграл A и соответственно закрыл этот ресурс для B . Гонку за ресурс y выиграл B и закрыл ресурс для A . Но A , чтобы закончить свою работу нужен ресурс y , поэтому он стал в очередь, ожидая освобождения ресурса. Симметрично, B находится в очереди, ожидая освобождения ресурса x . Возникает ситуация вечного ожидания (которое может разрешить только внешнее воздействие), когда ни A , ни B не могут продолжить свою работу. (Рис. 5):

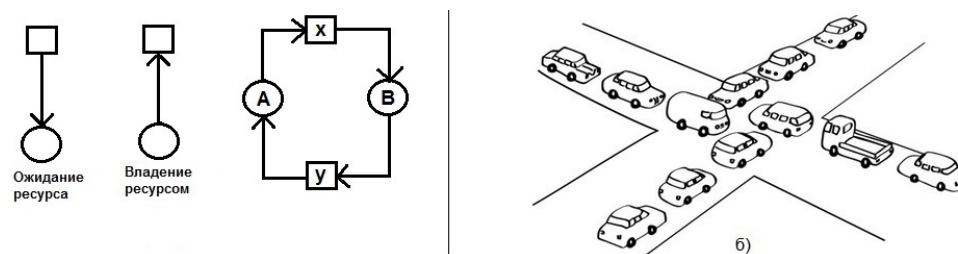


Рис. 5: Взаимная блокировка (клинч): а) Общий вид. б) На примере перекрестка.

4.4 Распарелеливание цикла

Думаю стоит перенести эту главу в практическую часть, и в ней предложить наиболее оптимальный параллельный алгоритм суммирования (т.к. тестовая программа решает задачу суммирования квадратов).

5 Функции замера времени

Эта глава посвящена функциям замера времени языка Си.

5.1 Категории функций?

Функции работы со временем можно отнести к трем категориям:

- Функции календарного времени
- Функции для измерения прошедшего времени CPU
- Функции для установки будильников и таймеров ⁷

5.1.1 Календарное время

Осуществляет для слежения за датами и временем согласно Грегорианскому календарю. Существуют несколько способов представления информации даты и времени. Т.к. нас интересует разрешающая способность (точность замеров времени) рассмотрим эти способы представления в зависимости от их разрешающей способности.

- Тип данных `time_t` — компактное представление, обычно дает число секунд, истекающих начиная с некоторого основного времени. Разрешающая способность одна секунда
- Тип данных `struct timeval` — представление времени с большей точностью. Разрешающая способность до микросекунд ⁸
- Тип данных `timespec` — представление времени с большей точностью. Разрешающая способность до наносекунд

⁷Данная категория функций не используется в практической работе, и рассматриваться не будет

⁸На некоторых платформах возможно отслеживание времени только в пределах разрешающей способности системного таймера, который в общем случае устанавливается на значение 100 Гц

5.1.2 Время процессора

Процессорное время отлично от фактических часов, тем что оно не включает времени выполнения другого процесса и все потраченное время на ожидание ввода-вывода. Процессорное время представляется типом данных `clock_t`, и дано как ряд импульсов времени относительно произвольного базового времени⁹, отмечающего начало одиночного вызова программы.

В зависимости от архитектуры компьютера и операционной системы способ слежения за процессорным временем может быть разным. Общее для внутренних часов процессора то, что разрешающая способность где-то между тысячной и миллионной долей секунды.

5.2 Функции

В данной главе будут описаны функции замера времени в операционных системах Windows и Linux.

5.2.1 Кроссплатформенные функции

`clock`

`clock_t clock (void)`

Функция возвращает прошедшее процессорное время. Базовое время произвольно, но не изменяется внутри одиночного процесса. Если процессорное время не доступно или не может представляться, `clock` возвращает значение (`clock_t`) (-1).

`time`

*`time_t time(time_t * timeptr)`*

⁹Для перевода количества импульсов в секунды, количество импульсов необходимо делить на `CLOCKS_PER_SEC` (число импульсов времени `clock` в секунду)

Функция возвращает текущее календарное значение времени в секундах. Если аргумент не является нулевым указателем, ей передается значение времени типа `time_t`.

omp_get_wtime

double omp_get_wtime(void)

Функция возвращает значение, прошедшее от некоторого "времени в прошлом которое гарантированно не затрагивает программа. The time is measured per thread, no guarantee can be made that two distinct threads measure the same time.?

gmtime

*struct tm * gmtime(const time_t * timeptr)*

Функция преобразует системное время в секундах в дату по Гринвичу. Результат помещается в структуру типа `tm` и функция возвращает указатель на эту структуру.

localtime

*struct tm * localtime(const time_t * timeptr)*

Функция преобразовывает текущее значение времени, передаваемое как аргумент, через указатель `timeptr` на `time_t` в структуру `tm` (местное время). Так же функция возвращает указатель на эту структуру.

asctime

*char * asctime (const struct tm * m_time)*

Функция преобразует локальное (местное) время представленное в виде структуры типа `struct tm`, на которую указывает аргумент `m_time` в

текстовую строку. Результат преобразования возвращается функцией в виде указателя на строку содержащую дату и время. Возвращаемая строка имеет следующий формат:

«ННН МММ ДД ЧЧ: ММ: СС ГГГГ \ n \ 0», где

ННН — это день недели,

МММ — месяц,

ДД — день,

ЧЧ: ММ: СС — время,

ГГГГ — год.

5.2.2 Windows

localtime_s

ctime_s

5.2.3 Linux

localtime_r

*struct tm * localtime_r (const time_t * s_time, struct tm * m_time);*

Функция преобразовывает текущее значение времени, передаваемое как аргумент, через указатель `timeptr` на `time_t` в структуру `tm` (местное время), на которую указывает аргумент `m_time`. Так же функция возвращает указатель на эту структуру.

asctime_r

*char * asctime_r (const struct tm * m_time, char * buf)*

Функция преобразует локальное (местное) время представленное в виде структуры типа `struct tm`, на которую указывает аргумент `m_time` в текстовую строку длиной 26 символов. Результат преобразования помещается в

строку, на которую указывает аргумент `buf`. Возвращаемая строка имеет формат аналогичный формату строки возвращаемый функцией `asctime`.

ctime_r

*char * ctime_r(const time_t * clock, char * buf)*

Функция эквивалентна последовательному выполнению функций `localtime_r()` и `asctime_r()`.

gettimeofday

*int gettimeofday(struct timeval *tv, struct timezone *tz)*

Функция возвращает системное время в виде структуры `timeval`.

clock_gettime

*int clock_gettime(clockid_t clk_id, struct timespec *tp)*

Функция обеспечивает доступ к нескольким видам системных таймеров и имеет наносекундное разрешение. `clk_id` — задает вид таймера, например:

- `CLOCK_THREAD_CPUTIME_ID` — Таймер процессора работающий с каждым потоком с высокой разрешающей способностью
- `CLOCK_PROCESS_CPUTIME_ID` — Таймер процессора работающий с каждым процессом с высокой разрешающей способностью
- `CLOCK_REALTIME` — Таймер реального времени в масштабе всей системы

5.3 Проблемы и сложности замеров времени при параллельный вычислениях

localtime gmtime - не хорошо, процессорное время тоже немного не то.
Возможно этот параграф не нужен, это все можно перести в практическую часть в качестве части вывода.

6 Практическая часть?

6.1 Описание экспериментальной программы

Эксперименты проводились с довольно таки простой программой код которой представлен ниже:

Листинг 1: Тестовая программа

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  double run_parallel_experiment(int amount_of_threads,
5      unsigned int experiment_size) {
6      unsigned int i;
7      double sum_of_squares = 0;
8      omp_set_num_threads(amount_of_threads);
9      #pragma omp parallel for reduction(+:sum_of_squares)
10     for (i = 0; i < experiment_size; ++i) {
11         sum_of_squares += i*i;
12     }
13     return sum_of_squares;
14 }
15
16 int main() {
17     double result;
18     double t1;
19     double t2;
20     //"Start timer"
21     result = run_parallel_experiment(2, 100000000);
22     //"Stop timer"
23     printf("result=%e, time_in_milliseconds=%f\n", result,
24         1000*(t2 - t1));
25
26     return 0;
27 }
```

С ходе экспериментов в строки №19 и 21 подставлялись различные функции замера времени, что в свою очередь сказывалось на выводе программы¹⁰.

Для получения более точных и достоверных результатов были предприняты следующие шаги:

- Было отключено большинство работающих программ, процессов.
- Запуск программы осуществлялся несколько раз, с последующим определением минимального, среднего и максимального значений

6.2 Результаты работы программы

В столбце "Итог" отображается вывод о том, стоит ли использовать данную функцию в параллельных программах. Эту таблицу можно отправить в главу функции замера времени.

¹⁰Вывод программы есть ни что иное как время выполнение программы, по крайней мере мы хотим, чтобы это было временем выполнения.

Функции	Критерии оценивания				
	Платформа	Точность	Thread-safety	Реента- бельность	Возв. значе- ние
clock	Кросс.		+	+	
time	Кросс.		+	+	
gmtime	Кросс.		-	-	
gmtime_r	Linux		+	+	
localtime	Кросс.		-	-	
localtime_r	Linux		+	+	
asctime	Кросс.		-	-	
asctime_r	Linux		+	+	
ctime	Кросс.		-	-	
ctime_r	Linux		+	+	
gettimeofday	Linux		+	+	
clock_gettime	Linux		+	+	
omp_get_wtime	Кросс.		+	+	
GetTickCount	Windows		+	+	
GetTickCount64	Windows		+	+	
GetLocalTime	Windows		?	?	
GetSystemTime	Windows		?	?	

Тут что-то в духе "в ходе проверки функций, тыли выявлены наиболее подходящие для пар. вычислений. Результаты работы этих функций представлены в таблице 2.

Или указать тут все функции и те, результат которых некорректен подкрывать красным. А ниже (например в выводе) описать причины каждого некорректного вывода.

(в практическом программировании под современные ОС термин «реентерабельный» на деле равносителен термину «thread-safe»).

W									
Функции	Кол-во ядер								
	1			2			3		
	Min	Ave	Max	Min	Ave	Max	Min	Ave	1
clock	32.609	32.641	32.673	16.422	16.445	16.469	11.187	11.226	1
time	32	34	36	16	16	17	11	11	
omp_get_wtime	32.61	32.649	32.688	16.442	16.455	16.469	11.172	11.227	1

6.3 Выводы

7 Вывод по производственной практике

8 Список литературы