Оптимизация программ

Оглавление

Введение

Довольно часто, имея дело с какой-нибудь программой, мы жалуемся на то, что она медленно работает, сожалеем о том, что компьютер у нас не такой, как у нашего соседа. “Вот на нем бы мы все сделали в два счета”. Но все ли дело только в компьютере? Да, несомненно, мегагерцы и большой объем памяти играют свою роль, но не менее важно то, как написана программа. Конечно с уже готовыми продуктами, поставляемыми в виде исполняемых файлов ничего сделать не получится, но если нам предоставлены исходные тексты, или если мы сами пишем программу, то за повышение производительности можно побороться.

Почему появился данный курс? Курс не является теоретическим. Здесь не будет сложных математических выкладок и доказательств теорем. В курсе собраны примеры оптимизации, с которыми пришлось столкнуться при разработке программ, их распараллеливанию, в результате преподавательской и научной деятельности. Все приведенные программы, из которых взяты примеры, если не оговаривается особо, разрабатывались под архитектуру x86 и собирались компилятором GNU C/C++.

В этом курсе планируется рассказать о том, что такое оптимизация, в чем она заключается, рассказать о методах оптимизации программ. В рамках курса будет рассматриваться оптимизация как последовательных, так и параллельных программ.

Данный курс будет полезен всем тем, кто увлекается оптимизацией программ и тем, кто планирует или уже связал свою жизнь с программированием. В курсе дается много полезных советов, которые гораздо выгоднее сразу применять на практике, чем потом тратить время на то, чтобы заставить программу выполняться быстрее.

Оптимизация – улучшение программы относительно каких-либо критериев (потребляемая память, ресурсы процессора, масштабируемость и т.д.). Мы будем понимать под оптимизацией увеличение скорости работы программы. Следует помнить, что оптимизация является палкой о двух концах. Для большинства случаев справедливы следующие утверждения:

- более быстрая программа требует больше оперативной памяти и имеет больший размер кода;
- для разработки более быстрой программы требуется затратить больше времени.

Принимая во внимание эти утверждения следует выбирать оптимальное соотношение затраченных сил и ожидаемого от этого результата.

Зачем нужна оптимизация? Ведь разработано множество теорий по создания компиляторов, написано множество книг и статей. Неужели компиляторы не смогут хорошо оптимизировать нашу программу? На такой вопрос можно возразить так:

- Ни один компилятор не в состояние полностью охватить и проанализировать вашу программу. Это очень сложный процесс.
- Компилятор не обладает базой алгоритмов.
- Может ваша программа так и должна выполняться как вы ее написали?
- Знание программиста о программе (предметной области) много шире, чем знание компилятора о ней.
- Плохую программу ни один компилятор не сможет хорошо оптимизировать.

Поэтому человек, обладающий достаточным количествам знаний, может написать более оптимальную программу на языке ассемблера, чем ее создаст компилятор из написанной программы на языке высокого уровня. Но не стоит рассматривать это утверждение, как призыв к отказу от языка C/C++ в пользу ассемблера.

В пользу оптимизации программ можно привести высказывание одного из разработчиков движка Counter-Strike Майкла Эбраша: "Всякая оптимизация, ощущаемая пользователем, заслуживает того, чтобы ее сделать"

Оптимизация должна происходить в несколько этапов:

1. на стадии проектирования программы (выбор оптимальных алгоритмов)
2. на стадии написания программы/после написания (оптимальная реализация выбранных алгоритмов)
3. оптимизация программы под заданную архитектуру (учет особенностей процессора, организации памяти)

Конечно, такое разбиение на этапы не всегда корректно и не редко приходится при выборе алгоритма учитывать особенности коммуникационной среды вычислительного комплекса, для которого предназначена программа. Так, например, может стоит выбрать менее оптимальный алгоритм по скорости вычисления в последовательном варианте, но который, тем не менее, будет хорошо параллелится и будет оптимальным для решения задачи на кластере.

В не зависимости от того, по каким критериям мы оптимизируем программу, наибольший вклад дает выбор алгоритма. Здесь можно добиться улучшения параметров в сотни и даже в тысячи раз. На втором месте стоит реализация алгоритмов. Здесь можно добиться выигрыша в десятки раз. На третьем месте – «заточка» программы под конкретную архитектуру – улучшение параметров от единиц процентов до нескольких раз. Некоторые программисты ошибочно полагают, что выучив язык ассемблер, они смогут создавать шустрые программы, однако это не так, поскольку основную роль все же играет выбор алгоритма и его реализация.

Поясним на простом примере высказанные ранее утверждения. Для этого рассмотрим различные реализации программы вычисления суммы ряда: [N/i], где квадратные скобки – целочисленое деление, N – положительное целое, переменная i пробегает значения от 1 до N

включительно.

Самый первый вариант, который приходит на ум – это просто в цикле подсчитать эту сумму.

```
for (s=0, i=1; i<=N; i++)
     s += N/i;
```

Все будет хорошо, и, возможно, мы даже не заметим как “быстро” будет исполняться данный фрагмент программы, если значение N не превышает нескольких сотен. А если N равно нескольким миллионам, или даже миллиардам? На Intel PIII -933 вычисление суммы элементов ряда из 40.000.000.000 элементов занимает около 6 часов!

Давайте подумаем и вспомним, что у нас в наличие двухпроцессорная машина и, следовательно, мы можем производить суммирование не последовательно, а параллельно. Перепишем слегка нашу программу: напишем функцию, которая будет производить подсчет переданной ей части последовательности (tt.n1, tt.n2) и помещать результат в свою ячейку результирующего массива (tt.k).

```
void *calc(void* arg){
     unsigned long long i;
     t tt = *((t*)arg);
     for(i=tt.n1; i<tt.n2; i++){
          sum[tt.k] += N/i;
     }
     return NULL;
}
```

В основной программе произведем инициализацию границ последовательности для каждого из потоков нашей программы, создадим два потока (pthread_create) и, дождавшись их завершения (pthread_wait), выведем результат.

```
tt[0].n1 = 1;
tt[0].n2 =N/2;
tt[0].k = 0;
tt[1].n1 = N/2;
tt[1].n2 = N+1;
tt[1].k = 1;

pthread_create(&t0, NULL, calc, (void*)&(tt[0]));
pthread_create(&t1, NULL, calc, (void*)&(tt[1]));

pthread_join(t0, NULL);
pthread_join(t1, NULL);
s =  sum[0]+sum[1];
```

Компилируем, запускаем. Теперь наша программа, если она занимается только подсчетом суммы ряда будет выполняться почти в два раза быстрее. Почти в два раз потому, что помимо

нашей программы в система выполняются и другие процессы, которым требуется процессорное время.

Можно ли добиться еще большего ускорения? “Да” - скажет кто-то, - “Давайте возьмем четырех, а лучше восмипроцессорный сервер и тогда наша программа будет в четыре/восемь раз быстрее”. Американцы, наверное, так бы и поступили. Если какую-то проблему можно решить деньгами, они ее так и решают. Но восемь раз мало. Да и кроме того, кто нам даст такой сервер?

Если расписать нашу последовательность хотя бы для N = 10, то можно заметить, что около половины элементов последовательности (ровно половина, если N четное) равно единицы, какое-то двум и т.д. И если теперь мы сможем определять количество элементов последовательности с равными значениями – границы перехода от одного значения к другому, то задача сильно упростится. А границы можно определить легко:

[N/k] – индекс правой границы

[N/(k+1)] – индекс левой границы

Здесь k – значение ряда. В результате первоначальный вариант программы можно переписать в следующем виде:

```
i = N;
while(i>0){
     //m - количество элементов с одинаковым значением
     m = i - N/(N/i + 1);
     s +=(N/i) * m;
     i -= m;
}
```

Такая программа для N порядка 40 миллиардов уже будет выполняться менее одной секунды.

Эти примеры призваны показать, что в первую очередь стоит выбрать правильный алгоритм, и лишь в последнюю очередь заниматься оптимизацией под архитектуру.

Выбор алгоритма

В разделе с таким названием довольно сложно давать какие-то советы. Заранее нельзя сказать, что этот алгоритм плох, а этот хорош. Все зависит от ситуации, в которой будет использоваться программа, которая его реализует. Да и кроме того, невозможно осветить все известные алгоритмы, а тем более те, которым еще суждено быть придуманными. Единственное, что здесь хочется посоветовать, это не стоит бояться потратить время на изучение работ Кнута в надежде подобрать требуемый алгоритм. В любом случае потраченное время не пропадет. И еще, если вашей программе суждено выполниться один раз и вы готовы подождать, то не стоит тратить время на выискивание более оптимальных алгоритмов и их реализацию. В противном случае вы рискуете потратить на это больше времени. А так у вас останется время для того, чтобы заняться чем-нибудь приятным, например, попить чай, или пообщаться с друзьями.

Реализация алгоритмов

Реализуя тот, или иной алгоритм программист стремится сделать так, чтобы структура программы, отдельных конструкций были максимально похожи на те, которые написаны на

бумаге (формулы, циклы и пр.) и это правильно, поскольку отлаживать такую программу гораздо проще. Оптимизация же приводит к тому, что код становится менее удобочитаемым в плане восприятия алгоритма. Поэтому прежде чем приступить к этому этапу оптимизации следует убедиться в том, что программа работает корректно. Иначе, в случае выявления ошибок, трудно будет сказать являются ли они результатом оптимизации или же ошибки – результат некачественной реализации алгоритма или самого алгоритма. При этом будет не плохо, если после каждого этапа оптимизации будет сохраняться промежуточная копия программы. В случае обнаружения ошибок это позволит легко сделать откат к работающей версии.

Существует правило “10-90”, которое гласит, что 90% времени программа проводит в 10% кода. К таким участкам кода относятся циклы. Как показывает практика, добиться повышения производительности программы можно в основном за счет оптимизации циклов и работы с памятью. В следующих разделах речь пойдет именно об этом.

Оптимизация циклов

Пожалуй, можно выделить четыре момента, которые упускаются при реализации циклов:

1. многократное вычисление одних и тех же данных на каждой итерации цикла, которое можно выполнить один раз за его пределами;
2. обход данных, расположение данных в памяти приводит к многократному переписыванию кэша;
3. наличие условных переходов внутри цикла;
4. “простые операции” внутри цикла, которые приводят к простоям процессора.

Разберемся со всем по порядку.

Рассмотрим участок кода программы вычисления интеграла от некой функции F:

```
for (s = 0.0, i=x1; i<=x2; i+=dx){
     s += F(i)*dx;
}
```

Здесь на каждой итерации цикла выполняется умножение на шаг интегрирования. Если итераций много, то вклад операции умножения может оказаться существенным. Если вспомнить уроки математики за первый класс, то участок кода можно будет переписать в следующем виде:

```
for (s = 0.0, i=x1; i<=x2; i+=dx){
     s += F(i);
}
s *= dx;
```

Такой код будет выполняться несколько быстрее. Стоит отметить, что компиляторы с каждой новой версией становятся все умнее и, вполне возможно, для такого простого цикла сами выполнят эту оптимизацию. Однако в более сложных случаях им требуется помощь программиста.

Рассмотрим еще один пример. Во второй половине 90-х одна компания, занимающаяся

разработкой программного обеспечения объявила конкурс, по результатам которого она намеривалась набрать себе сотрудников. Конкурс состоял в том, что претенденты пытаются наиболее оптимально, а главное правильно решить ряд поставленных перед ними задач. Чем-то напоминает олимпиады по программированию. Среди задач была такая: найти все такие параллелепипеды, в которых сумма кубов ребер равнялась бы кубу главной диагонали; при этом ребра и диагональ должны быть целым числом не более 250. Обозначим ребра параллелепипеда через переменные A, B, C, главную диагональ через D и запишем циклы перебора:

```
for(D=1; D<=250; D++){
     for(A=1; A<=250; A++){
          if(D <= A)
               break;
          for(B=1; B<=250; B++){
               if(D*D*D – A*A*A – B*B*B <=0)
                    break;
               for(C=1; C<=250; C++){
                    if(D*D*D – A*A*A -B*B*B – C*C*C < 0)
                         break;
                    if(D*D*D – A*A*A -B*B*B – C*C*C == 0){
                         my_print(A, B, C, D);
                         break;
                    }
               }
          }
     }
}
```

Уже в написанном коде программ есть элементы оптимизации: мы не перебираем значения ребер, которые приведут к тому, что не будет выполнено условие на длину главной диагонали. Вспоминая предыдущий пример можно предложить заранее подсчитать значение кубов целых чисел от одного до 250, поместить их в массив и вести работу с этим массивом.

```
for(D=1; D<=250; D++) cube[D] = D*D*D;
for(D=1; D<=250; D++){
     for(A=1; A<=250; A++){
          if(D <= A)
               break;
          for(B=1; B<=250; B++){
               tmp = cube[D] – cube[A] – cube[B];
               if(tmp <= 0)
                    break;
               for(C=1; C<=250; C++){
                    if(tmp – cube[C] < 0)
                         break;
                    if(tmp – cube[C] == 0){
                         my_print(A, B, C, D);
                         break;
                    }
               }
          }
```

```
        }
    }
```

Но и это еще не все. Параллелепипед со значениями ребер A, B, C может быть представлен шестью тройкам: ABC, ACB, BAC, BCA, CAB, CBA и достаточно найти только одну чтобы вывести все остальные. Можно математически доказать, что решение поставленной задачи может быть получено из решения следующей системы перестановкой A, B и C:

$$\begin{cases} A^3+B^3+C^3=D^3 \\ A \le B \le C \end{cases}$$

В результате получим следующий код:

```
for(D=1; D<=250; D++) cube[D] = D*D*D;
for(D=1; D<=250; D++){
     for(A=1; A<D; A++){
          for(B=A; B<=250; B++){
               tmp = cube[D] – cube[A] – cube[B];
               if(tmp <= 0)
                    break;
               for(C=B; C<=250; C++){
                    if(tmp – cube[C] < 0)
                         break;
                    if(tmp – cube[C] == 0){
                         my_print(A, B, C, D);
                         break;
                    }
               }
          }
     }
}
```

который работает более чем в два раза быстрее, чем исходный. Итак, ускорение было получено во-первых за счет вынесение за пределы основного цикла многократного вычисления куба диагонали и ребер и во-вторых за счет сокращения пространства перебора (основной вклад).

Для полноты картины оптимизации циклов необходимо привести еще один пример:

```
for(i=0; i<strlen(str);i++){
     a+=str[i];
}
```

Если посмотреть на ассемблерный текст этого фрагмента кода, то можно увидеть, что компилятор решил на каждой итерации вычислять длину строки str перед выполнением сравнения. Зачем? может в таких ситуациях компилятор не смог разобраться с работой с указателями, посчитал, что возможна модификация содержимого памяти и в конечном итоге решил оставить все как есть. В этом случае компилятору надо помочь и слегка переписать код:

```
// вычисление контрольной суммы
str_len = strlen(str);
for(i=0; i<str_len; i++){
     a += str[i];
```

```
}
```

или

```
// вычисление контрольной суммы
ptr = str;
while(*ptr){
     a += *ptr;
     ptr++;
}
```

Здесь стоит отметить, что если компилятор заранее знает границы цикла for, то в некоторых случаях он может этот цикл векторизовать и/или развернуть. Об этом пойдет речь ниже.

Посмотрим на следующие реализации копирования двумерного массива.

C/C++	***Fortran***

`for(i=0; i<N; i++){`
`  for(j=0; j<K; j++)`
`  m[i][j] = c[i][j];`
`}`
	`do I=1, N`
	`  do J=1, K`
	`  m(I,J) = c(I,J)`
	`  end do`
	`end do`

`for(i=0; i<N; i++){`
`  for(j=0; j<K; j++)`
`  m[j][i] = c[j][i];`
`}`
	`do I=1, N`
	`  do J=1, K`
	`  m(J,I) = c(J,I)`
	`  end do`
	`end do`

Для того, чтобы определить какой код более оптимален, необходимо вспомнить как располагаются в памяти массивы, объявленные на Фортране и на C/C++.

```
Fortran: m(1, 1), m(2, 1), m(3, 1), ... m(1, 2), m(2, 2), ...

C/C++  : m(0, 0), m(0, 1), m(0, 2), ... m(1, 0), m(1, 1), ...
```

Быстрее будет исполнятся тот код в котором обращение к памяти происходит последовательно. Поскольку в Фортране последовательно в памяти располагаются столбцы, а в C/C++ строки массивов, то будучи написанным на C/C++ будет работать быстрее первый вариант, а на Фортране второй.

Почему происходит так? кэш. переписывание кэша.

При работе с данными надо стремиться к тому, чтобы все они помещались в кэш. Если же это не возможно, то количество перезаписей кэша должно быть минимальным.

Довольно часто приходится встречаться со следующим кодом:

```
for(i=0; i<N; i++){
     if(i<N/2)
```

```
            // сделать что-то одно
        else
            // сделать что-то иное
    }
```

Не редко подобным образом выделяют первую и последнюю итерацию для обхода граничных условий. В плане оптимизации автор этого кода сделал как минимум одну ошибку: на каждой итерации цикла производится проверка условия i<N/2. В результате компилятор поместит лишние инструкции в тело цикла, а при выполнении программы модуль процессора, ответственный за предсказание ветвлений, неоднократно будет давать ложные предсказания, что в свою очередь будет приводить к отчистке конвейеров и простоям процессора. Выгоднее будет разбить этот цикл на два цикла от 0 до N/2-1 и от N/2 до N.

Посмотрим на другой распространенный пример:

```
    for(i=0; i<N; i++){
        if(a != b)
            // сделать что-то одно
        else
            // сделать что-то иное
    }
```

Здесь происходит тоже самое – сравнение на каждой итерации. Решение (если a и b не изменяются в теле цикла)– вынести оператор сравнения за пределы цикла.

```
    if(a != b)
        for(i=0; i<N; i++){
            // сделать что-то одно
        }
    else
        for(i=0; i<N; i++){
            // сделать что-то иное
        }
```

Вывод: стремиться избавиться от операторов сравнения/ветвления внутри тела цикла, например, одним из предложенных способов.

Развитие современной микропроцессорной техники давно дошло до того, что за один такт могут одновременно выполняться ряд операций сложения, умножения, деления. А раз так, то чтобы задействовать все ALU процессора надо, если это возможно, в цикле выполнять несколько различных операций. Так программу

```
    for(i=0; i<N; i++)
        a[i] = b[i]+c[i];
    for(i=0; i<N; i++)
        d[i] = k[i]*f;
```

рекомендуется переписать в виде

```
    for(i=0; i<N; i++){
```

```
        a[i] = b[i]+c[i];
        d[i] = k[i]*f;
    }
```

Другой вариант задействовать максимально возможное количество ALU – производить развертывание циклов. Результат будет достигнут в случае полной, или частичной независимости итераций исходного цикла. Например, первый цикл в предыдущем примере, при условии, что N кратно 4, можно переписать как

```
    for(i=0; i<N/4; i++){
        a[i] = b[i]+c[i];
        a[i+1] = b[i+1]+c[i+1];
        a[i+2] = b[i+2]+c[i+2];
        a[i+3] = b[i+3]+c[i+3];
    }
```

Автоматическое разворачивание циклов позволяет делать и компилятор. Для этого компиляторам gcc/g++ (GNU C/C++ compiler), icc (Intel C compiler) надо задать опцию -funroll-loops и -unroll соответственно.

Файловый ввод/вывод

Пожалуй основными причинами снижения производительности программ при работе с памятью является чтение/запись данных с диска/ на диск, размещение в памяти компьютера больше данных, чем объем оперативной памяти, что неминуемо приводит к использованию файла подкачки (своппинг).

Одна из стандартных ошибок, которую делают начинающие программисты выглядит так:

```
    char NextChar(void){
        char ch;
        fread(&ch, 1, sizeof(char), fd);
        return ch;
    }
```

Ошибка состоит в том, что на каждый вызов функции в лучшем случае происходит обращение к дисковому кэшу операционной системы, а в худшем к физическому устройству. При этом каждый раз в чтение участвует ядро операционной системы, что в совокупности существенно замедляет работу программы. Разумней будет завести массив, в который считывать блок данных с диска и из него отдавать данные пользователю.

```
    char NextChar1(void){
        static char    buf[256];
        static int     ptr=0;
        static int     tch=0;
            if(ptr==tch){
                ptr = 0;
                tch = fread(buf, sizeof(char), 256, fd);
            }
```

```
            return buf[ptr++];
    }
```

Ниже представлено среднее время считывания файла размером 1 мегабайт.

```
NextChar:  22759 mks
NextChar1:  2690 mks
```

В Unix системах программисту предоставляется возможность отображать файлы на память (mmap(2), munmap(2)). Этот механизм позволяет работать с файлами так, как буд-то они уже находятся в оперативной памяти. Операционная система сама заботится о том, чтобы считать с диска и разместить данные в памяти, к которым пользователь обратился. Используя этот механизм можно получить среднее время “посимвольного считывания” файла размером в 1 мегабайт 2054 мкс. Здесь, как и в предыдущих примерах, не учитывается время на открытие и отображение файла на память.

Аналогичная ситуация будет и при выводе данных в файл или на экран. При этом вывод на экран даже медленнее, чем посимвольная запись данных в файл.

буферизация ВВ

асинхронность

Оптимизация работы с памятью

Из предыдущего раздела стало понятно, что операции чтения/записи на физические носители занимают много больше времени, чем аналогичные операции с оперативной памятью. Следовательно было бы хорошо, если бы все требуемые данные, перед их использованием удалось в ней разместить. А если данных больше, чем объем ОЗУ? Вы просите у операционной системы еще чуть-чуть, она смотрит, что уже вся оперативная память занята, выявляет давно не используемые страницы памяти и записывает их на диск. В результате этих действий вся система(не только ваша программа) на некоторое время замирает. Как можно разрешить проблему нехватки памяти (оперативной памяти) вашей программе? Можно предложить следующие способы:

1. производить сжатие данны;
2. производить перевычисление данных;
3. избегать дублирования данных.

Сжатие данных здесь понимается двояко: как более компактное представление данных с которыми можно работать в сжатом виде и как сжатие только в целях минимизации памяти. Во втором случае предполагается распаковать данные перед их использованием. При сжатие данных и их перевычисление стоит учитывать что будет менее затратно по времени: сжатие и распаковка/перевычисление данных, или сохранение данных на диск и последующее их считывание.

Один студент как-то спросил нет ли какого-нибудь другого алгоритма решения его задачи поскольку тот, который он придумал работает слишком медленно (задача поиска подпоследовательностей в строках). Основная причина почему программа работала медленно

заключалась в том, что все данные не помещались в памяти. После распросов выяснилось, что алфавит состоит из 5-ти символов, а под каждый символ отводится по одному байту. Расточительно? Да, если учесть, что в один байт можно “упихать” два символа, а в int (4 байта) все десять. После небольшого усложнения программы – извлечение символа – студент сказал, что программа стала работать существенно быстрее и время ее работы стало для него приемлемым. Помимо того, что данные стали умещаться в память компьютера сыграло свою роль и то, что больше данных оказалось в кэше процессора.

А что делать, если приведенная в предыдущем примере упаковка данных не возможна? Не стоит отчаиваться. Рассмотрим следующий пример:

```
struct _t1_{
     char v1;
     int v2;
     char v3;
}
```

```
struct _t2_{
     char v1;
     char v3;
     int v2;
}
```

В первом случае структура будет занимать 12 байт, а во втором лишь 8. Если в программе используется массив таких структур, то 4 байта с каждой из них может очень сильно помочь. Почему так происходит? Неужели sizeof(char) + sizeof(int) + sizeof(char) не равно sizeof(char) + sizeof(char) + sizeof(int)? Чему тогда учат в первом классе? Но не стоит отчаиваться. Все дело в том, что компилятор производит выравнивание полей структуры различных типов на границу слова – 4 байта и в результате получаем следующее распределение полей по памяти:

байты	0	1	2	3	4	5	6	7	8	9	10	11
_t1_	v1				v2				v3			
_t2_	v1	v3			v2							

Таким образом в первом случае мы теряем 6 байт, и 2 во втором. Стоит заметить, что такой подход имеет один недостаток, который проявляется в том случае, если потребуется писать данные в файл или читать из него, и порядок данных в файле идет в строгом соответствии с первой структурой. При желание можно самому указать по какой границе производить выравнивание с помощью директивы компилятора #pragma pack.

Предположим, что наша программа работает с какими-то объектами, которые имеют координату в трехмерном пространстве, массу и цвет. Для хранения информации об объекте была создана структура, в которой в соответствии с предыдущим примером были упорядочены поля. Однако мы забыли, что координаты и масса объекта используются при вычислениях гораздо чаще, чем его цвет, который требуется только при отображении. И что нам это дает? А это дает следующее. При попадании объекта в кэш процессора туда попадает и информация о цвете, когда гораздо выгоднее было бы если бы туда попал следующий элемент (масса, координата). Этого можно добиться, например, перейдя от массива структур к структуре массивов. Другой способ – вынести информацию о цвете из структуры в отдельный массив.

```
struct _data {
     int x;
     unsigned m;
     unsigned char color;
} obj[512];
```

```
struct _data {
     int x[512];
     unsigned m[512];
     unsigned char color[512];
} obj;
```

Но давайте задумаемся всегда ли оправдан переход от массива структур к структуре массивов? Оказывается нет. Если потребуется вычислить суммарную массу и, не дай Бог, сумму всех координат, то во втором случае процесс будет идти гораздо медленнее, поскольку обращение к ячейкам памяти в первом случае будет идти в последовательности N, N+4, N+16, N+20..., а во втором N, N+2048, N+4, N+2052... Т.е. промахи кэша во втором случае более вероятны. Конечно современные процессоры могут распознавать некоторые последовательности выборки данных из памяти и соответственно подгружать их в кэш, но в первом случае переписывать кэш придется реже.

```
for(i=0; i<512; i++){
     s1 + = obj[i].x;
     s2 + = obj[i].m;
}
```

```
for(i=0; i<512; i++){
     s1 + = obj.x[i];
     s2 + = obj.m[i];
}
```

Порой, работая с графикой, программисты не задумываются, что использовать тип double не всегда целесообразно. Зачем знать 20-ый знак после запятой, если на экране и третий-то не всегда заметно? Зато изменив тип double на float мы можем не только сэкономить память, но и применяя потоковые инструкции процессора в два раза быстрее производить вычисления. Производя численное моделирование конечно же не стоит переходить на float (возрастает ошибка), но при сохранение результатов расчетов на диск точностью порой можно пренебречь.

За счет чего же еще работая с памятью можно повысить производительность программы? В процессе подготовки этого текста наткнулся на документ, в котором предлагалось перейти от связных списков к структуре, состоящей из двух полей: массива данных и массива с индексами следующего элемента. Работа с таким списком действительно оказывается удобной, а главное быстрой, если не приходится вставлять новые и удалять неиспользуемые элементы, Ниже приведено два варианта программы и время, полученное в результате их выполнения.

```
struct list{
#ifdef CLASSIC
     struct list *next;
     int dta;
#else
     int *next;
     int *dta;
#endif
};
struct list lst;

void main(void){
     int i;
     struct timeval tv1, tv2;
```

```
// init
#ifdef CLASSIC
     lst.next = NULL;
     for(i=0; i< N_ELEM; i++){
          struct list *ptr;
          ptr = (struct list *) malloc(sizeof( struct list));
          ptr->next = lst.next;
          lst.next = ptr;
          ptr->dta = i;
     }
#else
     lst.next = (int*)malloc(sizeof(int)*N_ELEM);
     lst.dta  = (int*)malloc(sizeof(int)*N_ELEM);
     for(i=0; i<N_ELEM-1; i++){
          lst.next[i] = i+1;
          lst.dta[i] = i;
     }
     lst.next[i] = -1; // absent
     lst.dta[i] = i;
#endif

// trace
     gettimeofday(&tv1, NULL);
     {
#ifdef CLASSIC
          struct list *ptr = &lst;
          while(ptr){ ptr = ptr->next;}
#else
          int c = 0;
          while(c != -1){ c = lst.next[c]; }
#endif
     }
     gettimeofday(&tv2, NULL);
     printf("Trace time: %d mks\n",
          (unsigned)((tv2.tv_sec – tv1.tv_sec)*1000000+
                     (tv2.tv_usec – tv1.tv_usec)));
     return;
}
```

Для списка из 1000000 элементов время обработки классического списка составило 65037 мкс, оптимизированного - 23535 мкс. Разница ощутимая.

Аналогично можно представлять и бинарные/тернарные деревья. Для этого на указатели следующего элемента надо отводить две/три ячейки памяти.

Вызов функций и передача параметров

Не смотря на то, что в приведенных ниже примерах присутствуют циклы, мы не будем заниматься их оптимизацией. Здесь хотелось бы обратить внимание на другие аспекты: вызов функций и передачу параметров.

Помогая пользователям с адаптацией и запуском их задач на кластере НГУ столкнулся со следующим кодом:

```
double Creconstruction::Integral(/* какие-то параметры */){
     Cpoint3D KSI;
     // что-то вычисляем
     for(int i = 0; i<512; i++){
          // что-то вычисляем
          KSI = GetKSI(NewBasic, LAMBDA, ALPHA, 0, source);
          // что-то вычисляем
          if(Function(LAMBDA, KSI, circle, &cell)){
               // что-то вычисляем
          }
     }
     // что-то вычисляем

}

bool CReconstruction::Function(double LAMBDA, CPoint3D KSI,
                    int circle, CMatrixElement *cell){
     // что-то делаем
}
```

Здесь стоит обратить внимание на то, что переменная KSI передается в функцию Function по значению и при каждом вызове этой функции вызывается конструктор для класса CPoint3D. Зачем так было сделано не смог объяснить даже автор программы. Подобных мест в программе было не мало и после замены передачи параметра по значению на передачу параметра по ссылке время работы программы уменьшилось почти в два раза.

Вспомним как происходит вызов функций. Перед вызовом функции в стек помещается адрес возврата из функции, ее параметры, локальные переменные. При входе в функцию могут сохраняться еще и некоторые регистры. Все это отнимает какое-то количество тактов процессора. Посмотрим на следующий пример:

```
double length(double x, double y){
     return sqrt(x*x+y*y);
}

// что-то делаем
     for(i=0; i<N; i++){
          l = length(a[i].x, a[i].y);
          // что-то делаем
     }
```

Здесь вызов функции происходит на каждой итерации цикла, что в конечном счете может быть очень расточительно, если учесть что сама функция маленькая. Рекомендуется оценить ускорение которое можно получить, если в приведенном примере функцию length объявить как макрос или как inline функцию. Иногда подставляя в цикл тело функции можно добиться того, что цикл будет векторизован.

Наверное, самая первая программа, которую когда-либо пишет прогаммист вычисляет факториал или печатает числа Фибоначи. Посмотрим с высоты своих знаний на два способа реализации этих алгоритмов.

```
unsigned Factorial(unsigned n){
     if(n == 0)
          return 1;
     return n*Factorial(n-1);
}
```

```
unsigned Factorial(unsigned n){
     unsigned res=1, i;
     for(i=1; i<=n; i++)
          res *= i;
     return res;
}
```

```
unsigned Fibonachi(unsigned n){
     if(n == 0 || n == 1)
          return 1;
     return Fibonachi(n-1)
          + Fibonachi(n-2);
}
```

```
unsigned Fibonachi(unsigned n){
     unsigned i;
     unsigned res[3];
     res[0] = res[1] = 1;
     for(i=2; i<=n; i++)
          res[i % 3] =
               res[(i-1) %3] +
               res[(i-2) %3];
     return res[n % 3];
}
```

Помимо описанного выше недостатка использование рекурсии имеет еще и то ограничение, что размер стека ограничен и реализация глубокой рекурсия часто бывает невозможна. Следовательно необходимо стремиться реализацию с применением рекурсии заменить на цикл, если это возможно и не сильно усложнит программу. Впрочем, если цель – оптимизация, то второе условие можно не рассматривать.

Прочее

Автоматическая оптимизация программ является проблемой для всех разработчиков компиляторов, которая порой решается успешно и не очень. Упрощая себе задачу, разработчики ограничиваются оптимизацией в рамках одной функции (в компиляторе, разрабатываемом Интел, есть возможность межпроцедурной оптимизации). Такая "ограниченность" компиляторов дает человеку преимущество и им необходимо пользоваться помогая компилятору решать эту непростую задачу. Пришлось как-то оптимизировать программу, которая была разработана в рамках темы кандидатской диссертации и уже некоторое время выполнялась на кластере в Москве. Ниже приведен часть кода этой программы.

```
typedef unsigned short US;
US   c12=0x1000,c0=0xe001,c1=0x0002,c2=0x0004,
     c3=0x0008,c4=0x0010,c5=0x0020,c6=0x0040,
     c7=0x0080,c8=0x0100, c9=0x0200,c10=0x0400,
```

```
     c11=0x0800;
//
void propagate(US ***f1,US ***f0){
     int x,y,z;
     for(x=1;x<=nx;x++)
          for(y=1;y<=ny;y++)
               for(z=1;z<=nz;z++)
                    f0[x][y][z]=
                         f1[x  ][y  ][z  ]&c0 |
                         f1[x  ][y-1][z-1]&c1 |
                         f1[x  ][y+1][z-1]&c2 |
                         f1[x-1][y-1][z  ]&c3 |
                         f1[x+1][y-1][z  ]&c4 |
                         f1[x-1][y  ][z-1]&c5 |
                         f1[x-1][y  ][z  ]&c6 |
                         f1[x  ][y+1][z  ]&c7 |
                         f1[x  ][y-1][z  ]&c8 |
                         f1[x+1][y+1][z  ]&c9 |
                         f1[x-1][y+1][z  ]&c10|
                         f1[x+1][y  ][z  ]&c11|
                         f1[x+1][y  ][z-1]&c12;
//
```

Анализ кода показал, что c0, c1, c2 ... являются константами. Компилятор же законно решил, что это переменные, значения которых могут меняться в других функциях, и оставил в цикле обращение к соответствующей области памяти. После замены US c0=0xe0001, ... на const US c0=0xe0001, ... время выполнения функции, которая состояла из четырех приведенных выше циклов, сократилось на тестовой задаче более чем в два раза. Еще около 30 процентов удалось выиграть за счет вынесение из внутреннего цикла вычисление адреса начала строк трехмерного массива f, выравниванием адресов начала массивов, векторизацией приведенного цикла и перегруппировки порядка выполнения операции "или" (удивительно, что последнее компилятор не сделал сам).

Изначально предполагалось, что программа будет выполняться на процессоре Alpha. Этот процессор обладает большим количеством регистров общего назначения и все константы c0, c1и т.д. компилятор располагает в этих регистрах. Так что заметного увеличения производительности способом аналогичным предыдущему, получить не удалось. Анализ же ассемблерного кода показал, что основное время в цикле уходит на приведение типов: int к short и обратно. Стоило задаться вопросом: из каких соображений был выбран именно тип short? Анализ программы показал, что это было сделано скорее всего для экономии памяти. И это при условие, что программа для работы требует 20-30 мегабайт оперативной памяти под данные! После замены unsigned short на int программа стала требовать около 50 мегабайт памяти под данные, но стала выполняться в 2 раза быстрее. Не экономьте память, если это вам даст выигрыш в производительности.

В упомянутой выше программе использовался генератор случайных чисел для определения вероятности происхождения того или иного события и даже на тестовом примере вызов функции rand производился десятки миллионов раз, а то, что вызов функции является очень тяжелой операцией обсуждалось выше. Как поступать со своими функциями понятно, можно

объявлять их как inline. А как подставить в код тело библиотечной функции? Похоже, что никак, а избавиться от накладных расходов на вызов функции хочется. Очевидным решением поставленной задачи является реализация собственной функции rand, тем более, что она является довольно простой. Один из вариантов функции перебирающей числа в псевдослучайной последовательности приведен ниже.

```
r = (r >> 1) | ((r ^ (r<<31))&0x80000000);
```

Период повторения чисел – 2^{31}.

Нормировка 100% вероятности на RAND_MAX и тем самым избавление от rand()%101.

Как правило, большинство программистов стремятся выделять памяти ровно столько, сколько требуется для размещения данных и в коде программы приходится встречать конструкции следующего вида:

```
short arr[110][121];
```

или

```
#define arr(a, b) array[(a)*121+(b)]
char array[110*121];
```

В результате начало строк матриц получаются не выровнены по границе слова что при работе программы приводит к потере в производительности.

```
#ifdef ALIGN
#define L 1024
#else
#definr L 1023
#endif

double src[1024][L];
double dst[1024][L];

int main(void){
     time_t t1, t2;
     int i,j,k;
     // .... инициализация .....
     t1 = clock();
     for(k=0; k<100; k++)
          for(j=1; j< (1024 - 1); j++)
               for(i=1; i<(1023 - 1); i++)
                    dst[j][i] = src[j][i] -
                         .25*(src[j][i-1]+src[j][i+1]+
                              src[j-1][i]+src[j+1][i]);
     t2 = clock();
     printf("%f sec\n",
               (float)(t2-t1)/(float)CLOCKS_PER_SEC);
```

Приведенная программа с выравниванием данных на Intel PIII 933MHz выполняется 7.3 секунд, без выравнивания – 8.5 секунды. В задачах численного моделирования подобные циклы могут выполняться не сотни, а сотни тысяч раз и выравнивая данные, время выполнения программы можно существенно сократить.

Оптимизация по памяти

Задача обратная оптимизации по быстродействию. Сворачивание циклов, перевычисление значений, ...

Оптимизация под архитектуру

Оптимизация под архитектуру заключается в учете особенностей процессора (количество и разрядности регистров, объемы и устройство кэшей, длина конвейера, количество АЛУ и пр.), работы памяти. В этом разделе не ставится задача рассказать обо всех особенностях этого типа оптимизации. Будут затронуты лишь общие положения; для детального рассмотрения следует изучать архитектуру процессоров и, возможно, спуститься на уровень ассемблерного кода. Рассмотрим некоторые опции компиляторов, позволяющие учитывать особенности того или иного процессора.

Первое с чего стоит начать оптимизацию вашей программы под заданную архитектуру – это попросить компилятор выполнить соответствующие действия. Если компилятор “умный”, то он знает какое количество регистров у процессора, объемы кэшей, наличие потоковых инструкций и многое другое и соответствующим способом сможет переупорядочить инструкции в вашей программе, развернуть и векторизовать циклы. Под какой процессор производить оптимизацию задается ключами компилятора, ряд из этих ключей приведено ниже.

```
gcc -mcpu={i586, i686,pentium, pentiumpro, k6, athlon, ...}

icc {-tpp5, -tpp6, -tpp7, ...}
```

Для компилятора gcc (GNU C compiler) значение ключей говорит само за себя. Для компилятора icc (Intel C compiler) соответственно Pentium (- tpp5), Pentium Pro, Intel Pentium II и Intel Pentium III (-tpp6), Pentium 4 (-tpp7). Сразу стоит сказать, что это далеко не все опции оптимизации. Полный их перечень есть в документации по компилятору.

Помимо опций, некоторые компиляторы предоставляют пользователю дополнительные типы данных, библиотечные функции для работы с этими данными и функции управления работой процессора, например, подгрузкой данных в кэш. Как правило работа с этими данными преобразуется к потоковым командам (SSE). К компиляторам, предоставляющим такие возможности относится компилятор от Intel. Ниже приведено ряд примеров оптимизации программ с использованием этого компилятора.

```
// исходный текст
// dest[i] = sqrt(src1[i]^2 + src2[i]^2)
void F(float *dest, float *src1, float *src2, unsigned len){
     unsigned i;
     for(i=0; i<len; i++)
```

```
            dest[i] = sqrt(src1[i]*src1[i] + src2[i]*src2[i]);
}

// вариант с использованием SSE инструкций
// dest[i] = sqrt(src1[i]^2 + src2[i]^2)
void F_SSE(float *dest, float *src1, float *src2,
                unsigned len){
     unsigned i;
     unsigned nLoops = len/4;
     __m128 m1, m2, m3;
     __m128 *arr1, *arr2, *arr3;

     arr1 = (__m128*)dest;
     arr2 = (__m128*)src1;
     arr3 = (__m128*)src2;
     for(i=0; i<nLoops; i++){
          m1 = _mm_mul_ps(*arr1, *arr1);
          m2 = _mm_mul_ps(*arr2, *arr2);
          m3 = _mm_add_ps(m1, m2);
          *arr3 = _mm_sqrt_ps(m3);
          arr1++;
          arr2++;
          arr3++;
     }
}
```

По приведенному коду следует сделать несколько замечаний. Прототипы всех SSE функций и тип данных __m128 определены в xmmintrin.h файле. Тип данных __m128, как не трудно догадаться из названия, занимает 128 бит и может использоваться для одновременного выполнения одних и тех же операций над 4-мя переменными типа float или 2-мя переменными типа double. Также этот тип данных может использоваться для выполнения арифметических и логических операций над целочисленными переменными. Непременными условиями использования функции F_SSE является то, что длинна массивов должна быть кратна четырем и массивы должны быть выровнены на границу 16-ти байт (проверки в примере опущены). Описание используемых функций приведено в таблице.

_mm_mul_ps	умножить 4 компоненты типа float 128-битного типа __m128
_mm_add_ps	сложить 4 компоненты типа float 128-битного типа __m128
_mm_sqrt_ps	выполнить взятие квадратного корня для 4- х компонент типа float 128-битного типа __m128

Скорость выполнения этих двух функций отличается в 3-4 раза.

Псевдооптимизация

Если предыдущие разделы рассчитаны в основном на оптимизацию реализаций численных методов и там получали какой-то прирост в производительности работы программы, то в этом разделе этого на будет. Здесь мы остановимся на создание иллюзии того, что программа работает быстро. В эпоху HyperThreading и Multi-core это не так. Да и вообще не так. На первый взгляд может показаться, что такая оптимизации не имеет смысла, но стоит вспомнить те моменты, когда приходилось ждать, пока Windows прекратить показывать часики, или

сохранится какой-то большой файл прежде чем можно будет продолжить работу и вопрос отпадет сам собой. Таким образом, в этом разделе пойдет речь о том как сделать работу пользователя с программой более комфортной.

Когда сталкиваешься с поведением программ, описанном выше, невольно возникает вопрос: “А нельзя ли производить эти операции параллельно?”. Собственно в этом вопросе и заключается ответ на вопрос как такие иллюзии создаются. Некоторые пользователи утверждают, что операционная система Windows XP по сравнению с Windows 2000 работает быстрее, поскольку быстрее после перезагрузки выдает приглашение на ввод пароля. Здесь разработчики пошли на описанною выше хитрость: они решили, что пользователь вводит пароль медленно и следовательно этот процесс можно совместить с подгрузкой драйверов. О таких хитростях и пойдет речь в этом разделе.

асинхронные операции В/В
F_ASYNC + F_SETSIG
неблокируюшие операции чтения/записи
F_NONBLK, select (poll)
выделение ресурсоемких обработок (графического интерфейса) в отдельный поток

Распараллеливание программ

OpenMP
Posix threads

Инструменты оптимизации программ

В предыдущих разделах было рассказано как можно добиться увеличения производительности программ. Однако встает ряд вопросов: с какой части программы начать ее оптимизацию и какие части программы вносят существенный вклад в падение производительности. Если программа небольшая, то можно наставить кучу вызовов функции замеров времени и вывод результатов замеров. Однако такой подход не совсем корректен, ведь вызов функции взятия времени, вывод результата оказывают влияние на измерения. Кроме того вам придется самому наводить статистику и усреднять полученные значения замеров.

С другой стороны, зачем делать то, что уже сделано для нас и при этом в более удобном виде? Существует множество программ, которые в паре с компиляторами способны выдавать различную статистику по работе программы, как-то: количество вызовов функций, время, затраченное на вызов (среднее, полное, с учетом подвызовов и без них), процентное соотношение между вызовами, и пр. Эти программы называются профилировшиками (profiler). К наиболее известным относятся VTune (Intel), Code Analyst (AMD), gprof (GNU). Стоит отметить, что первые две программы созданы разработчиками процессоров и профилирование с их помощью будет более качественным (больше информации о том, почему программа ведет себя именно так). Кроме того VTune и Code Analyst могут эммулировать различные процессоры и платформы и вам не потребуется иметь под рукой несколько компьютеров, чтобы оптимизировать программу под каждый из них.

Любой акт профилирования программы является экспериментом, и, как эксперимент, он не лишен погрешностей. Может во время профилирования запустилась другая программа, которая потребовала процессорное время, или сама операционная система решила положить что-то в

своп, или считать с диска необходимые данные. Поэтому профилирование следует проводить несколько раз, при этом отбрасывая крайние значения и усредняя оставшиеся.

примеры оптимизации с помощью gprof
Небольшой пример с VTune

Оптимизация параллельных программа

Параллельная программа (ПП) представляет собой систему последовательных взаимодействующих процессов и поэтому все то, что было сказано относительно оптимизации последовательных программ, относится и к ПП. Оптимизировать необходимо также и взаимодействия процессов. Об этом и пойдет речь в этом разделе.
Взаимодействие процессов ПП происходит с помощью передачи сообщений (рассматривается такая-то модель, которая является наиболее распространенной). Во время передачи/приема сообщений не редко не проводится никаких вычислений, что является плохим показателем для вычислительной программы. Конечно, не всегда удается совместить передачу данных с вычислениями и минимизировать простои процессора, но к этому следует стремиться.
И так, относительно оптимизации ПП программ будут затронуты следующие вопросы:

1. Минимизация передаваемых данных
2. Асинхронная передача/прием данных

Прежде чем переходить к примерам стоит сказать, что прирост производительности от применения того, или иного подхода будет различен для систем с различной коммуникационной средой и в каждом конкретном случае может потребоваться поварьировать параметры для получения наилучшего результата.

Пример – сортировка больших массивов.
Пусть два процесса хранят у себя по половине месива, который требуется отсортировать. Рассмотрим несколько вариантов реализации алгоритма сортировки слиянием (???).
Вариант 1.

- Отсортировать свою часть массива
- Передать элемент самый правый/левый элемент массива соседу
- Получить элемент от соседа, вставить его в свой массив выполнить сортировку
- Повторить прием/передачу, в случае если переданный элемент больше/меньше того, который был принят.

Этот алгоритм был предложен Диэкстрой и помимо того, что программа может попадать в состояние дедлока (для предотвращения требуются дополнительные проверки) обладает тем недостатком, что здесь передача осуществляется маленькими порциями и скорость передачи данных существенно ниже максимально возможной.
Если передача больших порций данных происходит с большей скоростью, то почему бы ни передавать большие участки массива.
Вариант 2.

- Отсортировать свою часть массива
- Передать ее соседу
- Один из процессов выполняет сортировку массива, получившегося слиянием двух его половинок.

Недостаток этого подхода в том, что, во-первых, две половинки могут не поместиться в оперативной памяти, что приведет к свопингу; во-вторых, на втором этапе работает только один

процесс (второй простаивает). И третье, возможно, что будут передаваться лишние данные, поскольку с некоторой долей вероятности в одной половине массива есть элементы меньшие (большие), чем минимальный (максимальный) элемент в другой половине массива.
Вариант 3.

- Отсортировать свою часть массива
- Выбрать из массива каждый 10/20/100-ый элемент и передать их соседу
- Сравнить полученные данные от соседа со своими данными и определить ту часть массива, которая нуждается в сортировке
- Передать соответствующую часть массива соседу
- Принять данные и выполнить сортировку

Этот способ на практике лучше предыдущих, но так же обладает недостатками. Так, в наихудшем случае может потребоваться обменяться всем массивом с соседом.
Вариант 4 (модернизированный вариант 1)

- Отсортировать свою часть массива
- Передать соседу 10/20/100 правых (левых) элементов массива
- Принять данные от соседа, вставить их в свою половину массива и произвести сортировку
- Если массив отсортирован, выйти. Иначе, повторить передачу/прием следующих 10/20/100 элементов

Используя последний подход, удается избавиться от всех недостатков предыдущих подходов.

В предыдущих разделах говорилось о том, что можно сэкономить память, если время на сжатие и распаковку данных меньше, чем время их перевычисление или время, которое затрачивается на запись и считывание этих данных с диска. Это можно применить и к передаче данных. Т.е. если время сжатия передачи и распаковки данных меньше времени передачи неупакованных данных, то этим следует воспользоваться. Например, если стартовая последовательность может быть вычислена из ее индекса и некоторых параметров, то выгоднее может быть передать эти параметры, чем всю последовательность.

Минимизировать можно не только размер передачи, но и количество передач. Так, иногда можно вообще не передавать данные по сети. Как? Большинство кластерных систем построены на базе двухпроцессорных модулей (узлов) и распределение процессов одной задачи по узлам происходит таким образом, что два процесса оказываются на одном узле. Если этим двум процессорам требуются одни и те же данные, то достаточно передать их только одному процессу, который через разделяемую память предоставит их для другого. Этот же подход можно использовать для минимизации используемой памяти.

Об асинхронной передаче данных упоминалось в разделе «Псевдооптимизация». Здесь смысл тот же – совмещение приема/передачи данных с другими действиями. В библиотеках для разработки ПП такие функции предусмотрены. В MPI это: MPI_Isend, MPI_Irecv, MPI_Test, и т.д., в PVM - ???? Порядок их использования следующий:

- Инициализировать прием/передачу данных и получить дескриптор
- Выполнить какие-то действия
- По дескриптору проверить статус завершения операции.

Например, такой подход широко используется при обмене границами областей при численном решении дифференциальных уравнении. Асинхронные операции приема/передачи могут использоваться и в приложениях построенных на архитектуре клиент-сервер.

Прочее

отказ от универсальности: k^(2n) -> k<<n
где рассказать об -O, -O1....