Análise e Otimização de Desempenho

Aleardo Manacero Jr.

O que é análise de desempenho

- É a principal técnica para se fornecer informações para o processo de otimização de um sistema/arquitetura

- Deve-se lembrar aqui que desempenho é uma medida subjetiva, que pode ter diferentes interpretações dependendo do objetivo do interessado

Desempenho em sistemas paralelos

- Envolve um conjunto de medidas diferente daquele que se entende por desempenho em sistemas sequenciais

- Em sistemas paralelos as principais medidas são:
  - Ganho de velocidade (*speedup*)
  - Capacidade de crescimento (*scalability*)

Escalabilidade

- Mede o quanto um problema ou sistema pode crescer sem prejuízo significativo de desempenho

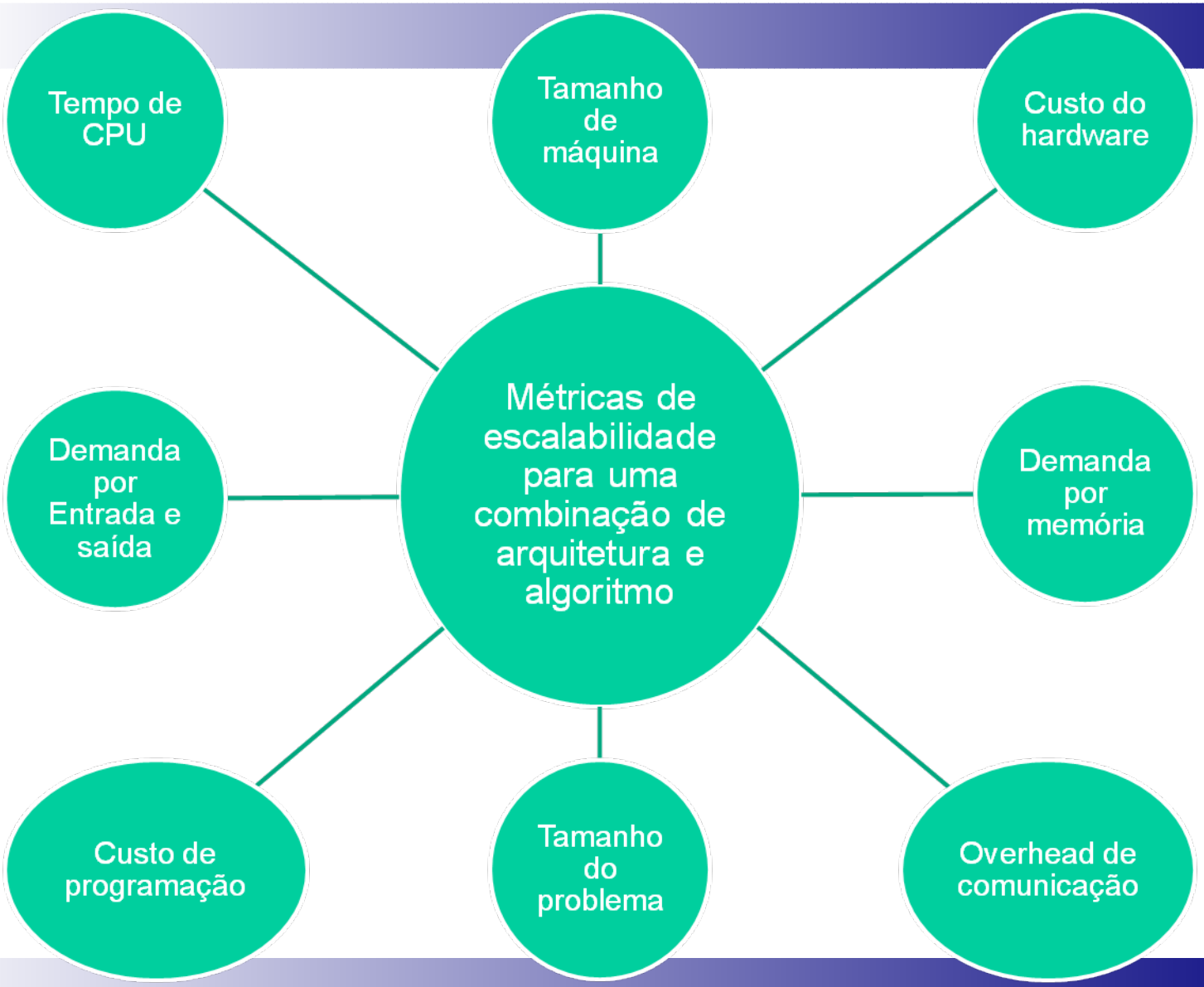
- Depende de características do ambiente (hardware e software), como:
  - Granulação (ou granulosidade)
  - Custo de comunicação
  - Necessidade de comunicação

Escalabilidade

- Por medir a capacidade de crescimento é de especial interesse a quem paga pelo ambiente paralelo

- Não é uma indicação de ganho máximo de velocidade, mas sim de quando esse ganho passa a crescer menos que o necessário (ou desejável)

Métricas de desempenho

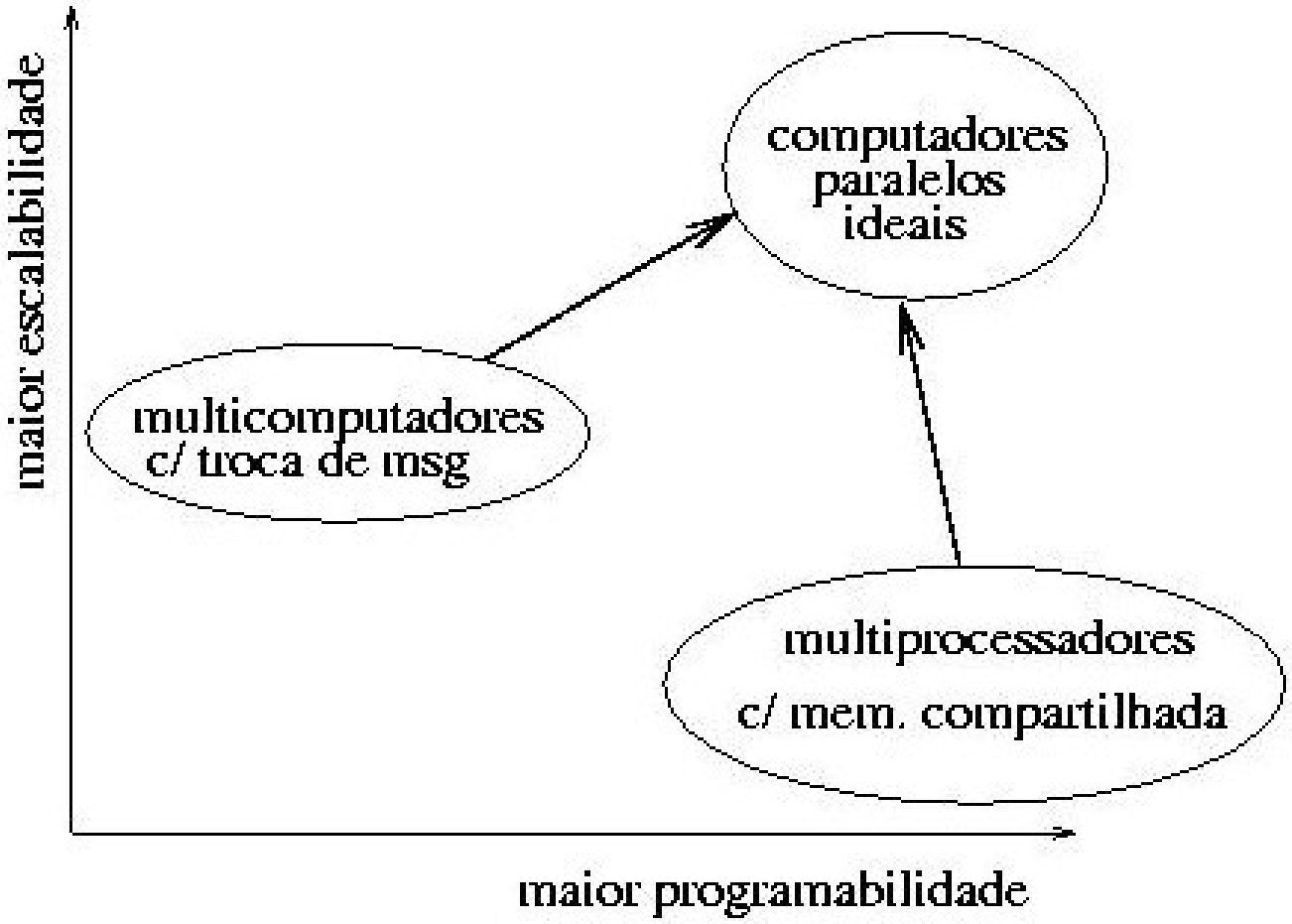


Escalabilidade X programação

- Um resultado importante das métricas apresentadas é que as arquiteturas disponíveis podem ser separadas entre as fáceis de programar e as fáceis de aumentar.

- O ideal é ter sistemas igualmente fáceis de programar e crescer.....

Escalabilidade X programação



Ganho de velocidade (speedup)

- Temos vários tipos de *speedup*:
  - Teórico
    - tamanho fixo
    - tempo fixo
  - Medido
    - pelo sequencial
    - pelo paralelo monoprocessado

Speedups teóricos

- Lei de Amdahl, que trata o problema de tamanho fixo

- Lei de Gustafson, que trata o problema de tempo fixo

Lei de Amdahl

- Define *speedup* como a razão entre os tempos de execução de um programa considerando tanto seus trechos paralelizáveis quanto os estritamente sequenciais

- Atua como um forte limitante ao paralelismo

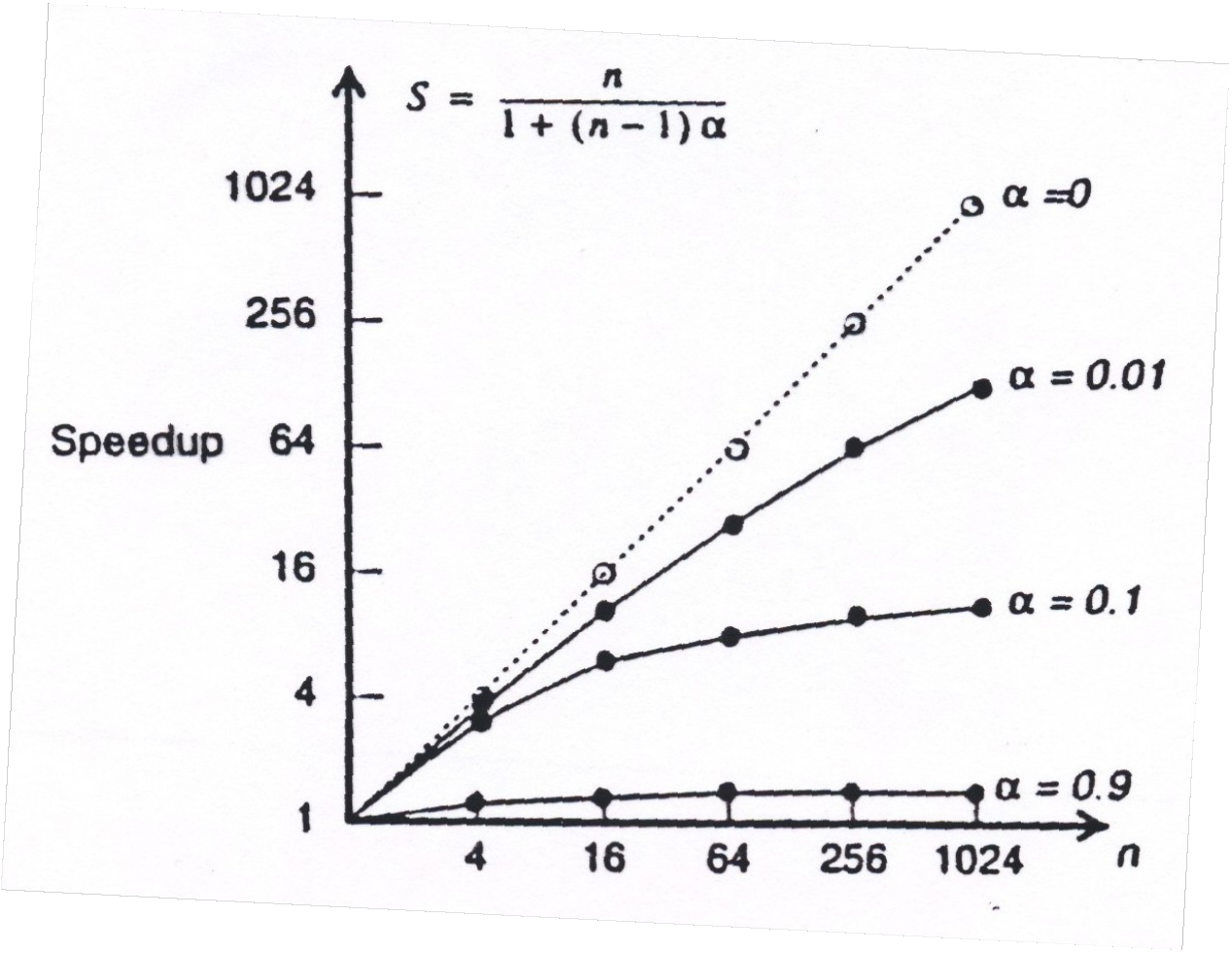
Lei de Amdahl

- Define α como sendo a probabilidade de processamento sequencial

- Então, o *speedup* para N elementos paralelos resulta em:

$$S_n = \frac{n}{1 + (n-1).\alpha}$$

Curvas de speedup

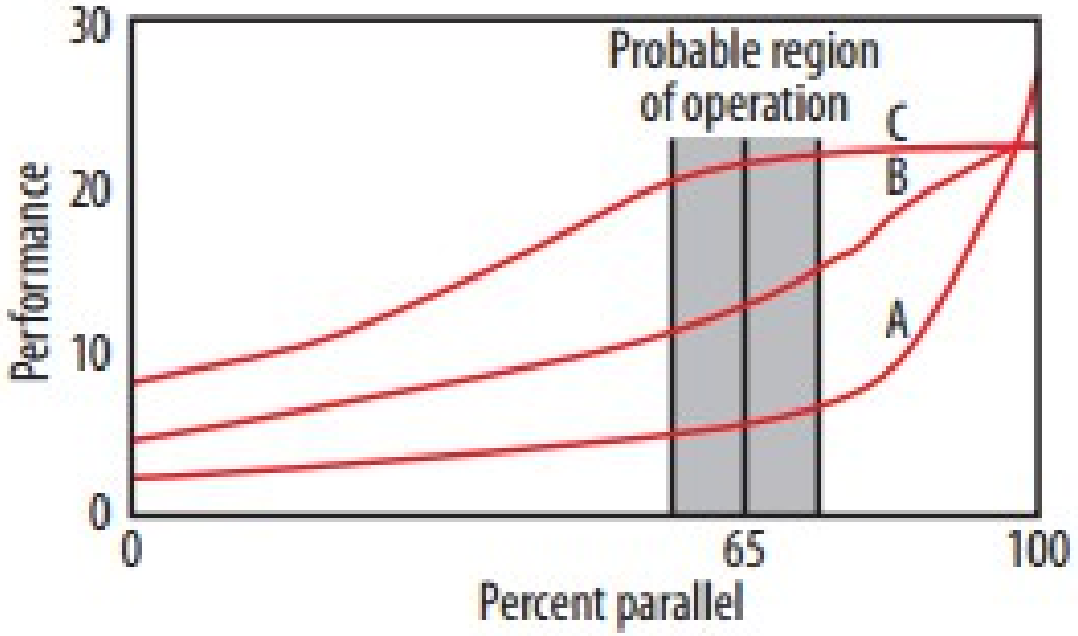


Lei de Amdahl – a verdade

- Amdahl, em 1967, apenas criticou o conceito de arquiteturas paralelas tipo SIMD (mais precisamente o Illiac IV), com apenas um fluxo de instruções
- No artigo original não se propõe nenhuma fórmula ou lei
- O equacionamento do que veio a se chamar Lei de Amdahl veio muito depois

Curvas de speedup (1967)

Curva do artigo original de Amdahl



Lei de Amdahl

- Problema: mesmo para pequenos valores de α o speedup é pequeno

- Exemplo:
  - Método do trapézio para integração:

$$F = \Sigma \left[(x_i - x_{i-1}).(f(x_i) + f(x_{i-1})) \right] / 2$$

Método do trapézio

- Numa execução sequencial, o método necessita:
  - 1 inicialização de somador
  - 2n-1 somas
  - n multiplicações
  - n subtrações
  - 1 divisão

- Numa execução paralela é preciso:

Em paralelo:

  - 1 inicialização de somador
  - 2(n/p) -1 somas
  - n/p multiplicações
  - n/p subtrações

Em série:

  - 1 inicialização de somador
  - p-1 somas
  - 1 divisão

Método do trapézio

- O que resulta, para n=10 pontos de discretização e p = 10 processadores, considerando 1 ciclo por operação, em:
  - 41 ciclos para execução sequencial
  - 15 ciclos para execução paralela
- O ganho “medido” de velocidade é de 41/15=2.73
- O valor de α é dado por 2/(4n+1)=0.04878
- Pela lei de Amdahl o valor máximo seria de 6.95

Resultados

- Já para n=10 e p=50 teríamos:

  Por Amdahl o speedup máximo fica em 14.75

- Conclusão:
  - Como um problema sempre tem uma parte sequencial, o paralelismo fica sempre severamente limitado.
- Erro de análise:
  - O tamanho do problema pode crescer!!

Resultados

- No caso do método do trapézio, se um problema é resolvido sequencialmente com 1000 pontos de discretização, ao resolve-lo em paralelo podemos aumentar a precisão da resposta resolvendo-se K*1000 iterações.
- Assim, ao aumentar-se p para 50, aumenta-se também n para 50. Com isso, o speedup teórico passaria a ser de 33.61 (ainda baixo!)
- Para se ter um melhor limite teórico para o speedup deve-se manter fixo o tempo de execução (speedup de tempo fixo)

Lei de Gustafson

- Corrige a lei de Amdahl ao manter a carga em cada nó fixa, através do aumento da carga total do problema.

$$S = n - \alpha.(n-1)$$

- Nos casos anteriores teríamos:
  - Para p=10 ⇒ s=9.56
  - Para p=50 ⇒ s=47.61

- Que representam limites teóricos muito mais aceitáveis.

Problemas com o speedup

- Os limitantes definidos pelas leis de Amdahl e Gustafson não representam de verdade o speedup efetivo de um sistema.

- Esse speedup é dado pela razão entre os tempos de execução sequencial e paralelo

- O problema é definir o tempo da execução sequencial

Tempo sequencial

- A definição do tempo sequencial segue uma de duas linhas:
  - É o tempo de execução do programa paralelo em uma única máquina

    Normalmente resulta num melhor valor de speedup
  - É o tempo de execução do melhor programa sequencial para o problema

    É mais conservador

Slowdown

- É uma medida de falta de desempenho
- Na prática indica um problema de escalabilidade de um sistema ou aplicação, em que o tempo de execução começa a aumentar a partir de um certo número de elementos de processamento
- Geralmente é resultado do aumento na comunicação entre elementos

Métrica Karp-Flatt

- Busca medir o grau de paralelismo em uma aplicação, relacionando o speedup obtido (S) e o número de elementos paralelos disponíveis (p)
- É dada por:

$$e = \frac{1/S - 1/p}{1 - 1/p}$$

Medindo desempenho

- É feita através de benchmarks
- Problemas em como e o que medir
- Existem dois tipos de benchmarks
  - Desenvolvidos por organizações
  - Desenvolvidos localmente

Problemas com medição

- Antes de medir é preciso definir várias coisas, ou melhor dizendo, responder às seguintes questões:
  - O que medir?
  - Como medir?
  - Por que medir?

Problemas da medição

- A solução das questões sobre o que e como medir partem da resposta sobre porque medir.

- Identificar precisamente o motivo de se estar medindo ajuda a identificação dos outros problemas (*mas não necessariamente sua solução*).

Por que medir?

- Respostas possíveis:
  - Medida do desempenho global
  - Informações para melhorar o desempenho global
  - Informações sobre partes específicas do sistema
  - etc.

O que medir?

- Partindo da escolha anterior, a resposta pode ficar entre:
  - Tempo de execução
  - Tempo de CPU
  - Tempo gasto em certas atividades
  - Ganhos de velocidade em sistemas paralelos
  - Eficiência na utilização de recursos
  - etc., etc., etc.

Como medir?

- Monitoração por hardware
  - Uso de analisadores lógicos e outros equipamentos
- Monitoração por software
  - Uso de interrupções do sistema operacional
- Modificação de código
  - Inserção de código, **fonte, objeto ou executável**, para medição

Monitoração X Modificação

- Monitoração
  - é mais precisa
  - exige conhecimentos sobre hardware e sistema operacional
- Modificação de código
  - é feita com o uso de ferramentas prontas
  - usa técnicas como *profiling* e extração de eventos

Monitoração X Modificação

- Monitoração
  - apesar de sua precisão quase não é utilizada, por ser cara e complexa
- Modificação de código
  - apesar dos vários problemas com as ferramentas baseadas em modificação de código, seu uso é bastante intenso

Profiling / extração de eventos

- Várias ferramentas existentes

  prof, gprof, pixie, tcov, etc.

- Problemas:
  - Precisão da amostragem
  - Não tratam paralelismo

Modificação de objeto/executável


```
main() {
   int l;
   for (l=0; l < 10000; l++) {
      if (l%2 == 0) foo();
      bar();
      baz();
   }
}
```

Modificação de objeto/executável


```
foo(){
  int j;
  for (j=0; j<200000; j++);
}
```

```
bar(){
  int i;
  for (i=0; i<200000; i++);
}
```

Modificação de objeto/executável


```
baz(){
  int k;
  for (k=0; k<300000; k++);
}
```

```
$[1] gcc –pg loops.c -o loops
$[2] ./loops
$[3] gprof > loops.prof
$[4] more loops.prof
```

Modificação de objeto/executável

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
53.89	0.08	0.08	10000	8.08	8.08	baz
33.68	0.13	0.05	10000	5.05	5.05	bar
13.47	0.15	0.02	5000	4.04	4.04	foo

Modificação de objeto/executável


```
index  % time    self  children    called        name
                                                 <spontaneous>
[1]    100.0    0.00    0.15                     main [1]
                0.08    0.00   10000/10000           baz [2]
                0.05    0.00   10000/10000           bar [3]
                0.02    0.00    5000/5000            foo [4]
-----------------------------------------------
                0.08    0.00   10000/10000           main [1]
[2]     53.3    0.08    0.00   10000                 baz [2]
-----------------------------------------------
                0.05    0.00   10000/10000           main [1]
[3]     33.3    0.05    0.00   10000                 bar [3]
-----------------------------------------------
                0.02    0.00    5000/5000            main [1]
[4]     13.3    0.02    0.00    5000                 foo [4]
-----------------------------------------------
```

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Modificação de código fonte

- Aqui o próprio programador insere chamadas de funções para contagem de tempo (instrumentação) dentro de seu código.
- A medição, portanto, depende da existência de acesso ao código fonte do programa.
- Também tem problemas de precisão e amostragem.

Modificação de código fonte


```
#include <time.h>
#include <sys/times.h>
#include “stdio.h”

float etime ()
{struct tms local;
 times (&local);
 return ((float)local.tms_utime/100.0 +
   (float)local.tms_stime/100.0);
}
```

Modificação de código fonte


```
main()
{float Duration;
…...
   Duration = etimes();
   do_whatever_has_to_be_measured();
   Duration = etimes() - Duration;
   printf (“Function took %f seconds\n”,Duration);
}
```

Modificação do executável

- Pode ser feita por instrumentação *off-line*, como fazem *prof* e similares
- Ou por instrumentação dinâmica, como faz o *Paradyn*
- Também pode ser feito por extração de características do programa, que então são simuladas *off-line*

Problemas de benchmarks

- Definição da carga de trabalho
  - Qual será a carga, como ela será aplicada ao sistema, que padrão estatístico ela terá?
- Definição dos padrões de medida
  - O que será medido, qual o arranjo de memórias, processadores, rede de comunicação, compiladores, etc.

Benchmarks por organizações

- Servem como padrões de referência na avaliação de sistemas computacionais

- Normalmente são compostos por vários programas complexos e que usam diferentes características do ambiente, como entrada/saída, operações com inteiros, ponto-flutuante, etc.

Exemplos

- SPEC (www.specbench.org)
  - É um dos principais, apresentando resultados diferenciados por segmentos (cpu, www, etc)
- NAS (www.nas.nasa.gov/publications/npb.html)
- Linpack/Lapack (www.netlib.org/lapack)
  - Historicamente é a origem da padronização de benchmarks, sendo usado no TOP500
- Rodinia (procurar projeto a partir de www.cs.virginia.edu/research)

Benchmarks locais

- São especialmente desenvolvidos para uma dada aplicação
- Os programas/casos de teste são escolhidos localmente, de acordo com os objetivos da organização
- A validade dos resultados também é local, não podendo ser generalizada para ambientes similares

Fazendo um benchmark

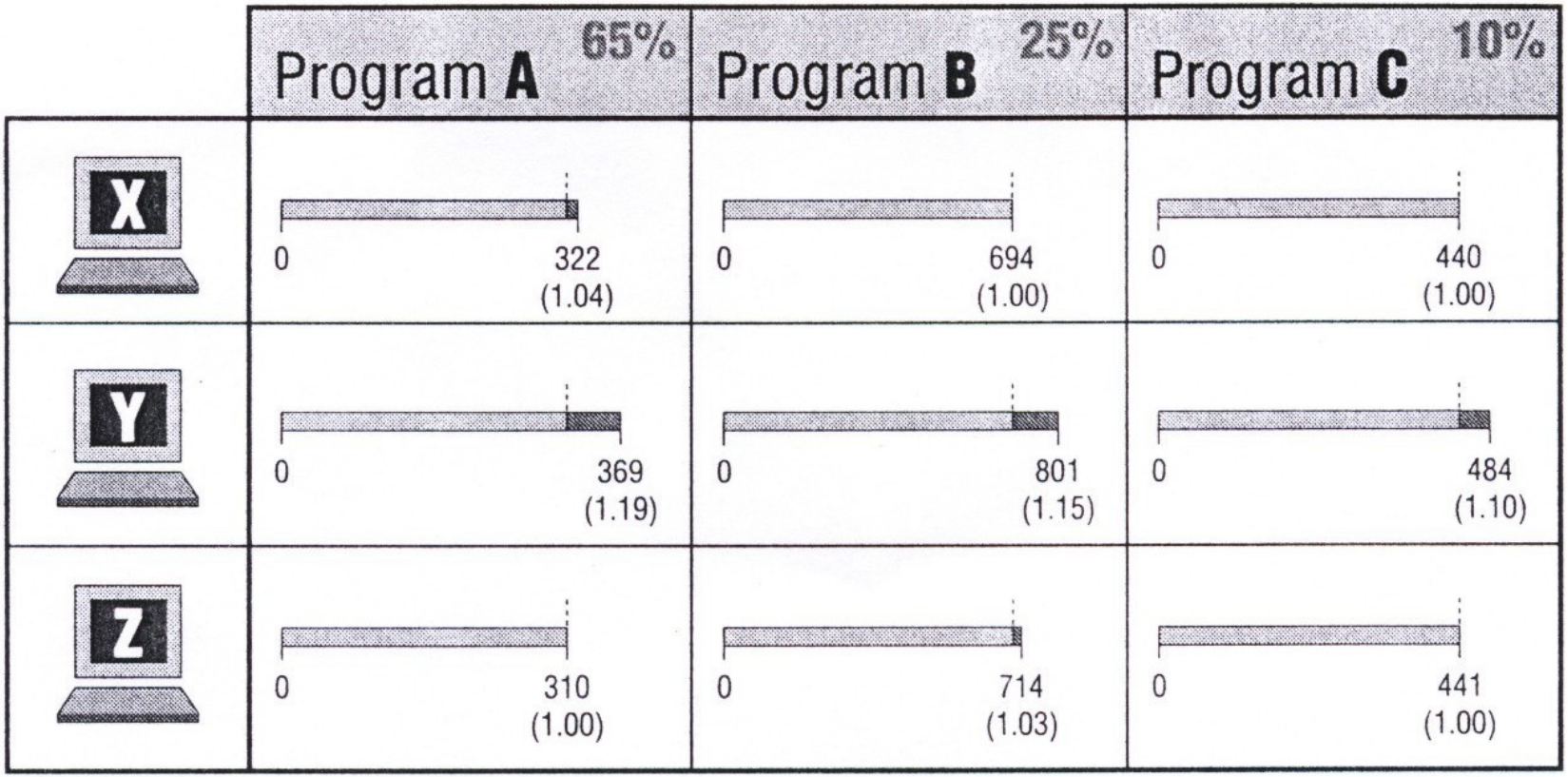
- Em qualquer benchmark deve-se medir os tempos de execução dos vários programas de teste

- Os resultados devem ser normalizados pelo volume de uso de cada programa testado

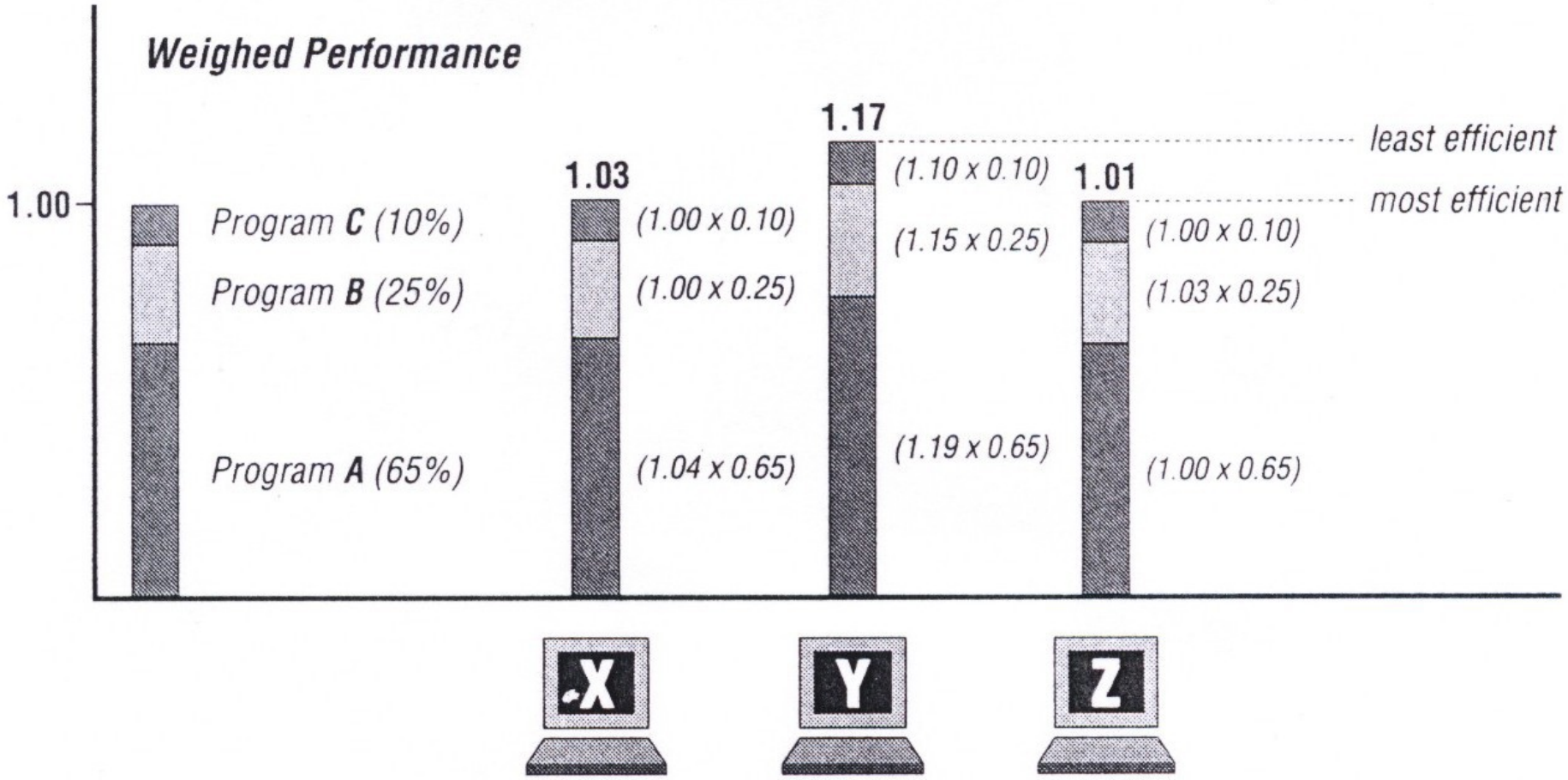
Exemplo

	Sistema		
Programa	X	Y	Z
A	322	369	310
B	694	801	714
C	440	484	441

Utilização dos programas



Desempenho normalizado



Prevendo o desempenho

- Além de medir efetivamente o desempenho de um sistema, através de benchmarking, é possível obter indicações sobre seu desempenho sem a necessidade de medição física

- Essa previsão de desempenho pode ser feita por modelos analíticos ou de simulação

Modelos analíticos

- Se caracterizam por ter uma formulação analítica (algébrica)

- Algumas vezes são resolvidos por simulação, apesar da formulação analítica

- Envolvem modelos como cadeias de markov, teoria de filas, redes de Petri

Modelos de simulação

- Trabalham a partir de características funcionais do sistema (descrições de funcionamento, interações entre partes, etc.) e não de equações analíticas

- Simuladores são construídos a partir de redes de Petri, Monte Carlo, caminhos em grafos, etc.

Medindo X Prevendo

- A diferença entre medir e prever está na segurança sobre os resultados e na possibilidade de se “medir” sem ter o sistema

- A escolha por um método envolve várias considerações, como pressa, disponibilidade, custo, exatidão, etc.

Critérios de escolha

Critério	Analíticos	Simulação	Benchmarking
Estágio de de-senvolvimento	Qualquer	Qualquer	Pós-protótipo
Tempo na obtenção de resultados	Pequeno	Médio	Variado
Ferramentas para análise	Humanos	Linguagens	instrumentação

Critérios de escolha

Critério	Analíticos	Simulação	Benchmarking
Exatidão	Baixa	Moderada	Variada
Avaliação de alternativas	Fácil	Moderada	Difícil
Custo	Baixo	Médio	Alto
Credibilidade do método	Baixo	Médio	Alto

Ferramentas de benchmarking

Software

- Etnus Total View
- Pablo
- Paradyn
- Tau
- Vampir

Hardware

- Linpack
- Lapack
- SPEC
- VTune

Ferramentas de simulação

Software

- PAWS
- PDL
- Simics
- SimpleScalar

Hardware

- PDL
- Asim
- SimpleScalar
- PAWS

Considerações finais

- Medir ou prever o desempenho de um sistema é uma atividade complexa pois envolve muitas métricas e diferentes objetivos

- Não existem soluções definitivas para nenhuma métrica nem sistema

- Cada tipo de técnica ou ferramenta tem um escopo de aplicação bem definido

Otimização de Código

O que é otimização de código?

- É a operação de identificar pontos de um programa que podem ser melhorados e realizar tais melhorias.

Quem faz a otimização?

- O programador

ou

- O compilador

Otimização pelo compilador

- Depende da existência de código visível, isto é, código que possa ser identificado como passível de otimização.
- Usa técnicas clássicas de otimização de código, examinadas no estudo de compiladores.

Otimização pelo programador

- Necessita de informações sobre o código (do ponto de vista de quais partes estão sendo executadas e em que volume)
- Usa, também, uma grande quantidade de técnicas vindas de compiladores.
- Além disso usa técnicas de estilo de programação voltadas para desempenho.

Otimizando o código

- Com as medidas em mãos passa-se a fase de otimização.

- Para essa fase podem ser usadas algumas ferramentas automáticas e, principalmente, alterações de estilo de programação.

Gargalos de software

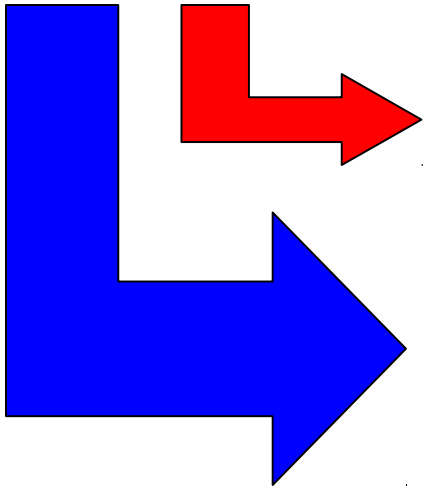
- Os pontos críticos para a melhoria de desempenho são chamados gargalos.
- Podem ser de três tipos:
  - Linguagem (ou estruturas de linguagem utilizadas)
  - Funções
  - Blocos de repetição

Gargalos de linguagem

- A escolha da linguagem de programação para sistemas de alto desempenho deve levar em consideração:
  - Favorecimento de estruturas simples e estáticas
  - Não uso de ponteiros e estruturas dinâmicas

Gargalos de linguagem

- Um exemplo:
- Montagem de um grafo de +/- 70 mil vértices



- 2 horas com ponteiros
- Menos que 30 segundos com vetores

Gargalos de linguagem

- Características de simplicidade de estruturas e do não uso de ponteiros leva ao uso do fortran 77, ou nos dias de hoje, do High Performance fortran (HPF).

- Com isso a otimização por compiladores fica mais fácil e simples.

Gargalos em funções

- Programação estruturada é elegante e deve ser sempre buscada.

entretanto.....

- Sobrecarrega o sistema com excessos de modularização e estruturação do código fonte

Gargalos em funções

- Para obter desempenho deve-se evitar o uso de funções curtas (antigamente economizava-se memória com elas).

- Deve-se usar alinhamento de funções (*function inlining*) no lugar de suas chamadas.

Gargalos em funções

```
#define average(x,y) ((x+y)/2)
main()
{
 float q=100, p=50;
 float a;
 a = average (p,q);
 printf("%f\n",a);
}
```

- Substitui-se

```
a = average (p,q);
```

- Pelo uso da macro definida

```
a = ((p+q)/2);
```

Gargalos em blocos de repetição

- São a maior fonte para otimizações, envolvendo:
  - Desdobramento de código
  - Remoção de testes desnecessários
  - Inversão de aninhamentos
  - Fissão ou fusão de laços

Gargalos em blocos de repetição

- Desdobramentos

```
DO I=1,N
     A(I) = A(I)+ B(I)* C
ENDDO
```

- Para situações em que existem k processadores (ou vias de execução) e em que N é múltiplo de k é possível modificar o código acima para aproveitar um maior paralelismo.
- Isso é feito a seguir para k=4

Desdobramento de ciclos

- Desdobramentos

```
DO I=1,N
     A(I) = A(I)+ B(I)* C
ENDDO
```

```
DO I=1,N,4
     A(I)   = A(I)+ B(I)* C
     A(I+1) = A(I+1)+ B(I+1)* C
     A(I+2) = A(I+2)+ B(I+2)* C
     A(I+3) = A(I+3)+ B(I+3)* C
ENDDO
```

Desdobramento de ciclos

- Desdobramentos

```
DO I=1,N
     A(I) = A(I)+ B(I)* C
ENDDO
```

- E quando N não é múltiplo do número k?

- Adiciona-se um ciclo inicial para acerto de módulo, como é feito a seguir:

Desdobramento de ciclos


```
II = IMOD (N/4)
DO I=1,II
     A(I) = A(I)+ B(I)* C
ENDDO

DO I=1+II,N,4
     A(I)   = A(I)  + B(I)  * C
     A(I+1) = A(I+1)+ B(I+1)* C
     A(I+2) = A(I+2)+ B(I+2)* C
     A(I+3) = A(I+3)+ B(I+3)* C
ENDDO
```

Desdobramento de ciclos

- O uso de desdobramentos pode ser útil também na ausência de múltiplos processadores.

- Para tanto é preciso um número fixo (e normalmente pequeno) de iterações.

Desdobramento de ciclos


```
PARAMETER (NITER=3)
DO I=1, NITER
   A(I) = B(I) * C
ENDDO
```

```
PARAMETER (NITER=3)
  A(1) = B(1) * C
  A(2) = B(2) * C
  A(3) = B(3) * C
```

Teste desnecessário

- Muitas vezes fazemos testes que, a menos por precisão, poderiam ser evitados

```
PARAMETER (SMALL = 1.E-20)
DO I=1,N
   IF (ABS(A(I)) .GE. SMALL) THEN
      B(I) = B(I) + A(I) * C
   ENDIF
ENDDO
```

Teste com invariante

- Muitas vezes um teste interno ao ciclo é feito sobre variável que não tem seu valor alterado internamente ao ciclo.
- Nesses casos é possível retirar o teste, desdobrando o ciclo em duas partes.

Teste com invariante


```
DO I = 1, K
 IF (N .EQ. 0) THEN
   A(I) = A(I) + B(I)*C
  ELSE
   A(I) = 0
 ENDIF
ENDDO
```

```
IF (N .EQ. 0) THEN
  DO I=1,K
   A(I) = A(I) + B(I)*C
  ENDDO
 ELSE
  DO I=1,K
   A(I) = 0
  ENDDO
ENDIF
```

Teste com desdobramento de ciclos

- Outras vezes o teste tem um resultado previsível a partir dos índices do ciclo.
- Nesses casos também é possível eliminar o teste dividindo o ciclo em partes previsíveis.

Teste com desdobramento de ciclos


```
DO I = 1,N
 DO J = 1,N
  IF (J .LT. I)
    A(J,I) = A(J,I)
  +B(J,I)*C
   ELSE
    A(J,I) = 0.0
  ENDIF
 ENDDO
ENDDO
```

```
DO I = 1,N
 DO J = 1,I-1
   A(J,I) = A(J,I)+B(J,I)*C
 ENDDO
 DO J = I,N
   A(J,I) = 0.0
 ENDDO
ENDDO
```

Teste sem desdobramento

- Testes que não podem ser eliminados por desdobramentos.

```
DO I = 1, N
   DO J = 1, N
    IF (B(J,I) .GT. 1.0) A(J,I) = A(J,I) + B(J,I)*C
   ENDDO
ENDDO
```

- Ainda podem ser paralelizados se o hardware permitir

Inversão de aninhamentos

- Ciclos aninhados usualmente indicam acesso a matrizes.
- Matrizes têm seu armazenamento normalizado pela linguagem escolhida.
- Exemplos:
  - Fortran armazena por colunas
  - C armazena por linhas

Inversão de aninhamentos

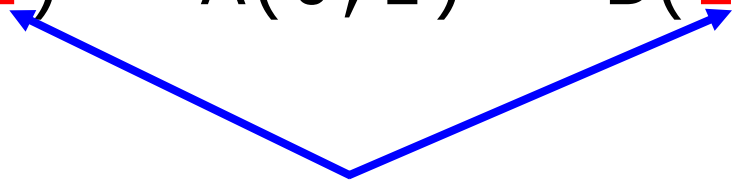

```
DO J = 1,N
 DO I = 1,N
  A(J,I) = A(J,I) + B(J,I)*C
 ENDDO
ENDDO
```

```
DO I = 1,N
 DO J = 1,N
  A(J,I) = A(J,I) + B(J,I)*C
 ENDDO
ENDDO
```

Situações não recuperáveis

- Existem situações em que é impraticável inverter a ordem dos ciclos para obter o casamento correto com a memória.
- Exemplo:

```
DO I = 1,N
   A(J,I) = A(J,I) + B(I,J)*C
```



Multiplicação de matrizes


```
DO I = 1,N
 DO J = 1,N
  SUM = 0
  DO K = 1,N
    SUM = SUM + A(I,K) * B(K,J)
  ENDDO
  C(I,J) = SUM
 ENDDO
ENDDO
```

- Problema com o acesso a A(I,K)

Multiplicação de matrizes


```
DO I = 1,N
 DO J = 1,N
   C(I,J) = 0.0
 ENDDO
ENDDO
DO K = 1,N
  DO J = 1,N
    SCALE = B(K,J)
    DO I = 1,N
      C(I,J) = C(I,J) + A(I,K) * SCALE
    ENDDO
  ENDDO
ENDDO
```

Fusão de laços

- Fusão de laços é possível quando dois laços consecutivos operam sobre os mesmos indexadores de controle
- Representam uma redução no volume de controle de fluxo necessário

Fissão de laços

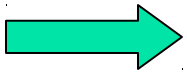
- Fissão de laços é possível quando um laço pode ser dividido em dois grupos independentes de comandos
- Representam uma redução no volume de trocas de conteúdo em *caches*

Otimizações pelo compilador

- Faz basicamente todas as otimizações já examinadas.
- Outras otimizações envolvem:
  - Propagação de código
  - Renomeação de variáveis
  - Remoção de código morto
  - Desdobramento de constantes

Propagação de código


```
x = y
z = 1.0 + x
```



```
x = y
z = 1.0 + y
```

Renomeação de variáveis


```
x = y * z;
q = r + x + x;
x = a + b;
```

```
x0 = y;
q = r + x0 + x0;
x = a + b;
```

Desdobramento de constantes

- Uso de constantes renomeando posições do código.

```
PROGRAM MAIN
INTEGER I, K
PARAMETER (I=100)
K=200
J=I+K
END
```

(J também é constante!)

Redução de força

- Certas operações custam mais que outras, como por exemplo:

```
Y = X ** 2      CONTRA    Y = X * X

J = K * 3       CONTRA    J = K + K + K
```

O trabalho do compilador

- Aplica as técnicas examinadas até agora
- Necessita do apoio do programador para facilitar/orientar seu trabalho
- Não faz milagres, portanto precisa de um código fonte claro e que evite uso de recursos complexos da linguagem

Considerações finais

- Otimização depende fortemente do trabalho do programador
- Estruturas simples resultam em melhor desempenho
- Opções erradas podem resultar em enormes diferenças de desempenho