Capítulo 6

Técnicas para análise e otimização de programas

Aleardo Manacero (UNESP)

Abstract

Parallel programs are solutions to enable results production for high computing costs problems. Even though the simple parallelization of such programs produce better execution times, it does not assure achieving the best possible efficiency. In order to achieve such efficiency one has to apply optimization techniques all over the original program. However, when you have a very large code to optimize, it is virtually impossible to perform optimizations uniformly over the whole code. This demands the application of a preliminary step, which is performance analysis of the program. This will identify parts of the code that are more relevant for optimizations than others. In this chapter we will study some of the definitions about performance analysis, the major techniques used and some of the available tools to perform it. english

Resumo

Programas paralelos são soluções para viabilizar a produção de resultados em problemas de alto custo computacional. Embora a simples paralelização desses programas já melhore os tempos necessários para execução, ela não garante a efetiva obtenção da maior eficiência possível. Para obter tal eficiência é preciso aplicar técnicas de otimização sobre todo o programa original. Entretanto, quando se tem um código bastante grande para otimizar é virtualmente impossível aplicar otimizações de modo uniforme sobre todo o código. Isto demanda a aplicação de uma etapa preliminar, que é a análise de desempenho do programa. Esta análise identificará partes do código que são mais relevantes para otimização do que outras. Neste capítulo serão estudadas as definições sobre análise de desempenho, as principais técnicas utilizadas e algumas das ferramentas disponíveis para realizar a análise.

6.1. Objetivos de análise de desempenho

O processo de análise de desempenho é necessário para que se tenha informações adequadas para otimizar um programa. A otimização, que basicamente envolve a aplicação de determinadas técnicas de redução de esforço computacional sobre um programa, é uma ação necessária em qualquer sistema (basta imaginar aplicativos lentos em seu smartphone). Para sistemas de alto desempenho a otimização de uma dada aplicação é ainda mais necessária, pois nesse caso ineficiências implicam em desperdício de ciclos de máquina, ou seja, desperdício de dinheiro [1, 2].

Entretanto, otimizar um dado programa pode se tornar uma tarefa bastante complicada caso o código seja muito longo. Para esses casos a análise de desempenho se torna vital, pois é por meio dessa análise que teremos informações sobre que partes do programa consomem mais tempo de CPU, por exemplo.

A análise de desempenho é, na prática, uma atividade cíclica, em que um determinado programa é medido, analisado e otimizado. Esse ciclo pode ser visto na Figura 6.1, sendo que basicamente o processo consiste em medir a execução do programa, fazer seu perfilamento[1], aplicar possíveis otimizações e verificar se os resultados obtidos continuam formalmente válidos, antes de um novo ciclo.

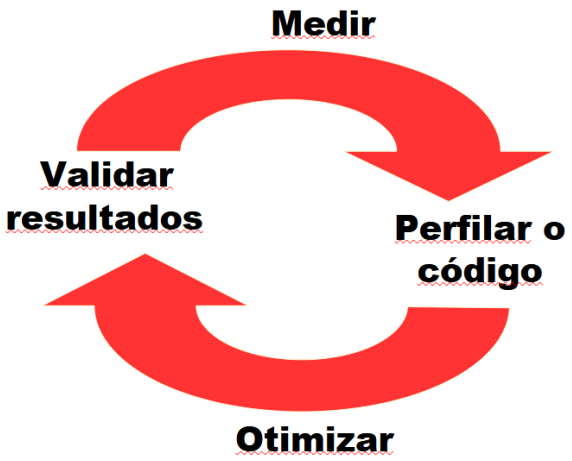

Figura 6.1. Ciclo de atividades no processo de otimização e análise de desempenho.

Mesmo considerando o ciclo indicado, que parece ser simples, ainda existem questões importantes a serem definidas. A principal delas é definir o que se quer de fato, isto é, qual o motivo de otimizar um programa. Mais acima indicamos que, por exemplo, podemos querer reduzir o tempo de CPU consumido. Entretanto, existem objetivos alternativos, como a redução no espaço ocupado em memória, ou ainda a redução no tempo gasto com a troca de dados pela rede. Logo, decidir o que medir e otimizar passa a ser função de quem faz a análise, embora exista uma forte tendência em considerar que o principal parâmetro é tempo [3].

[1]Perfilamento, como veremos mais adiante, é o ato de identificar quais partes do programa estão sendo executadas e em qual proporção isso ocorre

6.1.1. Formas de "medir" desempenho

Outro aspecto importante na definição de medição e otimização de desempenho envolve a definição de quais métodos serão aplicados na obtenção dos perfis de execução. Raj Jain [4] nos ensina que esses métodos podem ser classificados em três grupos: analíticos, baseados em simulação e medição efetiva (que chamaremos de *benchmarking* a partir daqui).

Desses métodos, apenas *benchmarking* é efetivamente uma técnica de medição, enquanto os métodos analíticos e de simulação, embora importantes, são técnicas para predição de desempenho. Na seção 6.4 detalharemos mais os conceitos envolvidos com os três métodos para produzir dados para a análise de desempenho.

6.1.2. Desempenho em sistemas paralelos

Para sistemas paralelos a definição do que devemos otimizar e como obteremos dados para essa otimização ficam ainda mais complicados. Nesses sistemas devemos considerar aspectos não tratados em sistemas sequenciais, como a competição por recursos, atrasos por comunicação, necessidade de sincronismo entre processos paralelos, etc.

Além disso, o fato de paralelizar a execução implica em otimizar o nível de paralelismo, isto é, quantos processos devemos executar em paralelo. A obtenção dessa medida implica em definir novas métricas, além de tempo de execução por exemplo. Na prática essas métricas envolvem o quanto uma execução será acelerada (*speedup*) e o ponto em que essa aceleração é de fato vantajosa (escalabilidade). Analisamos melhor esses aspectos na próxima seção.

6.2. Métricas para análise de desempenho: *speedup*, escalabilidade, eficiência

O aproveitamento de paralelismo introduz dificuldades adicionas, como a existência de atrasos criados pela necessidade de sincronismo, ou a impossibilidade de paralelizar determinadas atividades da aplicação. Esses fatores de degradação de desempenho acabam por definir, que dentro de ambientes paralelos, as métricas relevantes são *speedup* e escalabilidade.

6.2.1. Escalabilidade

A escalabilidade é, na prática, uma medida de até que ponto o paralelismo é vantajoso. Isso significa determinar o quanto um problema ou sistema pode crescer sem prejuízo significativo de desempenho. Sendo assim, a escalabilidade não é uma métrica que indique um valor máximo para algum parâmetro, mas sim de um valor a partir do qual fazer um sistema crescer passa a ser desvantajoso.

É importante destacar que escalabilidade não é uma medida única, ou seja, existem diferentes parâmetros que podem afetar a escalabilidade de um sistema. Entre esses parâmetros podemos destacar:

1. Custo de comunicação, em que a paralelização (aumento no número de processadores paralelos) fica limitada pelo que custarão roteadores, *switches*, cabos e placas de rede;

2. Necessidade de comunicação, em que o crescimento no número de processos paralelos pode ser limitado pelo volume adicional de comunicação;

3. Granulosidade do programa, em que o tamanho dos grãos (trechos de código executados em uma máquina) determina a necessidade de comunicação e uma eventual ociosidade na espera por novos dados;

4. Tamanho do problema, quando se identifica o quanto um problema pode crescer a partir dos recursos de hardware (discos, memória, etc.) disponíveis;

5. Custo de programação, uma vez que o nível e forma de paralelismo resulta em formas mais ou menos fáceis de se programar, podendo aumentar o custo no desenvolvimento de uma aplicação.

Outros parâmetros que impactam na escalabilidade incluem demanda por memória, demanda por entrada/saída, tamanho físico do sistema, etc.

Slowdown Deve ter ficado claro que a escalabilidade é uma medida de limitação ao paralelismo e desempenho. Um sistema pouco escalável implica em se ter um sistema que não pode crescer ou aumentar seu desempenho. A falta de escalabilidade de um sistema paralelo resulta, em geral, no que se chama de *slowdown* ou *parallel slowdown*. Um sistema entra em *slowdown* quando, a partir de um certo grau de paralelismo, o sistema passa a executar mais lentamente, perdendo desempenho apesar do crescimento no número de processos paralelos. Vejam que isso é ainda pior do que determinar o ponto no qual o crescimento deixa de ser vantajoso.

6.2.2. *Speedup*

Essa métrica indica o grau de aceleração obtido com a paralelização de um programa. Essa aceleração pode ser entendida como a razão entre o tempo necessário para executar o programa de forma sequencial e o tempo consumido em sua execução paralela, como visto na equação 1.

$$S = \frac{T_{sequencial}}{T_{paralelo}} \tag{1}$$

Embora essa definição pareça simples, a prática nos mostrou que existem formas diferentes de definir ou medir essa aceleração. Temos então, inicialmente, duas formas de *speedup*: teórico e medido. O *speedup* teórico é dado por relações matemáticas cujos parâmetros envolvem qual porcentagem de um código pode ser paralelizado e o número de processadores paralelos que serão usados. Já o *speedup* medido é obtido por meio de uma comparação direta entre os tempos medidos para execuções sequenciais e paralelas.

6.2.2.1. *Speedup* teórico

Speedups teóricos servem, basicamente, para predizer qual seria o grau ótimo de paralelismo para uma dada aplicação. Na prática não são uma medida real, mas sim uma

indicação sobre a vantagem ou não em se paralelizar. A primeira definição de *speedup* veio de um trabalho de Gene Amdahl, que buscava desestimular o uso de máquinas SIMD ao final da década de 60, mais precisamente o Illiac IV [5]. Os resultados apresentados nesse trabalho foram posteriormente formalizados e implicaram no que se conhece como Lei de Amdahl, dada pela equação 2.

$$S = \frac{n}{1 + (n-1) \times \alpha} \tag{2}$$

Em que n é o número de máquinas paralelas e α representa a probabilidade de processamento sequencial no programa. O problema dessa expressão é que sua formulação resulta num forte desestímulo ao paralelismo, uma vez que qualquer aplicação sempre terá alguma parte que deverá ser executada de forma estritamente sequencial. A figura 6.2 mostra essa característica de forma bastante clara. Observa-se, por exemplo, que com 128 máquinas temos um *speedup* de apenas 1,98 se $\alpha = 0,5$, ou de 56,4 com um $\alpha = 0,01$ (ou apenas 1% de processamento sequencial).

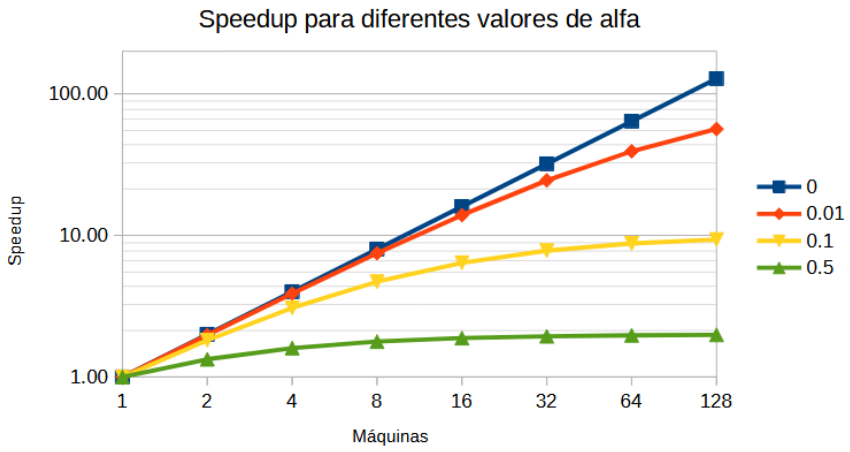

Figura 6.2. Curvas de *speedup*, segundo a Lei de Amdahl, para diferentes valores de α.

O problema com a Lei de Amdahl é que ela desconsidera um fator bastante relevante quando se usa um equipamento mais potente para resolver um problema, que é a ganância. Na prática, se recebemos uma máquina mais potente não a usaremos para resolver os mesmos problemas de antes. A tendência é que se passe a resolver problemas maiores, tanto aumentando a precisão dos resultados como trabalhando com maior quantidade de dados. Para Amdahl isso não foi considerado, levando a que seu equacionamento para *speedup* ficasse conhecido como sendo de tamanho fixo.

Gustafson, como muitos outros, percebeu que os valores de *speedup* dados pela Lei de Amdahl não correspondiam aos valores efetivamente medidos. Com isso acabou propondo o que se conhece como *speedup* de tempo fixo [6]. A ideia basica de Gustafson é considerar que ao paralelizar um programa podemos consumir, em paralelo, o mesmo tempo consumido em sua versão sequencial. Com isso a sua porção sequencial é significantemente reduzida, como vemos na Figura 6.3.

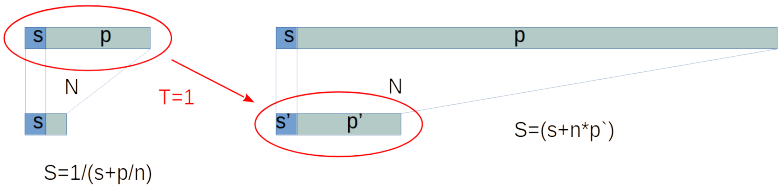

Figura 6.3. Concepção de speedup de tempo fixo.

A partir da ideia de manter o tempo de execução inicial, a razão entre os tempos de execução sequencial e paralelo são fortemente alterados. Isso leva à uma nova formulação, dada pela equação 3.

$$S = n - \alpha \times (n-1) \tag{3}$$

Os resultados obtidos com cada lei podem ser comparados ao examinar a figura 6.4. Fica evidente que os valores de *speedup* são bastante superiores quando determinados por Gustafson, sendo que para 128 máquinas temos S=64,5 para $\alpha = 0,5$ e S=126,7 para $\alpha = 0,01$.

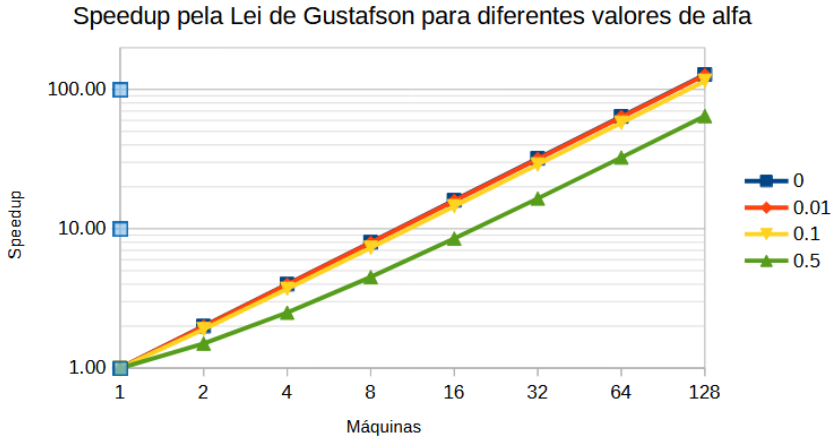

Figura 6.4. Curvas de *speedup*, segundo a Lei de Gustafson, para diferentes valores de α.

6.2.2.2. *Speedup* medido

Apesar de fornecer resultados mais realistas, a Lei de Gustafson ainda não é o procedimento mais adequado para se conhecer a aceleração propiciada pela paralelização de um programa. A forma mais realista de se obter o *speedup* efetivo de um programa é medindo os tempos de execução. Mas, obviamente, essa medição não é feita de forma única.

A medição do *speedup*, como definido pela equação 1, implica em medir dois tempos. Um deles só pode ser obtido de uma forma, que é com a execução paralela

do programa. O problema reside na determinação do tempo sequencial, que pode ser determinado de duas formas distintas. Numa delas esse tempo seria o da execução da versão estritamente sequencial do programa, se possível em sua forma mais eficiente. Na outra forma o tempo sequencial seria medido pela execução do programa paralelo usando apenas uma máquina.

Esses dois tempos sequenciais são primordialmente distintos, sendo o uso da versão sequencial a forma mais conservadora, uma vez que tipicamente será mais rápido do que a versão paralela executando em uma única máquina. Essa diferença é natural pois o programa paralelo sempre conterá instruções adicionais para efetivar o paralelismo. Essas instruções serão executadas, mesmo que a execução seja feita com apenas um processo paralelo, causando um aumento natural (*overhead*) no tempo de execução.

Este autor, assim como Lin e Snyder [7], entre outros, considera que a abordagem usando o tempo do programa paralelo executado em uma máquina deve ser evitado. A razão para isso é simples, pois se queremos saber quanto podemos acelerar um programa, temos que compará-lo com o que alguém executaria se não tivesse máquina paralela, ou seja, o melhor programa sequencial possível.

6.2.3. Eficiência

No início deste capítulo indicou-se que todo o processo de análise de desempenho é uma ferramenta para direcionar o processo de otimização. Neste sentido, uma métrica de desempenho importante é a eficiência. A medida de eficiência apresenta um certo paralelismo com a medida de escalabilidade, no sentido de que um sistema é escalável enquanto tiver uma alta eficiência.

Embora eficiência possa ser definida a partir de vários parâmetros, como estamos falando de computação de alto desempenho, e consequentemente de sistemas paralelos, adotaremos aqui uma definição de trate destes aspectos. Um sistema paralelo eficiente pode ser entendido como aquele em que a aceleração obtida com a sua paralelização é bastante próxima do grau de paralelismo utilizado. Assim, a eficiência de um sistema paralelo pode ser determinada pela equação 4, em que S é o valor de *speedup* e n o número de máquinas paralelas.

$$E(n) = \frac{S(n)}{n} \tag{4}$$

6.3. Medição: monitorar x instrumentar

A atividade de medir desempenho não se limita a determinar o *speedup* ou a eficiência do programa paralelo. Na verdade existem diversos outros aspectos que podem interessar a alguém, dependendo de seus objetivos. Com isso, objetivos diferentes levarão a métricas diferentes, sendo que a sua determinação é fundamental para a execução da análise de desempenho. Essa determinação resulta de respondermos a três perguntas, que são “para quê medir?”, “o quê medir?” e “como medir?”, que devem ser respondidas nessa ordem.

A primeira pergunta, ao ser respondida, restringe (ou direciona) as respostas às perguntas seguintes. Isso ocorre pois ao se definir para quê mediremos acabamos por eliminar certas respostas ao que deve ser medido e, consequentemente, como isso será

medido. Exemplos de respostas à esta pergunta incluem:

- Determinar o *speedup*,
- determinar o nível de ocupação de memória,
- determinar o tempo consumido em determinadas funções do programa,
- determinar o volume de tráfego na rede, etc.

Uma vez determinada a razão para quê realizaremos medidas no sistema é preciso identificar o que será efetivamente medido. Como já indicado, essa resposta depende, em parte, da resposta anterior. A lista a seguir representa possíveis respostas ao que deve ser medido, mas deve ficar claro que podem ou não ser uma opção, a depender do motivo da medição.

- Medir o tempo de entrega do programa,
- medir o tempo gasto em uma dada função,
- medir a quantidade de bytes transferidos por mensagem,
- medir o número de mensagens enviadas, etc.

Por fim, temos que determinar como mediremos os parâmetros desejados. Técnicas de medição podem ser divididas em duas grandes categorias, que são monitoração e instrumentação.

6.3.1. Monitoração

Técnicas de monitoração se baseiam no uso de mecanismos físicos, ou próximos disso, para medir aquilo que se deseja medir. A monitoração pode ser realizada por hardware especializado, como analisadores lógicos, contadores digitais, osciloscópios, etc. Também pode ser realizada por software, por meio da programação de determinadas interrupções do *kernel* do sistema operacional para a obtenção de medidas.

A principal vantagem da monitoração é fornecer medidas mais precisas, principalmente quando realizada com hardware especializado. Entretanto, tem como desvantagem a necessidade de equipamentos específicos e conhecimento detalhado do funcionamento da máquina ou do sistema operacional.

Na prática, técnicas de monitoração são aplicadas apenas pelos próprios fabricantes de hardware, durante o seu desenvolvimento. Para situações mais mundanas a opção recai em alguma das formas de instrumentação, que além de ser mais barata, exige menor conhecimento sobre o sistema.

```
1  #include <time.h>
2  #include "stdio.h"
3
4  double etimes()
5  {struct timespec tp;
6        clock_gettime(CLOCK_REALTIME, &tp);
7        return(tp.tv_sec + tp.tv_nsec/1000000000.0);
8  }
9
10 main()
11 {double Duration;
12 ......
13    Duration = etimes();
14    do_whatever_has_to_be_measured();
15    Duration = etimes() - Duration;
16    printf ("Function took %f seconds\n", Duration);
17 }
```

Figura 6.5. Instrumentação por modificação do código fonte.

6.3.2. Instrumentação

Técnicas de instrumentação fazem a modificação da aplicação que se quer analisar, de forma a que sua execução produza os resultados de interesse. A instrumentação pode ocorrer em três momentos da produção de um programa. Isso implica em modificar o programa, instrumentando-o para a geração das medidas, diretamente no código fonte, ou em seu objeto, ou ainda diretamente no executável.

O processo de instrumentação produz resultados que podem ser vistos como a descrição do perfil de uma execução. Essa descrição é o chamado perfilamento do programa (*profiling*), formando o conjunto de dados para sua análise de desempenho. Embora o perfilamento seja a forma mais comum de produção de resultados, é importante mencionar que um outro mecanismo usado é a geração de traços de eventos (*event tracing*). A diferença entre os dois é que o perfilamento produz resultados pela amostragem do que está sendo executado, enquanto os traços de eventos consistem em registrar o instante em que determinados eventos ocorrem.

6.3.2.1. Modificação do código fonte

A instrumentação de um programa por meio da modificação do código fonte é a que permite detalhar pontos mais específicos de interesse. Seu problema é que ela é aplicável apenas se o código fonte estiver acessível, o que nem sempre é uma realidade. A modificação do código fonte depende de funções de medição de tempo na linguagem utilizada. O processo de medição é bastante simples, consistindo em inserir chamadas para as funções de tempo antes e depois da parte do código que queremos medir.

O exemplo da Figura 6.5 usa a função `clock_gettime` para obter o tempo medido pelo sistema operacional para construir a função `etimes`. No caso se quer medir

o tempo consumido na chamada da função `do_whatever_has_to_be_measured` (linha 15), que é então cercada por duas chamadas para a função `etimes`, sendo que a diferença de valores retornados pela primeira e segunda chamadas o tempo consumido pela função medida.

Esse processo é intrusivo e acaba por criar uma carga adicional de execução, pelas chamadas de funções adicionais. No exemplo dado podemos diminuir um pouco a intrusão colocando chamadas para `clock_gettime` diretamente antes e depois da função medida. Mas, mesmo assim, ainda temos alguma imprecisão na medição.

Um outro problema, que ocorre com qualquer técnica de instrumentação, é a imprecisão decorrente da forma como o sistema atualiza seu relógio. Apesar de funções como a apresentada acima permitirem a medição de nanossegundos, essa não é a precisão das mesmas. O que ocorre é que apesar do relógio de um computador oscilar a mais de 3 GHz, o que dá um ciclo de relógio a cada 0,33 ns, a variável que armazena tempo não é atualizada a cada ciclo e sim a cada intervalo de alguns microssegundos. Assim, a leitura do tempo atual sempre incorre em um certo erro.

6.3.2.2. Modificação do código objeto ou do executável

Aqui as funções para medição de tempo são inseridas no momento da compilação. Com isso temos que usar ferramentas projetadas para essa função, sendo que boa parte delas também depende do acesso ao código fonte (para fazer a compilação, é claro). Talvez a ferramenta mais conhecida para este tipo de instrumentação é `gprof`, que faz parte do pacote *gnu*. Diferentes versões do gprof foram desenvolvidas para sistemas específicos, como Solaris, HPUX, etc., mas estão descontinuadas hoje em dia.

O procedimento para a instrumentação de código com a modificação de código objeto consiste em compilar o código utilizando o um parâmetro que faça a instrumentação. Após a compilação o programa deve ser executado, gerando seus resultados de perfilamento para análise. Por exemplo, o código apresentado na figura 6.6, pode ser compilado e medido da seguinte forma:

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

O parâmetro *pg* no comando de compilação é que insere chamadas para funções de perfilamento. Essas funções, ao serem executadas, produzem medidas que serão listadas pela chamada de *gprof*, cuja saída está direcionada para o arquivo *loops.prof*. Parte desse arquivo é apresentada na figura 6.7.

Nessa figura é possível ver que o *gprof* produz sobre quanto tempo é consumido em cada função, tanto no total como em cada chamada. Deve ser observado que apesar da contagem de chamadas estar correta (ele de fato conta quantas foram), os tempos consumidos não são estritamente proporcionais ao que se esperaria pelo que é executado. Essa diferença é causada pela amostragem, que como já indicado, depende da frequência de atualização do valor do relógio. Apesar desse erro, é importante lembrar que o objetivo

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

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

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

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

Figura 6.6. Código do programa *loops.c*, usado como exemplo de modificação do código objeto.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self                         self    total
 time    seconds  seconds    calls  us/call  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

 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]
```

Figura 6.7. Trechos de informações produzidas pelo *gprof*.

desse tipo de medida é a indicação das proporções de uso para a otimização, e é fácil perceber que a função *baz()* é a que merece maior atenção, por ter um laço mais longo.

6.3.2.3. Modificação dinâmica do executável

A modificação do código executável pode ser feita de modo *offline*, como ocorre com o uso de perfiladores como o *gprof*. Existem, entretanto, mecanismos de modificação de código que operam de modo *online*. A ferramenta mais utilizada que faz esse tipo de abordagem é o **Dyninst**, que é uma API de instrumentação dinâmica desenvolvida em um projeto entre a Universidade de Wisconsin-Madison e Universidade de Maryland [8, 9].

O que o *Dyninst* faz é inserir a instrumentação para medição enquanto o programa está sendo executado. A vantagem com essa abordagem é que a instrumentação pode ser direcionada aos trechos mais custosos do programa, reduzindo a sobrecarga com códigos intrusivos. O principal problema dela é que o código a ser medido tem que executar por alguns minutos, de forma a permitir que a ferramenta execute a instrumentação dinâmica em tempo de realizar as medições.

6.3.3. Observações sobre monitoração e modificação de código

Para finalizar esta seção faremos uma breve comparação entre os métodos de medição baseados em monitoração e modificação de código. Já antecipamos que os processos de monitoração são mais precisos, uma vez que suas medições ocorrem de forma exata, com instrumentos físicos e sem a necessidade de codificação adicional. Já os mecanismos baseados em modificação de código, ao acrescerem comandos de instrumentação ao código original, acabam gerando erros por métodos intrusivos, além dos problemas inerentes aos erros de amostragem do relógio.

Apesar de mais precisos, métodos de monitoração têm uso bastante restrito por demandarem um bom conhecimento sobre instrumentos e sobre o funcionamento do sistema operacional e o hardware. Já os métodos baseados em modificação do código não necessitam de conhecimento muito elaborado, permitindo um uso mais amplo por parte de desenvolvedores e analistas.

Como conclusão, vemos que apesar de certa imprecisão, os métodos baseados em modificação de código são os mais utilizados.

6.4. *Benchmarks*

Os métodos apresentados até aqui se enquadram na categoria de métodos baseados em medição, ou *benchmarking*, que serão melhor examinados nesta seção. Antes, porém, descreveremos de modo mais abreviado os métodos baseados em predição, que são os métodos analíticos e baseados em simulação.

6.4.1. Predição de desempenho

Apesar de aparecer estranho, existem situações em que a análise de desempenho pode ser feita com métodos preditivos, em que não ocorrem medições efetivas sobre o objeto da análise. Essas situações envolvem, por exemplo, a análise do possível desempenho de um

sistema que ainda não exista fisicamente, mas cujo modelo possa ser elaborado de forma razoável.

Os métodos preditivos podem ser baseados em modelos analíticos, como cadeias de Markov, redes de Petri e redes de filas [10]. A análise de desempenho usando essas técnicas se baseia na criação de de modelos matemáticos, que ao serem resolvidos fornecem previsões sobre o comportamento de um sistema computacional. A precisão desses modelos depende, fundamentalmente, da qualidade dos modelos construídos, ou seja, se o modelo consegue representar o ambiente software-hardware mais precisamente, então os resultados da predição serão mais precisos.

Deve ficar claro, entretanto, que a resolução dos modelos analíticos só é feita por métodos não computacionais para modelos relativamente pequenos. Modelos mais complexos, como sistemas paralelos e aplicações neles executadas, demandam a solução por meio de simulações computacionais.

Simulação de eventos discretos é, de forma isolada, outra classe de métodos preditivos. A diferença aqui é que os modelos não são resultantes de formulações algébricas, como cadeias de Markov, e sim de descrições qualitativas do sistema. Técnicas baseadas em simulação fazem a modelagem usando características observáveis do sistema, como a descrição de seu funcionamento, das interações entre partes, etc. Esses modelos são construídos usando técnicas conhecidas, como redes de Petri, Monte Carlo e redes de filas [11].

Mais uma vez, a precisão de técnicas baseadas em simulação depende da qualidade do modelo criado para o sistema. Uma vantagem da simulação, em relação aos modelos analíticos, é que a qualidade do modelo é mais facilmente ajustada, permitindo possivelmente resultados mais precisos.

6.4.1.1. Comparando métodos preditivos e *benchmarking*

A análise de desempenho feita por diferentes métodos tem, naturalmente, diferentes impactos sobre o processo. A comparação entre os métodos pode ser feita considerando alguns aspectos como em qual estágio de desenvolvimento do sistema se pode aplicar uma dada técnica. A tabela 6.1 apresenta tal comparação, sendo que dela podemos destacar os seguintes pontos:

Tabela 6.1. Quadro comparativo entre métodos preditivos e *benchmarking*.

Critério	Analíticos	Simulação	*Benchmarking*
Estágio de desenvolvimento	Qualquer	Qualquer	Pós-Protótipo
Tempo na obtenção de resultados	Baixo	Médio	Variável
Ferramentas para modelagem	Humanos	Linguagens	Instrumentação
Exatidão	Baixa	Moderada	Variável
Avaliação de alternativas	Fácil	Moderada	Difícil
Custo	Baixo	Médio	Alto
Credibilidade do método	Baixo	Médio	Alto

- Exatidão, em que mostra que métodos preditivos de fato não apresentam grande precisão, mas também mostra que a precisão de *benchmarking* pode variar, dependendo de como é feita a instrumentação e, principalmente, como é definida a carga de trabalho durante as medições.
- Avaliação de alternativas, ou seja, como avaliar mudanças no software ou hardware podem ser usadas na busca por melhoria de desempenho, o que é fácil de fazer em modelos abstratos (tanto analíticos como de simulação), mas é difícil de fazer com o sistema real, pois demanda modificações no hardware ou a implementação de novos algoritmos no software.
- Credibilidade do método, que é um indicativo de como um leigo enxerga os resultados apresentados, que aponta para maior credibilidade para medições reais sobre o ambiente real, e menor para medições geradas por modelos abstratos.

6.4.2. Introdução ao *benchmarking*

O processo de *benchmarking* envolve a medição física de parâmetros de um sistema. É, portanto, um mecanismo que atua diretamente sobre o objeto de nosso interesse, tipicamente utilizando técnicas de modificação de código. Apesar de sua efetividade e aparente simplicidade, existem problemas importantes que devem ser resolvidos antes de sua aplicação, que são a definição da carga de trabalho (o que será medido) e como a sua execução pode ser comparada com outras situações. Esses aspectos nos levam à existência de dois tipos básicos de *benchmarking*, que são:

1. ***Benchmarks* produzidos por organizações**, tipicamente utilizados como padrões para comparação de sistemas de hardware. Nesses casos se têm uma organização, como a SPEC [12] ou NASA [13], que define um conjunto de aplicações que devem ser executadas e medidas, de forma a produzir uma medida global de desempenho. É interessante notar que esse tipo de medição produz resultados que podem ser comparados entre si, desde que a carga de trabalho seja equivalente.

   Tipicamente as diferentes organizações indicam várias aplicações complexas e que demandem características distintas do sistema, como operações de ponto flutuante (dinânica de fluídos, previsão metereológica, modelagem oceânica, etc.), operações com inteiros (compilação do gcc, simulação discreta, compressão de dados, etc.), uso de memória, etc.

   Na mesma categoria podemos encontrar outros *benchmarks*, como o Rodinia[2], que tem como atratividade ser desenvolvido dentro de uma universidade, o Linpack/Lapack[3], que é o *benchmark* utilizado para classificar os sistemas da lista **TOP500**, e o Passmark[4] (entre outros), que apresentam resultados de desempenho de CPUs, por exemplo.

2. ***Benchmarks* produzidos com objetivos locais**, usados para mensurar uma aplicação especifica, com validade local. São especificamente desenvolvidos para uma

[2]Disponível em www.cs.virginia.edu/rodinia
[3]Disponível em www.netlib.org/lapack
[4]Disponível em www.cpubenchmark.net

dada aplicação, em que os programas/casos de teste são escolhidos localmente, de acordo com os objetivos da entidade interessada na análise. Por terem essa característica a validade dos resultados também é local, não podendo ser generalizada nem mesmo para ambientes similares.

Na criação desses *benchmarks* é importante observar que os resultados, quando envolverem vários programas de teste, devem ser normalizados de acordo com sua relevância.

6.4.2.1. Normalização de *benchmarks* locais

A normalização dos resultados de um *benchmark* local deve considerar que a comparação não pode usar apenas tempos absolutos de execução. Ela deve considerar também o grau de importância dos programas que estão sendo medidos. Para entender como a normalização funciona podemos trabalhar um exemplo simples, em que queremos comprar um sistema, entre três possíveis (X, Y e Z), para executar três programas (A, B e C). As medidas obtidas com esses programas e sistemas é vista na tabela 6.2.

Tabela 6.2. Tempos medidos (em u.t.) durante os *benchmarks* realizados.

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

Essas medidas, se desconsiderados quaisquer outros fatores, indicaria que o sistema X levaria 1456 u.t. para executar uma instância de cada um dos três programas, enquanto o sistema Y levaria 1654 u.t. e o sistema Z levaria 1465 u.t. Com tais resultados um analista diria que o sistema X é que seria o mais rápido entre os três. Essa análise, entretanto, pode ser falha caso os três programas tenham diferentes graus de uso ao longo de um dia. Se, por exemplo, o programa A ocupar 65% do tempo de uso do sistema, o programa B ocupar 25% e o programa C ocupar 10%, então os resultados da tabela 6.2 podem ser normalizados considerando essas novas informações.

O processo de normalização pode ser feito transformando os tempos apresentados em valores relativos ao menor tempo de execução para cada programa. Por exemplo, para o programa A consideramos 310 u.t. como sendo 1,00 (no sistema Z). Assim, os tempos de execução nos sistemas X e Y seriam respectivamente 1,04 e 1,19. Após aplicarmos a normalização aos três programas, podemos colocar os pesos relativos ao uso de cada programa, multiplicando seu uso pelo tempo normalizado, resultando nos valores apresentados na tabela 6.3, em que se observa que o sistema Z é que produziria os melhores resultados para o padrão de uso da empresa.

6.5. Exemplos de ferramentas

Apesar de vários obstáculos para a medição do desempenho de um sistema, como a definição correta da carga de trabalho, existem várias ferramentas produzidas com essa

Tabela 6.3. Tempos normalizados e ponderados para os *benchmarks* realizados.

	Sistema X	Sistema Y	Sistema Z
Programa A	1,04*0,65	1,19*0,65	1,00*0,65
Programa B	1,00*0,25	1,15*0,25	1,03*0,25
Programa C	1,00*0,10	1,10*0,10	1,00*0,10
Tempo normalizado	1,03	1,17	1,01

finalidade, algumas pagas outras de livre acesso. Descreveremos aqui algumas dessas ferramentas.

6.5.1. Paradyn/Dyninst [9]

É uma ferramenta de acesso livre, cujo desenvolvimento se iniciou há mais de 20 anos. Seu princípio de medição é fazer a instrumentação dinâmica do código. Esse processo consiste em criar uma cópia da função que será medida, com as chamadas para instrumentação e alterar a chamada para essa função, o que se denomina trampolim, para que fizesse a chamada da função modificada e não da original. Após a medida o trampolim é retirado e a função cópia é removida. A figura 6.8 mostra o grafo de chamadas de funções avaliadas em um dado programa, no qual as funções com nomes em preto ainda não foram instrumentadas.

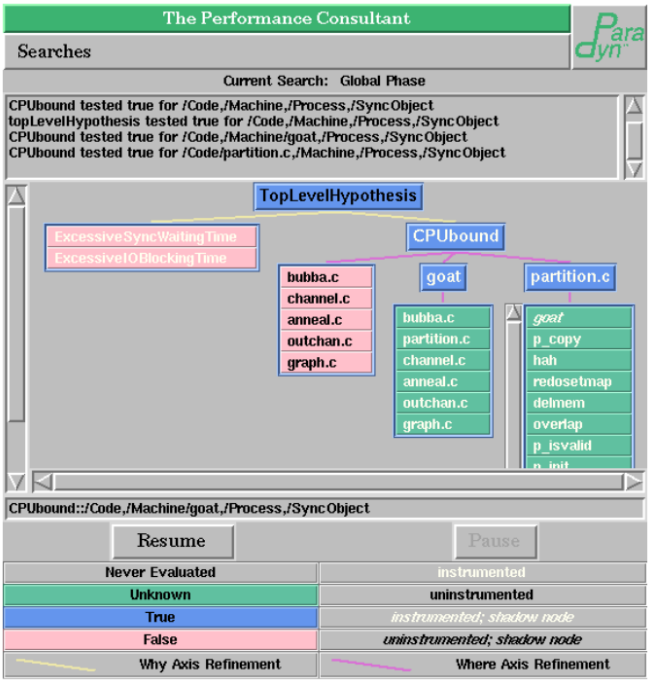

Figura 6.8. Interface do grafo de chamadas do Paradyn.

Um aspecto interessante nessa ferramenta é que sua base, o Dyninst, também é usada em várias outras ferramentas de análise de desempenho, como HPCToolkit, TAU e Omnitrace, por exemplo.

6.5.2. Vampir [14]

Diferente do Paradyn, esta é uma ferramenta com uma versão funcional em caráter demonstrativo, porém seu uso continuo depende de licenças pagas (algo entre 720 e 30.000 euros anuais). Na figura 6.9 temos a interface que mostra a linha de tempo de execução de um conjunto de processos, mostrando os trechos do código executados a cada momento.

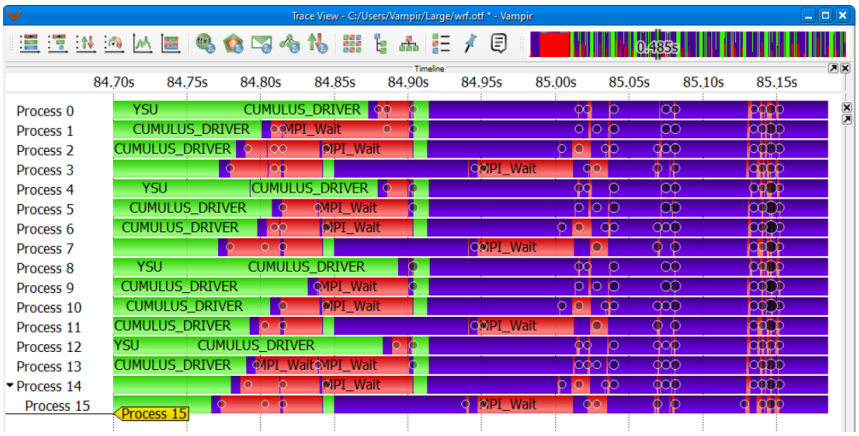

Figura 6.9. Interface de linha do tempo do Vampir.

6.5.3. vTune/Advisor [15]

Estas ferramentas fazem parte de uma suíte para análise de sistemas (oneAPI) oferecida pela Intel. Elas permitem análises em sistemas com CPU, GPU e FPGA, sendo portanto bastante versáteis. Advisor é uma ferramenta que indica otimizações em um programa, incluindo quais trechos do código deveriam ser executados por um acelerador (GPU, por exemplo), enquanto vTune faz o perfilamento da execução.

Uma métrica interessante oferecida pelo Advisor é o que se chama *roofline analysis*, que pode ser usada para identificar que volume de processamento pode ser levado para a GPU, ou ser vetorizado, como visto na figura 6.10.

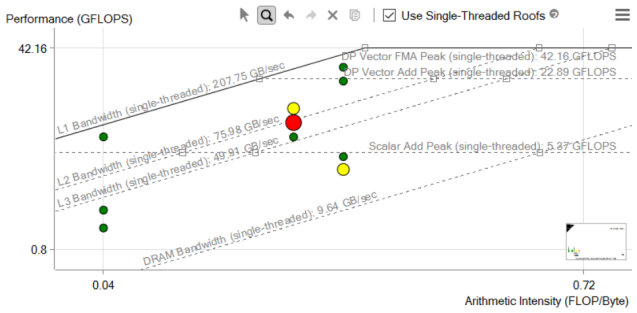

Figura 6.10. Interface do Advisor com informações sobre rooflines.

Em github.com/pvelesko/nbody-demo.git são encontrados vários exemplos de aplicação dessas ferramentas, que são disponibilizados por Paulius Velesko. A partir deles gerou-se, por exemplo, os dados mostrados na figura 6.11, que mostra a sequência de

execução de threads no programa. Na versão com interface gráfica se deve configurar o projeto, com indicação de executável e parâmetros de execução.

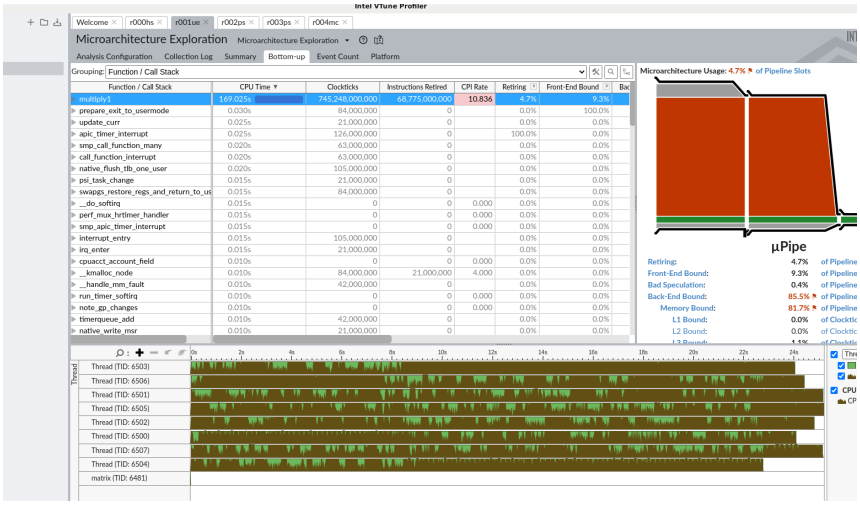

Figura 6.11. Interface do vTune com informações sobre threads executados.

6.5.4. TAU [16]

Desenvolvido na Universidade do Oregon, em parceria com Los Alamos National Laboratory (LANL) e Jülich Research Centre. É uma ferramenta bastante flexível, que pode ser configurada para diferentes máquinas e ambientes, gerando traços/perfilamentos da aplicação, que podem ser inclusive manipulados por outras ferramentas, como Vampir.

O programa a ser analisado deve ser compilado usando parâmetros específicos que indiquem o que deve ser medido. A execução do programa gera arquivos de traços para cada núcleo utilizado, que podem ser visualizados com duas ferramentas, que são o PerfExplorer e ParaProf.

Na figura 6.12, por exemplo, temos uma janela do Paraprof mostrando onde cada processo foi executado em um programa em MPI.

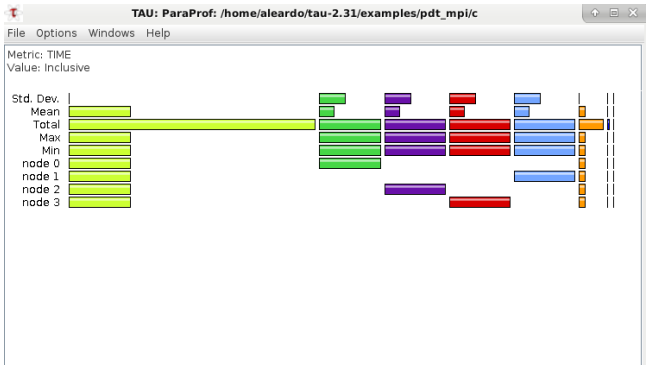

Figura 6.12. Interface do Tau/Paraprof, com a distribuição de processos por nó de processamento.

Com as ferramentas de visualização é possível obter um conjunto bastante amplo de métricas, incluindo a clusterização de resultados no caso de elevado número de processos paralelos, como pode ser visto na figura 6.13.

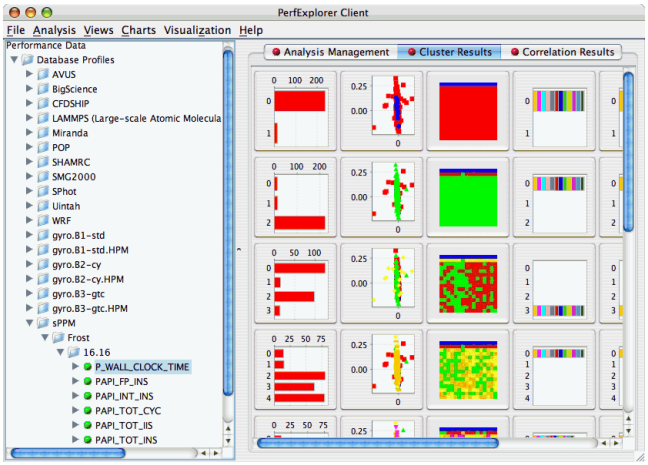

Figura 6.13. Interface do Tau/PerfExplorer, com uma análise clusterizada dos resultados.

6.6. Técnicas para otimização de desempenho

Otimização é a identificação de pontos em um programa que podem ser melhorados e, principalmente, a realização de tais melhorias. Esse processo pode ser feito de forma automatizada, pelo compilador, ou pelo próprio programador. Quando feita pelo compilador se utiliza as técnicas clássicas de otimização em compilação, como alinhamento de funções ou desenrolamento de laços.

6.6.1. Aspectos que impactam o desempenho

Um programa pode ser otimizado em três aspectos, que são o estilo de programação, a arquitetura geral do sistema ou o código do programa. Quando falamos em arquitetura geral do sistema, falamos em topologia, elementos de processamento ou conexão e modelos de particionamento do software. Destes, apenas a forma de particionamento é que pode ser modificada de modo simples, embora implique em modificar como dados são separados entre os processadores, de modo a garantir a melhor granulosidade dos processos paralelos.

Otimizar baseado em estilo de programação significa escolher uma linguagem mais eficiente para programar (para muitas aplicações numericamente intensivas isso ainda recai no Fortran), ou estruturas de dados mais eficientes. Sobre estas um aspecto importante é que se deve evitar o uso desnecessário de ponteiros. Por exemplo, apesar de ser possível armazenar um grafo usando ponteiros, evitando desperdício de memória com uma matriz de incidência, por exemplo, isso tem um custo computacional elevado[5].

[5]Um grafo de cerca de 70000 vértices, é lido e montado numa estrutura dinâmica em um tempo cerca de 500 vezes mais lento do que numa estrutura estática.

6.6.2. Otimizando o código

A otimização diretamente no código pode ocorrer em dois níveis: funções e laços. Para a otimização de funções basicamente se usa o conceito de *function inlining*, isto é, em vez de se fazer uma chamada convencional da função se faz a substituição direta no código. Isso reduz a legibilidade do código, assim como aumenta o tamanho do programa na memória. Quanto ao aspecto de legibilidade, podemos fazer uso de macros da linguagem, em que uma macro é definida e a chamada do que seria a função é substituída pela chamada da macro, como vemos no código da figura 6.14.

```
1  #define average(x,y) ((x+y)/2)
2
3  main()
4  {float a, q=100, p=50;
5      a = average (p,q);
6      printf("%f\n",a);
7  }
8  //  average (p, q)  vai ser trocada por (p+q)/2
```

Figura 6.14. Alinhamento de função com o uso de macros da linguagem.

Laços de repetição, são, por sua vez, uma fonte bastante ampla para otimizações. Em laços podemos der seu desenrolamento, desdobramento, remoção de testes desnecessários, fusão ou mesmo fissão do laço.

Desenrolamento de laço Quando se tem conhecimento do tamanho do laço (ou pelo menos um mínimo de iterações) é possível trocar o laço pela repetição explícita de seu corpo. Por exemplo:

```
1  for (i=0; i<3; i++)        pode ser        a[0] = b[0] + c[0];
2    a[i] = b[i] + c[i];    trocado por      a[1] = b[1] + c[1];
3                                            a[2] = b[2] + c[2];
```

Remoção de testes desnecessários Muitas vezes incluímos testes em laços que podem ser removidos por vários motivos. Um deles pode envolver um teste para o qual, a menos por problemas bastante intensos de precisão, podemos considerar que o resultado será sempre o mesmo, eliminando o teste. Outra situação, mais comum, é ter um teste para o qual o resultado será sempre igual pois envolve alguma variável não modificada no laço, como vemos na figura 6.15.

A mesma técnica pode ser aplicada se o teste interno, em laços aninhados, envolver a comparação entre as variáveis de controle dos laços. Nesse caso o laço interno pode ser desdobrado em dois laços, com iterações definidas pelo resultado previsível da comparação.

```
1  for (i=0; i<n; i++)       pode ser     if (k != 0)
2    if (k != 0)           trocado por       for (i=0; i<n; i++)
3        a[i] = b[i] + k;                       a[i] = b[i] + k;
4      else                                   else
5        a[i] = c[i] + k;                       for (i=0; i<n; i++)
6                                                 a[i] = c[i] + k;
```

Figura 6.15. Teste desnecessário com desdobramento de laço.

inversão de laço em laços aninhados A otimização nesse caso depende da forma como os laços, que envolvam matrizes, são percorridos. Sabemos que matrizes são armazenadas de modo vetorizado, sendo que em C (entre outras) isso ocorre linha por linha, enquanto em Fortran ocorre coluna por coluna. Assim, percorrer a matriz é feito mais eficientemente quando ocorre na mesma ordem de armazenamento.

Desse modo, laços aninhados podem ser percorridos mais eficientemente se o laço interno percorrer a matriz na ordem ótima (colunas da mesma linha) e o externo na outra ordem (linhas), exigindo menos mudanças de *cache*. O problema é que nem sempre isso é possível, como é o caso da multiplicação de matrizes, em que uma matriz é percorrida linha a linha e a outra coluna a coluna (embora existam alternativas para contornar esse problema).

Fusão e fissão de laços Fusão de laço ocorre quando temos laços consecutivos que executarão exatamente o mesmo número de iterações. Desse modo, para evitar a duplicidade de controle, os corpos dos dois laços podem ser aglomerados em um único laço, caso sejam independentes.

Já a fissão de um laço em dois novos laços pode ocorrer para reduzir a necessidade de trocas de linhas de *cache*. Assim as iterações em cada laço são modificadas para, em C por exemplo, envolver uma menor quantidade de colunas por laço.

6.6.3. Redução de força

Determinados operadores demandam mais esforço computacional que outros, principalmente quando tratamos de operandos de ponto flutuante. Em algumas situações é possível reduzir esse esforço trocando uma operação por outra que demande menos força. Por exemplo, podemos trocar o operador de potenciação por sua execução explícita, como em:

Multiplicação	$(y*y*y)$ é mais rápida que potenciação $(pow(y,3))$
Soma	$(k+k+k)$ é mais rápida que multiplicação $(3*k)$

Referências

[1] STERLING, T.; ANDERSON, M.; BRODOWICZ, M. *High Performance Computing: Modern Systems and Practices*. EUA: Morgan Kaufmann, 2017. 718 p.

[2] MANACERO, A. *Predição do desempenho de programas paralelos por simulação do grafo de execução*. Tese (Doutorado) — University of Campinas, Brazil, 1997. Disponível em: <https://doi.org/10.47749/T/UNICAMP.1997.118682>.

[3] SAHNI, S.; THANVANTRI, V. Performance metrics: keeping the focus on runtime. *IEEE Parallel & Distributed Technology: Systems & Applications*, v. 4, n. 1, p. 43–56, 1996.

[4] JAIN, R. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling*. [S.l.]: Wiley, 1991. 685 p. (Wiley professional computing). ISBN 978-0-471-50336-1.

[5] AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. New York, NY, USA: Association for Computing Machinery, 1967. (AFIPS '67 (Spring)), p. 483–485. ISBN 9781450378956. Disponível em: <https://doi.org/10.1145/1465482.1465560>.

[6] GUSTAFSON, J. L. Reevaluating amdahl's law. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 31, n. 5, p. 532–533, may 1988. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/42411.42415>.

[7] LIN, Y. C.; SNYDER, L. *Principles of Parallel Programming*. Boston, Mass: Pearson/Addison Wesley, 2008. ISBN 978-0321487902. Disponível em: <http://www.amazon.com/Principles-Parallel-Programming-Calvin-Lin/dp/0321487907>.

[8] MILLER, B.; HOLLINGSWORTH, J. *Paradyn Tools Project*. 2024. Accessed: 2024-09-13. Disponível em: <https://www.paradyn.org/>.

[9] HOLLINGSWORTH, J.; MILLER, B.; CARGILLE, J. Dynamic program instrumentation for scalable performance tools. In: *Proceedings of IEEE Scalable High Performance Computing Conference*. [S.l.: s.n.], 1994. p. 841–850.

[10] TAY, Y. C. *Analytical Performance Modeling for Computer Systems*. Springer International Publishing, 2014. ISSN 1932-1686. ISBN 9783031018008. Disponível em: <http://dx.doi.org/10.1007/978-3-031-01800-8>.

[11] CHEN, K. *Performance Evaluation by Simulation and Analysis with Applications to Computer Networks*. [S.l.]: John Wiley & Sons, Ltd, 2015. 286 p. ISBN 9781119006190.

[12] SPEC. *Standard Performance Evaluation Corporation*. 2024. Accessed: 2024-09-13. Disponível em: <https://www.spec.org>.

[13] NASA Advanced Suporcomputing Division. *NAS Parallel Benchmarks*. 2024. Accessed: 2024-09-13. Disponível em: <https://www.nas.nasa.gov/software/npb.html>.

[14] GWT-TUD GmbH. *Vampir 10.5*. 2024. Accessed: 2024-09-13. Disponível em: <https://vampir.eu>.

[15] REINDERS, J. *Vtune performance analyzer essentials*. [S.l.]: Intel Press, 2007.

[16] PERFORMANCE RESEARCH LAB. *TAU - Tuning and Analysis Utilities*. 2006. `https://www.cs.uoregon.edu/research/tau/home.php`. Accesseda em Julho de 2024.