Análise da Escalabilidade de um Programa Multicore

Weverson dos Santos Gomes[1]

[1]DTEC - Departamento de Tecnologia
Universidade Estadual de Feira de Santana (UEFS)
Feira de Santana, BA – Brasil

weversondsg@gmail.com

1. Introdução

Muitas aplicações são primeiramente programadas para rodar em um único núcleo. No campo da computação de alto desempenho, diferentes estratégias podem ser adotadas para acelerar cômputos quando diferentes recursos computacionais estão disponíveis.

A resolução de problemas computacionalmente densos, necessita de um tempo longo para a obtenção do resultado final. Uma técnica utilizada para acelerar o cômputo destes tipos de problemas é por meio da programação paralela, que permite a execução de múltiplas tarefas simultâneas. Uma das arquiteturas que permitem a utilização de programação paralela é a arquitetura multicore. A API OpenMP permite acrescentar simultaneidade aos programas escritos em C, C++ e Fortran sobre a base do modelo de execução fork-join.

O objetivo deste trabalho analisar a escalabilidade para arquiteturas multi-core de um problema computacionalmente denso. A análise será feita variando o número de núcleos.

2. Fundamentação Teórica

Algumas questões importantes a se avaliar quando se trabalha com computação de alto desempenho são: É melhor deixar a aplicação serial ou paralela? Se é melhor paralelizar, até que ponto vale a pena aumentar o número de processadores? A minha aplicação está fazendo uso eficiente dos recursos disponíveis? A seguir veremos algumas métricas que podem ajudar a responder estas perguntas.

2.1. Escalabilidade

Aplicações de alto desempenho dependem fortemente da capacidade de uma aplicação de escalar tarefas eficientemente em paralelo entre os múltiplos recursos computacionais [Hedges 2016]. A esta capacidade de escalar tarefas dá-se o nome de escalabilidade. Misurar a escalabilidade de uma aplicação é uma tarefa comum em computação de alto desempenho. Esta medida indica o quão eficiente uma aplicação é ao passo que aumenta o número de unidades de processamento paralelo (CPUs, cores, threads etc) [Sharcnet 2017].

A escalabilidade de um aplicação pode ser classificada como forte ou fraca [Hedges 2016], dependendo do tipo da aplicação, se é cpu-bound ou memory-bound [Sharcnet 2017]. Observe que o termo ”fraco”, como usado aqui, não significa inadequado ou ruim, mas é um termo técnico que facilita a descrição do tipo de escala [Hedges 2016].

Em aplicações de escalabilidade forte, o tamanho do problema permanece fixo, mas o número de elementos de processamento é aumentado. Isso é usado como justificativa para programas que demoram muito para ser executados (algo que é cpu-bound). O objetivo, neste caso, é encontrar um ponto de equilíbrio que permita completar o cômputo em um período de tempo razoável, mas sem desperdiçar muitos recursos [Sharcnet 2017].

Se a quantidade de tempo para completar uma unidade de trabalho com 1 elemento de processamento é t1 e a quantidade de tempo para completar a mesma unidade de trabalho com N elementos de processamento é tN, a eficiência de escalabilidade forte é dada como:

Es = t1 / (N * tN) * 100

Em aplicações de escalabilidade fraca, o tamanho do problema (carga de trabalho) atribuído a cada elemento de processamento permanece constante e elementos adicionais são usados para resolver um problema total maior (um que não caberia na RAM de um único nó, por exemplo). Portanto, esse tipo de medição é ideal para programas que levam muita memória ou outros recursos do sistema. A eficiência de escalabilidade fraca é dada como:

EW = (t1 / tN) * 100

2.2. Speed Up

Outra métrica importante para avaliar o desempenho de uma aplicação é o speedup.

O speedup pode ser definido como a relação entre o tempo gasto para executar uma tarefa com um único processador e o tempo gasto com N processadores. Representa o ganho efetivo em tempo para um dado aumento no número de unidades de processamento [Lantz 2015]. É dado como:

S = t1 / tN

2.3. OpenMP

OpenMP (Open Multi-Processing) é uma API para programação de sistemas paralelos de memória compartilhada. Portanto, OpenMP é projetado para sistemas em que cada *thread* ou processo pode potencialmente ter acesso à toda memória disponível [**?**]. Em programas desenvolvidos em OpenMP essas *threads* podem ser manipuladas através de diretivas que são passadas para o compilador.

Os algoritmos que utilizam a tecnologia OpenMP são executados serialmente até que uma seção paralela seja declarada pelo comando *omp parallel* passado através de um *pragma*. Os *pragmas* são recursos da linguagem C que permitem comportamentos que não estão diretamente especificados e são identificados pelo símbolo #. As políticas de *pragmas* dependem de cada sistema operacional e compilador. Portanto, programas implementados em OpenMP requerem um compilador que dê suporte a tal para ser executado. Caso o compilador não ofereça suporte à plataforma o código será normalmente executado, porém os *pragmas* não serão processados.

Ao encontrar um o comando *#pragma omp parallel* um grupo de *threads* é criado. O número de *threads* alocadas para execução do programa pode ser definido pelo programador através de variáveis de ambiente ou usando funções do próprio OpenMP. A

thread 0, ou *master*, é definida como sendo a primeira *thread* a executar a diretiva *parallel* [Grama et al. 2003].

O ciclo de execução segue pela *thread master* até que uma nova diretiva para paralelização seja encontrada. Nesse momento novas *threads* são chamadas para que o trabalho seja distribuído e ao final da execução o ciclo principal volta para a *thread 0*. Esse modelo é conhecido como *fork/join* e é ilustrado para o caso de duas *threads* na figura 1.

Figura 1. Divisão de uma tarefa (*forking*) entre duas *threads*. Fonte: [Pacheco 2011]

OpenMP constitui uma solução eficiente para programadores que necessitam paralelizar explicitamente um sistema de memória compartilhada por ser compatível com a maioria dos fabricantes de *hardware* e é relativamente fácil de ser utilizado. Uma desvantagem dessa tecnologia é que é limitado a sistemas de memória compartilhada.

3. Metodologia

3.1. Implementação

Primeiramente foi criado o código serial. A partir do código serial foi feita uma análise do código utilizando a ferramenta GNU Profiling para entender o comportamento do código serial durante a execução.

3.2. Compilação

A compilação usou o argumento de otimização -O3, priorizando a redução do tempo de execução em relação ao uso de memória pelo programa gerado e ao tempo de compilação. Usou-se também o argumento gcc -Wall, habilitando todos os alertas do compilador para melhorar a qualidade do código e obter melhor desempenho.

3.3. Validação

Na ausência de resultados que poderiam servir como referência para validação da saída do programa, optou-se por implementar as equações do problema em um outro ambiente para que, a partir de então, fosse possível verificar a exatidão dos valores calculados. Desse modo, todas equações foram escritas em uma planilha de testes e foram fornecidos dados iguais de entrada para o programa e para a planilha, em seguida os resultados eram comparados.

3.4. Execução

O programa gerado foi executado em um computador com sistema operacional Linux, 8GB de memória RAM e processador Intel Core i5 de 4 núcleos.

Os dados iniciais para o cômputo são lidos de um arquivo texto. Neste arquivo cada linha contém novos dados de entrada. Para cada linha lida são realizadas

1.468.800.000 chamadas a uma função dX, a qual internamente possui um laço ”for”que executa 20 vezes. Isto resulta em um total de 29.376.000.000 de operações para cada linha lida. A equação calculada é:

$$x(t) = 2 * (A\sin(wt) - B\cos(wt)) + Et + \sum_{n=1}^{N} F_n e^{-n\gamma t} + G$$

Figura 2. Equação calculada dentro da função dX

Para cada linha de arquivo foram analisados os tempos de execução do programa usando 1, 2, 3 e 4 núcleos do processador. Os tempos totais de execução são colhidos por meio do comando ”time”do Linux, inserido junto ao comando ”./”, que é usado para executar o arquivo. O programa recebe dois parâmetros. O primeiro se refere ao número de linhas que devem ser lidas do arquivo de entrada. O segundo se refere à quantidade de núcleos que devem ser usados. Desse modo o comando para executar o programa via linha de comando é:

time ./x N n

onde x é o nome do programa, N é o número de linhas do arquivo e n é o número de núcleos.

4. Resultados

As tabelas 1, 2 e 3 mostram os resultados obtidos para 1, 2 e 3 linhas de arquivo, variando a quantidade de núcleos utilizados do processador.

Tabela 1. Tempos de execução para 1 linha de arquivo

Quantidade de núcleos	1	2	3	4
Tempo de execução	35m35s	18m47s	17m13s	14m02s

Tabela 2. Tempos de execução para 2 linhas de arquivo

Quantidade de núcleos	1	2	3	4
Tempo de execução	70m38s	39m59s	34m35s	27m42s

Tabela 3. Tempos de execução para 3 linhas de arquivo

Quantidade de núcleos	1	2	3	4
Tempo de execução	108m0s	56m53s	51m7s	45m59s

Com estes dados em mãos podemos obter o speedup e a eficiência do programa à medida que aumentamos o número de núcleos.

A figura 3 mostra o speedup obtido para 2, 3 e 4 núcleos. A linha reta marca o speedup ideal.

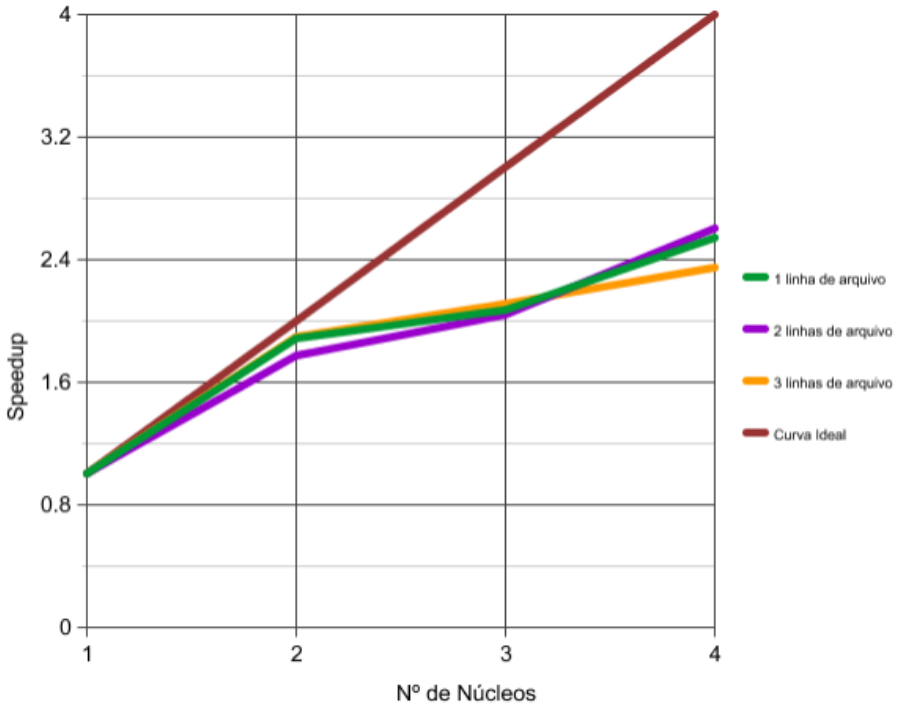

Figura 3. Curvas de speedup para 1, 2 e 3 linhas do arquivo de entrada

A figura 4 mostra a eficiência obtida para 2, 3 e 4 núcleos.

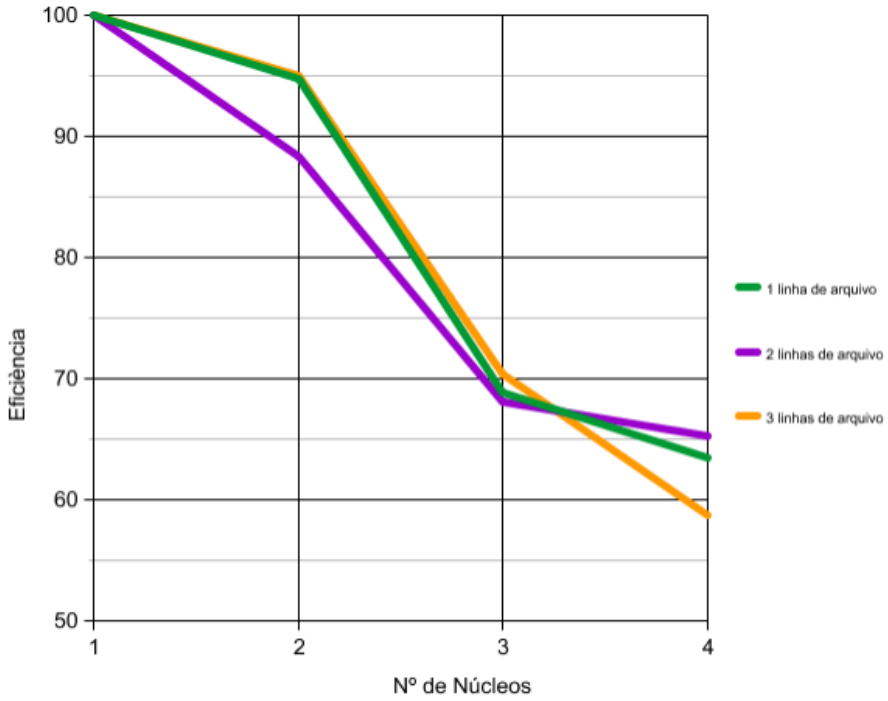

Figura 4. Curvas de eficiência para 1, 2 e 3 linhas do arquivo de entrada

5. Considerações Finais

Neste trabalho foi feita uma análise do desempenho de um programa executando em serial e em paralelo. Tal análise foi feita a partir do tempo necessário para a execução do programa utilizando 1, 2, 3 ou 4 núcleos, variando também a quantidade de linhas lidas do arquivo de entrada.

Na análise foi possível observar que apesar de o tempo necessário para completar o cálculo continuar caindo, à medida que aumentados o número de núcleos a eficiência do uso de recursos pelo programa cai.

Referências

Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). *Introduction to Parallel Computing*. Pearson Education.

Hedges, L. (2016). Real world aws scalability. AWS Compute Blog.

Lantz, S. (2015). Scalability. Workshop: High Performance Computing on Stampede.

Pacheco, P. S. (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann.

Sharcnet (2017). Measuring parallel scaling performance. Sharcnet.