Estudo do desempenho do algoritmo de seleção negativa aplicado à detecção de *builds* não-determinísticas

Daniel Matrone[1], **Roberto Tsuneki**[1], **Rodrigo P. Pasquale**[1], **Calebe P. Bianchini**[1]

[1]Faculdade de Computação de Informática – Universidade Presbiteriana Mackenzie (UPM)
São Paulo, Brasil

Resumo. *O uso de builds, que são entregas automatizadas de software e prática comum em ambientes de desenvolvimento ágil, tem como objetivo compilar, empacotar, testar e entregar diferentes versões de um sistema - o que pode resultar em sucesso ou falha, sendo este último investigado manualmente. Por meio do Algoritmo Bioinspirado de Seleção Negativa, as falhas podem ser identificadas de maneira automática, com boa precisão dado um número de detectores. Porém, o tempo dessa automação ainda é elevado. O presente artigo tem como objetivo apresentar os estudos dos pontos críticos encontrados nesse algoritmo, sendo que a solução atual reduz o tempo total de execução sequencial de* $119,13$*s para* $30,44$*s com 48 threads.*

1. Introdução

Uma *build* é o processo de construção automatizado das etapas de compilação, de execução de testes, de empacotamento e de entrega de sistemas [Cao et al. 2017]. Durante todo esse processo, uma *build* pode validar o código fonte e obter um *feedback* com relação aos processos de testes de um software.

De forma geral, uma etapa do processo de *build* pode resultar em sucesso ou falha. Caso resulte em falha, essa falha é propagada para as demais etapas e, consequentemente, resulta na falha do processo todo. Tais falhas podem ser causadas por motivos externos ao código, como, por exemplo, falha de conexão ao instalar os pacotes e dependências necessárias para uma das etapas do processo de *build* [de Lima Costa 2020], dificultando a identificação dessa falha [Labuschagne et al. 2017]. A fim de identificar esta falha de forma automatizada, pode ser aplicado um algoritmo bio-inspirado que identifica os resultados não-determinísticos como anomalias ajudando a equipe de desenvolvimento a entender os motivos de falha de uma *build* [de Castro 2002].

A solução utilizada para a detecção de *builds* anômalas foi o Algoritmo de Seleção Negativa devido à sua eficiência para detecção de *builds* não-determinística, com uma taxa próxima de 99% [de Lima Costa et al. 2019]. Porém, os custos computacionais de uso de CPU para a execução de experimentos mais densos utilizando esse algoritmo inviabiliza a solução atual, conforme descrito em [de Lima Costa et al. 2019].

Este trabalho tem como objetivo apresentar as otimizações feitas no ASN, bem como a paralelização do algoritmo utilizando OpenMP.

2. Algoritmo de Seleção Negativa (ASN)

O Algoritmo bio-inspirado de Seleção Negativa (ASN) para a identificação de anomalias consiste em duas fases. A primeira é uma fase de treinamento e a outra fase é de detecção

de anomalias. Durante a fase de treinamento, são gerados detectores de forma aleatória seguindo critérios de criação [Ji and Dasgupta 2009]: detectores válidos são aqueles que não detectam um outro detector. Ao atingir um numero pré-definido de detectores, é encerrada a fase de treinamento e inicia-se a segunda etapa.

A fase de detecção consiste em identificar se um objeto desconhecido é classificado como uma anomalia através do nível de similaridade entre o objeto e os detectores. Caso essa similaridade seja maior que um limiar, o objeto é considerado uma anomalia.

Por exemplo, pode-se assumir que cada detector reconhece um conjunto de anomalias em uma vizinhança de raio pré-estabelecido de tal forma que essa distribuição pode ser representada por meio da Figura 1 [Ji and Dasgupta 2009, de Lima Costa et al. 2019].

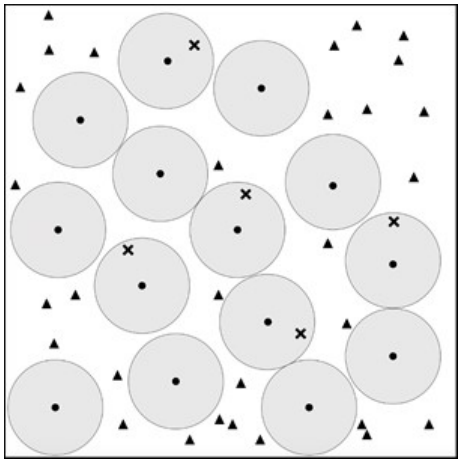

Figura 1. Imagem dos detectores (pontos pretos e seus respectivos raios), das anomalias (x), e dos padrões normais (triângulos).

3. Computação Paralela

Uma maneira de melhorar o desempenho de um algoritmo é reprojetá-lo para o uso de paralelismo e *threads* e aproveitar a arquitetura de um sistema de memória principal compartilhada entre os processadores por meio de uma rede de interconexão interna [Pas et al. 2017]. OpenMP é a combinação (i) de diretivas de compilação que definem e caracterizam o paralelismo; (ii) de bibliotecas disponíveis em tempo de compilação e execução, bem como (iii) de variáveis de ambientes para os ajustes da execução paralela em conjunto com o sistema operacional. Com o OpenMP é possível construir *threads* que podem acessar toda a memória principal de forma compartilhada através do modelo de execução *fork*/*join*.

4. Metodologia

A estratégia inicial adotada para o Algoritmo de Seleção Negativa foi instrumentar o código para descobrir quais os pontos críticos do desempenho do algoritmo. Esses experimentos utilizaram os mesmo cenários apresentados em [de Lima Costa et al. 2019] como, por exemplo, a utilização de 64.000 detectores em uma versão do código desenvolvida em C++[1]. Observou-se que o a função responsável pela geração aleatória dos detectores, denominada *generateDetectors()*, consumia mais de 90% do tempo de execução.

[1]Veja mais em `https://github.com/hpc-fci-mackenzie/FlakyBuild`

A estratégia utilizada para aumentar o desempenho desse bloco foi paralelizar totalmente a geração de detectores e, durante sua verificação, utilizava-se uma seção crítica para que as estruturas de armazenamento dos detectores selecionados não fosse corrompida. Apesar desse controle de coerência de dados, optou-se pelo relaxamento no controle de geração de detectores, sendo que a quantidade gerada poderia ser maior que solicitada em, no máximo, a quantidade de *threads* utilizadas - as estruturas de controles garantem que nunca seriam menos do que o especificado nos arquivos de configuração. Assim, por fim, ainda foi necessário um último ajuste no tamanho do conjunto de detectores gerados para manter a coerência com o tamanho que havia sido solicitado.

Na construção o experimento, foi utilizado o OpenMP 4.5 para a implementação dessa estratégia utilizando-se as diretivas *#pragma omp parallel* e *#pragma omp critical*. O experimento foi realizado em dois equipamentos denominados: Local (1x AMD(R) Ryzen(R) 7 1700x CPU @ 3.40GHz 8c/16t, 16GB RAM) e MackCloud (2 x Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz 12c/24t, 128Gb RAM). Os resultados do *speedup* e eficiência estão representados na Figura 2.

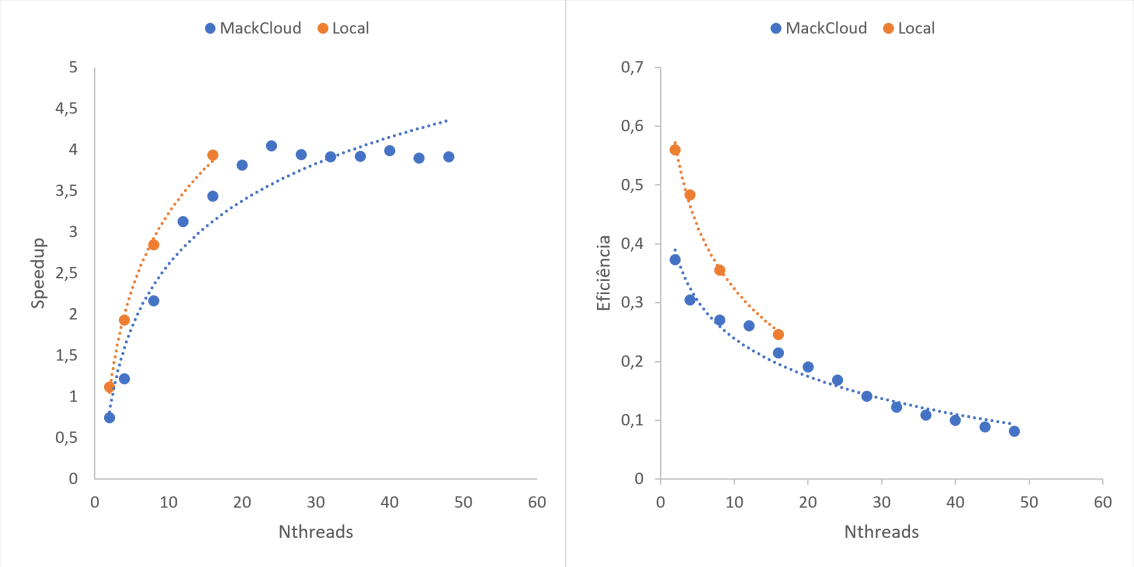

Figura 2. Gráfico de *speedup* vs número de *threads*. Fonte: Autoria própria

O *speedup* máximo obtido nas medições na solução paralela proposta foi de $4,04$ com 48 *threads* no MackCloud e $3,93$ com 16 threads na máquina local. Esses valores foram obtidos com o uso de *threads* lógicas dos processadores (*hyperthreading*).

Na própria Figura 2, o gráfico de eficiência do algoritmo também é apresentado para o mesmo experimento realizado. Fica evidente que os maiores *speedups* alcançados possuem uma baixíssima eficiência. Na verdade, nem mesmo o experimento com a quantidade de *threads* sendo coincidentes com a quantidade de cores (e, portanto, não utilizando *hyperthreading*), a eficiência foi satisfatória. A eficiência é a razão entre o *speedup* para um número de *threads* e a respectiva quantidade de *threads*. Ela mede o grau de aproveitamento dos processadores disponíveis.

5. Considerações Finais

A estratégia utilizada para melhorar o desempenho do Algoritmo de Seleção Negativa trouxe resultados apenas no tempo total de execução (de $119,13$s para $30,44$s, com 48 *threads*). Percebeu-se que a sincronização necessária para o controle das estruturas de armazenamento comprometeram a escalabilidade da solução, algo inicialmente previsto. Como próximos passos pretende-se aplicar outras técnicas para minimizar o impacto dos pontos de sincronização, bem como experimentar outras soluções, como produtor/consumidor [Pas et al. 2017].

Agradecimentos

Os autores agradecem ao MackCloud[2], Centro Multidisciplinar de Computação Científica e Nuvem da Universidade Presbiteriana Mackenzie, ao MackPesquisa, ao CNPq, à FAPESP pelo apoio financeiro.

Referências

Cao, Q., Wen, R., and Mcintosh, S. (2017). Forecasting the duration of incremental build jobs. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

de Castro, L. N. (2002). *Artificial immune systems: a new computational intelligence approach*. Springer.

de Lima Costa, J. C. (2020). Uma heurística de rotulação de builds com resultado não determinístico. Master's thesis, Programa de Pós-Graduação em Engenharia Elétrica e Computação - Universidade Presbiteriana Mackenzie.

de Lima Costa, J. C., de Castro, L. N., and de Paula Bianchini, C. (2019). Sensitivity analysis of the negative selection algorithm applied to anomalies identification in builds. In *2019 XLV Latin American Computing Conference (CLEI)*, pages 1–6.

Ji, Z. and Dasgupta, D. (2009). V-detector: An efficient negative selection algorithm with "probably adequate" detector coverage. *Information Sciences*, 179(10):1390–1406.

Labuschagne, A., Inozemtseva, L., and Holmes, R. (2017). Measuring the cost of regression testing in practice: a study of java projects using continuous integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 821–830.

Pas, R. v. d., Stotzer, E., and Terboven, C. (2017). *Using OpenMP - the next step: affinity, accelerators, tasking, and SIMD*. the MIT Press.

[2] `https://mackcloud.mackenzie.br`