AUMENTO DO DESEMPENHO DO ALGORITMO DE SELECÇÃO NEGATIVA APLICADO À DETECÇÃO DE BUILDS NÃO-DETERMINÍSTICAS

Roberto Tsuneki Sá Freire e Calebe de Paula Bianchini

Apoio:PIBIC Mackpesquisa

RESUMO

Em ambientes de desenvolvimento ágil uma estratégia para entregas mais rápidas é a utilização de *builds*, sendo esta um processo automatizado de compilação, execução de testes (também chamado de certificação), empacotamento e entrega de sistemas. Cada etapa de uma *build* pode resultar em sucesso ou falha (ou estados intermediários de execução, dentre outros que podem ser definidos). No caso de falha, esta é propagada para o resultado da execução geral, necessitando de uma investigação manual do problema, o que pode exigir um grande período de tempo. As falhas de um sistema automatizado podem estar relacionadas a eventos externos como falhas nos recursos de software, atrasos em conexões, processos indefinidos, tempo de execução excedido, dentre outras falhas que, neste caso, são denominadas não-determinísticas. O Algoritmo Bio-Inspirado de Seleção Negativa pode ser utilizado para identificar, no caso de falha da *build*, se existe uma falha não-determinística de forma automatizada com uma alta taxa de detecção dado uma quantidade de detectores. Apesar da eficiência para detecção do Algoritmo de Seleção Negativa, sua fase de treinamento demanda bastante tempo, sendo justificável o estudo do aumento de desempenho tendo em mente o tempo total na procura de erros. Os resultados obtidos em uma nova solução do Algoritmo de Seleção Negativa conseguiram diminuir o tempo total de 441,6 segundos para 89,8 em um ambiente com 16 *threads*. Em outro experimento, o *speedup* alcançado foi de 4.0

Palavras-chave: Algoritmo de seleção negativa. Computação paralela. *Builds* automatizadas.

ABSTRACT

In agile development environments, a strategy for faster delivery is the use of builds, which is an automated process of compilation, test execution (also called certification), packaging, and system delivery. Each step of a build results in success or failure (or intermediate states of execution, among others that can be defined). In case of failure, it is propagated to the result of the general execution, requiring a manual investigation of the problem, which may require a great amount of time. The failures of an automated system can be related to external events such as failures in software resources, delays in connections, undefined processes, exceeded execution time, among other failures that, in this case, are called non-deterministic. The Bio-Inspired Negative Selection Algorithm can be used to identify, in case of a build failure, if there is a non-deterministic failure in an automated way with a high detection rate given many

detectors. Despite the efficiency for detecting the Negative Selection Algorithm, its training phase demands a great amount of time, being justifiable to study the performance increase by keeping in mind the time taken in looking for errors. The results obtained by a new Negative Selection Algorithm solution decreased the total time from 441,6 seconds to 89,8 in an environment with 16 threads. In another experiment, the speedup achieved was 4.0.

Keywords: Negative Selection Algorithm. Parallel computing. Automated builds.

1. INTRODUÇÃO

O 13º Relatório Anual do Estado Ágil (VERSIONONE, 2019) aponta que 97% das empresas participantes utilizam metodologias de desenvolvimento ágil em suas organizações. Esse aumento de metodologias ágeis exige que o time de engenharia a tenha que se dedicar cada vez mais ao uso de práticas de automação e entrega de produtos com mais eficiência e qualidade. Essa automação demanda que entregas do produto gerado sejam frequentes. Por isso, usam-se práticas e processos de integração frequente de código para sustentar a automação visando alcançar, inclusive, o sucesso do modelo ágil (STOLBERG, 2009).

Integração contínua é o nome dado à prática de, dada uma mesma base de código-fonte, unificar diferentes códigos de diversos desenvolvedores (que, ao mesmo, tempo desenvolvem com um mesmo objetivo) de maneira automática. (FOWLER; FOEMMEL, 2006). Um *build* é o processo automatizado de compilação, execução de testes (também chamado certificação), empacotamento e entrega de sistemas (CAO; WEN; MCINTOSH, 2017). Um *build* pode resultar em sucesso ou falha (além dos estados intermediários de execução, dentre outros que podem ser definidos). Um *build* permite à equipe de desenvolvimento avaliar o código integrado e obter feedback antecipado quanto aos processos de integração e construção de código.

A quantidade elevada de processos automatizados traz benefícios aos times de engenharia. Porém, quando alguma das etapas falha, essa falha é propagada para o resultado da execução geral, necessitando, nesses casos, uma investigação manual do problema. Alguns estudos apontam que as falhas apresentadas no processo automatizado podem estar relacionadas a eventos externos, como falha nos recursos de software, atrasos em conexões, processos indefinidos, tempo de execução excedido, dentre outras falhas que, neste caso, são denominadas não-determinísticas (COSTA, BIANCHINI, DE CASTRO, 2019).

O esforço manual na inspeção das falhas, especialmente em *builds*, provenientes de situações externas justificam o uso de algoritmos de aprendizado de máquina (COSTA, BIANCHINI, DE CASTRO, 2019). Em particular, ao usar algoritmos de classificação é possível automatizar essa tarefa de identificação por meio da criação de modelos que classificam os resultados não-determinísticos como anomalias (DE CASTRO; TIMMIS, 2002). Por isso, usar um Algoritmo de Seleção Negativa (ASN), um algoritmo bio-inspirado, pode ser uma ferramenta eficaz para classificar automaticamente os resultados de *builds* com comportamento não determinístico antes mesmo da inspeção detalhadas da causa raiz dessas mesmas falhas.

O ASN se mostrou eficiente para a detecção de *build* não-determinísticas, com taxa de detecção por volta de 99% (COSTA, BIANCHINI, DE CASTRO, 2019). Portanto, a alta taxa

de detecção, alinha à necessidade de diminuição do esforço manual de inspeção de *builds* pela equipe de desenvolvimento, justifica o refinamento da versão atual do ASN aplicado à identificação de *build* não-determinísticas. Neste refinamento estão previstos a melhoria das métricas aplicadas ao processo de treinamento (como, por exemplo, desempenho e *speedup*, dentre outros) e ao processo de detecção (como, por exemplo, acurácia, taxa de detecção, falsos alarmes, dentre outros).

Apesar dos resultados promissores para a detecção de *build* não-determinística apresentados pelo ASN, o processo de treinamento desse algoritmo bio-inspirado ainda demanda bastante tempo (COSTA, BIANCHINI, DE CASTRO, 2019). Além disso, a variação das configurações do ASN, que pode apresentar resultados mais precisos na fase de detecção dessas *builds*, demandará ainda mais recursos computacionais para a fase de treinamento. É importante mencionar que, dentre essas variações estão parâmetros como o tipo de geometria utilizada, a combinação de geometria de tamanhos diferentes, dentre outros parâmetros que podem ser utilizados no ASN (JI et al, 2005; JI; DASGUPTA, 2009).

Sendo assim, o objetivo deste trabalho é apresentar os resultados do aumento do desempenho do Algoritmo de Seleção Negativa aplicado a *builds* utilizando um conjunto de parâmetros especializados. Foram alcançados também alguns objetivos secundários, como a adoção de modelos de computação paralela para aumentar o desempenho computacional e diminuir o tempo de resposta.

2. DESENVOLVIMENTO DO ARGUMENTO

Esta seção apresenta os principais tópicos relacionados aos resultados desse projeto de pesquisa.

2.1. Algoritmo de Seleção Negativa

Uma forma de realizar tarefas de análise e monitoramento de ambientes, detecção de anomalias, reconhecimento de padrões, otimização, dentre outros, é por meio da aplicação dos conceitos de Sistemas Imunológicos Artificiais (SIA), ou seja, sistemas inspirados na capacidade dos seres vivos de identificar anomalias presentes em seus corpos. A premissa dos SIA consiste no uso de modelos baseados no funcionamento do sistema imunológico, principalmente dos vertebrados, para a criação de algoritmos (DE CASTRO; TIMMIS, 2002).

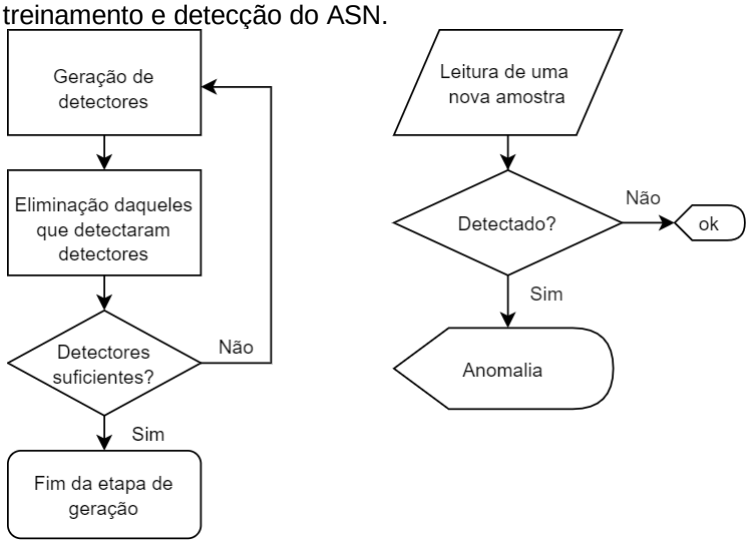
Uma das técnicas de SIA utilizada para identificação de anomalias é o Algoritmo de Seleção Negativa (ASN) (ZENG et al., 2007). O ASN é inspirado no funcionamento do sistema imunológico que (i) diferencia, por meio das células T, células ou moléculas não pertencentes

ao corpo, chamadas de anomalias, e (ii) filtra as células T na glândula Timo (FORREST, 1994). Em tal processo podem ocorrer 3 ações:

- não seleção: ausência de afinidade entre a célula T e o peptídeo;
- seleção positiva: a afinidade entre a célula T e o peptídeo causa a ativação das células T; e
- seleção negativa: a afinidade entre a célula T e o peptídeo causa a morte ou anergia das células T.

Basicamente, o ASN consiste em duas fases: uma fase de treinamento e uma fase de detecção de anomalias. Ambas as etapas estão demonstradas na Figura 1.

Figura 1. Fases de treinamento e detecção do ASN.

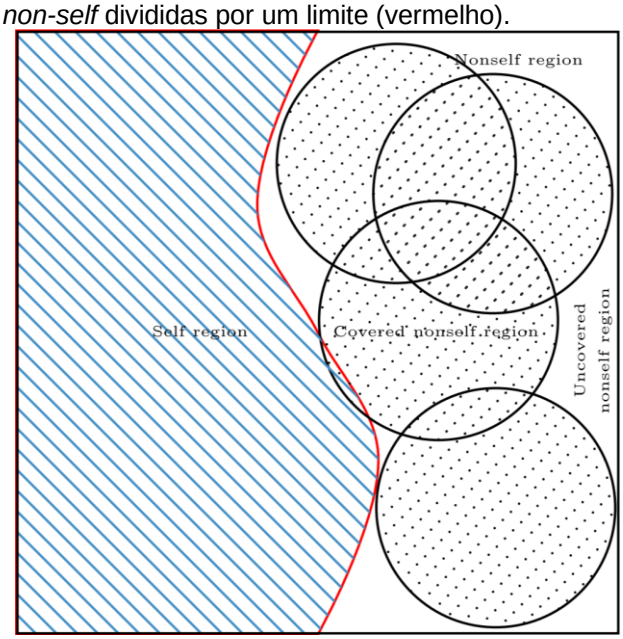


Fonte: FORREST et al, 1994

A fase de treinamento consiste na definição de duas regiões chamadas de *self* e *non-self* (JI; DASGUPTA, 2009). A região *self* consiste em um conjunto de objetos já rotulados como normais e compreende um espaço onde qualquer objeto será considerado normal. A região *non-self* será o espaço onde os detectores serão gerados. Essa separação pode ser percebida na Figura 2.

Ainda nessa fase de treinamento há a geração aleatória de detectores e sua validação. Neste caso, ela consiste em avaliar se há algum detector reconhecendo outro detector como anômalo, e eliminando aqueles que não passaram nessa validação. Ao atingir um número predefinido de detectores válidos é encerrada a fase de treinamento e todos os detectores existentes se tornam detectores de anomalias.

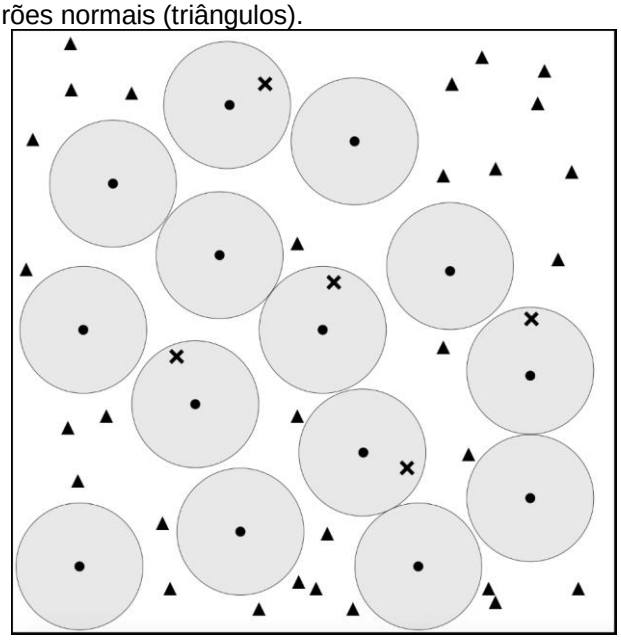
Figura 2. Regiões *self* e *non-self* divididas por um limite (vermelho).



Fonte: JI; DASGUPTA, 2009

A fase de detecção consiste em identificar se um objeto não rotulado é ou não uma anomalia através do nível de similaridade entre os detectores e o objeto avaliado. Caso esta similaridade seja superior a um limite pré-estabelecido, o objeto será rotulado como uma anomalia. Assumindo que cada detector reconhece um conjunto de anomalias em uma vizinhança de raio pré-estabelecido, a Figura 3 ilustra um resultado de uma aplicação do ASN (JI; DASGUPTA, 2009).

Figura 3. Imagem dos detectores (pontos pretos e seus respectivos raios) criados para detectar anomalias (x), e dos padrões normais (triângulos).



Fonte: COSTA, BIANCHINI, DE CASTRO, 2019

2.2. Computação Paralela

O aumento de frequência de *clock* em sistemas de um único processador para aumentar o desempenho gerou um problema de geração excessiva de calor, visto que a quantidade de calor e a frequência do *clock* estão intimamente ligadas, e os sistemas de refrigeração a ar estão chegando a seu limite. Então, para atingir um aumento de desempenho sem depender de melhorias dos processadores ou sistemas de refrigeração, foi aumentado o número de unidades de processamento em um único *chip*, tornando necessário a aplicação do conceito de paralelismo, ou seja, dividir a tarefa em subtarefas e executando essas de forma concorrente (PACHECO, 2011).

A maneira como será realizada a divisão de está intimamente relacionada com a forma como a memória é utilizada. Devido a isso, pode-se dividir os sistemas paralelos em dois tipos em relação ao uso da memória: sistemas de memória compartilhada e sistemas de memória distribuída. No sistema de memória compartilhada, os processadores têm acesso a uma memória principal que é compartilhada com todos por meio de uma intercomunicação interna. Por outro lado, no sistema de memória distribuída, cada processador possui sua própria memória principal e utiliza uma rede externa para se comunicarem entre si (SANTANA, 2020).

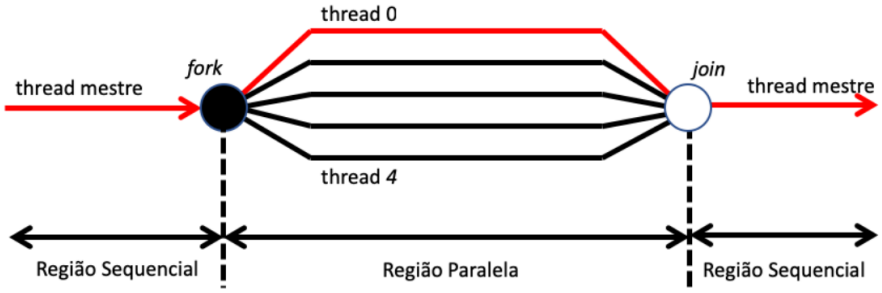
Uma maneira de desenvolver um algoritmo paralelo com uso de memória compartilhada e utilizando do OpenMP (VAN DER PAS, STOTZER, TERBOVEN, 2017). O OpenMP é a combinação (i) de diretivas de compilação, que definem e caracterizam o paralelismo; (ii) de bibliotecas disponíveis em tempo de compilação e execução, bem como (iii) de variáveis de ambientes para os ajustes da execução paralela em conjunto com o sistema operacional. Com OpenMP é possível ter a construção de threads que podem acessar toda a memória do processo de forma compartilhada.

O OpenMP faz uso do modelo de execução *fork*/*join*, sendo que a região sequencial é executada pela *thread* mestre. Quando essa *thread* encontra uma região paralela, outras threads serão iniciadas (*fork*) para a execução da região paralela. Quando todas as *threads* terminarem sua execução e chegarem ao final de uma região paralela, a *thread* mestre retornará à execução (*join*) da parte sequencial. A Figura 4 representa modelo *fork*/*join* para o OpenMP (BIANCHINI, VILASBÔAS, DE CASTRO, 2019).

Como um dos principais objetivos de uma solução paralela é aumentar o desempenho, pode-se verificar o aumento alcançado utilizando algumas métricas, como o *speedup*. O *speedup* é a razão entre a medição da execução do algoritmo paralelo e de uma solução sequencial equivalente, dada pela Equação 1 (PACHECO, 2011):

$$S_p(n) = \frac{T^*(n)}{T_p(n)} \quad (1)$$

Figura 4. Modelo *fork/join* de execução do OpenMP com 5 *threads*.



Fonte: BIANCHINI, VILASBÔAS, DE CASTRO, 2019

onde $S_p(n)$ é o *speedup*, $T_p(n)$ é a medição da execução do algoritmo paralelo com p processadores com a entrada n, e $T^*(n)$ é a medição da execução do algoritmo sequencial equivalente para a mesma entrada n. O *speedup* mostra quantas vezes mais rápida foi a execução do algoritmo paralelo em relação ao algoritmo sequencial.

Uma das principais métricas derivadas do *speedup* é a eficiência, ou seja, qual fração do total de processadores disponíveis foi utilizado durante o experimento. O cálculo da eficiência de um *speedup* é dado pela Equação 2 (PACHECO, 2011):

$$E = \frac{S_p(n)}{p} \tag{2}$$

onde E é a eficiência, $S_p(n)$ é o *speedup* do experimento e p é o número de processadores utilizados.

A Lei de Amdahl mostra qual o *speedup* máximo que pode ser atingido na paralelização de um algoritmo sequencial utilizando uma quantidade determinada de processadores. Essa lei constata que uma parte do algoritmo paralelo será sempre executado de forma sequencial. O cálculo da Lei de Amdahl é dado pela Equação 3 (PACHECO, 2011):

$$S_p(n) = \frac{T^*(n)}{\tau T^*(n) + \frac{1-\tau}{p} T^*(n)} = \frac{1}{\tau + \frac{1-\tau}{p}} \leq \frac{1}{\tau} \tag{3}$$

onde $T^*(n)$ é o tempo de execução do algoritmo sequencial, p é o número de processadores e τ é a fração do algoritmo que é executado de forma inerentemente sequencial.

Em contrapartida, a Lei de Gustafson explica como se comporta o *speedup* de um algoritmo cuja parte sequencial tem um tempo constante, independentemente do tamanho do problema. Considerando l como o tempo da parte sequencial e $l(n,p)$ o tempo para a resolução do problema de tamanho n com p processadores, o *speedup* é definido pela Equação 4 (PACHECO, 2011):

$$S_p(n) = \frac{l + l(n,1)}{l + l(n,p)} \tag{4}$$

2.3. 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 mesmos cenários apresentados em (COSTA, BIANCHINI, DE CASTRO, 2019) e (COSTA, 2020) como, por exemplo, a utilização de 64.000 detectores em uma versão do código desenvolvida em C++[1]. Observou-se nestes experimentos que o a função responsável pela geração aleatória dos detectores, denominada `generateDetectors()`, consumia mais de 90% do tempo de execução do algoritmo.

Para aumentar o desempenho desse bloco decidiu-se paralelizar totalmente a geração de detectores. Durante a verificação do detector gerado, utilizava-se uma seção crítica para que as estruturas de armazenamento dos detectores selecionados não fossem corrompidas. 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 do que a 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 foi solicitado pelo usuário do ASN.

Foi adotado o OpenMP 4.5 para a implementação dessa estratégia utilizando as diretivas `#pragma omp parallel` e `#pragma omp critical` (PACHECO, 2011). A primeira diretiva foi utilizada para a criação de diversas *threads*, conforme descreve o modelo *fork/join*, e pode ser visto na Figura 5. Cada thread cria aleatoriamente os detectores (função `randomVector()`) de forma paralela e, conforme a validação desse detector (a condição `if (!geometry.matches(...))`), ele é guardado e a quantidade de detectores válidos aumenta.

A segunda diretiva faz o controle de seção crítica para que a centralização da quantidade de controle não se torne inconsistente, conforme mostra a Figura 6. No primeiro uso, o container de detectores é utilizado na seção crítica para armazenar um novo detector encontrado e válido (`detectors->push_back(detector)`); no segundo uso, a variável local responsável pelo controle do laço de repetição geral é também atualizada em uma seção crítica (`size = detectors->size()`).

Em seguida, o experimento foi realizado em dois tipos de processadores diferentes. Para facilitar o entendimento, eles foram denominados como *Local*, que utiliza o processador AMD(R) Ryzen(R) 7 1700x CPU @ 3.40GHz 8c/16t, com 16GB de RAM; e *MackCloud*, que

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

possui dois processadores Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz 12c/24t, com 128Gb de RAM. Esses experimentos foram mensurados e os cálculos de *speedup* e eficiência foram calculados, conforme pode ser visto na Tabela 1 para o processador Local e na Tabela 2 para o processador MackCloud.

Figura 5. Trecho da solução do ASN utilizando o modelo fork/join com OpenMP 4.5.

```
std::vector<datatype *> *Detector::generateDetectors()
{
    std::vector<datatype *> *detectors = new std::vector<datatype *>();
    std::cout << "Generating detectors..." << std::endl;
    size_t size = 0;
#pragma omp parallel
{
    do
    {
        ...
        datatype *detector = new datatype[fConfigFile.getProblemSize()];
        randomVector(detector);
        if (!geometry.matches(detector, fSelfDataset, fConfigFile.getMinDist
()))
        {
            ...
        }
    } while (size < fConfigFile.getMaxDetectors());
}

    ...
    return detectors;
}
```

Fonte: Próprio autor.

Figura 6. Trecho da solução do ASN utilizando seção crítica.

```
            ...
            if (!geometry.matches(detector, copy, 0.0))
            {
#pragma omp critical
{
                detectors->push_back(detector);
}
                ...
            }
            ...
#pragma omp critical
{
        size = detectors->size();
}
        ...
```

Fonte: Próprio autor.

Tabela 1. Medição de tempo para o processador Local.

Quantidade de *threads*	**Tempo médio de execução (s)**	***Speedup***	**Eficiência**
1	441,6	0,8	0,8
2	315,7	1,1	0,6
4	182,8	1,9	0,5
8	124,3	2,8	0,4
16	89,8	3,9	0,2

Fonte: Próprio autor.

Tabela 2. Medição de tempo para o processador MackCloud.

Quantidade de *threads*	**Tempo médio de execução (s)**	***Speedup***	**Eficiência**
1	119,1	1,0	1,0
2	159,7	0,7	0,4
4	97,9	1,2	0,3
8	55,1	2,2	0,3
12	38,1	3,1	0,3
16	34,7	3,4	0,2
20	31,3	3,8	0,2
24	29,4	4,0	0,2
28	30,2	3,9	0,1
32	30,4	3,9	0,1
36	30,4	3,9	0,1
40	29,9	4,0	0,1
44	30,6	3,9	0,1
48	30,4	3,9	0,1

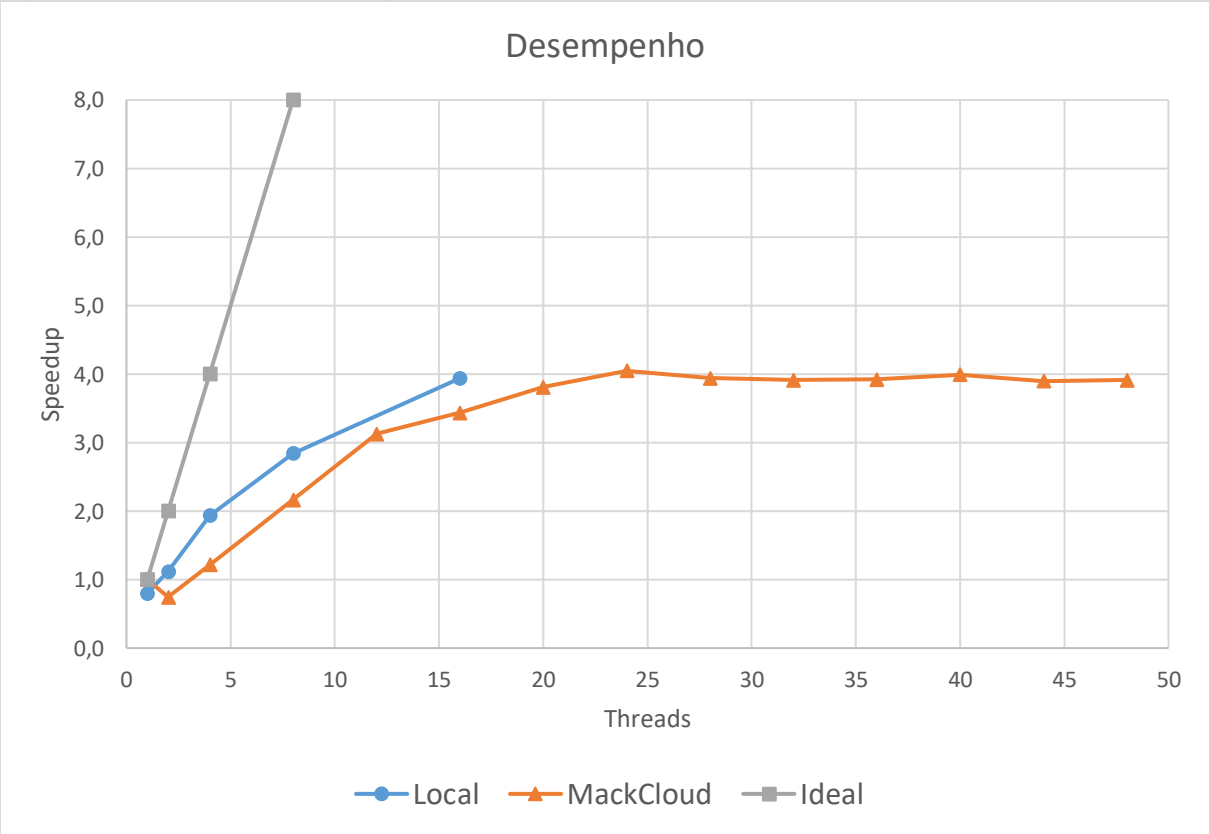
Fonte: Próprio autor.

O *speedup* máximo calculado a partir da execução da solução paralela no processador *Local* foi de 3,9 com 16 *threads*. Apesar disso, fica evidente a queda da eficiência conforme aumentamos a quantidade de *threads*. Isso se deve ao ponto de sincronização existente, especialmente, na adição de novos detectores na estrutura de dados, conforme mostra a Figura 5. Além disso, a partir da solução sequencial original, que possui um tempo total de execução de 353,4 segundos, o tempo da solução paralela com apenas 1 *thread* apresenta

um *overhead* inesperado, impactando negativamente a solução – isso pode ser perceptível tanto pelo *speedup* (0,8) quanto pela eficiência (0,8) apresentados na Tabela 1.

Ao coletar os resultados do experimento nos processadores *MackCloud*, o *speedup* máximo calculado foi de 4,0 com 24 e 40 *threads*. Essa estagnação no desempenho ficou mais evidente em função da quantidade de *cores* total existentes nesse ambiente. Assim, a partir de 24 *threads*, o uso da tecnologia de *hyper-threading* (ou *threads* lógicas) não colaborou com o desempenho dessa solução no ambiente. Um ponto importante a destacar nesse experimento é que a solução original sequencial não foi utilizada como base do cálculo do *speedup*, mas tentou-se observar o desempenho a partir da versão com 1 thread – já que sabemos que existe um *overhead*. Fica evidente também que o tempo total absoluto das soluções são bem diferentes, especialmente com 1 *thread*: 441,6 segundo no processador *Local* e 119,1 segundos no processador *MackCloud*. Maiores detalhes sobre o desempenho das duas soluções também podem ser observados na Figura 7, onde a curva com marcações em quadrado indica o *speedup* ideal, a curva com marcações em círculos apresenta o *speedup* do experimento no processador *Local* e a curva com marcações em triângulo apresenta o *speedup* do experimento no processador *MackCloud*.

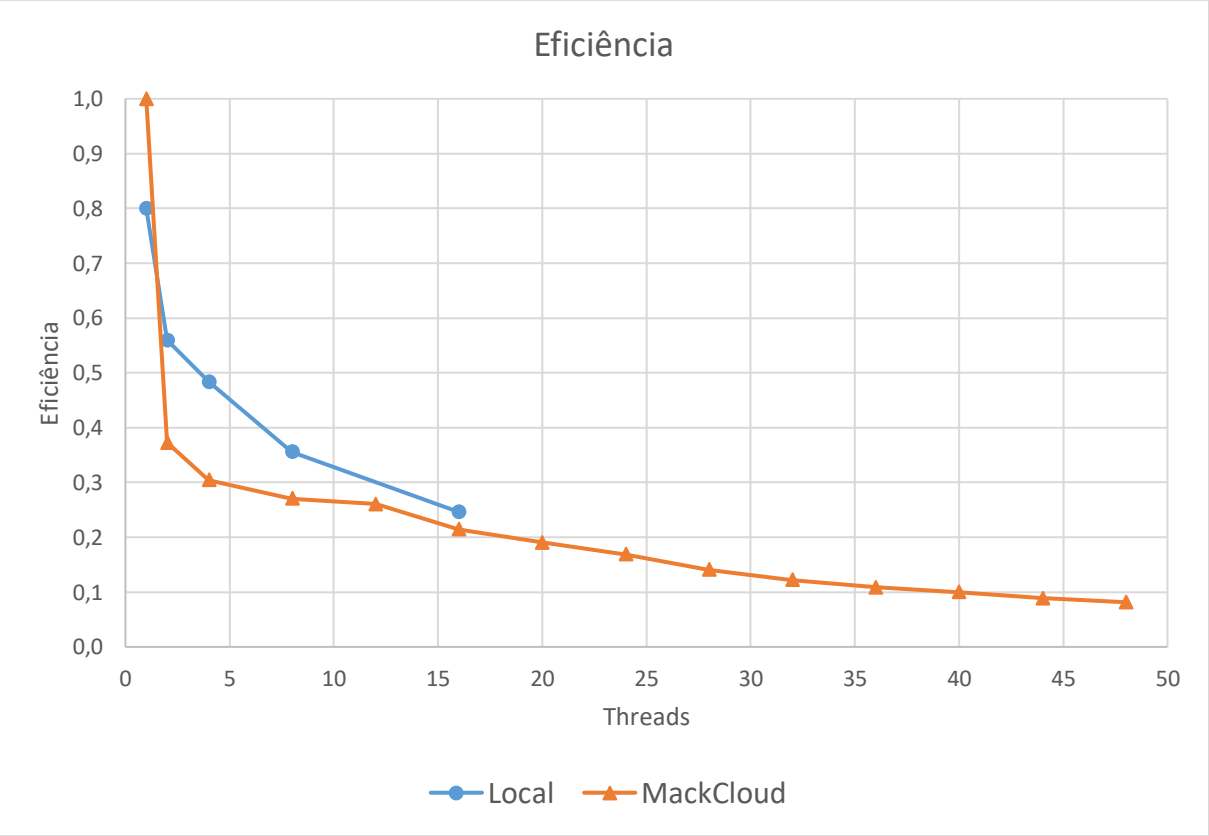
Figura 7. Desempenho das soluções *Local* e *MackCloud*.



Fonte: Próprio autor

Também é possível avaliar a eficiência das soluções por meio da Figura 8, onde a curva com marcações em círculos apresenta a eficiência da solução no processador *Local* e a curva com marcações em triângulo apresenta a eficiência da solução no processador *MackCloud*. Percebe-se, novamente, que a estagnação no desempenho, tanto no ambiente *Local* quanto no ambiente *MackCloud*, interferiu intensamente na escalabilidade e, consequentemente, nos cálculos de eficiência da solução paralela.

Figura 8. Eficiência das soluções *Local* e *MackCloud*.



Fonte: Próprio autor

3. CONSIDERAÇÕES FINAIS

A estratégia utilizada para melhorar o desempenho do Algoritmo de Seleção Negativa trouxe resultados tanto em relação ao desempenho, quanto em relação ao tempo total, sendo o *speedup* alcançado por volta de 4,0 e o tempo total de execução por volta de 29,4 segundos (no processador *MackCloud*, com 24 *threads*).

Apesar disso, a sincronização necessária para o controle das estruturas de armazenamento comprometeu a escalabilidade da solução, deteriorando a eficiência da solução: para o mesmo ambiente do processador MackCloud, com 24 *threads*, a eficiência

ficou por volta de 0,2, ou seja, a solução utilizou aproximadamente 20% dos recursos computacionais disponíveis no ambiente.

Os esforços dessa estratégia foram documentados e seus resultados divulgado para a comunidade científica na Escola Regional de Alto Desempenho de São Paulo – ERAD-SP 2021, com a publicação do resumo (MATRONE et al, 2021).

Ainda existe alguns estudos que podem ser feitos a partir dos resultados alcançados neste projeto de iniciação científica, dentre os quais podem ser citados: (i) outras técnicas para minimizar o impacto dos pontos de sincronização; (ii) experimentar outras soluções de paralelismo, como produtor/consumidor (VAN DER PAS; STOTZER; TERBOVEN, 2017); (iii) combinar estratégias diferentes para o cálculo de distância entre detectores (JI; DASGUPTA, 2009); (iv) modificar a geração dos detectores para tamanhos e formas aleatórias (JI; DASGUPTA, 2009); dentre outras estratégias tanto para a solução do ASN quanto para sua paralelização disponíveis na literatura.

3.1. Agradecimentos

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

Agradecemos também aos alunos e ex-alunos que participaram ativamente das discussões relacionadas a este trabalho, em especial: Daniel Matrone, Rodrigo Pigatto Pasquale e Fabrício Gomes Vilasbôas.

4. REFERÊNCIAS

BIANCHINI, C, VILASBÔAS, F, DE CASTRO, L. Paralelismo de tarefas utilizando OpenMP 4.5. I. 65. 2019.

CAO, Q; WEN, R; MCINTOSH, S. Forecasting the duration of incremental build jobs. In: 2017 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), 2017, Shanghai. China. Proceedings [...]. [S. l.: s. n.], 2017. p. 524-528.

COSTA, J C L. Uma heurística de rotulação de builds com resultado não determinístico. 2020. 59 f. Dissertação (Engenharia Elétrica) - Universidade Presbiteriana Mackenzie, São Paulo

[2] Veja o site em https://mackcloud.mackenzie.br

COSTA, J. C. L; BIANCHINI, C. P.; DE CASTRO, L. N. Análise de Sensibilidade do Algoritmo de Seleção Negativa Aplicado à Identicação de Anomalias em Builds. In: CLEI. 45TH CONFERÊNCIA LATINO-AMERICANA DE INFORMÁTICA., 2019, Cidade do Panamá. Panamá. Proceedings [...]. [S. l.: s. n.], 2019.

DE CASTRO, L N; TIMMIS, J. Artificial immune systems: a new computational intelligence approach. [S. l.]: Springer Science & Business Media, 2002.

FORREST, S. Self-nonself discrimination in a computer. In: 1994 IEEE COMPUTER SOCIETY SYMPOSIUM ON RESEARCH IN SECURITY AND PRIVACY, 1994, Oakland, CA, USA. Proceedings [...]. [S. l.]: IEEE, 1994. p. 202-212.

FOWLER, M; FOEMMEL, M. Continuous integration. Thought-Works). http://www.thoughtworks.com/ContinuousIntegration.pdf, v. 122, p. 14, 2006.

JI, Z; DASGUPTA, D. V-detector: An efficient negative selection algorithm with “probably adequate” detector coverage. In: INFORMATION Sciences. 10. ed. [S. l.]: Elsevier, 2009. v. 179, p. 1390-1406.

JI, Z et al. A boundary-aware negative selection algorithm. In: 9TH INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE AND SOFT COMPUTING, 2005, Benidorm, Espanha. Proceedings [...]. [S. l.: s. n.], 2005.

MATRONE, D.; et al. Estudo do desempenho do algoritmo de seleção negativa aplicado à detecção de builds não-determinísticas. In: ESCOLA REGIONAL DE ALTO DESEMPENHO DE SÃO PAULO (ERAD-SP), 12. , 2021, Evento Online. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2021 . p. 53-56.

PACHECO, P. S. An Introduction to Parallel Programming: 1. ed. San Francisco: Elsevier, 2011.

SANTANA, C. Workload scheduling analysis in geophysical numerical methods. 2020. 78f. Dissertação (Mestrado em Engenharia Elétrica e de Computação) - Centro de Tecnologia, Universidade Federal do Rio Grande do Norte, Natal, 2020.

STOLBERG, S. Enabling agile testing through continuous integration. In: IEEE. 2009 Agile Conference. [S.l.: s.n.], 2009. p. 369–374.

VAN DER PAS, Ruud; STOTZER, Eric; TERBOVEN, Christian. Using OpenMP - the next step: affinity, accelerators, tasking, and SIMD. [S. l.]: The MIT Press, 2017.

VERSIONONE, Collabnet. The 13th Annual State of Agile Report; 2018. [S.l.: s.n.], 2019.

ZENG, J et al. A Feedback Negative Selection Algorithm to Anomaly Detection. In: THIRD INTERNATIONAL CONFERENCE ON NATURAL COMPUTATION (ICNC 2007), 2007, Haikou, China. Proceedings [...]. [S. l.]: IEEE, 2007. p. 604–608.

Contatos: contato.tsuneki@gmail.com e calebe.bianchini@mackenzie.br