RedBlue: um *cluster* para simulações computacionais

Marcelo B. Pedras[1], Euler G. Horta[1], Alexandre R. Fonseca[1]

[1]Programa de Pós-Graduação em Educação - PPGED
Universidade Federal dos Vales do Jequitinhonha e Mucuri (UFVJM)
Diamantina – MG – Brasil

marcelo.pedras, euler.horta, arfonseca}@ict.ufvjm.edu.br

Abstract. *Computational simulations are widely used in scientific research. However, they may require a lot of computational resources. This can be solved by distributing the application between multiple computers. In this way, this work presents the RedBlue platform that allows users with less knowledge in distributed programming to adopt this strategy in computer labs. The platform uses common computers as clusters. To demonstrate the feasibility of the platform, tests were performed with 60 machines, and it was possible to reduce the execution time by 98% for a neural network and 99.3% for a simple genetic algorithm.*

Resumo. *Simulações computacionais são muito utilizadas na pesquisa científica. Porém, podem requerer muitos recursos computacionais. Isso pode ser resolvido distribuindo a aplicação entre vários computadores. Dessa forma, este trabalho apresenta a plataforma RedBlue que possibilita que usuários com menos conhecimentos em programação distribuída possam adotar essa estratégia em laboratórios de informática. A plataforma utiliza computadores comuns como* clusters. *Para demonstrar a viabilidade da plataforma, foram realizados testes com 60 máquinas, sendo possível reduzir o tempo de execução em 98% para uma rede neural e 99,3% para um algoritmo genético simples.*

1. Introdução

Apesar do desenvolvimento expressivo dos computadores pessoais, seus recursos podem ser insuficientes para a execução de alguns problemas computacionais em um tempo aceitável. Geralmente, esses problemas envolvem uma série de cálculos complexos e/ou são muito sensíveis a variação dos parâmetros de entrada. Logo, é comum ter um alto tempo de execução por instância, além de ter que repeti-las até encontrar um conjunto de parâmetros adequado à situação. Essas características normalmente estão presentes em simulações computacionais nas diversas áreas da ciência, e de modo geral, em aplicações de alto custo computacional. O elevado tempo de execução pode inviabilizar ou atrasar a obtenção de resultados.

Uma solução óbvia para reduzir o tempo de execução seria adquirir equipamentos mais robustos, como um supercomputador. Porém, esse tipo de equipamento possui alto custo de aquisição, exige grande investimento em infraestrutura e possui um alto custo de operação [Gholkar et al. 2016]. Logo, é uma opção restrita a grandes centros de pesquisa. Uma alternativa aos supercomputadores

comerciais são os *clusters* de computadores comuns. Um *cluster* de computadores é um sistema distribuído formado por um conjunto de computadores interligados por uma rede de alta velocidade com objetivo de processar pequenas porções da aplicação, chamadas tarefas, de modo colaborativo. No entanto, uma aplicação só terá ganho de desempenho quando aplicada a um *cluster* se for paralelizável e possuir granularidade grossa. Ser paralelizável significa que a aplicação deve ser divisível em partes menores que podem executar simultaneamente em vários computadores. Granularidade grossa significa que o tempo de execução da tarefa deve ser maior que o tempo gasto com a criação, comunicação e sincronização dessas tarefas [Eijkhout et al., 2016]. Além disso, *clusters* possuem peculiaridades quanto à estrutura dos programas executados devido ao ambiente distribuído em vários computadores, necessitando de conhecimentos mais aprofundados em programação paralela e distribuída. O perfil dos potenciais usuários também é uma restrição significativa. No instituto onde se desenvolveu este trabalho, os potenciais usuários normalmente possuem ou estão em formação em engenharia e, em sua maioria, não possuem conhecimento avançado em computação distribuída.

Com o objetivo de permitir que seja possível desenvolver aplicações sobre um *cluster* de computadores, mesmo sem conhecimentos avançados em aplicações distribuídas, apresenta-se a plataforma RedBlue. Ela utiliza uma árvore de sincronização de tarefas onde os nós são representados pelas cores vermelha e azul, o que motivou o nome da plataforma. O RedBlue permite combinar o poder computacional de vários computadores, como os disponíveis em laboratórios de informática, bastando que os usuários definam a precedência das tarefas. Embora a plataforma tenha sido idealizada para utilização em simulações computacionais, ela pode ser utilizada para outras categorias de *software*, desde que seja respeitado o mecanismo de precedência e o formato de comunicação. Dessa forma, espera-se que a plataforma contribua para o desenvolvimento e aprimoramento de pesquisas científicas.

Este artigo está organizado em 6 partes. A Introdução contextualiza o problema da demora na obtenção de resultados em aplicações de alto custo computacional e duas possíveis soluções; o Estado da Arte descreve alguns conceitos importantes e apresenta os trabalhos relacionados; a modelagem da plataforma é detalhada em Materiais e Métodos; os testes executados e os resultados obtidos são mostrados em Experimentos e Resultados e são comentados em Discussão; por fim, na Conclusão são apresentadas as considerações finais e algumas sugestões para trabalhos futuros.

2. Estado da Arte

Simulação computacional é uma metodologia poderosa para projetar e analisar sistemas complexos. A simulação busca representar as características dinâmicas de um sistema real por meio de um modelo computacional [Robinson 2004]. Na literatura encontram-se várias definições para simulação computacional. Por exemplo, “o uso do computador para imitar as operações de um processo do mundo real, considerando os relacionamentos lógicos, estatísticos ou matemáticos desenvolvidos em um modelo” [McHaney 2009, tradução nossa]. Outros autores levam em consideração a simplificação da realidade, a variabilidade do sistema e o propósito da simulação, definindo-a como: “experimentação em um computador da imitação simplificada de um sistema, bem como seu progresso através do tempo, com propósito de melhor entender e/ou melhorar o sistema” [Robinson, 2004, tradução nossa].

A indústria utiliza simulações para projetar e testar produtos, equipamentos e até processos do sistema produtivo antes que esses sejam implementados. Dessa forma é possível identificar muitos dos problemas com antecedência, diminuindo o retrabalho e consequentemente reduzindo custos. Como exemplos de utilização se destacam as indústrias automobilística [Bohn et al. 2013], construção civil [Tang e Ren, 2008], de microprocessadores [Chen et al. 2013], entre outras.

O aumento da complexidade das simulações e a manipulação de um grande volume de dados são dois dos principais desafios ao desenvolvimento de simulações atualmente. Isso motivou o surgimento de propostas que facilitem seu desenvolvimento e ainda assim obtenham maior poder computacional. Gosney et al. (2011) apresentam um mecanismo capaz de otimizar a alocação de tarefas de forma dinâmica em *clusters* multiusuários. Malysiak e Handmann (2014) propõem um *framework* para desenvolvimento de aplicações distribuídas chamado SimpleHydra. Esse possui funcionalidades que tornam possível a utilização tanto de CPUs como GPUs, gerenciamento dos recursos computacionais e adaptação automática ao número de computadores. Goyal et al. (2017) propõem um *framework* para desenvolvimento de aplicações distribuídas no domínio da mineração de dados. Esse utiliza uma linguagem própria de programação chamada DWARDF, que permite que o compilador identifique pontos de código paralelizáveis e os traduza em código C++ paralelizado.

Nesse cenário, percebe-se a importância de facilitar o desenvolvimento de simulações, além de reduzir seu tempo de execução. Dessa forma, este trabalho apresenta a plataforma RedBlue, que foi desenvolvida para obter um maior poder computacional ao se utilizar um *cluster* ao mesmo tempo que reduz a complexidade para desenvolver aplicações que executem sobre ele. Na próxima seção são apresentadas as estratégias, arquitetura, e os recursos utilizados.

3. Materiais e Métodos

Normalmente *clusters* são empregados para tornar possível a obtenção de resultados em um tempo de execução viável. Seu ganho de desempenho depende de quão eficiente é a paralelização da aplicação e a comunicação entre os computadores. Em alguns casos, optar por otimizar ao máximo o tempo de execução implica em uma interface de programação mais complexa. Portanto, será exigido mais conhecimento em programação distribuída por parte do usuário. Devido ao perfil do usuário que a plataforma pretende atender, optou-se por ter uma interface de programação mais amigável. Para isso, separou-se o gerenciamento do *cluster* da aplicação cliente, de forma que o cliente consome os serviços oferecidos pelo *cluster*, ao mesmo tempo que oculta as chamadas remotas. Dessa forma, o usuário não precisa se preocupar com os detalhes de implementação da distribuição das tarefas. Ele precisa apenas compreender o mecanismo de indicação de precedência entre tarefas, que será apresentado na seção seguinte. Para permitir que o cliente possa ser implementado em diversas linguagens de programação, as aplicações de gerenciamento e cliente foram implementadas como processos separados. Técnicas de intercomunicação entre processos foram utilizadas para a troca de mensagens.

Para flexibilizar o uso da plataforma, escolheu-se utilizar apenas as operações de leitura e escrita em terminal para a comunicação entre aplicação cliente e *cluster*. Assim, as informações enviadas do *cluster* para aplicação cliente são recebidas por meio

de uma operação de leitura em terminal (STDIN) no cliente. Analogamente, a aplicação cliente informa o resultado de sua execução para o *cluster* utilizando uma operação de escrita em terminal (STDOUT).

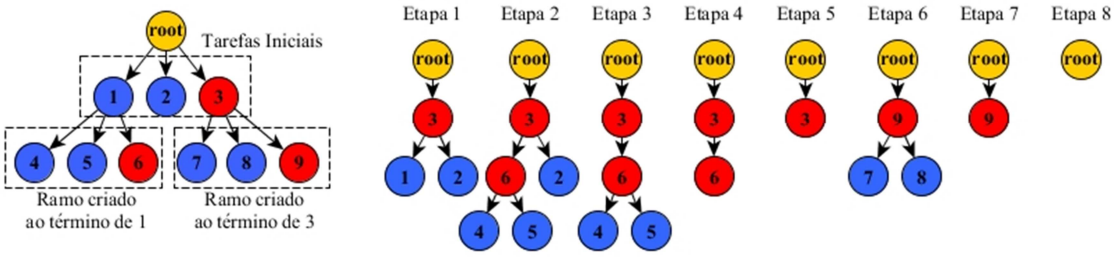
Uma vez que a aplicação cliente não controla explicitamente a distribuição das tarefas, é necessário um meio para que o usuário informe ao *cluster* em que sequência as tarefas devem ser processadas. Para isso, foi criada uma estrutura em forma de árvore para organizar a precedência das tarefas e o fluxo da informação.

3.1. Árvore de tarefas

A vantagem de se utilizar um *cluster* advém da possibilidade de executar diferentes partes do código simultaneamente em diferentes máquinas. Essas porções são chamadas de *jobs*, tarefas. Porém, mesmo aplicações altamente paralelizáveis possuem porções sequenciais de código que normalmente funcionam como pontos de sincronização. Para definir as dependências de execução, organizou-se as tarefas em uma estrutura em forma de árvore, chamada Árvore de Tarefas (*JobsTree*).

Nessa estrutura, seus elementos recebem o nome de nós. Os nós são organizados hierarquicamente por meio de arestas. Cada nó possui um único nó superior, chamado nó pai, com exceção do nó raiz (*root*). Os descendentes diretos de um nó chamam-se nós filhos. Um nó que não possui descendentes recebe o nome de nó folha. Nós que possuem pai comum chamam-se irmãos. Um conjunto de nós com pai comum recebe o nome de ramo. Cada nó representa uma tarefa, com exceção do nó raiz. Este nó especial é utilizado para indicar o ponto de entrada da árvore, logo não armazena dados referentes às tarefas. As tarefas iniciais são adicionadas a esse nó. À medida em que terminam, elas podem adicionar novas tarefas, de forma que a árvore é definida dinamicamente. Em cada ramo deve existir obrigatoriamente um nó que atuará como ponto de sincronização. Este é identificado pelo campo *wait* com valor *"true"*, enquanto os demais irmãos recebem o valor *"false"*.

Na Figura 1a é apresentado um exemplo de uma Árvore de Tarefas. Um nó que atua como ponto de sincronização é classificado como *RedNode*, representado em vermelho, enquanto os demais são classificados como *BlueNode*, representado em azul. Um *RedNode* será executado apenas após seus irmãos *BlueNode* e seus descendentes terminarem a execução. Os *BlueNodes*, por sua vez, não necessitam verificar o estado da execução dos nós irmãos, de forma que podem executar assim que houver computadores disponíveis.



(a) Visão do Usuário (b) Visão do Escalonador de Tarefas

Figura 1. Árvore de Tarefas e Escalonador

Por exemplo, ainda considerando a Figura 1a, as tarefas dos nós 1 e 2 (azuis) podem executar em paralelo, mas a tarefa do nó 3 (vermelho) só iniciará sua execução após o término dos nós 1 e 2 e seus descendentes (nós 4, 5 e 6). Considerando o ramo com as folhas 4, 5 e 6, as tarefas dos nós 4 e 5 (azuis) executarão em paralelo, enquanto a tarefa do nó 6 (vermelho) esperará a conclusão dessas duas para poder executar. O mesmo acontece com as tarefas do ramo com folhas 7, 8 (azuis) e 9 (vermelha). A tarefa do nó 9 irá executar apenas depois que as tarefas dos nós 7 e 8 terminarem. Logo, uma possível sequência de execução seria 1, 2, 4, 5, 6, 3, 7, 8 e 9.

Além da ordem de execução, também é necessário definir para onde as informações serão repassadas quando uma tarefa termina. Quando uma tarefa termina e cria tarefas adicionais, as informações necessárias são repassadas para as suas tarefas filhas, independente dessa ser um *RedNode* ou *BlueNode*. Por exemplo, ao terminar, a tarefa 1 repassa as informações necessárias para as tarefas 4, 5 e 6. A tarefa 3 repassa as informações para as tarefas 7, 8 e 9. Quando a tarefa recém-terminada é um *BlueNode*, sua informação é repassada para o nó de sincronização (*RedNode*) do ramo corrente. Por exemplo, as saídas das tarefas 4 e 5 são repassadas à tarefa 6. Quando a tarefa recém-terminada é um *RedNode* que não cria tarefas adicionais, sua informação é repassada para o primeiro *RedNode* de um ramo anterior que ainda não tenha sido executado. Por exemplo, a saída da tarefa 6 é repassada à tarefa 3. Caso não exista um *RedNode* anterior que ainda não foi executado, significa que a execução da Árvore de Tarefas chegou ao resultado final. Essa situação ocorre quando a tarefa 9 termina.

Nota-se que na representação da Árvore de Tarefas, as arestas indicam o parentesco entre as tarefas à medida que novos conjuntos de tarefas (ramos) são inseridos. Ou seja, uma tarefa pai que cria outras filhas. Isso facilita a adição de novas tarefas por parte do usuário, no entanto, dificulta a descoberta de tarefas aptas à execução e a descoberta de que tarefa receberá a informação de outra. Para simplificar a descoberta de tarefas, o *cluster* modifica a ordem em que essas são inseridas na Árvore de Tarefas. Esse mecanismo é chamado de Escalonador de Tarefas, apresentado na Figura 1b.

No Escalonador de Tarefas, o *RedNode* de cada novo ramo é promovido a pai de seus irmãos e é inserido no lugar do nó recém-terminado. Caso o nó recém-terminado não possua descendentes, sua informação será repassada ao seu nó pai antes de ser removido. Logo, as arestas representarão o fluxo da informação e a árvore armazenará apenas nós que possuem tarefas ainda não executadas. Dessa forma, os nós folhas passam a conter tarefas aptas a execução. Ainda considerando a Figura 1b, cada etapa representa um estado da árvore após o fim de uma ou mais tarefas. Por exemplo, da etapa 1 para 2, a tarefa contida no nó 1 terminou, dando origem aos nós 4, 5 e 6. Como o nó 6 é vermelho, foi promovido a pai dos nós 4 e 5, e o ramo foi anexado ao nó 3. Nas etapas seguintes não houve a criação de novas tarefas, apenas a remoção de nós. A condição de parada passa a ser a inexistência de nós na Árvore de Tarefas.

Além da definição de qual ordem de execução das tarefas, também é necessário saber onde essas tarefas serão processadas, bem como o estado atual dos computadores que compõem o *cluster*. Dessa forma, a seção seguinte discute os demais componentes necessários para a implementação do *cluster* desenvolvido.

3.2. Arquitetura do *cluster* Redblue

Um *cluster* é composto por um conjunto de máquinas que executam uma aplicação distribuída de forma colaborativa. Normalmente um desses computadores é responsável exclusivamente por coordenar os demais. Nessa estrutura, o computador responsável pelo gerenciamento é classificado como Cabeça, enquanto os demais são chamados Escravos. A aplicação cliente é dividida em unidades menores, chamadas tarefas, que executam nos Escravos. Para liberar a aplicação cliente da responsabilidade de gerenciar as comunicações e a paralelização das tarefas, desenvolveu-se a arquitetura apresentada na Figura 2. Essa está dividida em três partes principais: Cabeça, Escravos e DRBL.

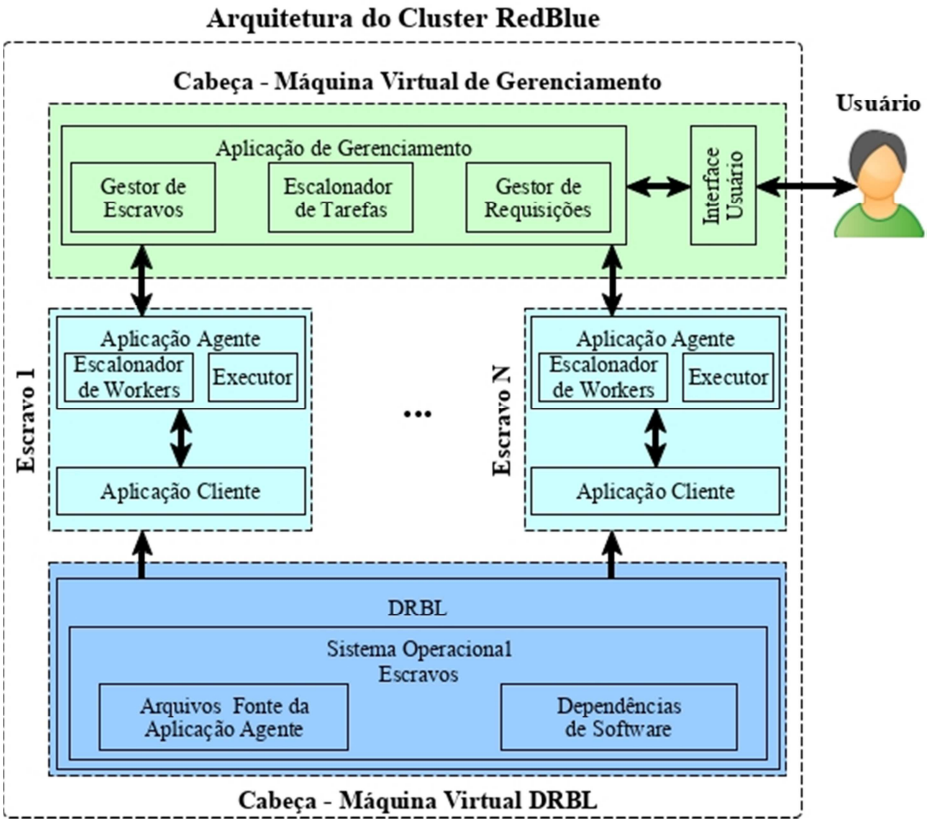


Figura 2. Arquitetura do cluster. Divisão de responsabilidades entre componentes do cluster, seus relacionamentos e módulos principais

A Cabeça é responsável por receber as requisições do usuário (Interface Usuário) e por gerenciar o *cluster* por meio da Aplicação de Gerenciamento. Essa, por sua vez, contabiliza o estado de carga de trabalho dos Escravos em um dado momento e qual tarefa esses estão executando (Gestor de Escravos). O Escalonador de Tarefas define quais tarefas estão prontas para executar, criando um fila de execução. O Gestor de Requisições envia informações sobre novas tarefas aos Escravos, bem como recebe a saída das tarefas que terminaram, atualizando a Árvore de Tarefas.

Os Escravos são responsáveis por receber ordens da Cabeça, executar a Aplicação Cliente, recolher a saída e enviá-la de volta à Cabeça. Isso é realizado por meio da Aplicação Agente, um serviço que inicia junto ao sistema operacional dos Escravos. Essa é composta pelo Escalonador de *Workers*, responsável por definir se a

Aplicação Cliente executará em um núcleo específico ou não, e um Executor, responsável por colocar a Aplicação Cliente em execução, monitorar o estado do processo da Aplicação Cliente e capturar os resultados da execução.

Para evitar causar alterações nos computadores que serão utilizados como Escravos, optou-se por carregar um sistema operacional através da rede. Assim, é possível utilizar um sistema operacional único sem alterar a configuração das máquinas.

O DRBL (*Diskless Remote Boot in Linux*) [Shiau et al. 2017] é incumbido de disponibilizar uma imagem do sistema operacional, previamente configurado com as dependências de *software*, para os Escravos. Essa imagem contém o código fonte da Aplicação Agente (Arquivos Fonte da Aplicação Agente) e as bibliotecas necessárias para execução da Aplicação Agente e da Aplicação Cliente (Dependências de Software).

A Cabeça e o DRBL são instalados em máquinas virtuais (*Virtual Machines - VMs*) distintas. Optou-se pela virtualização para manter simultaneamente versões pré-configuradas da VM DRBL com as dependências para diferentes linguagens. Dessa forma é possível trocar as dependências de *software* apenas ligando outra VM DRBL. A VM Cabeça foi instalada na mesma máquina física para utilizar apenas um computador para gerenciamento e distribuição da imagem do sistema operacional. A comunicação entre Aplicação de Gerenciamento e Aplicações Agentes é realizada por meio de chamadas de métodos remotos, assim como a comunicação entre Interface Usuário e Aplicação de Gerenciamento. Entretanto, os arquivos executáveis da Aplicação Cliente são disponibilizados via NFS (*Network File System*). A Cabeça disponibiliza uma pasta compartilhada onde os arquivos do usuário são copiados e mapeados para os Escravos.

3.2.1. Atribuição de tarefas e balanceamento de carga

Atualmente é muito comum que os computadores possuam processadores com vários núcleos. Essa característica pode ser explorada para que se possa submeter mais de uma tarefa a um computador. No entanto isso pode ser ruim dependendo do consumo de memória e do número de *threads* utilizado pela tarefa. Para tornar possível que o usuário escolha a melhor alternativa para sua aplicação, optou-se por dividir cada computador (Escravo) em estruturas menores chamadas *Workers* (Trabalhadores). O número de *Workers* determina quantas tarefas um computador pode alocar simultaneamente. Por padrão, o número de *Workers* é igual ao número de núcleos físicos, mas isso pode ser alterado pelo usuário.

O número de tarefas simultaneamente em execução depende do número de computadores disponíveis e da própria aplicação cliente. Em alguns casos, as tarefas aguardarão até que algum computador esteja livre, em outros, vários estarão disponíveis. Nesta situação se opta pelo Escravo menos sobrecarregado, ou seja, que está executando menos tarefas. Isso foi implementado criando filas separadas para tarefas (*jobs_queue*) e computadores (*nodes_queue*), conforme apresentado na Figura 3. Dessa forma o *cluster* se adapta automaticamente ao número de computadores. A fila de tarefas é construída a partir das tarefas aptas à execução na Árvore de Tarefas, ou seja, dos nós folhas. Uma verificação do estado dos nós garante que uma tarefa será enfileirada apenas uma vez em *jobs_queue*. Essas operações são executadas pelo Escalonador de Tarefas. A fila de computadores é gerenciada pelo Gestor de Escravos, que é ordenada de forma decrescente pelo número de *Workers* livres (*free*). Quando

uma tarefa é alocada em um computador, este é movido para o fim da fila e tem o número de *Workers* livres decrementado e de ocupados (*busy*) incrementado. Esse mecanismo faz com que a fila seja circular. Após um ciclo completo a fila é reordenada para manter a carga balanceada entre os computadores. Uma atribuição de tarefa pode ser interrompida até que existam computadores livres. Quando isso ocorre, o Escalonador de Tarefas passa a aguardar a notificação do Gestor de Escravos. Ao término de uma tarefa, a notificação é disparada, reiniciando o processo de alocação de tarefas.

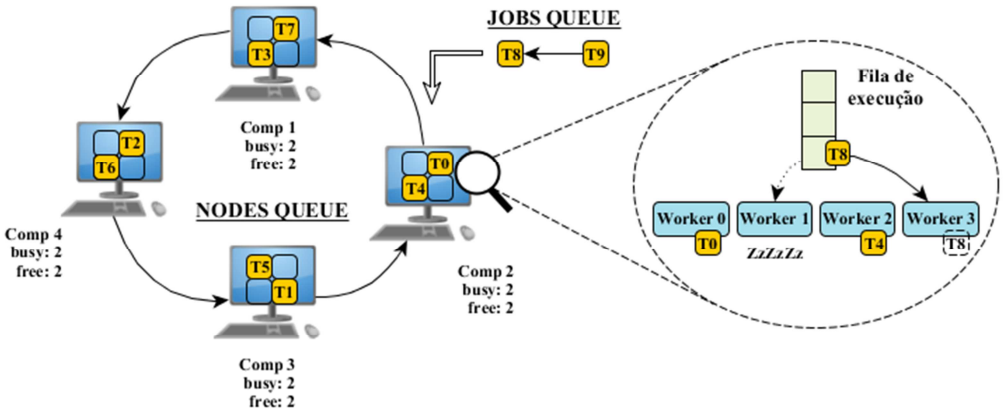


Figura 3. Atribuição de tarefas. Os retângulos amarelos representam tarefas. Cada quadrante representa um *Worker*. Uma tarefa alocada no quadrante representa uma atribuição de tarefa ao computador

Cada Escravo também possui seu próprio sistema de alocação de tarefas, destacado à direita da Figura 3, o Escalonador de *Workers*. Um *Worker* é um processo gerenciado pela Aplicação Agente. Por sua vez, cada *Worker* gerencia o estado do processo da Aplicação Cliente (tarefa) alocada a ele. Isso permite associar um *Worker,* e consequentemente a tarefa, a um núcleo especifico do processador por meio da técnica de afinidade entre processos, *CPU Affinity*. O usuário pode ativá-la por meio do parâmetro CPU_AFFINITY. Isso reduz as perdas causadas pela realocação de processos para aplicações *single thread* em processadores com múltiplos núcleos.

3.2.2. Formato de comunicação

Uma vez que a Aplicação Cliente não gerencia o *cluster* diretamente, é necessário definir uma interface de controle. A Aplicação Cliente recebe os parâmetros de entrada utilizando um mecanismo similar à passagem de argumentos por linha de comando do Linux. Um parâmetro é definido utilizando-se dois traços (--) seguidos do seu nome e uma lista de valores separados por espaços. Por exemplo, na Figura 4 uma tarefa carrega uma matriz 10x4 (--dim) a partir de um arquivo passado como parâmetro (--path). Esse mecanismo permite executar uma tarefa no *cluster* ou no terminal, facilitando os testes.

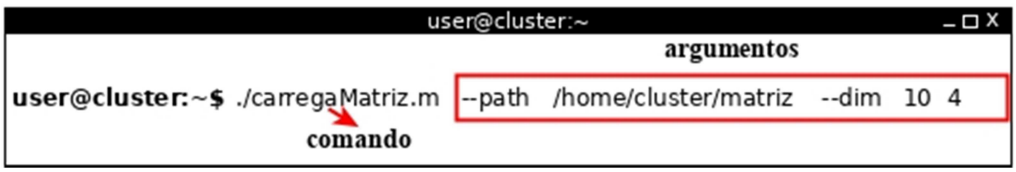


Figura 4. Exemplo de passagem de parâmetros via terminal no Linux

Por sua vez, o conjunto inicial de tarefas e a saída de uma Aplicação Cliente são retornadas em formato JSON (*Javascript Object Notation*), respeitando a estrutura de controle definida a seguir:

- *command* - Nome do arquivo que será executado. Por exemplo, caso o arquivo possua o nome job_setup.m, o campo *command* receberá o valor "job_setup.m".
- *args* - Recebe uma *string* contendo os parâmetros para aplicação definida em *command*. O formato de dados segue o mesmo padrão exibido na Figura 4.
- *wait* - Recebe os valores em formato texto "true" ou "false". É utilizado para definir a precedência de execução entre as tarefas da aplicação cliente. Em um conjunto inicial de tarefas e nas definidas em *next_jobs*, apenas uma pode ter valor "true".
- *job_output* - Recebe os resultados da execução da aplicação em formato JSON. Os valores definidos nesse campo serão redirecionados à tarefa responsável pelo agrupamento de informações ou serão a saída final da execução do conjunto de tarefas.
- *next_jobs* - Recebe um *array* de objetos em formato JSON contendo as chaves *command*, *args*, e *wait* quando deseja-se encadear novas tarefas ou um *array* vazio quando a tarefa não possui descendentes. Por exemplo, caso *next_jobs* esteja vazio, não serão criadas novas tarefas. No entanto, uma definição conforme: *{"next_jobs": [{"command": "job1", "args": "--valor 1", "wait":"true"},{"command": "job2", "args": "--opcao a", "wait":'false"}]}*, indica que serão criadas as tarefas *job1* e *job2*, onde *job1* tem precedência sobre *job2*.
- *job_input* - Recebe um *array* em formato JSON com os valores contidos no campo *job_output* de outras tarefas. Esse campo é criado pelo *cluster* e é utilizado para repassar as informações de outras tarefas à responsável por agrupar seus resultados. Por exemplo, se a tarefa "c" é responsável por agrupar os valores das tarefas "a" e "b" com saídas respectivamente iguais a {"val": "1"} e {"val": "2"}, a tarefa "c" receberá o parâmetro job_input da seguinte forma: *--job_input "[{\"val\": \"1\"}, {\"val\": \"2\"}]"*.

Na Figura 5 é apresentado um exemplo do fluxo e formato de dados utilizados da requisição inicial do usuário até o recebimento dos resultados. Em 1 o usuário submete um conjunto inicial de tarefas ao *cluster*. Em 2 o *cluster* extrai a tarefa que será executada e a coloca em execução. Observa-se que parâmetros mais complexos podem ser repassados em formato JSON. Em 3 o resultado da execução é capturado pelo *cluster*, que devolve ao usuário em 4 por meio de um arquivo. O resultado final é proveniente do campo *job_output* da instância final da Aplicação Cliente.

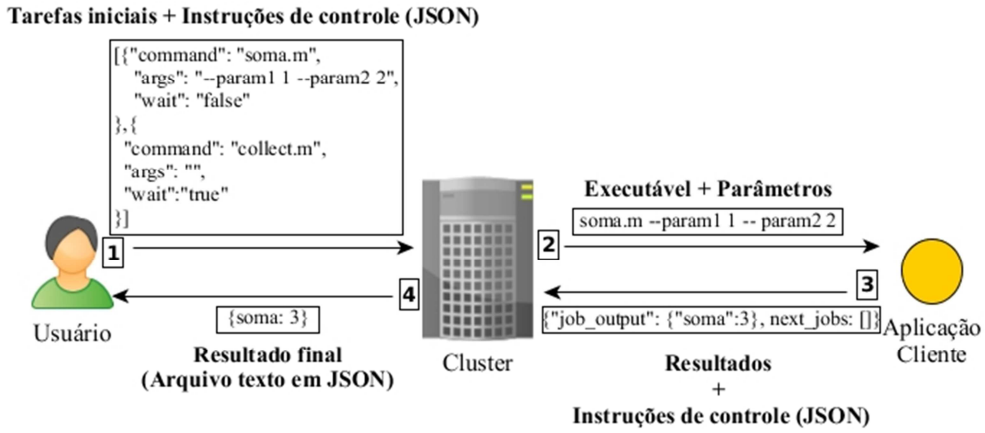


Figura 5. Formato e fluxo de informações no cluster. Exemplo das etapas de comunicação entre Usuário, Cluster e Aplicação Cliente

3.2.3. Execução de código cliente e meio de comunicação

A execução do código cliente é similar a inicialização de uma aplicação no terminal do Linux. Normalmente, aplicações via linha de comando possuem como saída padrão (STDOUT) o próprio terminal, e recebem dados via entradas de texto também pelo terminal, sendo essa a entrada padrão (STDIN). É comum que as exceções sejam exibidas na tela (STDERR). No *cluster,* os dados de entrada e saída são manipulados por outro processo (*Worker*), que tem acesso a essas informações ao redirecionar os descritores padrão. Dessa forma o processo *Worker*, por meio do Executor, cria um canal de comunicação via *pipes* com o processo da Aplicação Cliente. A entrada de dados é feita por meio de STDIN, utilizando o formato de dados descrito na Seção 3.2.2.

Um utilitário de recuperação de parâmetros, comumente conhecido como *option parser*, é utilizado para recuperar os parâmetros e transformá-los em variáveis. A saída de dados é obtida por meio da captura das mensagens que seriam impressas no terminal. Logo, a Aplicação Cliente precisa imprimir as informações no formato apresentado na Seção 3.2.2. O Executor sabe reconhecer o *status* de saída da Aplicação Cliente, sucesso ou erro, com base na análise do código de saída da aplicação. Caso a saída indique um erro, o *cluster* iniciará o tratamento de exceção.

O *cluster* possui suporte a linguagens compiladas e interpretadas. No segundo caso é necessário informar ao sistema operacional qual será o interpretador utilizado. Isso é feito diretamente no arquivo do *script* utilizando-se do *Shebang* "#!". Por exemplo, a instrução "*#!/usr/bin/octave*" na primeira indica que se utilizará o Octave.

Em resumo, para ocorrer a comunicação entre *cluster* e Aplicação Cliente é necessário que essa seja capaz de: ler e escrever de/para um terminal; identificar parâmetros e extrair valores numa estrutura idêntica à ARGV; e manipular dados em formato JSON. Uma vez que se utilizou apenas funcionalidades básicas, várias linguagens serão compatíveis com o *cluster*, tornando-o mais flexível.

3.3. Métricas e Testes de desempenho

As métricas utilizadas nos testes visam observar o ganho de desempenho da aplicação cliente e a eficiência do *software* de gerenciamento de operações distribuídas, possibilitando a descoberta de pontos a serem melhorados. Dessa forma, focou-se em verificar o aumento de desempenho percebido pelo usuário. Para isso mediu-se quantas vezes a aplicação se tornou mais rápida quando submetida ao *cluster* em comparação a execução em um único computador (*speedup*), dado pela Equação 1 (Amdahl,1967):

$$S(p) = \frac{T(1)}{T(p)} \quad , \tag{1}$$

onde *S(p)* é o *speedup*, *T(1)* o tempo de processamento sequencial e *T(p)* é o tempo de processamento paralelo para p processadores. Segundo Amdahl (1967), o *speedup* de uma aplicação é limitado pelo percentual de código inerentemente sequencial, tornando difícil uma paralelização perfeita. Além disso, aplicações distribuídas sofrem com perdas causadas pela latência da comunicação e pelo desbalanceamento da carga de trabalho entre os processadores. Por isso, dificilmente se obtém um ganho de desempenho igual ao número de processadores utilizados (*Speedup* Ideal), uma vez que sempre haverá algum tipo de perda (*overhead*) (Eijkhout, Chow, e van Geijin, 2016). Para se verificar o percentual de aproveitamento dos recursos computacionais, calculou-se a eficiência do *cluster*, dada pela Equação 2 [Eijkhout et al., 2016]:

$$E(p) = \frac{S(p)}{p} \quad , \tag{2}$$

onde *E(p)* é a eficiência e *S(p)* o *speedup* para *p* processadores. Essa é dada pela razão entre o *speedup* alcançado e o ideal esperado para o mesmo número de processadores.

3.4. Materiais

No ambiente de desenvolvimento, testes e produção utilizaram-se os mesmos modelos de computador, variando-se o número de equipamentos empregados simultaneamente. Respectivamente 4 máquinas como Escravos no *cluster* no ambiente de testes e 60 no ambiente de produção, onde utilizou-se um laboratório de informática. Esses ambientes compartilham o mesmo servidor de gerenciamento do *cluster*. Utilizou-se computadores Dell Optiplex 9020, equipados com processadores Intel Core i7-4770, 8Gb de memória RAM e placas de rede *gigabit ethernet*. Para conexões de rede se utilizou *switchs gigabit* interligados por fibra óptica. No ambiente de desenvolvimento se utilizou um único computador com 16 Gb de RAM. Nesse, máquinas virtuais atuaram como Escravos.

O sistema de gerenciamento do *cluster* foi desenvolvido na linguagem Ruby, utilizando-se a implementação oficial escrita em C, na versão 2.4.1. Utilizou-se os módulos *Open3* para executar outra aplicação e ter acesso às suas interfaces de comunicação e o módulo *DRb* para chamadas de método remoto, RMI. Alguns pacotes adicionais de *software*, chamados de *gems* (gemas) pela comunidade Ruby, também foram utilizados. As *gems*: *net-ssh*, *net-scp* e *net-ping* para incluir código atualizado no servidor (*deploy*) e testar se estão respondendo aos comandos; *daemons* para transformar programas que rodam em primeiro plano ininterruptamente em serviços que respondam aos comandos de gerenciamento de serviço do Linux; *rspec* para testes

automatizados; *msgpack* e *msgpack-rpc* para serialização/desserialização no formato *msgpack* e comunicação via interface *Unix Domain Socket*; *sys-cpu* para obter informações sobre número de núcleos e a associação entre núcleos lógicos e físicos do processador; *ffi* para utilizar trechos da API Linux escrita em C diretamente de um código Ruby, utilizada para definir a afinidade entre processos e núcleos; e *Oj* para conversão de dados no formato JSON.

No ambiente de testes e produção escolheu-se utilizar a distribuição Debian Jessie 8.x para ser o sistema operacional do *host* e *guests* devido a sua estabilidade e baixo consumo de recursos. Para a virtualização utilizou-se KVM/ QEMU com *drivers Virtio*. Para gerenciar as máquinas virtuais utilizou-se o software *Virt Manager*. Para disponibilizar o sistema operacional dos Escravos se utilizou o software DRBL sobre o sistema operacional Debian Jessie 8.x. O protocolo PXE (Preboot eXECUTION Environment) foi utilizado para que os computadores inicializassem pelo sistema operacional oferecido na rede. As aplicações clientes que executam sobre o *cluster* (*jobs*) e as bibliotecas utilitárias foram desenvolvidas na linguagem Octave versão 4.2. Utilizou-se o pacote o pacote JSONlab para manipulação e conversão de texto em JSON para estrutura de dados na linguagem Octave.

4. Experimentos e Resultados

A necessidade de otimização de parâmetros e repetição de simulações é uma prática comum a engenheiros e pesquisadores. Para se aproximar dessa realidade, desenvolveu-se duas aplicações de inteligência computacional que necessitam de otimização de parâmetros e possuem elevado tempo de execução.

A primeira foi baseada no trabalho de Vasconcelos et at. (2001), que buscava descobrir que tipos de operadores e procedimentos eram mais apropriados para determinados tipos de algoritmos genéticos. Contava-se o número de convergências para o mínimo global para cada conjunto de parâmetros. Parte desse trabalho foi implementada, atendo-se ao SGA (*Simple Genetic Algorithm*). Manteve-se a mesma metodologia de testes do trabalho base, partindo-se de uma configuração fixa para depois utilizar os melhores operadores/procedimentos da iteração anterior na próxima iteração. Isso resultou em 17 configurações possíveis e um tempo médio de execução de 46 minutos para um único processador. Nomeou-se essa aplicação como GA-01. Modificou-se a aplicação para testar todas as combinações, resultando em 432 configurações possíveis e um tempo médio de execução de 25,63 horas (1537,8 minutos) para um único processador. Nomeou-se essa versão como GA-02.

A segunda aplicação usou uma rede neural *Multilayer Perceptron* (MLP) com retropropagação para reconhecimento de caracteres através do pacote *nnet* para Octave. Buscava-se verificar o impacto do número de neurônios e camadas ocultas na acurácia das classificações obtidas. O conjunto de dados (*dataset*) *Letter Image Recognition Data* [Slate 1991] com 20000 padrões foi escolhido para testar a aplicação. O objetivo desse *dataset* é identificar qual letra maiúscula (A até Z) é formada a partir de um conjunto de pixeis. Modelou-se a rede neural com 16 neurônios na camada de entrada, uma para cada atributo, e 26 neurônios na de saída, um para cada letra. Variou-se o número de camadas ocultas de 1 a 2, com número de neurônios de 1 a 28 e 0 a 28, respectivamente. Para a fase de treinamento reservou-se 20% (4000 padrões) do *dataset*, 10% (2000 padrões) para validação e o restante para o teste de classificação. Ao variar o

número de camadas e de neurônios obteve-se 812 configurações possíveis. Salienta-se que pacote *nnet* utiliza todos os núcleos disponíveis por padrão. Dessa forma, considerou-se o tempo de execução T(1) para um computador (4 núcleos) e T(*p*) o tempo para *p* computadores no cálculo do *speedup*. T(1) totalizou cerca de 28 horas (1680 minutos). Essa aplicação foi nomeada como NN-01.

Cada aplicação foi modelada em três tarefas com responsabilidades bem definidas. A de configuração é responsável por criar as tarefas de processamento, distribuindo os parâmetros entre essas. A de processamento contém as regras de negócio para resolução do problema, recebendo um conjunto de parâmetros configuráveis. A tarefa de agregação de resultados é responsável por consolidar e formatar a saída das tarefas de processamento. Apenas as tarefas de processamento executam em paralelo. Porém, o tempo de execução dessas instâncias é elevado o suficiente para que se despreze o tempo gasto pelas tarefas de configuração e agregação de resultados. Logo, considerou-se que as aplicações desenvolvidas são altamente paralelizáveis.

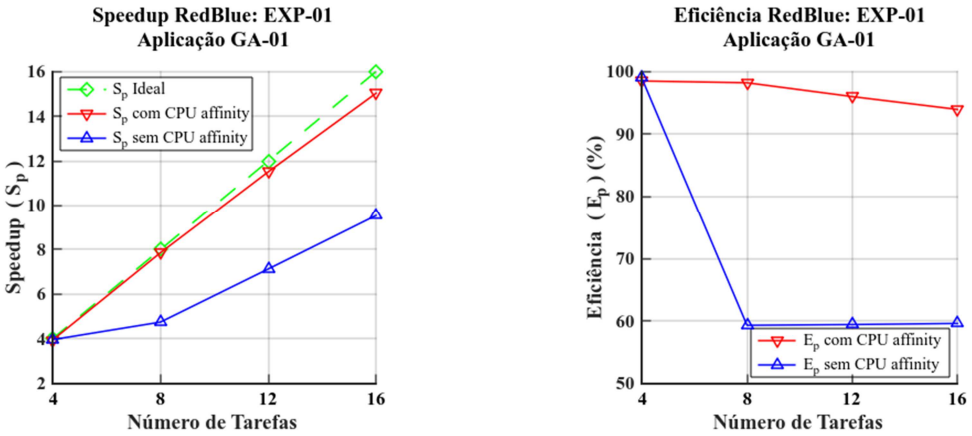
A aplicação GA-01 foi executada apenas no ambiente de testes do *cluster*, com 4 Escravos. As tarefas de processamento executaram cópias idênticas da versão sequencial da aplicação GA-01. Testou-se com 4, 8, 12 e 16 tarefas de processamento distribuídas igualmente entre as mesmas quantidades de *Workers*. Ou seja, com 4 tarefas, utilizou-se 1 *Worker* em cada computador, com 8 tarefas, 2 *Workers* em cada computador, e assim sucessivamente. Cada configuração foi testada em dois cenários: com o escalonador de processos padrão do sistema operacional e com escalonamento estático, *CPU Affinity*. Nomeou-se esse experimento como EXP-01.

A aplicação GA-02 foi executada no ambiente de produção com a opção CPU_AFFINITY ativa. Cada máquina executou 4 tarefas simultaneamente, isto é, cada Escravo instanciou 4 *Workers*. Para testar a escalabilidade do *cluster* se variou o número de Escravos, mas manteve-se o número de tarefas inalterado. Logo, 432 tarefas de processamento foram executadas em um *cluster* composto por 10, 20, 30, 40, 50 e 60 computadores. Nomeou-se esse experimento como EXP-02.

A aplicação NN-01 foi executada em *clusters* com os mesmos tamanhos utilizados por GA-02, de 10 à 60, porém com 812 tarefas de processamento. Cada computador recebeu uma única tarefa de cada vez devido ao alto consumo de memória. Todos os processadores disponíveis foram utilizados devido a implementação do pacote *nnet*. A opção CPU_AFFINITY foi desativada para não interferir no funcionamento do desse pacote. Nomeou-se esse experimento como EXP-03.

Para cada experimento se calculou o *speedup* e a eficiência do *cluster*, exibidos em forma de gráficos. O *speedup* ideal foi incluído para facilitar a análise dos resultados. Na Figura 6 são apresentados os resultados obtidos para o experimento EXP-01. Nota-se um desempenho superior ao se utilizar afinidade de processos frente ao mecanismo padrão de alocação de processos. O *speedup* para o cenário com *CPU Affinity* se aproximou do ideal enquanto que com o mecanismo padrão se manteve distante. Ao observar o gráfico de eficiência se verifica que essa diferença chegou a cerca 40% para 8 tarefas e a 35% para 16 tarefas (quatro tarefas por computador). Na abordagem com afinidade de processos houve uma perda de eficiência de apenas 5%, em relação ao *speedup* ideal, na execução de 16 tarefas. Para o mecanismo padrão houve queda brusca de eficiência na transição de 4 para 8 tarefas, estabilizando-se em

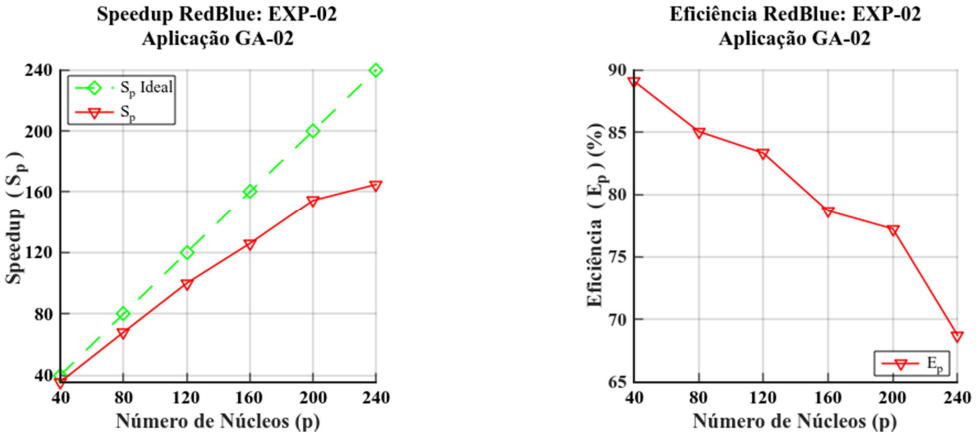
cerca de 60% para 12 e 16 tarefas. Verificou-se que o desempenho ruim do mecanismo padrão se deve a alocação de duas tarefas a núcleos lógicos pertencentes ao mesmo núcleo físico do processador, causando disputa por recursos. Isso não ocorre ao se definir a afinidade das tarefas, uma vez que o *cluster* faz considerações sobre a distribuição dos núcleos lógicos.



(a) *Speedup* (b) Eficiência

Figura 6. Gráficos do *speedup* e eficiência para experimento EXP-01

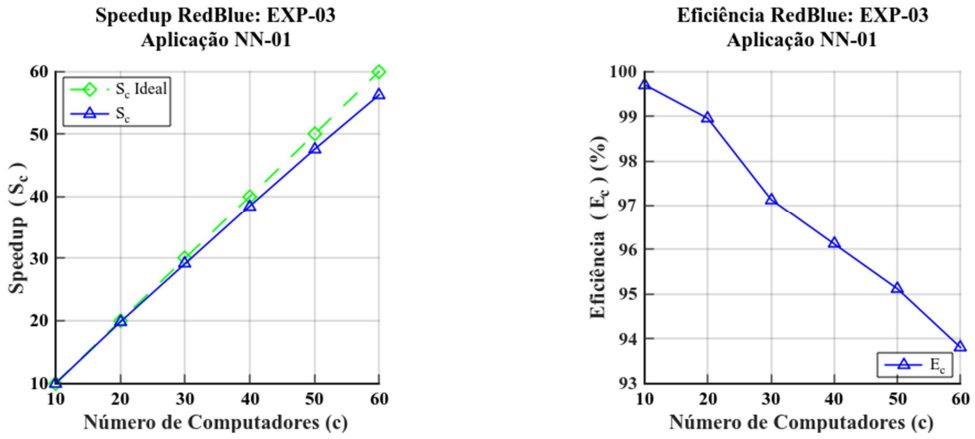
Na Figura 7 são apresentados os resultados obtidos para o experimento EXP-02. Observa-se uma diminuição gradual do *speedup* com redução mais acentuada entre 200 e 240 núcleos. Para 240 núcleos a eficiência foi de cerca de 68%. A queda mais expressiva de eficiência ocorreu no aumento de 200 para 240 núcleos, uma perda de cerca de 7%. Entre 80 e 120, e também entre 160 e 200 núcleos utilizados, houve uma perda de eficiência relativamente menor do que a observada entre 40 e 80 núcleos, 120 e 160 núcleos, e 200 e 240 núcleos. Isso indica que entre 80 e 120, e também entre 160 e 200 núcleos, houve uma melhor distribuição de tarefas.



(a) *Speedup* (b) Eficiência

Figura 7. Gráficos do *speedup* e eficiência para experimento EXP-02

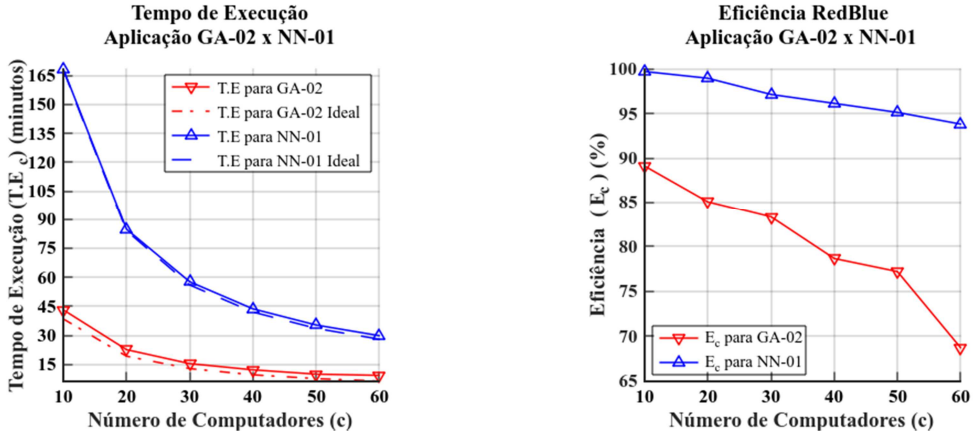
Na Figura 8 são apresentados os resultados obtidos para o experimento EXP-03. Observa-se baixa perda de eficiência para todos os tamanhos de *cluster*. A maior ocorreu na transição entre 20 e 30 computadores, cerca de 2%. As demais perdas se mantiveram em cerca de 1% a cada 10 computadores adicionados. Para 60 máquinas houve um aproveitamento de 93%. O *speedup* manteve-se próximo ao ideal.



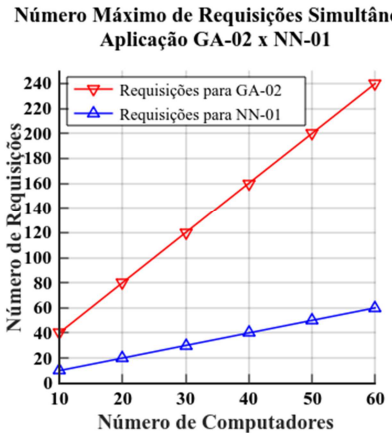
(a) *Speedup* (b) Eficiência

Figura 8. Gráficos do *speedup* e eficiência para experimento EXP-03

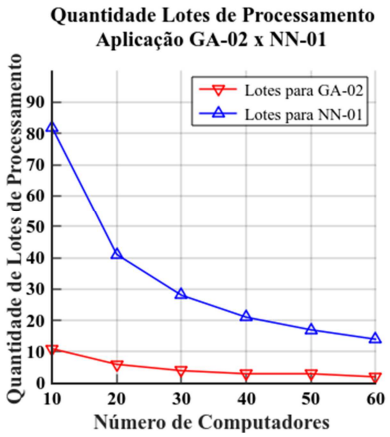
Na Figura 9 são apresentadas comparações entre as aplicações GA-02 e NN-01. Embora as aplicações GA-02 e NN-01 tenham obtido uma redução expressiva no tempo de execução, conforme a Figura 9a, notam-se diferenças na eficiência obtida para os mesmos tamanhos de *clusters*.



(a) Tempo de Execução (b) Eficiência



(c) Número máximo de requisições possíveis



(d) Número de lotes de processamento

Figura 9. Gráficos do tempo de execução, eficiência, número máximo de requisições simultâneas e quantidades de lotes utilizados, uma comparação entre as aplicações GA-02 e NN-01

Uma das explicações para isso é o número de requisições simultâneas. Na Figura 9c se apresenta uma comparação entre máximo de requisições simultâneas possíveis de cada aplicação ao variar o tamanho do *cluster*. O gráfico da eficiência, Figura 9b, foi adicionado para se verificar a relação entre número de requisições e a eficiência obtida. Observa-se que a diminuição da eficiência é proporcional ao aumento do número de requisições simultâneas. Isso indica que há um gargalo no Gestor de Requisições, fazendo com que atrasos adicionais apareçam a medida que o *cluster* aumenta de tamanho. Outro fator que pode reduzir o desempenho é o desbalanceamento de carga no lote final de processamento. O tamanho do lote é definido pelo número de tarefas que podem executar simultaneamente. Para GA-02 e NN-01 esse limite é número total de *Workers* do *cluster*. Isso acontece porque o número de tarefas de processamento é superior ao total de *Workers* para as configurações testadas. O número de lotes de processamento pode ser calculado se dividindo o total de tarefas pelo número total de *Workers*. Provavelmente o resultado dessa divisão não será exato, o que significa que será necessário adicionar mais um lote para se executar as tarefas remanescentes. Portanto, existirão processadores ociosos nesse último lote, o que acarreta na diminuição da eficiência. Na Figura 9d é apresentada uma comparação da quantidade de lotes necessários para aplicação GA-02 e NN-01. Observa-se que para aplicação GA-02 o aumento do *cluster*, de 30 a 50 computadores, possui pouco impacto na diminuição do número de lotes necessários. Enquanto o número de computadores cresceu em 66% nessa faixa, o número de lotes de processamento diminuiu em apenas 25%, constatando o desperdício de recursos. Isso pode ser atenuado ao se alocar mais tarefas que o número de processadores em lotes intermediários, caso haja memória disponível. O desempenho de algumas tarefas poderia ser sacrificado devido ao compartilhamento de recursos, mas a eficiência geral da aplicação seria melhor.

5. Discussão

A plataforma apresentada neste trabalho para o desenvolvimento de simulações computacionais se mostrou viável para utilização em laboratórios de informática. Em

especial para laboratórios que são utilizados para múltiplas finalidades, como em instituições de ensino, visto que é possível ativar o modo *cluster* apenas quando necessário. A capacidade do *cluster* de se adaptar à quantidade de máquinas automaticamente e a necessidade de instalação em apenas um computador também são pontos positivos para utilizá-lo nesse tipo de ambiente. Outra característica interessante é a possibilidade de utilizar apenas as duas primeiras camadas da arquitetura. Dessa forma é possível ter um ambiente de desenvolvimento "*stand alone*" em um computador isolado. Isso é útil para que os usuários possam desenvolver e testar as aplicações em seus computadores pessoais. Ambas propriedades tornam a plataforma RedBlue interessante para instituições de ensino que não possuem uma plataforma dedicada de alto desempenho, mas têm laboratórios de informática disponíveis.

A plataforma também se mostrou capaz de acelerar as simulações computacionais. Os resultados dos experimentos mostraram uma redução expressiva no tempo de execução das simulações testadas. No experimento EXP-02 o tempo de execução sequencial foi de cerca de 25,63 horas. Ao utilizar o *cluster* com 60 computadores o tempo de execução para esse experimento foi reduzido para 10 minutos em média. Para o experimento EXP-03 o tempo de execução foi reduzido de 28 horas para cerca de 30 minutos para a mesma quantidade de máquinas no *cluster*. No entanto, em todos os testes foram observadas perdas de eficiência à medida em que mais máquinas são adicionadas ao *cluster*. Porém, esse comportamento era esperado, visto que é comumente observado em outros trabalhos [Yang et al. 2008, Datti et al. 2015, Setiawan e Murdyantoro 2016].

A economia obtida com a redução no tempo de execução das simulações computacionais permite que se aproveite o tempo economizado em outras atividades. Na pesquisa, a obtenção rápida de resultados poderia reduzir o esforço realizado com soluções inadequadas, como, por exemplo, testes que se mostrem ineficazes após muitas horas de simulação. Além disso, o *cluster* poderia ser utilizado para testar simultaneamente várias configurações diferentes de uma simulação. Em ambos os casos essa ferramenta permite tornar o trabalho dos pesquisadores mais eficiente.

Cada um dos experimentos apresentados neste trabalho foi desenvolvido com uma finalidade específica. No experimento EXP-01 se observou que o escalonamento de processos tem forte impacto sobre o desempenho quando múltiplas instâncias estão em execução no mesmo computador. Originalmente esse experimento foi desenhado para verificar a degradação no desempenho a medida que o número de tarefas aumentava enquanto os recursos computacionais permaneciam constantes. Por essa razão se executou várias instâncias de uma tarefa com os mesmos parâmetros. Pensava-se que, ao executar instâncias idênticas de uma aplicação em um computador com múltiplos núcleos, o tempo de execução seria muito próximo em todas as instâncias. Logo, ao considerar o tempo de execução como uma constante, tornaria possível observar o *overhead* inserido pela paralelização de tarefas realizada pela plataforma. Porém, não foi o que aconteceu. Ao executar de duas a quatro instâncias no mesmo computador se verificou um aumento do tempo de execução da aplicação cliente, mesmo com memória RAM suficiente para isso. Esse problema foi identificado como sendo uma alocação inadequada de processos nos núcleos lógicos dos processadores causada pelo escalonador do sistema. Por essa razão se desenvolveu uma forma de alocação de processos otimizada para o *cluster*. Esse foi um achado importante, uma

vez que permitiu utilizar todos os núcleos de uma máquina com pouca perda se comparada à execução de uma instância única. Ao utilizar a afinidade entre processos, o tempo de cada instância foi penalizado em apenas 4,7% em média. Para o mecanismo padrão o aumento do tempo de execução de cada instância chegou a 63,2% para quatro núcleos em comparação ao tempo de uma instância em um computador ocioso. Logo, verificou-se que o mecanismo de alocação de processos desenvolvido para o *cluster* é mais adequado a aplicações *single thread*. Para aplicações com múltiplas *threads* é preferível utilizar o mecanismo de alocação padrão do sistema operacional para evitar a concentração das *threads* em um único núcleo.

O experimento EXP-02 foi desenvolvido para demonstrar o potencial do *cluster* para expandir o número de testes de configuração de parâmetros. Nesse experimento se executou a aplicação GA-02, que testou 432 combinações de parâmetros. Para 60 computadores foram gastos cerca de 10 minutos. A mesma foi desenvolvida a partir da aplicação GA-01, que testava apenas 17 variações de parâmetros, gastando cerca de 46 minutos. Ou seja, aumentou-se o número de combinações testadas em 25,4 vezes e ainda assim se reduziu o tempo de execução em 78%.

O experimento EXP-03 foi desenvolvido com o intuito de mostrar a aplicabilidade da plataforma a problemas muito sensíveis a variação de parâmetros, no caso, redes neurais artificiais. A calibração dos parâmetros em redes neurais é considerada uma tarefa complexa. Embora existam algumas estratégias para configurá-la, não há garantias que essas obterão bons resultados para todos os problemas. Dessa forma, é necessário modificar e testar os parâmetros várias vezes até encontrar uma boa configuração para o problema, uma tarefa que pode tomar muito tempo. Ao utilizar um *cluster* seria possível executar várias instâncias de uma rede neural com parâmetros diferentes simultaneamente e escolher aqueles que se mostrarem mais promissores. Dessa forma, o esforço e tempo gastos para encontrar a configuração ideal para o problema seriam reduzidos.

No experimento EXP-03 se obteve o melhor aproveitamento dos computadores disponíveis, cerca de 93% de eficiência, frente aos 67% de eficiência obtidos no experimento EXP-02 para 60 máquinas. Visto que ambos os experimentos foram executados com as mesmas quantidades de máquinas foi possível fazer comparações e especulações sobre a diferença obtida no critério eficiência. A primeira possibilidade é o número de requisições simultâneas. Para EXP-03 esse número poderia chegar a no máximo 60 enquanto que para EXP-02 poderia chegar a 240 requisições. Para 10 a 20 computadores no experimento EXP-02 poderiam ocorrer de 40 a 80 requisições simultâneas. Nesse intervalo a eficiência variou entre 89% a 85%, próximo a eficiência obtida para 60 computadores (60 requisições possíveis) em EXP-03, 93%. Logo, é possível concluir que a degradação no desempenho aumenta a medida que cresce o número de requisições simultâneas. A causa mais provável para isso é a incapacidade do Gestor de Requisições em tratar várias requisições simultaneamente. Isso provoca enfileiramento das requisições e consequentemente atrasos adicionais. Algumas soluções para esse problema seriam: utilizar apenas comunicação assíncrona; reduzir o tempo total de processamento das requisições otimizando o código; paralelizar os mecanismos da Aplicação de Gerenciamento; e utilizar múltiplas instâncias (processos) da Aplicação de Gerenciamento.

Outra explicação para a diferença da eficiência nos testes é causada pela estratégia de alocação de tarefas. O número de *Workers* é definido na inicialização do cluster e não é mais mudado. Dessa forma, tem-se um número máximo de tarefas que podem ser atendidas simultaneamente no cluster. Quando esse número de tarefas supera a quantidade de *Workers* é necessário que as demais tarefas esperem até que algum *Worker* esteja disponível. Em alguns casos o último lote terá poucas tarefas em execução, ou seja, a maioria das máquinas estará ociosa. Assim, a Aplicação de Gerenciamento irá esperar o término da última tarefa para retornar os resultados ao usuário. Observou-se esse comportamento nos experimentos realizados. Em EXP-03 houve um melhor aproveitamento dos lotes em comparação a EXP-02, o que explica a obtenção de uma eficiência melhor.

Algumas restrições para utilização da plataforma foram observadas. Apenas aplicações compatíveis com Linux funcionam no cluster. As aplicações utilizadas nos testes foram consideradas altamente paralelizáveis, visto que apenas as tarefas de configuração e coleta de resultados executaram sequencialmente. Logo, a utilização de aplicações menos paralelizáveis provavelmente resultará em piores tempos de execução. O conjunto de tarefas que executaram em paralelo tinha granularidade grossa. Isso significa que o tempo gasto para paralelizar as tarefas, iniciá-las e coletar os resultados é pequeno em relação ao tempo em que essas permanecem em execução. A utilização de tarefas com baixo tempo de execução traria deterioração drástica no desempenho do cluster. O cluster utiliza apenas troca de mensagens via rede como forma de comunicação, não havendo compartilhamento de memória. Isso implica que a quantidade de memória que uma tarefa pode utilizar está restrita a quantidade não utilizada no Escravo em que está executando. A comunicação entre *cluster* e aplicação cliente acontece apenas no início e no fim da execução de uma tarefa. Logo, não é possível que tarefas em execução se comuniquem. Os resultados intermediários das tarefas são armazenados em memória pela Aplicação de Gerenciamento, portanto está restrita a quantidade de memória disponível na Máquina Virtual de Gerenciamento. A maior parte dessas limitações é necessária para simplificar o processo de desenvolvimento das aplicações por parte do usuário.

Apesar das limitações é possível concluir que a plataforma RedBlue atingiu seus objetivos. As estratégias para distribuição de tarefas e a ocultação da complexidade de um sistema distribuído foram dois dos principais achados deste trabalho. Logo, além da utilização da plataforma em laboratórios de informática, espera-se que as ideias propostas possam contribuir para o surgimento de novas aplicações.

6. Conclusão

É indiscutível que o avanço tecnológico em diversas áreas do conhecimento só foi possível devido ao surgimento do computador. No entanto, ainda hoje, algumas dificuldades persistem devido à distância entre a computação como objeto de estudo e como ferramenta. Em virtude disso, muitos discentes, ou mesmo pesquisadores, deixam de utilizar ferramentas computacionais mais complexas por desconhecerem técnicas avançadas de programação. Um exemplo disso é a programação paralela e/ou distribuída, muito utilizada em supercomputadores ou *clusters*. Para contornar esse quadro, este trabalho propôs uma plataforma que permite utilizar os laboratórios de informática como *clusters* para desenvolvimento de simulações computacionais. A

plataforma permite abstrair conceitos complexos em computação paralela e distribuída, permitindo que pessoas com conhecimento básico em programação possam utilizá-la. Seu mecanismo de execução é flexível, tornando-a compatível com diferentes linguagens de programação. Além disso, a plataforma necessita ser instalada em apenas uma máquina.

Os resultados obtidos mostram uma redução de até 99% no tempo de execução do experimento no *cluster* com 60 máquinas, quando comparado a um único computador. Essa economia de tempo seria útil para pesquisas científicas. Atividades como refinamento de experimentos, desenvolvimento de novos testes ou análise mais profunda de resultados poderiam ser realizadas em menor tempo, aumentando a qualidade da pesquisa.

Dessa forma, considera-se que os objetivos propostos neste trabalho foram alcançados, visto que foi possível reduzir o tempo das simulações computacionais ao mesmo tempo que se simplificou o desenvolvimento das aplicações. Como trabalhos futuros, pretende-se desenvolver instaladores para várias distribuições Linux e criar um repositório para possibilitar a distribuição de atualizações da plataforma.

Agradecimentos

Os autores agradecem ao Instituto de Ciência e Tecnologia da UFVJM por disponibilizar os equipamentos para a realização desta pesquisa.

Referências

Amdahl, G. M. (1967). Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, spring jointcomputer conference on - AFIPS '67 (Spring)*, volume 30, pages 483–485, New York, New York, USA. ACM Press.

Bohn, B., Garcke, J., Iza-Teran, R., Paprotny, A., Peherstorfer, B., Schepsmeier, U., and Thole, C. A. (2013). Analysis of car crash simulation data with nonlinear machine learning methods. *Procedia Computer Science*, 18:621–630.

Chen, T., Chen, Y., Guo, Q., Zhou, Z.-H., Li, L., and Xu, Z. (2013). Effective and efficient microprocessor design space exploration using unlabeled design configurations. *ACM Transactions on Intelligent Systems and Technology*, 5(1):1–18.

Datti, A. A., Umar, H. A., and Galadanci, J. (2015). A Beowulf Cluster for Teaching and Learning. *Procedia Computer Science*, (70):62–68.

Eijkhout, V., Chow, E., and van Geijin, R. (2016). *Introduction to High Performance Scientific Computing*. Lulu.com, 2 edition.

Gholkar, N., Mueller, F., and Rountree, B. (2016). A Power-Aware Cost Model for HPC Procurement. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1110–1113. IEEE.

Gosney, A., Miller, J. H., Gorton, I., and Oehmen, C. (2011). An Adaptive Middleware Framework for Optimal Scheduling on Large Scale Compute Clusters. In *2011 Eighth International Conference on Information Technology: New Generations*, pages 713–718, Las Vegas, NV, USA. IEEE.

Goyal, N., Balasubramaniam, S., Goyal, P., Islam, S., and Sati, M. (2017). A high performance computing framework for data mining. *Proceedings - 23rd IEEE International Conference on High Performance Computing Workshops, HiPCW 2016*.

Malysiak, D. and Handmann, U. (2014). An efficient framework for distributed computing in heterogeneous Beowulf clusters and cluster-management. *CINTI 2014 - 15th IEEE International Symposium on Computational Intelligence and Informatics*, Proceedings, pages 169–178.

McHaney, R. (2009). *Understandind Computer Simulation*. Roger McHaney & bookboon.com, 1 edition.

Robinson, S. (2004). Simulation : *The Practice of Model Development and Use*. John Wiley & Sons Ltd, Chichester, England.

Setiawan, I. and Murdyantoro, E. (2016). Commodity cluster using single system image based on Linux/Kerrighed for high-performance computing. In *2016 3rd International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE)*, pages 367–372, Semarang, Indonésia. IEEE.

Shiau, S., Huang, K., Sun, C., Wang, J., Tsai, T., Nifenecker, J.-F., Chen, L., and Alagappan, N. (2017). DRBL Project. Disponível em: <http://drbl.org/>. Acesso em: 11 ago. 2017.

Slate, D. J. (1991). Letter Recognition Data Set. Disponível em: <https://archive.ics.uci.edu/ml/datasets/letter+recognition>. Acesso em: 19 ago. 2017.

Tang, F. and Ren, A. (2008). Agent-Based Evacuation Model Incorporating Fire Scene and Building Geometry. *Tsinghua Science and Technology*, 13(5):708–714.

Vasconcelos, J., Ramirez, J., Takahashi, R., and Saldanha, R. (2001). Improvements in genetic algorithms. *IEEE Transactions on Magnetics*, 37(5):3414–3417.

Yang, C. T., Hsieh, W. F., and Chen, H. Y. (2008). Implementation of a diskless cluster computing environment in a computer Classroom. *Proceedings of the 3rd IEEE Asia-Pacific Services Computing Conference, APSCC 2008*, pages 819–824.